

7

LOGICAL AGENTS

In which we design agents that can form representations of a complex world, use a process of inference to derive new representations about the world, and use these new representations to deduce what to do.

REASONING
REPRESENTATION
KNOWLEDGE-BASED
AGENTS

Humans, it seems, know things; and what they know helps them do things. These are not empty statements. They make strong claims about how the intelligence of humans is achieved—not by purely reflex mechanisms but by processes of **reasoning** that operate on internal **representations** of knowledge. In AI, this approach to intelligence is embodied in **knowledge-based agents**.

The problem-solving agents of Chapters 3 and 4 know things, but only in a very limited, inflexible sense. For example, the transition model for the 8-puzzle—knowledge of what the actions do—is hidden inside the domain-specific code of the `RESULT` function. It can be used to predict the outcome of actions but not to deduce that two tiles cannot occupy the same space or that states with odd parity cannot be reached from states with even parity. The atomic representations used by problem-solving agents are also very limiting. In a partially observable environment, an agent’s only choice for representing what it knows about the current state is to list all possible concrete states—a hopeless prospect in large environments.

LOGIC

Chapter 6 introduced the idea of representing states as assignments of values to variables; this is a step in the right direction, enabling some parts of the agent to work in a domain-independent way and allowing for more efficient algorithms. In this chapter and those that follow, we take this step to its logical conclusion, so to speak—we develop **logic** as a general class of representations to support knowledge-based agents. Such agents can combine and recombine information to suit myriad purposes. Often, this process can be quite far removed from the needs of the moment—as when a mathematician proves a theorem or an astronomer calculates the earth’s life expectancy. Knowledge-based agents can accept new tasks in the form of explicitly described goals; they can achieve competence quickly by being told or learning new knowledge about the environment; and they can adapt to changes in the environment by updating the relevant knowledge.

We begin in Section 7.1 with the overall agent design. Section 7.2 introduces a simple new environment, the wumpus world, and illustrates the operation of a knowledge-based agent without going into any technical detail. Then we explain the general principles of **logic**

in Section 7.3 and the specifics of **propositional logic** in Section 7.4. While less expressive than **first-order logic** (Chapter 8), propositional logic illustrates all the basic concepts of logic; it also comes with well-developed inference technologies, which we describe in sections 7.5 and 7.6. Finally, Section 7.7 combines the concept of knowledge-based agents with the technology of propositional logic to build some simple agents for the wumpus world.

7.1 KNOWLEDGE-BASED AGENTS

KNOWLEDGE BASE
SENTENCE

KNOWLEDGE
REPRESENTATION
LANGUAGE
AXIOM

The central component of a knowledge-based agent is its **knowledge base**, or KB. A knowledge base is a set of **sentences**. (Here “sentence” is used as a technical term. It is related but not identical to the sentences of English and other natural languages.) Each sentence is expressed in a language called a **knowledge representation language** and represents some assertion about the world. Sometimes we dignify a sentence with the name **axiom**, when the sentence is taken as given without being derived from other sentences.

INFERENCE

There must be a way to add new sentences to the knowledge base and a way to query what is known. The standard names for these operations are TELL and ASK, respectively. Both operations may involve **inference**—that is, deriving new sentences from old. Inference must obey the requirement that when one ASKs a question of the knowledge base, the answer should follow from what has been told (or TELLED) to the knowledge base previously. Later in this chapter, we will be more precise about the crucial word “follow.” For now, take it to mean that the inference process should not make things up as it goes along.

BACKGROUND
KNOWLEDGE

Figure 7.1 shows the outline of a knowledge-based agent program. Like all our agents, it takes a percept as input and returns an action. The agent maintains a knowledge base, *KB*, which may initially contain some **background knowledge**.

Each time the agent program is called, it does three things. First, it TELLS the knowledge base what it perceives. Second, it ASKS the knowledge base what action it should perform. In the process of answering this query, extensive reasoning may be done about the current state of the world, about the outcomes of possible action sequences, and so on. Third, the agent program TELLS the knowledge base which action was chosen, and the agent executes the action.

The details of the representation language are hidden inside three functions that implement the interface between the sensors and actuators on one side and the core representation and reasoning system on the other. MAKE-PERCEPT-SENTENCE constructs a sentence asserting that the agent perceived the given percept at the given time. MAKE-ACTION-QUERY constructs a sentence that asks what action should be done at the current time. Finally, MAKE-ACTION-SENTENCE constructs a sentence asserting that the chosen action was executed. The details of the inference mechanisms are hidden inside TELL and ASK. Later sections will reveal these details.

The agent in Figure 7.1 appears quite similar to the agents with internal state described in Chapter 2. Because of the definitions of TELL and ASK, however, the knowledge-based agent is not an arbitrary program for calculating actions. It is amenable to a description at

```

function KB-AGENT(percept) returns an action
  persistent: KB, a knowledge base
               t, a counter, initially 0, indicating time

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  action ← ASK(KB, MAKE-ACTION-QUERY(t))
  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t ← t + 1
  return action

```

Figure 7.1 A generic knowledge-based agent. Given a percept, the agent adds the percept to its knowledge base, asks the knowledge base for the best action, and tells the knowledge base that it has in fact taken that action.

KNOWLEDGE LEVEL

the **knowledge level**, where we need specify only what the agent knows and what its goals are, in order to fix its behavior. For example, an automated taxi might have the goal of taking a passenger from San Francisco to Marin County and might know that the Golden Gate Bridge is the only link between the two locations. Then we can expect it to cross the Golden Gate Bridge *because it knows that that will achieve its goal*. Notice that this analysis is independent of how the taxi works at the **implementation level**. It doesn't matter whether its geographical knowledge is implemented as linked lists or pixel maps, or whether it reasons by manipulating strings of symbols stored in registers or by propagating noisy signals in a network of neurons.

IMPLEMENTATION LEVEL

DECLARATIVE

A knowledge-based agent can be built simply by TELLing it what it needs to know. Starting with an empty knowledge base, the agent designer can TELL sentences one by one until the agent knows how to operate in its environment. This is called the **declarative** approach to system building. In contrast, the **procedural** approach encodes desired behaviors directly as program code. In the 1970s and 1980s, advocates of the two approaches engaged in heated debates. We now understand that a successful agent often combines both declarative and procedural elements in its design, and that declarative knowledge can often be compiled into more efficient procedural code.

We can also provide a knowledge-based agent with mechanisms that allow it to learn for itself. These mechanisms, which are discussed in Chapter 18, create general knowledge about the environment from a series of percepts. A learning agent can be fully autonomous.

7.2 THE WUMPUS WORLD

WUMPUS WORLD

In this section we describe an environment in which knowledge-based agents can show their worth. The **wumpus world** is a cave consisting of rooms connected by passageways. Lurking somewhere in the cave is the terrible wumpus, a beast that eats anyone who enters its room. The wumpus can be shot by an agent, but the agent has only one arrow. Some rooms contain

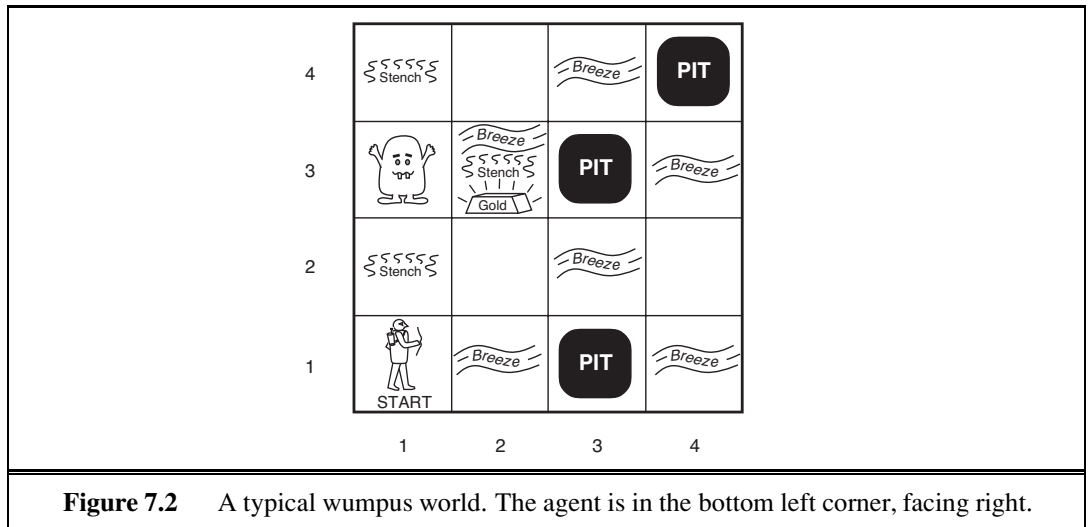
bottomless pits that will trap anyone who wanders into these rooms (except for the wumpus, which is too big to fall in). The only mitigating feature of this bleak environment is the possibility of finding a heap of gold. Although the wumpus world is rather tame by modern computer game standards, it illustrates some important points about intelligence.

A sample wumpus world is shown in Figure 7.2. The precise definition of the task environment is given, as suggested in Section 2.3, by the PEAS description:

- **Performance measure:** +1000 for climbing out of the cave with the gold, −1000 for falling into a pit or being eaten by the wumpus, −1 for each action taken and −10 for using up the arrow. The game ends either when the agent dies or when the agent climbs out of the cave.
- **Environment:** A 4×4 grid of rooms. The agent always starts in the square labeled [1,1], facing to the right. The locations of the gold and the wumpus are chosen randomly, with a uniform distribution, from the squares other than the start square. In addition, each square other than the start can be a pit, with probability 0.2.
- **Actuators:** The agent can move *Forward*, *TurnLeft* by 90° , or *TurnRight* by 90° . The agent dies a miserable death if it enters a square containing a pit or a live wumpus. (It is safe, albeit smelly, to enter a square with a dead wumpus.) If an agent tries to move forward and bumps into a wall, then the agent does not move. The action *Grab* can be used to pick up the gold if it is in the same square as the agent. The action *Shoot* can be used to fire an arrow in a straight line in the direction the agent is facing. The arrow continues until it either hits (and hence kills) the wumpus or hits a wall. The agent has only one arrow, so only the first *Shoot* action has any effect. Finally, the action *Climb* can be used to climb out of the cave, but only from square [1,1].
- **Sensors:** The agent has five sensors, each of which gives a single bit of information:
 - In the square containing the wumpus and in the directly (not diagonally) adjacent squares, the agent will perceive a *Stench*.
 - In the squares directly adjacent to a pit, the agent will perceive a *Breeze*.
 - In the square where the gold is, the agent will perceive a *Glitter*.
 - When an agent walks into a wall, it will perceive a *Bump*.
 - When the wumpus is killed, it emits a woeful *Scream* that can be perceived anywhere in the cave.

The percepts will be given to the agent program in the form of a list of five symbols; for example, if there is a stench and a breeze, but no glitter, bump, or scream, the agent program will get [*Stench*, *Breeze*, *None*, *None*, *None*].

We can characterize the wumpus environment along the various dimensions given in Chapter 2. Clearly, it is discrete, static, and single-agent. (The wumpus doesn't move, fortunately.) It is sequential, because rewards may come only after many actions are taken. It is partially observable, because some aspects of the state are not directly perceivable: the agent's location, the wumpus's state of health, and the availability of an arrow. As for the locations of the pits and the wumpus: we could treat them as unobserved parts of the state that happen to be immutable—in which case, the transition model for the environment is completely



known; or we could say that the transition model itself is unknown because the agent doesn't know which *Forward* actions are fatal—in which case, discovering the locations of pits and wumpus completes the agent's knowledge of the transition model.

For an agent in the environment, the main challenge is its initial ignorance of the configuration of the environment; overcoming this ignorance seems to require logical reasoning. In most instances of the wumpus world, it is possible for the agent to retrieve the gold safely. Occasionally, the agent must choose between going home empty-handed and risking death to find the gold. About 21% of the environments are utterly unfair, because the gold is in a pit or surrounded by pits.

Let us watch a knowledge-based wumpus agent exploring the environment shown in Figure 7.2. We use an informal knowledge representation language consisting of writing down symbols in a grid (as in Figures 7.3 and 7.4).

The agent's initial knowledge base contains the rules of the environment, as described previously; in particular, it knows that it is in [1,1] and that [1,1] is a safe square; we denote that with an "A" and "OK," respectively, in square [1,1].

The first percept is *[None, None, None, None, None]*, from which the agent can conclude that its neighboring squares, [1,2] and [2,1], are free of dangers—they are OK. Figure 7.3(a) shows the agent's state of knowledge at this point.

A cautious agent will move only into a square that it knows to be OK. Let us suppose the agent decides to move forward to [2,1]. The agent perceives a breeze (denoted by "B") in [2,1], so there must be a pit in a neighboring square. The pit cannot be in [1,1], by the rules of the game, so there must be a pit in [2,2] or [3,1] or both. The notation "P?" in Figure 7.3(b) indicates a possible pit in those squares. At this point, there is only one known square that is OK and that has not yet been visited. So the prudent agent will turn around, go back to [1,1], and then proceed to [1,2].

The agent perceives a stench in [1,2], resulting in the state of knowledge shown in Figure 7.4(a). The stench in [1,2] means that there must be a wumpus nearby. But the

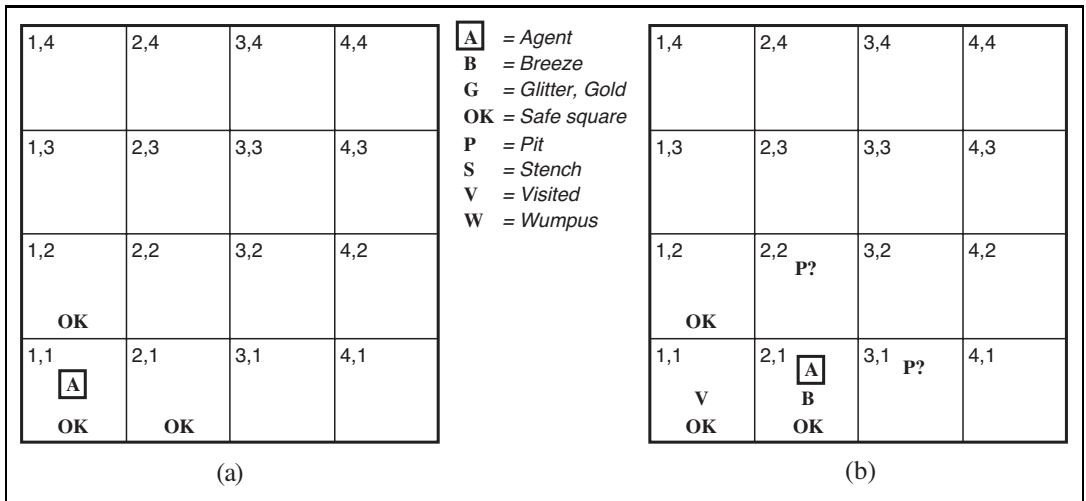


Figure 7.3 The first step taken by the agent in the wumpus world. (a) The initial situation, after percept [*None, None, None, None, None*]. (b) After one move, with percept [*None, Breeze, None, None, None*].

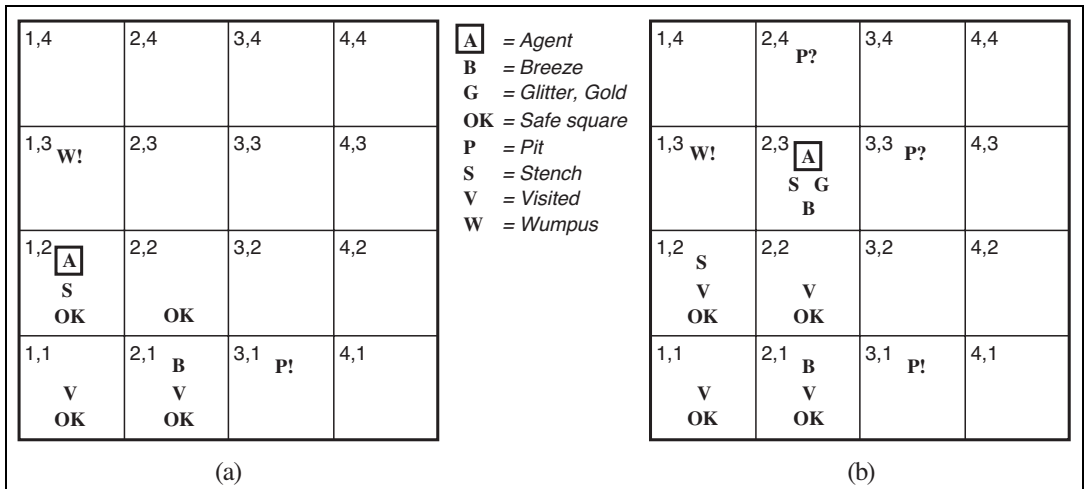


Figure 7.4 Two later stages in the progress of the agent. (a) After the third move, with percept [*Stench, None, None, None, None*]. (b) After the fifth move, with percept [*Stench, Breeze, Glitter, None, None*].

wumpus cannot be in [1,1], by the rules of the game, and it cannot be in [2,2] (or the agent would have detected a stench when it was in [2,1]). Therefore, the agent can infer that the wumpus is in [1,3]. The notation **W!** indicates this inference. Moreover, the lack of a breeze in [1,2] implies that there is no pit in [2,2]. Yet the agent has already inferred that there must be a pit in either [2,2] or [3,1], so this means it must be in [3,1]. This is a fairly difficult inference, because it combines knowledge gained at different times in different places and relies on the lack of a percept to make one crucial step.

The agent has now proved to itself that there is neither a pit nor a wumpus in [2,2], so it is OK to move there. We do not show the agent's state of knowledge at [2,2]; we just assume that the agent turns and moves to [2,3], giving us Figure 7.4(b). In [2,3], the agent detects a glitter, so it should grab the gold and then return home.

Note that in each case for which the agent draws a conclusion from the available information, that conclusion is *guaranteed* to be correct if the available information is correct. This is a fundamental property of logical reasoning. In the rest of this chapter, we describe how to build logical agents that can represent information and draw conclusions such as those described in the preceding paragraphs.

7.3 LOGIC

This section summarizes the fundamental concepts of logical representation and reasoning. These beautiful ideas are independent of any of logic's particular forms. We therefore postpone the technical details of those forms until the next section, using instead the familiar example of ordinary arithmetic.

SYNTAX

In Section 7.1, we said that knowledge bases consist of sentences. These sentences are expressed according to the **syntax** of the representation language, which specifies all the sentences that are well formed. The notion of syntax is clear enough in ordinary arithmetic: “ $x + y = 4$ ” is a well-formed sentence, whereas “ $x4y+ =$ ” is not.

SEMANTICS

TRUTH

POSSIBLE WORLD

A logic must also define the **semantics** or meaning of sentences. The semantics defines the **truth** of each sentence with respect to each **possible world**. For example, the semantics for arithmetic specifies that the sentence “ $x + y = 4$ ” is true in a world where x is 2 and y is 2, but false in a world where x is 1 and y is 1. In standard logics, every sentence must be either true or false in each possible world—there is no “in between.”¹

MODEL

When we need to be precise, we use the term **model** in place of “possible world.” Whereas possible worlds might be thought of as (potentially) real environments that the agent might or might not be in, models are mathematical abstractions, each of which simply fixes the truth or falsehood of every relevant sentence. Informally, we may think of a possible world as, for example, having x men and y women sitting at a table playing bridge, and the sentence $x + y = 4$ is true when there are four people in total. Formally, the possible models are just all possible assignments of real numbers to the variables x and y . Each such assignment fixes the truth of any sentence of arithmetic whose variables are x and y . If a sentence α is true in model m , we say that m **satisfies** α or sometimes m **is a model of** α . We use the notation $M(\alpha)$ to mean the set of all models of α .

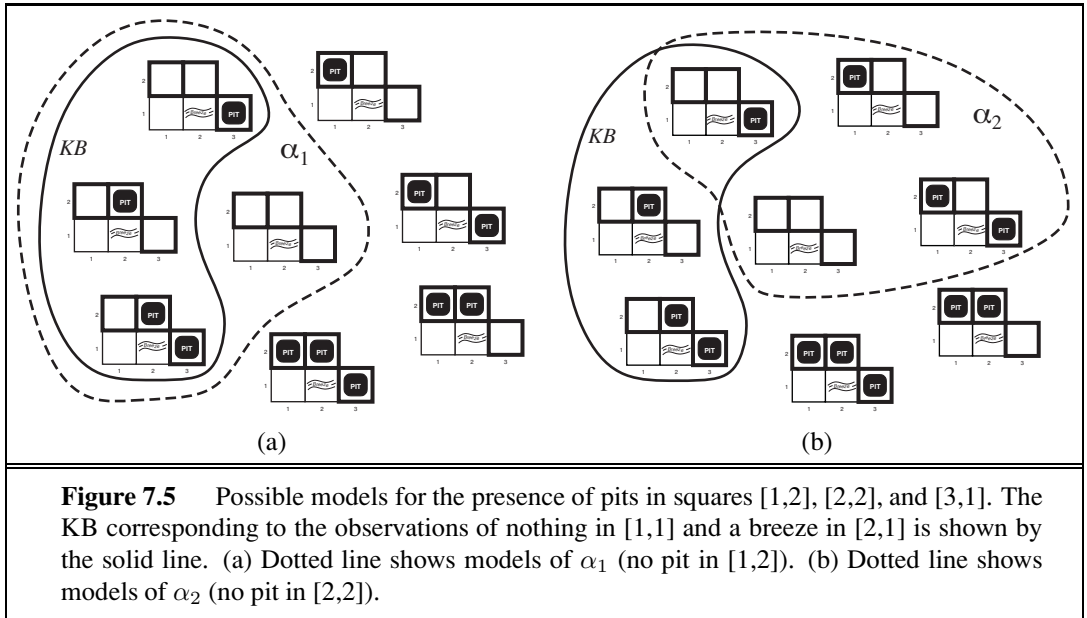
SATISFACTION

ENTAILMENT

Now that we have a notion of truth, we are ready to talk about logical reasoning. This involves the relation of logical **entailment** between sentences—the idea that a sentence *follows logically* from another sentence. In mathematical notation, we write

$$\alpha \models \beta$$

¹ **Fuzzy logic**, discussed in Chapter 14, allows for degrees of truth.



to mean that the sentence α entails the sentence β . The formal definition of entailment is this: $\alpha \models \beta$ if and only if, in every model in which α is true, β is also true. Using the notation just introduced, we can write

$$\alpha \models \beta \text{ if and only if } M(\alpha) \subseteq M(\beta).$$

(Note the direction of the \subseteq here: if $\alpha \models \beta$, then α is a *stronger* assertion than β : it rules out *more* possible worlds.) The relation of entailment is familiar from arithmetic; we are happy with the idea that the sentence $x = 0$ entails the sentence $xy = 0$. Obviously, in any model where x is zero, it is the case that xy is zero (regardless of the value of y).

We can apply the same kind of analysis to the wumpus-world reasoning example given in the preceding section. Consider the situation in Figure 7.3(b): the agent has detected nothing in [1,1] and a breeze in [2,1]. These percepts, combined with the agent's knowledge of the rules of the wumpus world, constitute the KB. The agent is interested (among other things) in whether the adjacent squares [1,2], [2,2], and [3,1] contain pits. Each of the three squares might or might not contain a pit, so (for the purposes of this example) there are $2^3 = 8$ possible models. These eight models are shown in Figure 7.5.²

The KB can be thought of as a set of sentences or as a single sentence that asserts all the individual sentences. The KB is false in models that contradict what the agent knows—for example, the KB is false in any model in which [1,2] contains a pit, because there is no breeze in [1,1]. There are in fact just three models in which the KB is true, and these are

² Although the figure shows the models as partial wumpus worlds, they are really nothing more than assignments of *true* and *false* to the sentences “there is a pit in [1,2]” etc. Models, in the mathematical sense, do not need to have ‘orrible’ airy wumpuses in them.

shown surrounded by a solid line in Figure 7.5. Now let us consider two possible conclusions:

$\alpha_1 = \text{“There is no pit in [1,2].”}$

$\alpha_2 = \text{“There is no pit in [2,2].”}$

We have surrounded the models of α_1 and α_2 with dotted lines in Figures 7.5(a) and 7.5(b), respectively. By inspection, we see the following:

in every model in which KB is true, α_1 is also true.

Hence, $KB \models \alpha_1$: there is no pit in [1,2]. We can also see that

in some models in which KB is true, α_2 is false.

Hence, $KB \not\models \alpha_2$: the agent *cannot* conclude that there is no pit in [2,2]. (Nor can it conclude that there *is* a pit in [2,2].)³

LOGICAL INFERENCE

MODEL CHECKING

The preceding example not only illustrates entailment but also shows how the definition of entailment can be applied to derive conclusions—that is, to carry out **logical inference**. The inference algorithm illustrated in Figure 7.5 is called **model checking**, because it enumerates all possible models to check that α is true in all models in which KB is true, that is, that $M(KB) \subseteq M(\alpha)$.

In understanding entailment and inference, it might help to think of the set of all consequences of KB as a haystack and of α as a needle. Entailment is like the needle being in the haystack; inference is like finding it. This distinction is embodied in some formal notation: if an inference algorithm i can derive α from KB , we write

$$KB \vdash_i \alpha,$$

which is pronounced “ α is derived from KB by i ” or “ i derives α from KB .”

SOUND

TRUTH-PRESERVING

An inference algorithm that derives only entailed sentences is called **sound** or **truth-preserving**. Soundness is a highly desirable property. An unsound inference procedure essentially makes things up as it goes along—it announces the discovery of nonexistent needles. It is easy to see that model checking, when it is applicable,⁴ is a sound procedure.

COMPLETENESS

The property of **completeness** is also desirable: an inference algorithm is complete if it can derive any sentence that is entailed. For real haystacks, which are finite in extent, it seems obvious that a systematic examination can always decide whether the needle is in the haystack. For many knowledge bases, however, the haystack of consequences is infinite, and completeness becomes an important issue.⁵ Fortunately, there are complete inference procedures for logics that are sufficiently expressive to handle many knowledge bases.

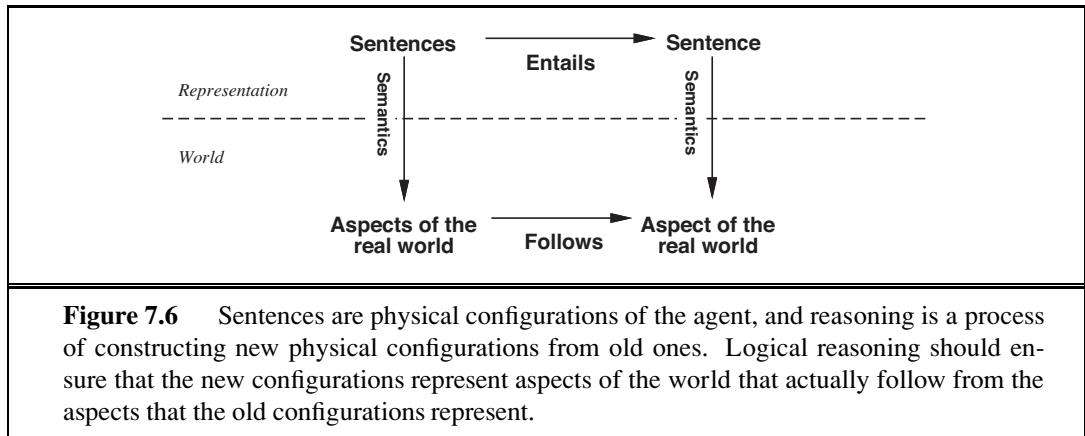


We have described a reasoning process whose conclusions are guaranteed to be true in any world in which the premises are true; in particular, *if KB is true in the real world, then any sentence α derived from KB by a sound inference procedure is also true in the real world*. So, while an inference process operates on “syntax”—internal physical configurations such as bits in registers or patterns of electrical blips in brains—the process *corresponds*

³ The agent can calculate the *probability* that there is a pit in [2,2]; Chapter 13 shows how.

⁴ Model checking works if the space of models is finite—for example, in wumpus worlds of fixed size. For arithmetic, on the other hand, the space of models is infinite: even if we restrict ourselves to the integers, there are infinitely many pairs of values for x and y in the sentence $x + y = 4$.

⁵ Compare with the case of infinite search spaces in Chapter 3, where depth-first search is not complete.



to the real-world relationship whereby some aspect of the real world is the case⁶ by virtue of other aspects of the real world being the case. This correspondence between world and representation is illustrated in Figure 7.6.

GROUNDING



The final issue to consider is **grounding**—the connection between logical reasoning processes and the real environment in which the agent exists. In particular, *how do we know that KB is true in the real world?* (After all, *KB* is just “syntax” inside the agent’s head.) This is a philosophical question about which many, many books have been written. (See Chapter 26.) A simple answer is that the agent’s sensors create the connection. For example, our wumpus-world agent has a smell sensor. The agent program creates a suitable sentence whenever there is a smell. Then, whenever that sentence is in the knowledge base, it is true in the real world. Thus, the meaning and truth of percept sentences are defined by the processes of sensing and sentence construction that produce them. What about the rest of the agent’s knowledge, such as its belief that wumpuses cause smells in adjacent squares? This is not a direct representation of a single percept, but a general rule—derived, perhaps, from perceptual experience but not identical to a statement of that experience. General rules like this are produced by a sentence construction process called **learning**, which is the subject of Part V. Learning is fallible. It could be the case that wumpuses cause smells *except on February 29 in leap years*, which is when they take their baths. Thus, *KB* may not be true in the real world, but with good learning procedures, there is reason for optimism.

7.4 PROPOSITIONAL LOGIC: A VERY SIMPLE LOGIC

PROPOSITIONAL LOGIC

We now present a simple but powerful logic called **propositional logic**. We cover the syntax of propositional logic and its semantics—the way in which the truth of sentences is determined. Then we look at **entailment**—the relation between a sentence and another sentence that follows from it—and see how this leads to a simple algorithm for logical inference. Everything takes place, of course, in the wumpus world.

⁶ As Wittgenstein (1922) put it in his famous *Tractatus*: “The world is everything that is the case.”

Figure 7.7 gives a formal grammar of propositional logic; see page 1060 if you are not familiar with the BNF notation. The BNF grammar by itself is ambiguous; a sentence with several operators can be parsed by the grammar in multiple ways. To eliminate the ambiguity we define a precedence for each operator. The “not” operator (\neg) has the highest precedence, which means that in the sentence $\neg A \wedge B$ the \neg binds most tightly, giving us the equivalent of $(\neg A) \wedge B$ rather than $\neg(A \wedge B)$. (The notation for ordinary arithmetic is the same: $-2 + 4$ is 2, not -6 .) When in doubt, use parentheses to make sure of the right interpretation. Square brackets mean the same thing as parentheses; the choice of square brackets or parentheses is solely to make it easier for a human to read a sentence.

7.4.2 Semantics

TRUTH VALUE

Having specified the syntax of propositional logic, we now specify its semantics. The semantics defines the rules for determining the truth of a sentence with respect to a particular model. In propositional logic, a model simply fixes the **truth value**—*true* or *false*—for every proposition symbol. For example, if the sentences in the knowledge base make use of the proposition symbols $P_{1,2}$, $P_{2,2}$, and $P_{3,1}$, then one possible model is

$$m_1 = \{P_{1,2} = \text{false}, P_{2,2} = \text{false}, P_{3,1} = \text{true}\}.$$

With three proposition symbols, there are $2^3 = 8$ possible models—exactly those depicted in Figure 7.5. Notice, however, that the models are purely mathematical objects with no necessary connection to wumpus worlds. $P_{1,2}$ is just a symbol; it might mean “there is a pit in [1,2]” or “I’m in Paris today and tomorrow.”

The semantics for propositional logic must specify how to compute the truth value of *any* sentence, given a model. This is done recursively. All sentences are constructed from atomic sentences and the five connectives; therefore, we need to specify how to compute the truth of atomic sentences and how to compute the truth of sentences formed with each of the five connectives. Atomic sentences are easy:

- *True* is true in every model and *False* is false in every model.
- The truth value of every other proposition symbol must be specified directly in the model. For example, in the model m_1 given earlier, $P_{1,2}$ is false.

For complex sentences, we have five rules, which hold for any subsentences P and Q in any model m (here “iff” means “if and only if”):

- $\neg P$ is true iff P is false in m .
- $P \wedge Q$ is true iff both P and Q are true in m .
- $P \vee Q$ is true iff either P or Q is true in m .
- $P \Rightarrow Q$ is true unless P is true and Q is false in m .
- $P \Leftrightarrow Q$ is true iff P and Q are both true or both false in m .

TRUTH TABLE

The rules can also be expressed with **truth tables** that specify the truth value of a complex sentence for each possible assignment of truth values to its components. Truth tables for the five connectives are given in Figure 7.8. From these tables, the truth value of any sentence s can be computed with respect to any model m by a simple recursive evaluation. For example,

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	false	true	false	false	true	true
false	true	true	false	true	true	false
true	false	false	false	true	false	false
true	true	false	true	true	true	true

Figure 7.8 Truth tables for the five logical connectives. To use the table to compute, for example, the value of $P \vee Q$ when P is true and Q is false, first look on the left for the row where P is true and Q is false (the third row). Then look in that row under the $P \vee Q$ column to see the result: *true*.

the sentence $\neg P_{1,2} \wedge (P_{2,2} \vee P_{3,1})$, evaluated in m_1 , gives $\text{true} \wedge (\text{false} \vee \text{true}) = \text{true} \wedge \text{true} = \text{true}$. Exercise 7.3 asks you to write the algorithm PL-TRUE?(s, m), which computes the truth value of a propositional logic sentence s in a model m .

The truth tables for “and,” “or,” and “not” are in close accord with our intuitions about the English words. The main point of possible confusion is that $P \vee Q$ is true when P is true or Q is true *or both*. A different connective, called “exclusive or” (“xor” for short), yields false when both disjuncts are true.⁷ There is no consensus on the symbol for exclusive or; some choices are $\dot{\vee}$ or \neq or \oplus .

The truth table for \Rightarrow may not quite fit one’s intuitive understanding of “ P implies Q ” or “if P then Q .” For one thing, propositional logic does not require any relation of *causation* or *relevance* between P and Q . The sentence “5 is odd implies Tokyo is the capital of Japan” is a true sentence of propositional logic (under the normal interpretation), even though it is a decidedly odd sentence of English. Another point of confusion is that any implication is true whenever its antecedent is false. For example, “5 is even implies Sam is smart” is true, regardless of whether Sam is smart. This seems bizarre, but it makes sense if you think of “ $P \Rightarrow Q$ ” as saying, “If P is true, then I am claiming that Q is true. Otherwise I am making no claim.” The only way for this sentence to be *false* is if P is true but Q is false.

The biconditional, $P \Leftrightarrow Q$, is true whenever both $P \Rightarrow Q$ and $Q \Rightarrow P$ are true. In English, this is often written as “ P if and only if Q .” Many of the rules of the wumpus world are best written using \Leftrightarrow . For example, a square is breezy *if* a neighboring square has a pit, and a square is breezy *only if* a neighboring square has a pit. So we need a biconditional,

$$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1}),$$

where $B_{1,1}$ means that there is a breeze in [1,1].

7.4.3 A simple knowledge base

Now that we have defined the semantics for propositional logic, we can construct a knowledge base for the wumpus world. We focus first on the *immutable* aspects of the wumpus world, leaving the mutable aspects for a later section. For now, we need the following symbols for each $[x, y]$ location:

⁷ Latin has a separate word, *aut*, for exclusive or.

$P_{x,y}$ is true if there is a pit in $[x, y]$.

$W_{x,y}$ is true if there is a wumpus in $[x, y]$, dead or alive.

$B_{x,y}$ is true if the agent perceives a breeze in $[x, y]$.

$S_{x,y}$ is true if the agent perceives a stench in $[x, y]$.

The sentences we write will suffice to derive $\neg P_{1,2}$ (there is no pit in $[1,2]$), as was done informally in Section 7.3. We label each sentence R_i so that we can refer to them:

- There is no pit in $[1,1]$:

$$R_1 : \neg P_{1,1} .$$

- A square is breezy if and only if there is a pit in a neighboring square. This has to be stated for each square; for now, we include just the relevant squares:

$$R_2 : B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1}) .$$

$$R_3 : B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1}) .$$

- The preceding sentences are true in all wumpus worlds. Now we include the breeze percepts for the first two squares visited in the specific world the agent is in, leading up to the situation in Figure 7.3(b).

$$R_4 : \neg B_{1,1} .$$

$$R_5 : B_{2,1} .$$

7.4.4 A simple inference procedure

Our goal now is to decide whether $KB \models \alpha$ for some sentence α . For example, is $\neg P_{1,2}$ entailed by our KB ? Our first algorithm for inference is a model-checking approach that is a direct implementation of the definition of entailment: enumerate the models, and check that α is true in every model in which KB is true. Models are assignments of *true* or *false* to every proposition symbol. Returning to our wumpus-world example, the relevant proposition symbols are $B_{1,1}$, $B_{2,1}$, $P_{1,1}$, $P_{1,2}$, $P_{2,1}$, $P_{2,2}$, and $P_{3,1}$. With seven symbols, there are $2^7 = 128$ possible models; in three of these, KB is true (Figure 7.9). In those three models, $\neg P_{1,2}$ is true, hence there is no pit in $[1,2]$. On the other hand, $P_{2,2}$ is true in two of the three models and false in one, so we cannot yet tell whether there is a pit in $[2,2]$.

Figure 7.9 reproduces in a more precise form the reasoning illustrated in Figure 7.5. A general algorithm for deciding entailment in propositional logic is shown in Figure 7.10. Like the BACKTRACKING-SEARCH algorithm on page 215, TT-ENTAILS? performs a recursive enumeration of a finite space of assignments to symbols. The algorithm is **sound** because it implements directly the definition of entailment, and **complete** because it works for any KB and α and always terminates—there are only finitely many models to examine.

Of course, “finitely many” is not always the same as “few.” If KB and α contain n symbols in all, then there are 2^n models. Thus, the time complexity of the algorithm is $O(2^n)$. (The space complexity is only $O(n)$ because the enumeration is depth-first.) Later in this chapter we show algorithms that are much more efficient in many cases. Unfortunately, propositional entailment is co-NP-complete (i.e., probably no easier than NP-complete—see Appendix A), so *every known inference algorithm for propositional logic has a worst-case complexity that is exponential in the size of the input.*



$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	R_1	R_2	R_3	R_4	R_5	KB
false	false	false	false	false	false	false	true	true	true	true	false	false
false	false	false	false	false	false	true	true	true	false	true	false	false
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
false	true	false	false	false	false	false	true	true	false	true	true	false
false	true	false	false	false	false	true	true	true	true	true	true	<u>true</u>
false	true	false	false	false	true	false	true	true	true	true	true	<u>true</u>
false	true	false	false	false	true	true	true	true	true	true	true	<u>true</u>
false	true	false	false	true	false	false	true	false	false	true	true	false
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
true	true	true	true	true	true	true	false	true	true	false	true	false

Figure 7.9 A truth table constructed for the knowledge base given in the text. KB is true if R_1 through R_5 are true, which occurs in just 3 of the 128 rows (the ones underlined in the right-hand column). In all 3 rows, $P_{1,2}$ is false, so there is no pit in [1,2]. On the other hand, there might (or might not) be a pit in [2,2].

```

function TT-ENTAILS?( $KB, \alpha$ ) returns true or false
  inputs:  $KB$ , the knowledge base, a sentence in propositional logic
            $\alpha$ , the query, a sentence in propositional logic

   $symbols \leftarrow$  a list of the proposition symbols in  $KB$  and  $\alpha$ 
  return TT-CHECK-ALL( $KB, \alpha, symbols, \{ \}$ )



---


function TT-CHECK-ALL( $KB, \alpha, symbols, model$ ) returns true or false
  if EMPTY?( $symbols$ ) then
    if PL-TRUE?( $KB, model$ ) then return PL-TRUE?( $\alpha, model$ )
    else return true // when  $KB$  is false, always return true
  else do
     $P \leftarrow$  FIRST( $symbols$ )
     $rest \leftarrow$  REST( $symbols$ )
    return (TT-CHECK-ALL( $KB, \alpha, rest, model \cup \{P = true\}$ )
      and
      TT-CHECK-ALL( $KB, \alpha, rest, model \cup \{P = false\}$ ))

```

Figure 7.10 A truth-table enumeration algorithm for deciding propositional entailment. (TT stands for truth table.) PL-TRUE? returns *true* if a sentence holds within a model. The variable *model* represents a partial model—an assignment to some of the symbols. The keyword “**and**” is used here as a logical operation on its two arguments, returning *true* or *false*.

$(\alpha \wedge \beta) \equiv (\beta \wedge \alpha)$	commutativity of \wedge
$(\alpha \vee \beta) \equiv (\beta \vee \alpha)$	commutativity of \vee
$((\alpha \wedge \beta) \wedge \gamma) \equiv (\alpha \wedge (\beta \wedge \gamma))$	associativity of \wedge
$((\alpha \vee \beta) \vee \gamma) \equiv (\alpha \vee (\beta \vee \gamma))$	associativity of \vee
$\neg(\neg\alpha) \equiv \alpha$	double-negation elimination
$(\alpha \Rightarrow \beta) \equiv (\neg\beta \Rightarrow \neg\alpha)$	contraposition
$(\alpha \Rightarrow \beta) \equiv (\neg\alpha \vee \beta)$	implication elimination
$(\alpha \Leftrightarrow \beta) \equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha))$	biconditional elimination
$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta)$	De Morgan
$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta)$	De Morgan
$(\alpha \wedge (\beta \vee \gamma)) \equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma))$	distributivity of \wedge over \vee
$(\alpha \vee (\beta \wedge \gamma)) \equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma))$	distributivity of \vee over \wedge

Figure 7.11 Standard logical equivalences. The symbols α , β , and γ stand for arbitrary sentences of propositional logic.

7.5 PROPOSITIONAL THEOREM PROVING

THEOREM PROVING

So far, we have shown how to determine entailment by *model checking*: enumerating models and showing that the sentence must hold in all models. In this section, we show how entailment can be done by **theorem proving**—applying rules of inference directly to the sentences in our knowledge base to construct a proof of the desired sentence without consulting models. If the number of models is large but the length of the proof is short, then theorem proving can be more efficient than model checking.

LOGICAL
EQUIVALENCE

Before we plunge into the details of theorem-proving algorithms, we will need some additional concepts related to entailment. The first concept is **logical equivalence**: two sentences α and β are logically equivalent if they are true in the same set of models. We write this as $\alpha \equiv \beta$. For example, we can easily show (using truth tables) that $P \wedge Q$ and $Q \wedge P$ are logically equivalent; other equivalences are shown in Figure 7.11. These equivalences play much the same role in logic as arithmetic identities do in ordinary mathematics. An alternative definition of equivalence is as follows: any two sentences α and β are equivalent only if each of them entails the other:

$$\alpha \equiv \beta \quad \text{if and only if} \quad \alpha \models \beta \text{ and } \beta \models \alpha.$$

VALIDITY

TAUTOLOGY

The second concept we will need is **validity**. A sentence is valid if it is true in *all* models. For example, the sentence $P \vee \neg P$ is valid. Valid sentences are also known as **tautologies**—they are *necessarily* true. Because the sentence *True* is true in all models, every valid sentence is logically equivalent to *True*. What good are valid sentences? From our definition of entailment, we can derive the **deduction theorem**, which was known to the ancient Greeks:

DEDUCTION
THEOREM



For any sentences α and β , $\alpha \models \beta$ if and only if the sentence $(\alpha \Rightarrow \beta)$ is valid.

(Exercise 7.5 asks for a proof.) Hence, we can decide if $\alpha \models \beta$ by checking that $(\alpha \Rightarrow \beta)$ is true in every model—which is essentially what the inference algorithm in Figure 7.10 does—

or by proving that $(\alpha \Rightarrow \beta)$ is equivalent to *True*. Conversely, the deduction theorem states that every valid implication sentence describes a legitimate inference.

SATISFIABILITY

The final concept we will need is **satisfiability**. A sentence is satisfiable if it is true in, or satisfied by, *some* model. For example, the knowledge base given earlier, $(R_1 \wedge R_2 \wedge R_3 \wedge R_4 \wedge R_5)$, is satisfiable because there are three models in which it is true, as shown in Figure 7.9. Satisfiability can be checked by enumerating the possible models until one is found that satisfies the sentence. The problem of determining the satisfiability of sentences in propositional logic—the **SAT** problem—was the first problem proved to be NP-complete. Many problems in computer science are really satisfiability problems. For example, all the constraint satisfaction problems in Chapter 6 ask whether the constraints are satisfiable by some assignment.

SAT

Validity and satisfiability are of course connected: α is valid iff $\neg\alpha$ is unsatisfiable; contrapositively, α is satisfiable iff $\neg\alpha$ is not valid. We also have the following useful result:



$\alpha \models \beta$ if and only if the sentence $(\alpha \wedge \neg\beta)$ is unsatisfiable.

REDUCTIO AD
ABSURDUM

REFUTATION

CONTRADICTION

Proving β from α by checking the unsatisfiability of $(\alpha \wedge \neg\beta)$ corresponds exactly to the standard mathematical proof technique of *reductio ad absurdum* (literally, “reduction to an absurd thing”). It is also called proof by **refutation** or proof by **contradiction**. One assumes a sentence β to be false and shows that this leads to a contradiction with known axioms α . This contradiction is exactly what is meant by saying that the sentence $(\alpha \wedge \neg\beta)$ is unsatisfiable.

7.5.1 Inference and proofs

INFERENCE RULES

PROOF

MODUS PONENS

This section covers **inference rules** that can be applied to derive a **proof**—a chain of conclusions that leads to the desired goal. The best-known rule is called **Modus Ponens** (Latin for *mode that affirms*) and is written

$$\frac{\alpha \Rightarrow \beta, \quad \alpha}{\beta}.$$

The notation means that, whenever any sentences of the form $\alpha \Rightarrow \beta$ and α are given, then the sentence β can be inferred. For example, if $(WumpusAhead \wedge WumpusAlive) \Rightarrow Shoot$ and $(WumpusAhead \wedge WumpusAlive)$ are given, then *Shoot* can be inferred.

AND-ELIMINATION

Another useful inference rule is **And-Elimination**, which says that, from a conjunction, any of the conjuncts can be inferred:

$$\frac{\alpha \wedge \beta}{\alpha}.$$

For example, from $(WumpusAhead \wedge WumpusAlive)$, *WumpusAlive* can be inferred.

By considering the possible truth values of α and β , one can show easily that Modus Ponens and And-Elimination are sound once and for all. These rules can then be used in any particular instances where they apply, generating sound inferences without the need for enumerating models.

All of the logical equivalences in Figure 7.11 can be used as inference rules. For example, the equivalence for biconditional elimination yields the two inference rules

$$\frac{\alpha \Leftrightarrow \beta}{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)} \quad \text{and} \quad \frac{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)}{\alpha \Leftrightarrow \beta}.$$

Not all inference rules work in both directions like this. For example, we cannot run Modus Ponens in the opposite direction to obtain $\alpha \Rightarrow \beta$ and α from β .

Let us see how these inference rules and equivalences can be used in the wumpus world. We start with the knowledge base containing R_1 through R_5 and show how to prove $\neg P_{1,2}$, that is, there is no pit in [1,2]. First, we apply biconditional elimination to R_2 to obtain

$$R_6 : (B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}) .$$

Then we apply And-Elimination to R_6 to obtain

$$R_7 : ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}) .$$

Logical equivalence for contrapositives gives

$$R_8 : (\neg B_{1,1} \Rightarrow \neg(P_{1,2} \vee P_{2,1})) .$$

Now we can apply Modus Ponens with R_8 and the percept R_4 (i.e., $\neg B_{1,1}$), to obtain

$$R_9 : \neg(P_{1,2} \vee P_{2,1}) .$$

Finally, we apply De Morgan's rule, giving the conclusion

$$R_{10} : \neg P_{1,2} \wedge \neg P_{2,1} .$$

That is, neither [1,2] nor [2,1] contains a pit.

We found this proof by hand, but we can apply any of the search algorithms in Chapter 3 to find a sequence of steps that constitutes a proof. We just need to define a proof problem as follows:

- INITIAL STATE: the initial knowledge base.
- ACTIONS: the set of actions consists of all the inference rules applied to all the sentences that match the top half of the inference rule.
- RESULT: the result of an action is to add the sentence in the bottom half of the inference rule.
- GOAL: the goal is a state that contains the sentence we are trying to prove.



Thus, searching for proofs is an alternative to enumerating models. In many practical cases *finding a proof can be more efficient because the proof can ignore irrelevant propositions, no matter how many of them there are*. For example, the proof given earlier leading to $\neg P_{1,2} \wedge \neg P_{2,1}$ does not mention the propositions $B_{2,1}$, $P_{1,1}$, $P_{2,2}$, or $P_{3,1}$. They can be ignored because the goal proposition, $P_{1,2}$, appears only in sentence R_2 ; the other propositions in R_2 appear only in R_4 and R_2 ; so R_1 , R_3 , and R_5 have no bearing on the proof. The same would hold even if we added a million more sentences to the knowledge base; the simple truth-table algorithm, on the other hand, would be overwhelmed by the exponential explosion of models.

One final property of logical systems is **monotonicity**, which says that the set of entailed sentences can only *increase* as information is added to the knowledge base.⁸ For any sentences α and β ,

$$\text{if } KB \models \alpha \text{ then } KB \wedge \beta \models \alpha .$$

⁸ **Nonmonotonic** logics, which violate the monotonicity property, capture a common property of human reasoning: changing one's mind. They are discussed in Section 12.6.

For example, suppose the knowledge base contains the additional assertion β stating that there are exactly eight pits in the world. This knowledge might help the agent draw *additional* conclusions, but it cannot invalidate any conclusion α already inferred—such as the conclusion that there is no pit in [1,2]. Monotonicity means that inference rules can be applied whenever suitable premises are found in the knowledge base—the conclusion of the rule must follow *regardless of what else is in the knowledge base*.

7.5.2 Proof by resolution

We have argued that the inference rules covered so far are *sound*, but we have not discussed the question of *completeness* for the inference algorithms that use them. Search algorithms such as iterative deepening search (page 89) are complete in the sense that they will find any reachable goal, but if the available inference rules are inadequate, then the goal is not reachable—no proof exists that uses only those inference rules. For example, if we removed the biconditional elimination rule, the proof in the preceding section would not go through. The current section introduces a single inference rule, **resolution**, that yields a complete inference algorithm when coupled with any complete search algorithm.

We begin by using a simple version of the resolution rule in the wumpus world. Let us consider the steps leading up to Figure 7.4(a): the agent returns from [2,1] to [1,1] and then goes to [1,2], where it perceives a stench, but no breeze. We add the following facts to the knowledge base:

$$\begin{aligned} R_{11} : & \quad \neg B_{1,2} . \\ R_{12} : & \quad B_{1,2} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{1,3}) . \end{aligned}$$

By the same process that led to R_{10} earlier, we can now derive the absence of pits in [2,2] and [1,3] (remember that [1,1] is already known to be pitless):

$$\begin{aligned} R_{13} : & \quad \neg P_{2,2} . \\ R_{14} : & \quad \neg P_{1,3} . \end{aligned}$$

We can also apply biconditional elimination to R_3 , followed by Modus Ponens with R_5 , to obtain the fact that there is a pit in [1,1], [2,2], or [3,1]:

$$R_{15} : \quad P_{1,1} \vee P_{2,2} \vee P_{3,1} .$$

Now comes the first application of the resolution rule: the literal $\neg P_{2,2}$ in R_{13} *resolves with* the literal $P_{2,2}$ in R_{15} to give the **resolvent**

$$R_{16} : \quad P_{1,1} \vee P_{3,1} .$$

In English; if there's a pit in one of [1,1], [2,2], and [3,1] and it's not in [2,2], then it's in [1,1] or [3,1]. Similarly, the literal $\neg P_{1,1}$ in R_1 resolves with the literal $P_{1,1}$ in R_{16} to give

$$R_{17} : \quad P_{3,1} .$$

In English: if there's a pit in [1,1] or [3,1] and it's not in [1,1], then it's in [3,1]. These last two inference steps are examples of the **unit resolution** inference rule,

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \quad m}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k} ,$$

where each ℓ is a literal and ℓ_i and m are **complementary literals** (i.e., one is the negation

RESOLVENT

UNIT RESOLUTION

COMPLEMENTARY LITERALS

CLAUSE

of the other). Thus, the unit resolution rule takes a **clause**—a disjunction of literals—and a literal and produces a new clause. Note that a single literal can be viewed as a disjunction of one literal, also known as a **unit clause**.

UNIT CLAUSE

RESOLUTION

The unit resolution rule can be generalized to the full **resolution** rule,

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \quad m_1 \vee \cdots \vee m_n}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n},$$

where ℓ_i and m_j are complementary literals. This says that resolution takes two clauses and produces a new clause containing all the literals of the two original clauses *except* the two complementary literals. For example, we have

$$\frac{P_{1,1} \vee P_{3,1}, \quad \neg P_{1,1} \vee \neg P_{2,2}}{P_{3,1} \vee \neg P_{2,2}}.$$

FACTORING

There is one more technical aspect of the resolution rule: the resulting clause should contain only one copy of each literal.⁹ The removal of multiple copies of literals is called **factoring**. For example, if we resolve $(A \vee B)$ with $(A \vee \neg B)$, we obtain $(A \vee A)$, which is reduced to just A .

The *soundness* of the resolution rule can be seen easily by considering the literal ℓ_i that is complementary to literal m_j in the other clause. If ℓ_i is true, then m_j is false, and hence $m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n$ must be true, because $m_1 \vee \cdots \vee m_n$ is given. If ℓ_i is false, then $\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k$ must be true because $\ell_1 \vee \cdots \vee \ell_k$ is given. Now ℓ_i is either true or false, so one or other of these conclusions holds—exactly as the resolution rule states.



What is more surprising about the resolution rule is that it forms the basis for a family of *complete* inference procedures. A *resolution-based theorem prover* can, for any sentences α and β in propositional logic, decide whether $\alpha \models \beta$. The next two subsections explain how resolution accomplishes this.

Conjunctive normal form

The resolution rule applies only to clauses (that is, disjunctions of literals), so it would seem to be relevant only to knowledge bases and queries consisting of clauses. How, then, can it lead to a complete inference procedure for all of propositional logic? The answer is that *every sentence of propositional logic is logically equivalent to a conjunction of clauses*. A sentence expressed as a conjunction of clauses is said to be in **conjunctive normal form** or **CNF** (see Figure 7.14). We now describe a procedure for converting to CNF. We illustrate the procedure by converting the sentence $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$ into CNF. The steps are as follows:



1. Eliminate \Leftrightarrow , replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.

$$(B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}).$$
2. Eliminate \Rightarrow , replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$:

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1}).$$

⁹ If a clause is viewed as a *set* of literals, then this restriction is automatically respected. Using set notation for clauses makes the resolution rule much cleaner, at the cost of introducing additional notation.

3. CNF requires \neg to appear only in literals, so we “move \neg inwards” by repeated application of the following equivalences from Figure 7.11:

$$\begin{aligned}\neg(\neg\alpha) &\equiv \alpha \quad (\text{double-negation elimination}) \\ \neg(\alpha \wedge \beta) &\equiv (\neg\alpha \vee \neg\beta) \quad (\text{De Morgan}) \\ \neg(\alpha \vee \beta) &\equiv (\neg\alpha \wedge \neg\beta) \quad (\text{De Morgan})\end{aligned}$$

In the example, we require just one application of the last rule:

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1}) .$$

4. Now we have a sentence containing nested \wedge and \vee operators applied to literals. We apply the distributivity law from Figure 7.11, distributing \vee over \wedge wherever possible.

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1}) .$$

The original sentence is now in CNF, as a conjunction of three clauses. It is much harder to read, but it can be used as input to a resolution procedure.

A resolution algorithm

Inference procedures based on resolution work by using the principle of proof by contradiction introduced on page 250. That is, to show that $KB \models \alpha$, we show that $(KB \wedge \neg\alpha)$ is unsatisfiable. We do this by proving a contradiction.

A resolution algorithm is shown in Figure 7.12. First, $(KB \wedge \neg\alpha)$ is converted into CNF. Then, the resolution rule is applied to the resulting clauses. Each pair that contains complementary literals is resolved to produce a new clause, which is added to the set if it is not already present. The process continues until one of two things happens:

- there are no new clauses that can be added, in which case KB does not entail α ; or,
- two clauses resolve to yield the *empty* clause, in which case KB entails α .

The empty clause—a disjunction of no disjuncts—is equivalent to *False* because a disjunction is true only if at least one of its disjuncts is true. Another way to see that an empty clause represents a contradiction is to observe that it arises only from resolving two complementary unit clauses such as P and $\neg P$.

We can apply the resolution procedure to a very simple inference in the wumpus world. When the agent is in [1,1], there is no breeze, so there can be no pits in neighboring squares. The relevant knowledge base is

$$KB = R_2 \wedge R_4 = (B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})) \wedge \neg B_{1,1}$$

and we wish to prove α which is, say, $\neg P_{1,2}$. When we convert $(KB \wedge \neg\alpha)$ into CNF, we obtain the clauses shown at the top of Figure 7.13. The second row of the figure shows clauses obtained by resolving pairs in the first row. Then, when $P_{1,2}$ is resolved with $\neg P_{1,2}$, we obtain the empty clause, shown as a small square. Inspection of Figure 7.13 reveals that many resolution steps are pointless. For example, the clause $B_{1,1} \vee \neg B_{1,1} \vee P_{1,2}$ is equivalent to $True \vee P_{1,2}$ which is equivalent to *True*. Deducing that *True* is true is not very helpful. Therefore, any clause in which two complementary literals appear can be discarded.

```

function PL-RESOLUTION( $KB, \alpha$ ) returns true or false
  inputs:  $KB$ , the knowledge base, a sentence in propositional logic
            $\alpha$ , the query, a sentence in propositional logic

   $clauses \leftarrow$  the set of clauses in the CNF representation of  $KB \wedge \neg\alpha$ 
   $new \leftarrow \{\}$ 
  loop do
    for each pair of clauses  $C_i, C_j$  in  $clauses$  do
       $resolvents \leftarrow$  PL-RESOLVE( $C_i, C_j$ )
      if  $resolvents$  contains the empty clause then return true
       $new \leftarrow new \cup resolvents$ 
    if  $new \subseteq clauses$  then return false
     $clauses \leftarrow clauses \cup new$ 

```

Figure 7.12 A simple resolution algorithm for propositional logic. The function PL-RESOLVE returns the set of all possible clauses obtained by resolving its two inputs.

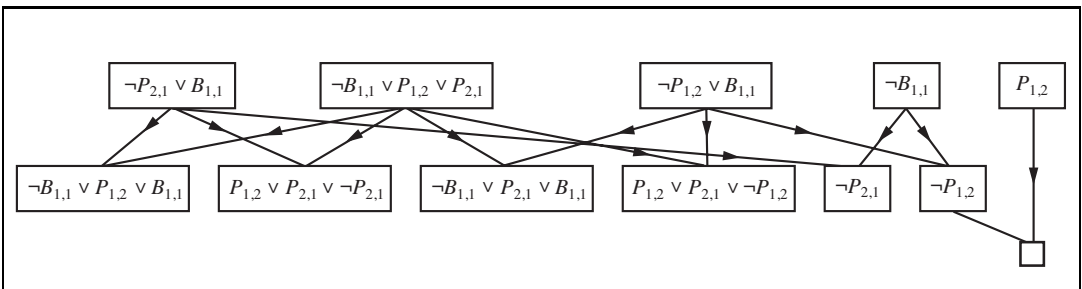


Figure 7.13 Partial application of PL-RESOLUTION to a simple inference in the wumpus world. $\neg P_{1,2}$ is shown to follow from the first four clauses in the top row.

Completeness of resolution

To conclude our discussion of resolution, we now show why PL-RESOLUTION is complete. To do this, we introduce the **resolution closure** $RC(S)$ of a set of clauses S , which is the set of all clauses derivable by repeated application of the resolution rule to clauses in S or their derivatives. The resolution closure is what PL-RESOLUTION computes as the final value of the variable $clauses$. It is easy to see that $RC(S)$ must be finite, because there are only finitely many distinct clauses that can be constructed out of the symbols P_1, \dots, P_k that appear in S . (Notice that this would not be true without the factoring step that removes multiple copies of literals.) Hence, PL-RESOLUTION always terminates.

The completeness theorem for resolution in propositional logic is called the **ground resolution theorem**:

If a set of clauses is unsatisfiable, then the resolution closure of those clauses contains the empty clause.

This theorem is proved by demonstrating its contrapositive: if the closure $RC(S)$ does *not*

RESOLUTION
CLOSURE

GROUND
RESOLUTION
THEOREM

contain the empty clause, then S is satisfiable. In fact, we can construct a model for S with suitable truth values for P_1, \dots, P_k . The construction procedure is as follows:

For i from 1 to k ,

- If a clause in $RC(S)$ contains the literal $\neg P_i$ and all its other literals are false under the assignment chosen for P_1, \dots, P_{i-1} , then assign *false* to P_i .
- Otherwise, assign *true* to P_i .

This assignment to P_1, \dots, P_k is a model of S . To see this, assume the opposite—that, at some stage i in the sequence, assigning symbol P_i causes some clause C to become false. For this to happen, it must be the case that all the *other* literals in C must already have been falsified by assignments to P_1, \dots, P_{i-1} . Thus, C must now look like either $(false \vee false \vee \dots false \vee P_i)$ or like $(false \vee false \vee \dots false \vee \neg P_i)$. If just one of these two is in $RC(S)$, then the algorithm will assign the appropriate truth value to P_i to make C true, so C can only be falsified if *both* of these clauses are in $RC(S)$. Now, since $RC(S)$ is closed under resolution, it will contain the resolvent of these two clauses, and that resolvent will have all of its literals already falsified by the assignments to P_1, \dots, P_{i-1} . This contradicts our assumption that the first falsified clause appears at stage i . Hence, we have proved that the construction never falsifies a clause in $RC(S)$; that is, it produces a model of $RC(S)$ and thus a model of S itself (since S is contained in $RC(S)$).

7.5.3 Horn clauses and definite clauses

The completeness of resolution makes it a very important inference method. In many practical situations, however, the full power of resolution is not needed. Some real-world knowledge bases satisfy certain restrictions on the form of sentences they contain, which enables them to use a more restricted and efficient inference algorithm.

DEFINITE CLAUSE

One such restricted form is the **definite clause**, which is a disjunction of literals of which *exactly one is positive*. For example, the clause $(\neg L_{1,1} \vee \neg Breeze \vee B_{1,1})$ is a definite clause, whereas $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1})$ is not.

HORN CLAUSE

Slightly more general is the **Horn clause**, which is a disjunction of literals of which *at most one is positive*. So all definite clauses are Horn clauses, as are clauses with no positive literals; these are called **goal clauses**. Horn clauses are closed under resolution: if you resolve two Horn clauses, you get back a Horn clause.

GOAL CLAUSES

Knowledge bases containing only definite clauses are interesting for three reasons:

1. Every definite clause can be written as an implication whose premise is a conjunction of positive literals and whose conclusion is a single positive literal. (See Exercise 7.13.) For example, the definite clause $(\neg L_{1,1} \vee \neg Breeze \vee B_{1,1})$ can be written as the implication $(L_{1,1} \wedge Breeze) \Rightarrow B_{1,1}$. In the implication form, the sentence is easier to understand: it says that if the agent is in [1,1] and there is a breeze, then [1,1] is breezy. In Horn form, the premise is called the **body** and the conclusion is called the **head**. A sentence consisting of a single positive literal, such as $L_{1,1}$, is called a **fact**. It too can be written in implication form as $True \Rightarrow L_{1,1}$, but it is simpler to write just $L_{1,1}$.

BODY

HEAD

FACT

$$\begin{aligned}
\text{CNFSentence} &\rightarrow \text{Clause}_1 \wedge \cdots \wedge \text{Clause}_n \\
\text{Clause} &\rightarrow \text{Literal}_1 \vee \cdots \vee \text{Literal}_m \\
\text{Literal} &\rightarrow \text{Symbol} \mid \neg \text{Symbol} \\
\text{Symbol} &\rightarrow P \mid Q \mid R \mid \dots \\
\text{HornClauseForm} &\rightarrow \text{DefiniteClauseForm} \mid \text{GoalClauseForm} \\
\text{DefiniteClauseForm} &\rightarrow (\text{Symbol}_1 \wedge \cdots \wedge \text{Symbol}_l) \Rightarrow \text{Symbol} \\
\text{GoalClauseForm} &\rightarrow (\text{Symbol}_1 \wedge \cdots \wedge \text{Symbol}_l) \Rightarrow \text{False}
\end{aligned}$$

Figure 7.14 A grammar for conjunctive normal form, Horn clauses, and definite clauses. A clause such as $A \wedge B \Rightarrow C$ is still a definite clause when it is written as $\neg A \vee \neg B \vee C$, but only the former is considered the canonical form for definite clauses. One more class is the k -CNF sentence, which is a CNF sentence where each clause has at most k literals.

FORWARD-CHAINING
BACKWARD-
CHAINING

2. Inference with Horn clauses can be done through the **forward-chaining** and **backward-chaining** algorithms, which we explain next. Both of these algorithms are natural, in that the inference steps are obvious and easy for humans to follow. This type of inference is the basis for **logic programming**, which is discussed in Chapter 9.
3. Deciding entailment with Horn clauses can be done in time that is *linear* in the size of the knowledge base—a pleasant surprise.

7.5.4 Forward and backward chaining

The forward-chaining algorithm $\text{PL-FC-ENTAILS?}(KB, q)$ determines if a single proposition symbol q —the query—is entailed by a knowledge base of definite clauses. It begins from known facts (positive literals) in the knowledge base. If all the premises of an implication are known, then its conclusion is added to the set of known facts. For example, if $L_{1,1}$ and *Breeze* are known and $(L_{1,1} \wedge \text{Breeze}) \Rightarrow B_{1,1}$ is in the knowledge base, then $B_{1,1}$ can be added. This process continues until the query q is added or until no further inferences can be made. The detailed algorithm is shown in Figure 7.15; the main point to remember is that it runs in linear time.

The best way to understand the algorithm is through an example and a picture. Figure 7.16(a) shows a simple knowledge base of Horn clauses with A and B as known facts. Figure 7.16(b) shows the same knowledge base drawn as an **AND-OR graph** (see Chapter 4). In AND-OR graphs, multiple links joined by an arc indicate a conjunction—every link must be proved—while multiple links without an arc indicate a disjunction—any link can be proved. It is easy to see how forward chaining works in the graph. The known leaves (here, A and B) are set, and inference propagates up the graph as far as possible. Whenever a conjunction appears, the propagation waits until all the conjuncts are known before proceeding. The reader is encouraged to work through the example in detail.


```

function PL-FC-ENTAILS?(KB, q) returns true or false
  inputs: KB, the knowledge base, a set of propositional definite clauses
           q, the query, a proposition symbol
  count  $\leftarrow$  a table, where count[c] is the number of symbols in c's premise
  inferred  $\leftarrow$  a table, where inferred[s] is initially false for all symbols
  agenda  $\leftarrow$  a queue of symbols, initially symbols known to be true in KB

  while agenda is not empty do
    p  $\leftarrow$  POP(agenda)
    if p = q then return true
    if inferred[p] = false then
      inferred[p]  $\leftarrow$  true
      for each clause c in KB where p is in c.PREMISE do
        decrement count[c]
        if count[c] = 0 then add c.CONCLUSION to agenda
  return false

```

Figure 7.15 The forward-chaining algorithm for propositional logic. The *agenda* keeps track of symbols known to be true but not yet “processed.” The *count* table keeps track of how many premises of each implication are as yet unknown. Whenever a new symbol *p* from the agenda is processed, the count is reduced by one for each implication in whose premise *p* appears (easily identified in constant time with appropriate indexing.) If a count reaches zero, all the premises of the implication are known, so its conclusion can be added to the agenda. Finally, we need to keep track of which symbols have been processed; a symbol that is already in the set of inferred symbols need not be added to the agenda again. This avoids redundant work and prevents loops caused by implications such as $P \Rightarrow Q$ and $Q \Rightarrow P$.

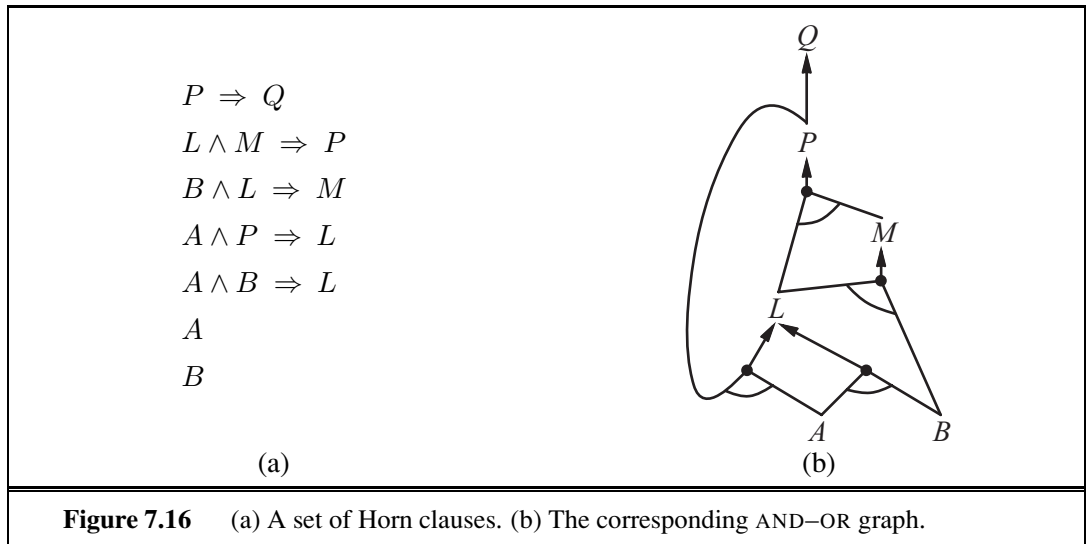
It is easy to see that forward chaining is **sound**: every inference is essentially an application of Modus Ponens. Forward chaining is also **complete**: every entailed atomic sentence will be derived. The easiest way to see this is to consider the final state of the *inferred* table (after the algorithm reaches a **fixed point** where no new inferences are possible). The table contains *true* for each symbol inferred during the process, and *false* for all other symbols. We can view the table as a logical model; moreover, *every definite clause in the original KB is true in this model*. To see this, assume the opposite, namely that some clause $a_1 \wedge \dots \wedge a_k \Rightarrow b$ is false in the model. Then $a_1 \wedge \dots \wedge a_k$ must be true in the model and *b* must be false in the model. But this contradicts our assumption that the algorithm has reached a fixed point! We can conclude, therefore, that the set of atomic sentences inferred at the fixed point defines a model of the original KB. Furthermore, any atomic sentence *q* that is entailed by the KB must be true in all its models and in this model in particular. Hence, every entailed atomic sentence *q* must be inferred by the algorithm.

Forward chaining is an example of the general concept of **data-driven** reasoning—that is, reasoning in which the focus of attention starts with the known data. It can be used within an agent to derive conclusions from incoming percepts, often without a specific query in mind. For example, the wumpus agent might TELL its percepts to the knowledge base using

FIXED POINT



DATA-DRIVEN



an incremental forward-chaining algorithm in which new facts can be added to the agenda to initiate new inferences. In humans, a certain amount of data-driven reasoning occurs as new information arrives. For example, if I am indoors and hear rain starting to fall, it might occur to me that the picnic will be canceled. Yet it will probably not occur to me that the seventeenth petal on the largest rose in my neighbor’s garden will get wet; humans keep forward chaining under careful control, lest they be swamped with irrelevant consequences.

The backward-chaining algorithm, as its name suggests, works backward from the query. If the query q is known to be true, then no work is needed. Otherwise, the algorithm finds those implications in the knowledge base whose conclusion is q . If all the premises of one of those implications can be proved true (by backward chaining), then q is true. When applied to the query Q in Figure 7.16, it works back down the graph until it reaches a set of known facts, A and B , that forms the basis for a proof. The algorithm is essentially identical to the AND-OR-GRAPH-SEARCH algorithm in Figure 4.11. As with forward chaining, an efficient implementation runs in linear time.

GOAL-DIRECTED
REASONING

Backward chaining is a form of **goal-directed reasoning**. It is useful for answering specific questions such as “What shall I do now?” and “Where are my keys?” Often, the cost of backward chaining is *much less* than linear in the size of the knowledge base, because the process touches only relevant facts.

7.6 EFFECTIVE PROPOSITIONAL MODEL CHECKING

In this section, we describe two families of efficient algorithms for general propositional inference based on model checking: One approach based on backtracking search, and one on local hill-climbing search. These algorithms are part of the “technology” of propositional logic. This section can be skimmed on a first reading of the chapter.

The algorithms we describe are for checking satisfiability: the SAT problem. (As noted earlier, testing entailment, $\alpha \models \beta$, can be done by testing *unsatisfiability* of $\alpha \wedge \neg\beta$.) We have already noted the connection between finding a satisfying model for a logical sentence and finding a solution for a constraint satisfaction problem, so it is perhaps not surprising that the two families of algorithms closely resemble the backtracking algorithms of Section 6.3 and the local search algorithms of Section 6.4. They are, however, extremely important in their own right because so many combinatorial problems in computer science can be reduced to checking the satisfiability of a propositional sentence. Any improvement in satisfiability algorithms has huge consequences for our ability to handle complexity in general.

7.6.1 A complete backtracking algorithm

DAVIS-PUTNAM
ALGORITHM

The first algorithm we consider is often called the **Davis–Putnam algorithm**, after the seminal paper by Martin Davis and Hilary Putnam (1960). The algorithm is in fact the version described by Davis, Logemann, and Loveland (1962), so we will call it DPLL after the initials of all four authors. DPLL takes as input a sentence in conjunctive normal form—a set of clauses. Like BACKTRACKING-SEARCH and TT-ENTAILS?, it is essentially a recursive, depth-first enumeration of possible models. It embodies three improvements over the simple scheme of TT-ENTAILS?:

- *Early termination*: The algorithm detects whether the sentence must be true or false, even with a partially completed model. A clause is true if *any* literal is true, even if the other literals do not yet have truth values; hence, the sentence as a whole could be judged true even before the model is complete. For example, the sentence $(A \vee B) \wedge (A \vee C)$ is true if A is true, regardless of the values of B and C . Similarly, a sentence is false if *any* clause is false, which occurs when each of its literals is false. Again, this can occur long before the model is complete. Early termination avoids examination of entire subtrees in the search space.
- *Pure symbol heuristic*: A **pure symbol** is a symbol that always appears with the same “sign” in all clauses. For example, in the three clauses $(A \vee \neg B)$, $(\neg B \vee \neg C)$, and $(C \vee A)$, the symbol A is pure because only the positive literal appears, B is pure because only the negative literal appears, and C is impure. It is easy to see that if a sentence has a model, then it has a model with the pure symbols assigned so as to make their literals *true*, because doing so can never make a clause false. Note that, in determining the purity of a symbol, the algorithm can ignore clauses that are already known to be true in the model constructed so far. For example, if the model contains $B = \text{false}$, then the clause $(\neg B \vee \neg C)$ is already true, and in the remaining clauses C appears only as a positive literal; therefore C becomes pure.
- *Unit clause heuristic*: A **unit clause** was defined earlier as a clause with just one literal. In the context of DPLL, it also means clauses in which all literals but one are already assigned *false* by the model. For example, if the model contains $B = \text{true}$, then $(\neg B \vee \neg C)$ simplifies to $\neg C$, which is a unit clause. Obviously, for this clause to be true, C must be set to *false*. The unit clause heuristic assigns all such symbols before branching on the remainder. One important consequence of the heuristic is that

PURE SYMBOL

```

function DPLL-SATISFIABLE?(s) returns true or false
  inputs: s, a sentence in propositional logic

  clauses  $\leftarrow$  the set of clauses in the CNF representation of s
  symbols  $\leftarrow$  a list of the proposition symbols in s
  return DPLL(clauses, symbols, { })

```

```

function DPLL(clauses, symbols, model) returns true or false

  if every clause in clauses is true in model then return true
  if some clause in clauses is false in model then return false
  P, value  $\leftarrow$  FIND-PURE-SYMBOL(symbols, clauses, model)
  if P is non-null then return DPLL(clauses, symbols – P, model  $\cup$  {P=value})
  P, value  $\leftarrow$  FIND-UNIT-CLAUSE(clauses, model)
  if P is non-null then return DPLL(clauses, symbols – P, model  $\cup$  {P=value})
  P  $\leftarrow$  FIRST(symbols); rest  $\leftarrow$  REST(symbols)
  return DPLL(clauses, rest, model  $\cup$  {P=true}) or
    DPLL(clauses, rest, model  $\cup$  {P=false})

```

Figure 7.17 The DPLL algorithm for checking satisfiability of a sentence in propositional logic. The ideas behind FIND-PURE-SYMBOL and FIND-UNIT-CLAUSE are described in the text; each returns a symbol (or null) and the truth value to assign to that symbol. Like TT-ENTAILS?, DPLL operates over partial models.

any attempt to prove (by refutation) a literal that is already in the knowledge base will succeed immediately (Exercise 7.23). Notice also that assigning one unit clause can create another unit clause—for example, when C is set to *false*, $(C \vee A)$ becomes a unit clause, causing *true* to be assigned to A . This “cascade” of forced assignments is called **unit propagation**. It resembles the process of forward chaining with definite clauses, and indeed, if the CNF expression contains only definite clauses then DPLL essentially replicates forward chaining. (See Exercise 7.24.)

UNIT PROPAGATION

The DPLL algorithm is shown in Figure 7.17, which gives the the essential skeleton of the search process.

What Figure 7.17 does not show are the tricks that enable SAT solvers to scale up to large problems. It is interesting that most of these tricks are in fact rather general, and we have seen them before in other guises:

1. **Component analysis** (as seen with Tasmania in CSPs): As DPLL assigns truth values to variables, the set of clauses may become separated into disjoint subsets, called **components**, that share no unassigned variables. Given an efficient way to detect when this occurs, a solver can gain considerable speed by working on each component separately.
2. **Variable and value ordering** (as seen in Section 6.3.1 for CSPs): Our simple implementation of DPLL uses an arbitrary variable ordering and always tries the value *true* before *false*. The **degree heuristic** (see page 216) suggests choosing the variable that appears most frequently over all remaining clauses.

3. **Intelligent backtracking** (as seen in Section 6.3 for CSPs): Many problems that cannot be solved in hours of run time with chronological backtracking can be solved in seconds with intelligent backtracking that backs up all the way to the relevant point of conflict. All SAT solvers that do intelligent backtracking use some form of **conflict clause learning** to record conflicts so that they won't be repeated later in the search. Usually a limited-size set of conflicts is kept, and rarely used ones are dropped.
4. **Random restarts** (as seen on page 124 for hill-climbing): Sometimes a run appears not to be making progress. In this case, we can start over from the top of the search tree, rather than trying to continue. After restarting, different random choices (in variable and value selection) are made. Clauses that are learned in the first run are retained after the restart and can help prune the search space. Restarting does not guarantee that a solution will be found faster, but it does reduce the variance on the time to solution.
5. **Clever indexing** (as seen in many algorithms): The speedup methods used in DPLL itself, as well as the tricks used in modern solvers, require fast indexing of such things as “the set of clauses in which variable X_i appears as a positive literal.” This task is complicated by the fact that the algorithms are interested only in the clauses that have not yet been satisfied by previous assignments to variables, so the indexing structures must be updated dynamically as the computation proceeds.

With these enhancements, modern solvers can handle problems with tens of millions of variables. They have revolutionized areas such as hardware verification and security protocol verification, which previously required laborious, hand-guided proofs.

7.6.2 Local search algorithms

We have seen several local search algorithms so far in this book, including HILL-CLIMBING (page 122) and SIMULATED-ANNEALING (page 126). These algorithms can be applied directly to satisfiability problems, provided that we choose the right evaluation function. Because the goal is to find an assignment that satisfies every clause, an evaluation function that counts the number of unsatisfied clauses will do the job. In fact, this is exactly the measure used by the MIN-CONFLICTS algorithm for CSPs (page 221). All these algorithms take steps in the space of complete assignments, flipping the truth value of one symbol at a time. The space usually contains many local minima, to escape from which various forms of randomness are required. In recent years, there has been a great deal of experimentation to find a good balance between greediness and randomness.

One of the simplest and most effective algorithms to emerge from all this work is called WALKSAT (Figure 7.18). On every iteration, the algorithm picks an unsatisfied clause and picks a symbol in the clause to flip. It chooses randomly between two ways to pick which symbol to flip: (1) a “min-conflicts” step that minimizes the number of unsatisfied clauses in the new state and (2) a “random walk” step that picks the symbol randomly.

When WALKSAT returns a model, the input sentence is indeed satisfiable, but when it returns *failure*, there are two possible causes: either the sentence is unsatisfiable or we need to give the algorithm more time. If we set $max_flips = \infty$ and $p > 0$, WALKSAT will eventually return a model (if one exists), because the random-walk steps will eventually hit

```

function WALKSAT(clauses, p, max_flips) returns a satisfying model or failure
  inputs: clauses, a set of clauses in propositional logic
           p, the probability of choosing to do a “random walk” move, typically around 0.5
           max_flips, number of flips allowed before giving up

  model ← a random assignment of true/false to the symbols in clauses
  for i = 1 to max_flips do
    if model satisfies clauses then return model
    clause ← a randomly selected clause from clauses that is false in model
    with probability p flip the value in model of a randomly selected symbol from clause
    else flip whichever symbol in clause maximizes the number of satisfied clauses
  return failure

```

Figure 7.18 The WALKSAT algorithm for checking satisfiability by randomly flipping the values of variables. Many versions of the algorithm exist.

upon the solution. Alas, if *max_flips* is infinity and the sentence is unsatisfiable, then the algorithm never terminates!

For this reason, WALKSAT is most useful when we expect a solution to exist—for example, the problems discussed in Chapters 3 and 6 usually have solutions. On the other hand, WALKSAT cannot always detect *unsatisfiability*, which is required for deciding entailment. For example, an agent cannot *reliably* use WALKSAT to prove that a square is safe in the wumpus world. Instead, it can say, “I thought about it for an hour and couldn’t come up with a possible world in which the square *isn’t* safe.” This may be a good empirical indicator that the square is safe, but it’s certainly not a proof.

7.6.3 The landscape of random SAT problems

Some SAT problems are harder than others. *Easy* problems can be solved by any old algorithm, but because we know that SAT is NP-complete, at least some problem instances must require exponential run time. In Chapter 6, we saw some surprising discoveries about certain kinds of problems. For example, the *n*-queens problem—thought to be quite tricky for backtracking search algorithms—turned out to be trivially easy for local search methods, such as min-conflicts. This is because solutions are very densely distributed in the space of assignments, and any initial assignment is guaranteed to have a solution nearby. Thus, *n*-queens is easy because it is **underconstrained**.

UNDERCONSTRAINED

When we look at satisfiability problems in conjunctive normal form, an underconstrained problem is one with relatively *few* clauses constraining the variables. For example, here is a randomly generated 3-CNF sentence with five symbols and five clauses:

$$(\neg D \vee \neg B \vee C) \wedge (B \vee \neg A \vee \neg C) \wedge (\neg C \vee \neg B \vee E) \\ \wedge (E \vee \neg D \vee B) \wedge (B \vee E \vee \neg C).$$

Sixteen of the 32 possible assignments are models of this sentence, so, on average, it would take just two random guesses to find a model. This is an easy satisfiability problem, as are

most such underconstrained problems. On the other hand, an *overconstrained* problem has many clauses relative to the number of variables and is likely to have no solutions.

To go beyond these basic intuitions, we must define exactly how random sentences are generated. The notation $CNF_k(m, n)$ denotes a k -CNF sentence with m clauses and n symbols, where the clauses are chosen uniformly, independently, and without replacement from among all clauses with k different literals, which are positive or negative at random. (A symbol may not appear twice in a clause, nor may a clause appear twice in a sentence.)

Given a source of random sentences, we can measure the probability of satisfiability. Figure 7.19(a) plots the probability for $CNF_3(m, 50)$, that is, sentences with 50 variables and 3 literals per clause, as a function of the clause/symbol ratio, m/n . As we expect, for small m/n the probability of satisfiability is close to 1, and at large m/n the probability is close to 0. The probability drops fairly sharply around $m/n = 4.3$. Empirically, we find that the “cliff” stays in roughly the same place (for $k = 3$) and gets sharper and sharper as n increases. Theoretically, the **satisfiability threshold conjecture** says that for every $k \geq 3$, there is a threshold ratio r_k such that, as n goes to infinity, the probability that $CNF_k(n, rn)$ is satisfiable becomes 1 for all values of r below the threshold, and 0 for all values above. The conjecture remains unproven.

SATISFIABILITY
THRESHOLD
CONJECTURE

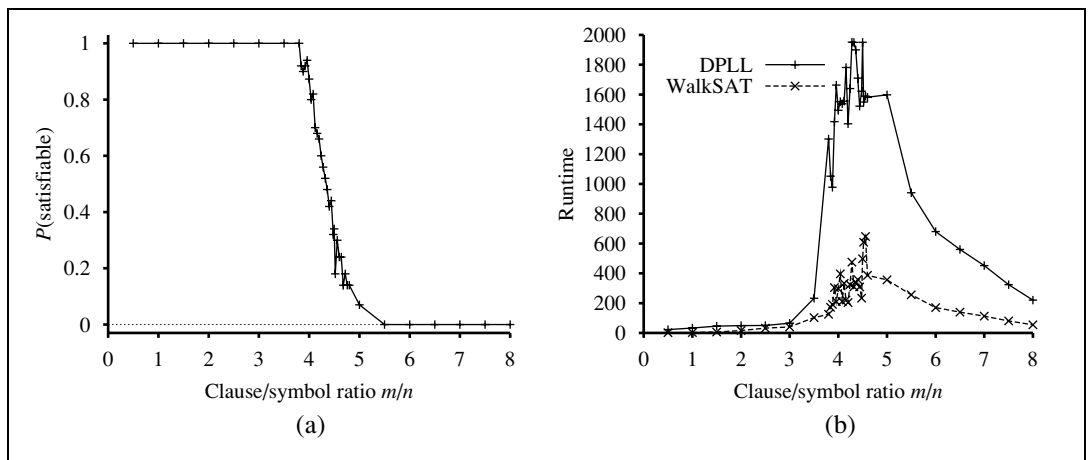


Figure 7.19 (a) Graph showing the probability that a random 3-CNF sentence with $n = 50$ symbols is satisfiable, as a function of the clause/symbol ratio m/n . (b) Graph of the median run time (measured in number of recursive calls to DPLL, a good proxy) on random 3-CNF sentences. The most difficult problems have a clause/symbol ratio of about 4.3.

Now that we have a good idea where the satisfiable and unsatisfiable problems are, the next question is, where are the hard problems? It turns out that they are also often at the threshold value. Figure 7.19(b) shows that 50-symbol problems at the threshold value of 4.3 are about 20 times more difficult to solve than those at a ratio of 3.3. The underconstrained problems are easiest to solve (because it is so easy to guess a solution); the overconstrained problems are not as easy as the underconstrained, but still are much easier than the ones right at the threshold.

7.7 AGENTS BASED ON PROPOSITIONAL LOGIC

In this section, we bring together what we have learned so far in order to construct wumpus world agents that use propositional logic. The first step is to enable the agent to deduce, to the extent possible, the state of the world given its percept history. This requires writing down a complete logical model of the effects of actions. We also show how the agent can keep track of the world efficiently without going back into the percept history for each inference. Finally, we show how the agent can use logical inference to construct plans that are guaranteed to achieve its goals.

7.7.1 The current state of the world

As stated at the beginning of the chapter, a logical agent operates by deducing what to do from a knowledge base of sentences about the world. The knowledge base is composed of axioms—general knowledge about how the world works—and percept sentences obtained from the agent’s experience in a particular world. In this section, we focus on the problem of deducing the current state of the wumpus world—where am I, is that square safe, and so on.

We began collecting axioms in Section 7.4.3. The agent knows that the starting square contains no pit ($\neg P_{1,1}$) and no wumpus ($\neg W_{1,1}$). Furthermore, for each square, it knows that the square is breezy if and only if a neighboring square has a pit; and a square is smelly if and only if a neighboring square has a wumpus. Thus, we include a large collection of sentences of the following form:

$$\begin{aligned} B_{1,1} &\Leftrightarrow (P_{1,2} \vee P_{2,1}) \\ S_{1,1} &\Leftrightarrow (W_{1,2} \vee W_{2,1}) \\ &\dots \end{aligned}$$

The agent also knows that there is exactly one wumpus. This is expressed in two parts. First, we have to say that there is *at least one* wumpus:

$$W_{1,1} \vee W_{1,2} \vee \dots \vee W_{4,3} \vee W_{4,4} .$$

Then, we have to say that there is *at most one* wumpus. For each pair of locations, we add a sentence saying that at least one of them must be wumpus-free:

$$\begin{aligned} &\neg W_{1,1} \vee \neg W_{1,2} \\ &\neg W_{1,1} \vee \neg W_{1,3} \\ &\dots \\ &\neg W_{4,3} \vee \neg W_{4,4} . \end{aligned}$$

So far, so good. Now let’s consider the agent’s percepts. If there is currently a stench, one might suppose that a proposition *Stench* should be added to the knowledge base. This is not quite right, however: if there was no stench at the previous time step, then $\neg \text{Stench}$ would already be asserted, and the new assertion would simply result in a contradiction. The problem is solved when we realize that a percept asserts something *only about the current time*. Thus, if the time step (as supplied to MAKE-PERCEPT-SENTENCE in Figure 7.1) is 4, then we add

$Stench^4$ to the knowledge base, rather than $Stench$ —neatly avoiding any contradiction with $\neg Stench^3$. The same goes for the breeze, bump, glitter, and scream percepts.

FLUENT

ATEMPORAL
VARIABLE

The idea of associating propositions with time steps extends to any aspect of the world that changes over time. For example, the initial knowledge base includes $L_{1,1}^0$ —the agent is in square $[1, 1]$ at time 0—as well as $FacingEast^0$, $HaveArrow^0$, and $WumpusAlive^0$. We use the word **fluent** (from the Latin *fluens*, flowing) to refer an aspect of the world that changes. “Fluent” is a synonym for “state variable,” in the sense described in the discussion of factored representations in Section 2.4.7 on page 57. Symbols associated with permanent aspects of the world do not need a time superscript and are sometimes called **atemporal variables**.

We can connect stench and breeze percepts directly to the properties of the squares where they are experienced through the location fluent as follows.¹⁰ For any time step t and any square $[x, y]$, we assert

$$\begin{aligned} L_{x,y}^t &\Rightarrow (Breeze^t \Leftrightarrow B_{x,y}) \\ L_{x,y}^t &\Rightarrow (Stench^t \Leftrightarrow S_{x,y}). \end{aligned}$$

Now, of course, we need axioms that allow the agent to keep track of fluents such as $L_{x,y}^t$. These fluents change as the result of actions taken by the agent, so, in the terminology of Chapter 3, we need to write down the **transition model** of the wumpus world as a set of logical sentences.

First, we need proposition symbols for the occurrences of actions. As with percepts, these symbols are indexed by time; thus, $Forward^0$ means that the agent executes the *Forward* action at time 0. By convention, the percept for a given time step happens first, followed by the action for that time step, followed by a transition to the next time step.

EFFECT AXIOM

To describe how the world changes, we can try writing **effect axioms** that specify the outcome of an action at the next time step. For example, if the agent is at location $[1, 1]$ facing east at time 0 and goes *Forward*, the result is that the agent is in square $[2, 1]$ and no longer is in $[1, 1]$:

$$L_{1,1}^0 \wedge FacingEast^0 \wedge Forward^0 \Rightarrow (L_{2,1}^1 \wedge \neg L_{1,1}^1). \quad (7.1)$$

We would need one such sentence for each possible time step, for each of the 16 squares, and each of the four orientations. We would also need similar sentences for the other actions: *Grab*, *Shoot*, *Climb*, *TurnLeft*, and *TurnRight*.

FRAME PROBLEM

Let us suppose that the agent does decide to move *Forward* at time 0 and asserts this fact into its knowledge base. Given the effect axiom in Equation (7.1), combined with the initial assertions about the state at time 0, the agent can now deduce that it is in $[2, 1]$. That is, $ASK(KB, L_{2,1}^1) = true$. So far, so good. Unfortunately, the news elsewhere is less good: if we $ASK(KB, HaveArrow^1)$, the answer is *false*, that is, the agent cannot prove it still has the arrow; nor can it prove it *doesn't* have it! The information has been lost because the effect axiom fails to state what remains *unchanged* as the result of an action. The need to do this gives rise to the **frame problem**.¹¹ One possible solution to the frame problem would

¹⁰ Section 7.4.3 conveniently glossed over this requirement.

¹¹ The name “frame problem” comes from “frame of reference” in physics—the assumed stationary background with respect to which motion is measured. It also has an analogy to the frames of a movie, in which normally most of the background stays constant while changes occur in the foreground.

FRAME AXIOM

be to add **frame axioms** explicitly asserting all the propositions that remain the same. For example, for each time t we would have

$$\begin{aligned} Forward^t &\Rightarrow (HaveArrow^t \Leftrightarrow HaveArrow^{t+1}) \\ Forward^t &\Rightarrow (WumpusAlive^t \Leftrightarrow WumpusAlive^{t+1}) \\ &\dots \end{aligned}$$

REPRESENTATIONAL
FRAME PROBLEM

where we explicitly mention every proposition that stays unchanged from time t to time $t + 1$ under the action *Forward*. Although the agent now knows that it still has the arrow after moving forward and that the wumpus hasn't died or come back to life, the proliferation of frame axioms seems remarkably inefficient. In a world with m different actions and n fluents, the set of frame axioms will be of size $O(mn)$. This specific manifestation of the frame problem is sometimes called the **representational frame problem**. Historically, the problem was a significant one for AI researchers; we explore it further in the notes at the end of the chapter.

LOCALITY

INFERENTIAL FRAME
PROBLEM

The representational frame problem is significant because the real world has very many fluents, to put it mildly. Fortunately for us humans, each action typically changes no more than some small number k of those fluents—the world exhibits **locality**. Solving the representational frame problem requires defining the transition model with a set of axioms of size $O(mk)$ rather than size $O(mn)$. There is also an **inferential frame problem**: the problem of projecting forward the results of a t step plan of action in time $O(kt)$ rather than $O(nt)$.

SUCCESSOR-STATE
AXIOM

The solution to the problem involves changing one's focus from writing axioms about *actions* to writing axioms about *fluents*. Thus, for each fluent F , we will have an axiom that defines the truth value of F^{t+1} in terms of fluents (including F itself) at time t and the actions that may have occurred at time t . Now, the truth value of F^{t+1} can be set in one of two ways: either the action at time t causes F to be true at $t + 1$, or F was already true at time t and the action at time t does not cause it to be false. An axiom of this form is called a **successor-state axiom** and has this schema:

$$F^{t+1} \Leftrightarrow ActionCausesF^t \vee (F^t \wedge \neg ActionCausesNotF^t).$$

One of the simplest successor-state axioms is the one for *HaveArrow*. Because there is no action for reloading, the *ActionCausesF^t* part goes away and we are left with

$$HaveArrow^{t+1} \Leftrightarrow (HaveArrow^t \wedge \neg Shoot^t). \quad (7.2)$$

For the agent's location, the successor-state axioms are more elaborate. For example, $L_{1,1}^{t+1}$ is true if either (a) the agent moved *Forward* from $[1, 2]$ when facing south, or from $[2, 1]$ when facing west; or (b) $L_{1,1}^t$ was already true and the action did not cause movement (either because the action was not *Forward* or because the action bumped into a wall). Written out in propositional logic, this becomes

$$\begin{aligned} L_{1,1}^{t+1} &\Leftrightarrow (L_{1,1}^t \wedge (\neg Forward^t \vee Bump^{t+1})) \\ &\vee (L_{1,2}^t \wedge (South^t \wedge Forward^t)) \\ &\vee (L_{2,1}^t \wedge (West^t \wedge Forward^t)). \end{aligned} \quad (7.3)$$

Exercise 7.26 asks you to write out axioms for the remaining wumpus world fluents.

Given a complete set of successor-state axioms and the other axioms listed at the beginning of this section, the agent will be able to ASK and answer any answerable question about the current state of the world. For example, in Section 7.2 the initial sequence of percepts and actions is

$$\begin{aligned}
 &\neg Stench^0 \wedge \neg Breeze^0 \wedge \neg Glitter^0 \wedge \neg Bump^0 \wedge \neg Scream^0 ; Forward^0 \\
 &\neg Stench^1 \wedge Breeze^1 \wedge \neg Glitter^1 \wedge \neg Bump^1 \wedge \neg Scream^1 ; TurnRight^1 \\
 &\neg Stench^2 \wedge Breeze^2 \wedge \neg Glitter^2 \wedge \neg Bump^2 \wedge \neg Scream^2 ; TurnRight^2 \\
 &\neg Stench^3 \wedge Breeze^3 \wedge \neg Glitter^3 \wedge \neg Bump^3 \wedge \neg Scream^3 ; Forward^3 \\
 &\neg Stench^4 \wedge \neg Breeze^4 \wedge \neg Glitter^4 \wedge \neg Bump^4 \wedge \neg Scream^4 ; TurnRight^4 \\
 &\neg Stench^5 \wedge \neg Breeze^5 \wedge \neg Glitter^5 \wedge \neg Bump^5 \wedge \neg Scream^5 ; Forward^5 \\
 &Stench^6 \wedge \neg Breeze^6 \wedge \neg Glitter^6 \wedge \neg Bump^6 \wedge \neg Scream^6
 \end{aligned}$$

At this point, we have $ASK(KB, L_{1,2}^6) = true$, so the agent knows where it is. Moreover, $ASK(KB, W_{1,3}) = true$ and $ASK(KB, P_{3,1}) = true$, so the agent has found the wumpus and one of the pits. The most important question for the agent is whether a square is OK to move into, that is, the square contains no pit nor live wumpus. It's convenient to add axioms for this, having the form

$$OK_{x,y}^t \Leftrightarrow \neg P_{x,y} \wedge \neg (W_{x,y} \wedge WumpusAlive^t).$$

Finally, $ASK(KB, OK_{2,2}^6) = true$, so the square $[2, 2]$ is OK to move into. In fact, given a sound and complete inference algorithm such as DPLL, the agent can answer any answerable question about which squares are OK—and can do so in just a few milliseconds for small-to-medium wumpus worlds.

Solving the representational and inferential frame problems is a big step forward, but a pernicious problem remains: we need to confirm that *all* the necessary preconditions of an action hold for it to have its intended effect. We said that the *Forward* action moves the agent ahead unless there is a wall in the way, but there are many other unusual exceptions that could cause the action to fail: the agent might trip and fall, be stricken with a heart attack, be carried away by giant bats, etc. Specifying all these exceptions is called the **qualification problem**. There is no complete solution within logic; system designers have to use good judgment in deciding how detailed they want to be in specifying their model, and what details they want to leave out. We will see in Chapter 13 that probability theory allows us to summarize all the exceptions without explicitly naming them.

QUALIFICATION
PROBLEM

7.7.2 A hybrid agent

The ability to deduce various aspects of the state of the world can be combined fairly straightforwardly with condition–action rules and with problem-solving algorithms from Chapters 3 and 4 to produce a **hybrid agent** for the wumpus world. Figure 7.20 shows one possible way to do this. The agent program maintains and updates a knowledge base as well as a current plan. The initial knowledge base contains the *atemporal* axioms—those that don't depend on t , such as the axiom relating the breeziness of squares to the presence of pits. At each time step, the new percept sentence is added along with all the axioms that depend on t , such

HYBRID AGENT

as the successor-state axioms. (The next section explains why the agent doesn't need axioms for *future* time steps.) Then, the agent uses logical inference, by ASKing questions of the knowledge base, to work out which squares are safe and which have yet to be visited.

The main body of the agent program constructs a plan based on a decreasing priority of goals. First, if there is a glitter, the program constructs a plan to grab the gold, follow a route back to the initial location, and climb out of the cave. Otherwise, if there is no current plan, the program plans a route to the closest safe square that it has not visited yet, making sure the route goes through only safe squares. Route planning is done with A* search, not with ASK. If there are no safe squares to explore, the next step—if the agent still has an arrow—is to try to make a safe square by shooting at one of the possible wumpus locations. These are determined by asking where $\text{ASK}(KB, \neg W_{x,y})$ is false—that is, where it is *not* known that there is *not* a wumpus. The function PLAN-SHOT (not shown) uses PLAN-ROUTE to plan a sequence of actions that will line up this shot. If this fails, the program looks for a square to explore that is not provably unsafe—that is, a square for which $\text{ASK}(KB, \neg OK_{x,y}^t)$ returns false. If there is no such square, then the mission is impossible and the agent retreats to [1, 1] and climbs out of the cave.

7.7.3 Logical state estimation

The agent program in Figure 7.20 works quite well, but it has one major weakness: as time goes by, the computational expense involved in the calls to ASK goes up and up. This happens mainly because the required inferences have to go back further and further in time and involve more and more proposition symbols. Obviously, this is unsustainable—we cannot have an agent whose time to process each percept grows in proportion to the length of its life! What we really need is a *constant* update time—that is, independent of t . The obvious answer is to save, or **cache**, the results of inference, so that the inference process at the next time step can build on the results of earlier steps instead of having to start again from scratch.

CACHING

As we saw in Section 4.4, the past history of percepts and all their ramifications can be replaced by the **belief state**—that is, some representation of the set of all possible current states of the world.¹² The process of updating the belief state as new percepts arrive is called **state estimation**. Whereas in Section 4.4 the belief state was an explicit list of states, here we can use a logical sentence involving the proposition symbols associated with the current time step, as well as the atemporal symbols. For example, the logical sentence

$$WumpusAlive^1 \wedge L_{2,1}^1 \wedge B_{2,1} \wedge (P_{3,1} \vee P_{2,2}) \quad (7.4)$$

represents the set of all states at time 1 in which the wumpus is alive, the agent is at [2, 1], that square is breezy, and there is a pit in [3, 1] or [2, 2] or both.

Maintaining an exact belief state as a logical formula turns out not to be easy. If there are n fluent symbols for time t , then there are 2^n possible states—that is, assignments of truth values to those symbols. 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, hence 2^{2^n} belief states. Even if we used the most compact possible encoding of logical formulas, with each belief state represented

¹² We can think of the percept history itself as a representation of the belief state, but one that makes inference increasingly expensive as the history gets longer.

```

function HYBRID-WUMPUS-AGENT(percept) returns an action
  inputs: percept, a list, [stench,breeze,glitter,bump,scream]
  persistent: KB, a knowledge base, initially the atemporal “wumpus physics”
               t, a counter, initially 0, indicating time
               plan, an action sequence, initially empty

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  TELL the KB the temporal “physics” sentences for time t
  safe  $\leftarrow \{[x, y] : \text{ASK}(\text{KB}, OK_{x,y}^t) = \text{true}\}$ 
  if ASK(KB, Glittert) = true then
    plan  $\leftarrow [Grab] + \text{PLAN-ROUTE}(\text{current}, \{[1,1]\}, \text{safe}) + [Climb]$ 
  if plan is empty then
    unvisited  $\leftarrow \{[x, y] : \text{ASK}(\text{KB}, L_{x,y}^{t'}) = \text{false} \text{ for all } t' \leq t\}$ 
    plan  $\leftarrow \text{PLAN-ROUTE}(\text{current}, \text{unvisited} \cap \text{safe}, \text{safe})$ 
  if plan is empty and ASK(KB, HaveArrowt) = true then
    possible_wumpus  $\leftarrow \{[x, y] : \text{ASK}(\text{KB}, \neg W_{x,y}) = \text{false}\}$ 
    plan  $\leftarrow \text{PLAN-SHOT}(\text{current}, \text{possible\_wumpus}, \text{safe})$ 
  if plan is empty then // no choice but to take a risk
    not_unsafe  $\leftarrow \{[x, y] : \text{ASK}(\text{KB}, \neg OK_{x,y}^t) = \text{false}\}$ 
    plan  $\leftarrow \text{PLAN-ROUTE}(\text{current}, \text{unvisited} \cap \text{not\_unsafe}, \text{safe})$ 
  if plan is empty then
    plan  $\leftarrow \text{PLAN-ROUTE}(\text{current}, \{[1,1]\}, \text{safe}) + [Climb]$ 
  action  $\leftarrow \text{POP}(\text{plan})$ 
  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t  $\leftarrow t + 1$ 
  return action

```

```

function PLAN-ROUTE(current, goals, allowed) returns an action sequence
  inputs: current, the agent’s current position
          goals, a set of squares; try to plan a route to one of them
          allowed, a set of squares that can form part of the route

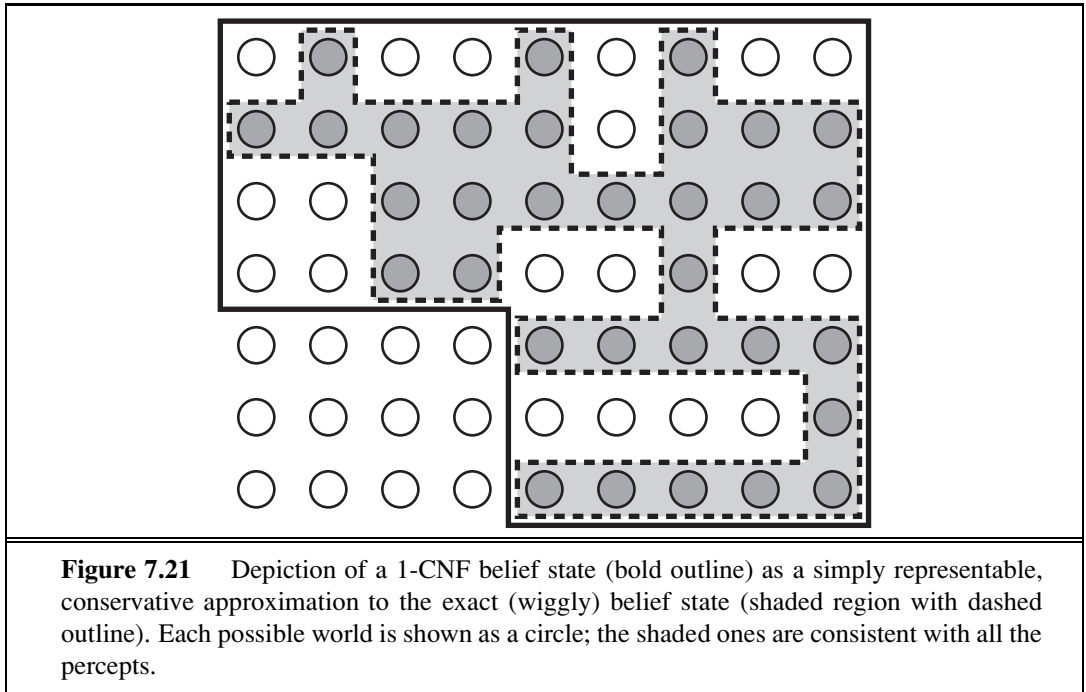
  problem  $\leftarrow \text{ROUTE-PROBLEM}(\text{current}, \text{goals}, \text{allowed})$ 
  return A*-GRAPH-SEARCH(problem)

```

Figure 7.20 A hybrid agent program for the wumpus world. It uses a propositional knowledge base to infer the state of the world, and a combination of problem-solving search and domain-specific code to decide what actions to take.

by a unique binary number, we would need numbers with $\log_2(2^{2^n}) = 2^n$ bits to label the current belief state. That is, exact state estimation may require logical formulas whose size is exponential in the number of symbols.

One very common and natural scheme for *approximate* state estimation is to represent belief states as conjunctions of literals, that is, 1-CNF formulas. To do this, the agent program simply tries to prove X^t and $\neg X^t$ for each symbol X^t (as well as each atemporal symbol whose truth value is not yet known), given the belief state at $t - 1$. The conjunction of



provable literals becomes the new belief state, and the previous belief state is discarded.

It is important to understand that this scheme may lose some information as time goes along. For example, if the sentence in Equation (7.4) were the true belief state, then neither $P_{3,1}$ nor $P_{2,2}$ would be provable individually and neither would appear in the 1-CNF belief state. (Exercise 7.27 explores one possible solution to this problem.) On the other hand, because every literal in the 1-CNF belief state is proved from the previous belief state, and the initial belief state is a true assertion, we know that entire 1-CNF belief state must be true. Thus, *the set of possible states represented by the 1-CNF belief state includes all states that are in fact possible given the full percept history*. As illustrated in Figure 7.21, the 1-CNF belief state acts as a simple outer envelope, or **conservative approximation**, around the exact belief state. We see this idea of conservative approximations to complicated sets as a recurring theme in many areas of AI.



7.7.4 Making plans by propositional inference

The agent in Figure 7.20 uses logical inference to determine which squares are safe, but uses A* search to make plans. In this section, we show how to make plans by logical inference. The basic idea is very simple:

1. Construct a sentence that includes
 - (a) $Init^0$, a collection of assertions about the initial state;
 - (b) $Transition^1, \dots, Transition^t$, the successor-state axioms for all possible actions at each time up to some maximum time t ;
 - (c) the assertion that the goal is achieved at time t : $HaveGold^t \wedge ClimbedOut^t$.

2. Present the whole sentence to a SAT solver. If the solver finds a satisfying model, then the goal is achievable; if the sentence is unsatisfiable, then the planning problem is impossible.
3. Assuming a model is found, extract from the model those variables that represent actions and are assigned *true*. Together they represent a plan to achieve the goals.

A propositional planning procedure, SATPLAN, is shown in Figure 7.22. It implements the basic idea just given, with one twist. Because the agent does not know how many steps it will take to reach the goal, the algorithm tries each possible number of steps t , up to some maximum conceivable plan length T_{\max} . In this way, it is guaranteed to find the shortest plan if one exists. Because of the way SATPLAN searches for a solution, this approach cannot be used in a partially observable environment; SATPLAN would just set the unobservable variables to the values it needs to create a solution.

```

function SATPLAN(init, transition, goal,  $T_{\max}$ ) returns solution or failure
  inputs: init, transition, goal, constitute a description of the problem
            $T_{\max}$ , an upper limit for plan length

  for  $t = 0$  to  $T_{\max}$  do
     $cnf \leftarrow$  TRANSLATE-TO-SAT(init, transition, goal,  $t$ )
     $model \leftarrow$  SAT-SOLVER( $cnf$ )
    if  $model$  is not null then
      return EXTRACT-SOLUTION( $model$ )
  return failure

```

Figure 7.22 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 . 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 key step in using SATPLAN is the construction of the knowledge base. It might seem, on casual inspection, that the wumpus world axioms in Section 7.7.1 suffice for steps 1(a) and 1(b) above. There is, however, a significant difference between the requirements for entailment (as tested by ASK) and those for satisfiability. Consider, for example, the agent's location, initially $[1, 1]$, and suppose the agent's unambitious goal is to be in $[2, 1]$ at time 1. The initial knowledge base contains $L_{1,1}^0$ and the goal is $L_{2,1}^1$. Using ASK, we can prove $L_{2,1}^1$ if $Forward^0$ is asserted, and, reassuringly, we cannot prove $L_{2,1}^1$ if, say, $Shoot^0$ is asserted instead. Now, SATPLAN will find the plan $[Forward^0]$; so far, so good. Unfortunately, SATPLAN also finds the plan $[Shoot^0]$. How could this be? To find out, we inspect the model that SATPLAN constructs: it includes the assignment $L_{2,1}^0$, that is, the agent can be in $[2, 1]$ at time 1 by being there at time 0 and shooting. One might ask, "Didn't we say the agent is in $[1, 1]$ at time 0?" Yes, we did, but we didn't tell the agent that it can't be in two places at once! For entailment, $L_{2,1}^0$ is unknown and cannot, therefore, be used in a proof; for satisfiability,

on the other hand, $L_{2,1}^0$ is unknown and can, therefore, be set to whatever value helps to make the goal true. For this reason, SATPLAN is a good debugging tool for knowledge bases because it reveals places where knowledge is missing. In this particular case, we can fix the knowledge base by asserting that, at each time step, the agent is in exactly one location, using a collection of sentences similar to those used to assert the existence of exactly one wumpus. Alternatively, we can assert $\neg L_{x,y}^0$ for all locations other than $[1, 1]$; the successor-state axiom for location takes care of subsequent time steps. The same fixes also work to make sure the agent has only one orientation.

SATPLAN has more surprises in store, however. The first is that it finds models with impossible actions, such as shooting with no arrow. To understand why, we need to look more carefully at what the successor-state axioms (such as Equation (7.3)) 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 10.14), 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.¹³ For example, we need to say, for each time t , that

$$\text{Shoot}^t \Rightarrow \text{HaveArrow}^t.$$

This ensures that if a plan selects the *Shoot* action at any time, it must be the case that the agent has an arrow at that time.

SATPLAN's second surprise is the creation of plans with multiple simultaneous actions. For example, it may come up with a model in which both Forward^0 and Shoot^0 are true, which is not allowed. To eliminate this problem, we introduce **action exclusion axioms**: for every pair of actions A_i^t and A_j^t we add the axiom

$$\neg A_i^t \vee \neg A_j^t.$$

It might be pointed out that walking forward and shooting at the same time is not so hard to do, whereas, say, shooting and grabbing at the same time is rather impractical. By imposing action exclusion axioms only on pairs of actions that really do interfere with each other, we can allow for plans that include multiple simultaneous actions—and because SATPLAN finds the shortest legal plan, we can be sure that it will take advantage of this capability.

To summarize, SATPLAN finds models for a sentence containing the initial state, the goal, the successor-state axioms, the precondition axioms, and the action exclusion axioms. 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. Modern SAT-solving technology makes the approach quite practical. For example, a DPLL-style solver has no difficulty in generating the 11-step solution for the wumpus world instance shown in Figure 7.2.

This section has described a declarative approach to agent construction: the agent works by a combination of asserting sentences in the knowledge base and performing logical inference. This approach has some weaknesses hidden in phrases such as “for each time t ” and

¹³ Notice that the addition of precondition axioms means that we need not include preconditions for actions in the successor-state axioms.

PRECONDITION
AXIOMS

ACTION EXCLUSION
AXIOM

“for each square $[x, y]$.” For any practical agent, these phrases have to be implemented by code that generates instances of the general sentence schema automatically for insertion into the knowledge base. For a wumpus world of reasonable size—one comparable to a smallish computer game—we might need a 100×100 board and 1000 time steps, leading to knowledge bases with tens or hundreds of millions of sentences. Not only does this become rather impractical, but it also illustrates a deeper problem: we know something about the wumpus world—namely, that the “physics” works the same way across all squares and all time steps—that we cannot express directly in the language of propositional logic. To solve this problem, we need a more expressive language, one in which phrases like “for each time t ” and “for each square $[x, y]$ ” can be written in a natural way. First-order logic, described in Chapter 8, is such a language; in first-order logic a wumpus world of any size and duration can be described in about ten sentences rather than ten million or ten trillion.

7.8 SUMMARY

We have introduced knowledge-based agents and have shown how to define a logic with which such agents can reason about the world. The main points are as follows:

- Intelligent agents need knowledge about the world in order to reach good decisions.
- Knowledge is contained in agents in the form of **sentences** in a **knowledge representation language** that are stored in a **knowledge base**.
- A knowledge-based agent is composed of a knowledge base and an inference mechanism. It operates by storing sentences about the world in its knowledge base, using the inference mechanism to infer new sentences, and using these sentences to decide what action to take.
- A representation language is defined by its **syntax**, which specifies the structure of sentences, and its **semantics**, which defines the **truth** of each sentence in each **possible world** or **model**.
- The relationship of **entailment** between sentences is crucial to our understanding of reasoning. A sentence α entails another sentence β if β is true in all worlds where α is true. Equivalent definitions include the **validity** of the sentence $\alpha \Rightarrow \beta$ and the **unsatisfiability** of the sentence $\alpha \wedge \neg\beta$.
- Inference is the process of deriving new sentences from old ones. **Sound** inference algorithms derive *only* sentences that are entailed; **complete** algorithms derive *all* sentences that are entailed.
- **Propositional logic** is a simple language consisting of **proposition symbols** and **logical connectives**. It can handle propositions that are known true, known false, or completely unknown.
- The set of possible models, given a fixed propositional vocabulary, is finite, so entailment can be checked by enumerating models. Efficient **model-checking** inference algorithms for propositional logic include backtracking and local search methods and can often solve large problems quickly.

- **Inference rules** are patterns of sound inference that can be used to find proofs. The **resolution** rule yields a complete inference algorithm for knowledge bases that are expressed in **conjunctive normal form**. **Forward chaining** and **backward chaining** are very natural reasoning algorithms for knowledge bases in **Horn form**.
- **Local search** methods such as WALKSAT can be used to find solutions. Such algorithms are sound but not complete.
- Logical **state estimation** involves maintaining a logical sentence that describes the set of possible states consistent with the observation history. Each update step requires inference using the transition model of the environment, which is built from **successor-state axioms** that specify how each **fluent** changes.
- Decisions within a logical agent can be made by SAT solving: finding possible models specifying future action sequences that reach the goal. This approach works only for fully observable or sensorless environments.
- Propositional logic does not scale to environments of unbounded size because it lacks the expressive power to deal concisely with time, space, and universal patterns of relationships among objects.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

John McCarthy's paper "Programs with Common Sense" (McCarthy, 1958, 1968) promulgated the notion of agents that use logical reasoning to mediate between percepts and actions. It also raised the flag of declarativism, pointing out that telling an agent what it needs to know is an elegant way to build software. Allen Newell's (1982) article "The Knowledge Level" makes the case that rational agents can be described and analyzed at an abstract level defined by the knowledge they possess rather than the programs they run. The declarative and procedural approaches to AI are analyzed in depth by Boden (1977). The debate was revived by, among others, Brooks (1991) and Nilsson (1991), and continues to this day (Shaparaou *et al.*, 2008). Meanwhile, the declarative approach has spread into other areas of computer science such as networking (Loo *et al.*, 2006).

Logic itself had its origins in ancient Greek philosophy and mathematics. Various logical principles—principles connecting the syntactic structure of sentences with their truth and falsity, with their meaning, or with the validity of arguments in which they figure—are scattered in the works of Plato. The first known systematic study of logic was carried out by Aristotle, whose work was assembled by his students after his death in 322 B.C. as a treatise called the *Organon*. Aristotle's **sylogisms** were what we would now call inference rules. Although the syllogisms included elements of both propositional and first-order logic, the system as a whole lacked the compositional properties required to handle sentences of arbitrary complexity.

The closely related Megarian and Stoic schools (originating in the fifth century B.C. and continuing for several centuries thereafter) began the systematic study of the basic logical connectives. The use of truth tables for defining connectives is due to Philo of Megara. The

Stoics took five basic inference rules as valid without proof, including the rule we now call Modus Ponens. They derived a number of other rules from these five, using, among other principles, the deduction theorem (page 249) and were much clearer about the notion of proof than was Aristotle. A good account of the history of Megarian and Stoic logic is given by Benson Mates (1953).

The idea of reducing logical inference to a purely mechanical process applied to a formal language is due to Wilhelm Leibniz (1646–1716), although he had limited success in implementing the ideas. George Boole (1847) introduced the first comprehensive and workable system of formal logic in his book *The Mathematical Analysis of Logic*. Boole’s logic was closely modeled on the ordinary algebra of real numbers and used substitution of logically equivalent expressions as its primary inference method. Although Boole’s system still fell short of full propositional logic, it was close enough that other mathematicians could quickly fill in the gaps. Schröder (1877) described conjunctive normal form, while Horn form was introduced much later by Alfred Horn (1951). The first comprehensive exposition of modern propositional logic (and first-order logic) is found in Gottlob Frege’s (1879) *Begriffsschrift* (“Concept Writing” or “Conceptual Notation”).

The first mechanical device to carry out logical inferences was constructed by the third Earl of Stanhope (1753–1816). The Stanhope Demonstrator could handle syllogisms and certain inferences in the theory of probability. William Stanley Jevons, one of those who improved upon and extended Boole’s work, constructed his “logical piano” in 1869 to perform inferences in Boolean logic. An entertaining and instructive history of these and other early mechanical devices for reasoning is given by Martin Gardner (1968). The first published computer program for logical inference was the Logic Theorist of Newell, Shaw, and Simon (1957). This program was intended to model human thought processes. Martin Davis (1957) had actually designed a program that came up with a proof in 1954, but the Logic Theorist’s results were published slightly earlier.

Truth tables as a method of testing validity or unsatisfiability in propositional logic were introduced independently by Emil Post (1921) and Ludwig Wittgenstein (1922). In the 1930s, a great deal of progress was made on inference methods for first-order logic. In particular, Gödel (1930) showed that a complete procedure for inference in first-order logic could be obtained via a reduction to propositional logic, using Herbrand’s theorem (Herbrand, 1930). We take up this history again in Chapter 9; the important point here is that the development of efficient propositional algorithms in the 1960s was motivated largely by the interest of mathematicians in an effective theorem prover for first-order logic. The Davis–Putnam algorithm (Davis and Putnam, 1960) was the first effective algorithm for propositional resolution but was in most cases much less efficient than the DPLL backtracking algorithm introduced two years later (1962). The full resolution rule and a proof of its completeness appeared in a seminal paper by J. A. Robinson (1965), which also showed how to do first-order reasoning without resort to propositional techniques.

Stephen Cook (1971) showed that deciding satisfiability of a sentence in propositional logic (the SAT problem) is NP-complete. Since deciding entailment is equivalent to deciding unsatisfiability, it is co-NP-complete. Many subsets of propositional logic are known for which the satisfiability problem is polynomially solvable; Horn clauses are one such subset.

The linear-time forward-chaining algorithm for Horn clauses is due to Dowling and Gallier (1984), who describe their algorithm as a dataflow process similar to the propagation of signals in a circuit.

Early theoretical investigations showed that DPLL has polynomial average-case complexity for certain natural distributions of problems. This potentially exciting fact became less exciting when Franco and Paull (1983) showed that the same problems could be solved in constant time simply by guessing random assignments. The random-generation method described in the chapter produces much harder problems. Motivated by the empirical success of local search on these problems, Koutsoupias and Papadimitriou (1992) showed that a simple hill-climbing algorithm can solve *almost all* satisfiability problem instances very quickly, suggesting that hard problems are rare. Moreover, Schöning (1999) exhibited a randomized hill-climbing algorithm whose *worst-case* expected run time on 3-SAT problems (that is, satisfiability of 3-CNF sentences) is $O(1.333^n)$ —still exponential, but substantially faster than previous worst-case bounds. The current record is $O(1.324^n)$ (Iwama and Tamaki, 2004). Achlioptas *et al.* (2004) and Alekhovich *et al.* (2005) exhibit families of 3-SAT instances for which all known DPLL-like algorithms require exponential running time.

On the practical side, efficiency gains in propositional solvers have been marked. Given ten minutes of computing time, the original DPLL algorithm in 1962 could only solve problems with no more than 10 or 15 variables. By 1995 the SATZ solver (Li and Anbulagan, 1997) could handle 1,000 variables, thanks to optimized data structures for indexing variables. Two crucial contributions were the **watched literal** indexing technique of Zhang and Stickel (1996), which makes unit propagation very efficient, and the introduction of clause (i.e., constraint) learning techniques from the CSP community by Bayardo and Schrag (1997). Using these ideas, and spurred by the prospect of solving industrial-scale circuit verification problems, Moskewicz *et al.* (2001) developed the CHAFF solver, which could handle problems with millions of variables. Beginning in 2002, SAT competitions have been held regularly; most of the winning entries have either been descendants of CHAFF or have used the same general approach. RSAT (Pipatsrisawat and Darwiche, 2007), the 2007 winner, falls in the latter category. Also noteworthy is MINISAT (Een and Sörensson, 2003), an open-source implementation available at <http://minisat.se> that is designed to be easily modified and improved. The current landscape of solvers is surveyed by Gomes *et al.* (2008).

Local search algorithms for satisfiability were tried by various authors throughout the 1980s; all of the algorithms were based on the idea of minimizing the number of unsatisfied clauses (Hansen and Jaumard, 1990). A particularly effective algorithm was developed by Gu (1989) and independently by Selman *et al.* (1992), who called it GSAT and showed that it was capable of solving a wide range of very hard problems very quickly. The WALKSAT algorithm described in the chapter is due to Selman *et al.* (1996).

The “phase transition” in satisfiability of random k -SAT problems was first observed by Simon and Dubois (1989) and has given rise to a great deal of theoretical and empirical research—due, in part, to the obvious connection to phase transition phenomena in statistical physics. Cheeseman *et al.* (1991) observed phase transitions in several CSPs and conjecture that all NP-hard problems have a phase transition. Crawford and Auton (1993) located the 3-SAT transition at a clause/variable ratio of around 4.26, noting that this coincides with a

sharp peak in the run time of their SAT solver. Cook and Mitchell (1997) provide an excellent summary of the early literature on the problem.

The current state of theoretical understanding is summarized by Achlioptas (2009). The **satisfiability threshold conjecture** states that, for each k , there is a sharp satisfiability threshold r_k , such that as the number of variables $n \rightarrow \infty$, instances below the threshold are *satisfiable* with probability 1, while those above the threshold are *unsatisfiable* with probability 1. The conjecture was not quite proved by Friedgut (1999): a sharp threshold exists but its location might depend on n even as $n \rightarrow \infty$. Despite significant progress in asymptotic analysis of the threshold location for large k (Achlioptas and Peres, 2004; Achlioptas *et al.*, 2007), all that can be proved for $k = 3$ is that it lies in the range $[3.52, 4.51]$. Current theory suggests that a peak in the run time of a SAT solver is not necessarily related to the satisfiability threshold, but instead to a phase transition in the solution distribution and structure of SAT instances. Empirical results due to Coarfa *et al.* (2003) support this view. In fact, algorithms such as **survey propagation** (Parisi and Zecchina, 2002; Maneva *et al.*, 2007) take advantage of special properties of random SAT instances near the satisfiability threshold and greatly outperform general SAT solvers on such instances.

The best sources for information on satisfiability, both theoretical and practical, are the *Handbook of Satisfiability* (Biere *et al.*, 2009) and the regular *International Conferences on Theory and Applications of Satisfiability Testing*, known as SAT.

The idea of building agents with propositional logic can be traced back to the seminal paper of McCulloch and Pitts (1943), which initiated the field of neural networks. Contrary to popular supposition, the paper was concerned with the implementation of a Boolean circuit-based agent design in the brain. Circuit-based agents, which perform computation by propagating signals in hardware circuits rather than running algorithms in general-purpose computers, have received little attention in AI, however. The most notable exception is the work of Stan Rosenschein (Rosenstein, 1985; Kaelbling and Rosenschein, 1990), who developed ways to compile circuit-based agents from declarative descriptions of the task environment. (Rosenstein's approach is described at some length in the second edition of this book.) The work of Rod Brooks (1986, 1989) demonstrates the effectiveness of circuit-based designs for controlling robots—a topic we take up in Chapter 25. Brooks (1991) argues that circuit-based designs are *all* that is needed for AI—that representation and reasoning are cumbersome, expensive, and unnecessary. In our view, neither approach is sufficient by itself. Williams *et al.* (2003) show how a hybrid agent design not too different from our wumpus agent has been used to control NASA spacecraft, planning sequences of actions and diagnosing and recovering from faults.

The general problem of keeping track of a partially observable environment was introduced for state-based representations in Chapter 4. Its instantiation for propositional representations was studied by Amir and Russell (2003), who identified several classes of environments that admit efficient state-estimation algorithms and showed that for several other classes the problem is intractable. The **temporal-projection** problem, which involves determining what propositions hold true after an action sequence is executed, can be seen as a special case of state estimation with empty percepts. Many authors have studied this problem because of its importance in planning; some important hardness results were established by

Liberatore (1997). The idea of representing a belief state with propositions can be traced to Wittgenstein (1922).

Logical state estimation, of course, requires a logical representation of the effects of actions—a key problem in AI since the late 1950s. The dominant proposal has been the **situation calculus** formalism (McCarthy, 1963), which is couched within first-order logic. We discuss situation calculus, and various extensions and alternatives, in Chapters 10 and 12. The approach taken in this chapter—using temporal indices on propositional variables—is more restrictive but has the benefit of simplicity. The general approach embodied in the SATPLAN algorithm was proposed by Kautz and Selman (1992). Later generations of SATPLAN were able to take advantage of the advances in SAT solvers, described earlier, and remain among the most effective ways of solving difficult problems (Kautz, 2006).

The **frame problem** was first recognized by McCarthy and Hayes (1969). Many researchers considered the problem unsolvable 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 solution of the frame problem with successor-state axioms is due to Ray Reiter (1991). Thielscher (1999) identifies the inferential frame problem as a separate idea and provides a solution. In retrospect, one can see that Rosenschein's (1985) agents were using circuits that implemented successor-state axioms, but Rosenschein did not notice that the frame problem was thereby largely solved. Foo (2001) explains why the discrete-event control theory models typically used by engineers do not have to explicitly deal with the frame problem: because they are dealing with prediction and control, not with explanation and reasoning about counterfactual situations.

Modern propositional solvers have wide applicability in industrial applications. The application of propositional inference in the synthesis of computer hardware is now a standard technique having many large-scale deployments (Nowick *et al.*, 1993). The SATMC satisfiability checker was used to detect a previously unknown vulnerability in a Web browser user sign-on protocol (Armando *et al.*, 2008).

The wumpus world was invented by Gregory Yob (1975). Ironically, Yob developed it because he was bored with games played on a rectangular grid: the topology of his original wumpus world was a dodecahedron, and we put it back in the boring old grid. Michael Genesereth was the first to suggest that the wumpus world be used as an agent testbed.

EXERCISES

7.1 Suppose the agent has progressed to the point shown in Figure 7.4(a), page 239, having perceived nothing in [1,1], a breeze in [2,1], and a stench in [1,2], and is now concerned with the contents of [1,3], [2,2], and [3,1]. Each of these can contain a pit, and at most one can contain a wumpus. Following the example of Figure 7.5, construct the set of possible worlds. (You should find 32 of them.) Mark the worlds in which the KB is true and those in which

each of the following sentences is true:

$\alpha_2 = \text{"There is no pit in [2,2]."}'$

$\alpha_3 = \text{"There is a wumpus in [1,3]."}'$

Hence show that $KB \models \alpha_2$ and $KB \models \alpha_3$.

7.2 (Adapted from Barwise and Etchemendy (1993).) Given the following, can you prove that the unicorn is mythical? How about magical? Horned?

If the unicorn is mythical, then it is immortal, but if it is not mythical, then it is a mortal mammal. If the unicorn is either immortal or a mammal, then it is horned. The unicorn is magical if it is horned.

7.3 Consider the problem of deciding whether a propositional logic sentence is true in a given model.

- Write a recursive algorithm $\text{PL-TRUE?}(s, m)$ that returns *true* if and only if the sentence s is true in the model m (where m assigns a truth value for every symbol in s). The algorithm should run in time linear in the size of the sentence. (Alternatively, use a version of this function from the online code repository.)
- Give three examples of sentences that can be determined to be true or false in a *partial* model that does not specify a truth value for some of the symbols.
- Show that the truth value (if any) of a sentence in a partial model cannot be determined efficiently in general.
- Modify your PL-TRUE? algorithm so that it can sometimes judge truth from partial models, while retaining its recursive structure and linear run time. Give three examples of sentences whose truth in a partial model is *not* detected by your algorithm.
- Investigate whether the modified algorithm makes TT-ENTAILS? more efficient.

7.4 Which of the following are correct?

- $\text{False} \models \text{True}$.
- $\text{True} \models \text{False}$.
- $(A \wedge B) \models (A \Leftrightarrow B)$.
- $A \Leftrightarrow B \models A \vee B$.
- $A \Leftrightarrow B \models \neg A \vee B$.
- $(A \wedge B) \Rightarrow C \models (A \Rightarrow C) \vee (B \Rightarrow C)$.
- $(C \vee (\neg A \wedge \neg B)) \equiv ((A \Rightarrow C) \wedge (B \Rightarrow C))$.
- $(A \vee B) \wedge (\neg C \vee \neg D \vee E) \models (A \vee B)$.
- $(A \vee B) \wedge (\neg C \vee \neg D \vee E) \models (A \vee B) \wedge (\neg D \vee E)$.
- $(A \vee B) \wedge \neg(A \Rightarrow B)$ is satisfiable.
- $(A \Leftrightarrow B) \wedge (\neg A \vee B)$ is satisfiable.
- $(A \Leftrightarrow B) \Leftrightarrow C$ has the same number of models as $(A \Leftrightarrow B)$ for any fixed set of proposition symbols that includes A, B, C .

7.5 Prove each of the following assertions:

- a. α is valid if and only if $\text{True} \models \alpha$.
- b. For any α , $\text{False} \models \alpha$.
- c. $\alpha \models \beta$ if and only if the sentence $(\alpha \Rightarrow \beta)$ is valid.
- d. $\alpha \equiv \beta$ if and only if the sentence $(\alpha \Leftrightarrow \beta)$ is valid.
- e. $\alpha \models \beta$ if and only if the sentence $(\alpha \wedge \neg\beta)$ is unsatisfiable.

7.6 Prove, or find a counterexample to, each of the following assertions:

- a. If $\alpha \models \gamma$ or $\beta \models \gamma$ (or both) then $(\alpha \wedge \beta) \models \gamma$
- b. If $\alpha \models (\beta \wedge \gamma)$ then $\alpha \models \beta$ and $\alpha \models \gamma$.
- c. If $\alpha \models (\beta \vee \gamma)$ then $\alpha \models \beta$ or $\alpha \models \gamma$ (or both).

7.7 Consider a vocabulary with only four propositions, A , B , C , and D . How many models are there for the following sentences?

- a. $B \vee C$.
- b. $\neg A \vee \neg B \vee \neg C \vee \neg D$.
- c. $(A \Rightarrow B) \wedge A \wedge \neg B \wedge C \wedge D$.

7.8 We have defined four binary logical connectives.

- a. Are there any others that might be useful?
- b. How many binary connectives can there be?
- c. Why are some of them not very useful?

7.9 Using a method of your choice, verify each of the equivalences in Figure 7.11 (page 249).

7.10 Decide whether each of the following sentences is valid, unsatisfiable, or neither. Verify your decisions using truth tables or the equivalence rules of Figure 7.11 (page 249).

- a. $\text{Smoke} \Rightarrow \text{Smoke}$
- b. $\text{Smoke} \Rightarrow \text{Fire}$
- c. $(\text{Smoke} \Rightarrow \text{Fire}) \Rightarrow (\neg\text{Smoke} \Rightarrow \neg\text{Fire})$
- d. $\text{Smoke} \vee \text{Fire} \vee \neg\text{Fire}$
- e. $((\text{Smoke} \wedge \text{Heat}) \Rightarrow \text{Fire}) \Leftrightarrow ((\text{Smoke} \Rightarrow \text{Fire}) \vee (\text{Heat} \Rightarrow \text{Fire}))$
- f. $(\text{Smoke} \Rightarrow \text{Fire}) \Rightarrow ((\text{Smoke} \wedge \text{Heat}) \Rightarrow \text{Fire})$
- g. $\text{Big} \vee \text{Dumb} \vee (\text{Big} \Rightarrow \text{Dumb})$

7.11 Any propositional logic sentence is logically equivalent to the assertion that each possible world in which it would be false is not the case. From this observation, prove that any sentence can be written in CNF.

7.12 Use resolution to prove the sentence $\neg A \wedge \neg B$ from the clauses in Exercise 7.20.

7.13 This exercise looks into the relationship between clauses and implication sentences.

- a. Show that the clause $(\neg P_1 \vee \dots \vee \neg P_m \vee Q)$ is logically equivalent to the implication sentence $(P_1 \wedge \dots \wedge P_m) \Rightarrow Q$.
- b. Show that every clause (regardless of the number of positive literals) can be written in the form $(P_1 \wedge \dots \wedge P_m) \Rightarrow (Q_1 \vee \dots \vee Q_n)$, where the P s and Q s are proposition symbols. A knowledge base consisting of such sentences is in **implicative normal form** or **Kowalski form** (Kowalski, 1979).
- c. Write down the full resolution rule for sentences in implicative normal form.

7.14 According to some political pundits, a person who is radical (R) is electable (E) if he/she is conservative (C), but otherwise is not electable.

- a. Which of the following are correct representations of this assertion?

- (i) $(R \wedge E) \iff C$
- (ii) $R \Rightarrow (E \iff C)$
- (iii) $R \Rightarrow ((C \Rightarrow E) \vee \neg E)$

- b. Which of the sentences in (a) can be expressed in Horn form?

7.15 This question considers representing satisfiability (SAT) problems as CSPs.

- a. Draw the constraint graph corresponding to the SAT problem

$$(\neg X_1 \vee X_2) \wedge (\neg X_2 \vee X_3) \wedge \dots \wedge (\neg X_{n-1} \vee X_n)$$

for the particular case $n = 5$.

- b. How many solutions are there for this general SAT problem as a function of n ?
- c. Suppose we apply BACKTRACKING-SEARCH (page 215) to find *all* solutions to a SAT CSP of the type given in (a). (To find *all* solutions to a CSP, we simply modify the basic algorithm so it continues searching after each solution is found.) Assume that variables are ordered X_1, \dots, X_n and *false* is ordered before *true*. How much time will the algorithm take to terminate? (Write an $O(\cdot)$ expression as a function of n .)
- d. We know that SAT problems in Horn form can be solved in linear time by forward chaining (unit propagation). We also know that every tree-structured binary CSP with discrete, finite domains can be solved in time linear in the number of variables (Section 6.5). Are these two facts connected? Discuss.

7.16 Explain why every nonempty propositional clause, by itself, is satisfiable. Prove rigorously that every set of five 3-SAT clauses is satisfiable, provided that each clause mentions exactly three distinct variables. What is the smallest set of such clauses that is unsatisfiable? Construct such a set.

7.17 A propositional 2-CNF expression is a conjunction of clauses, each containing *exactly* 2 literals, e.g.,

$$(A \vee B) \wedge (\neg A \vee C) \wedge (\neg B \vee D) \wedge (\neg C \vee G) \wedge (\neg D \vee G).$$

- a. Prove using resolution that the above sentence entails G .

- b. Two clauses are *semantically distinct* if they are not logically equivalent. How many semantically distinct 2-CNF clauses can be constructed from n proposition symbols?
- c. Using your answer to (b), prove that propositional resolution always terminates in time polynomial in n given a 2-CNF sentence containing no more than n distinct symbols.
- d. Explain why your argument in (c) does not apply to 3-CNF.

7.18 Consider the following sentence:

$$[(Food \Rightarrow Party) \vee (Drinks \Rightarrow Party)] \Rightarrow [(Food \wedge Drinks) \Rightarrow Party] .$$

- a. Determine, using enumeration, whether this sentence is valid, satisfiable (but not valid), or unsatisfiable.
- b. Convert the left-hand and right-hand sides of the main implication into CNF, showing each step, and explain how the results confirm your answer to (a).
- c. Prove your answer to (a) using resolution.

DISJUNCTIVE
NORMAL FORM

7.19 A sentence is in **disjunctive normal form** (DNF) if it is the disjunction of conjunctions of literals. For example, the sentence $(A \wedge B \wedge \neg C) \vee (\neg A \wedge C) \vee (B \wedge \neg C)$ is in DNF.

- a. Any propositional logic sentence is logically equivalent to the assertion that some possible world in which it would be true is in fact the case. From this observation, prove that any sentence can be written in DNF.
- b. Construct an algorithm that converts any sentence in propositional logic into DNF. (*Hint*: The algorithm is similar to the algorithm for conversion to CNF given in Section 7.5.2.)
- c. Construct a simple algorithm that takes as input a sentence in DNF and returns a satisfying assignment if one exists, or reports that no satisfying assignment exists.
- d. Apply the algorithms in (b) and (c) to the following set of sentences:

$$\begin{aligned} A &\Rightarrow B \\ B &\Rightarrow C \\ C &\Rightarrow \neg A . \end{aligned}$$

- e. Since the algorithm in (b) is very similar to the algorithm for conversion to CNF, and since the algorithm in (c) is much simpler than any algorithm for solving a set of sentences in CNF, why is this technique not used in automated reasoning?

7.20 Convert the following set of sentences to clausal form.

$$\begin{aligned} \text{S1: } A &\Leftrightarrow (B \vee E). \\ \text{S2: } E &\Rightarrow D. \\ \text{S3: } C \wedge F &\Rightarrow \neg B. \\ \text{S4: } E &\Rightarrow B. \\ \text{S5: } B &\Rightarrow F. \\ \text{S6: } B &\Rightarrow C \end{aligned}$$

Give a trace of the execution of DPLL on the conjunction of these clauses.

7.21 Is a randomly generated 4-CNF sentence with n symbols and m clauses more or less likely to be solvable than a randomly generated 3-CNF sentence with n symbols and m clauses? Explain.

7.22 Minesweeper, the well-known computer game, is closely related to the wumpus world. A minesweeper world is a rectangular grid of N squares with M invisible mines scattered among them. Any square may be probed by the agent; instant death follows if a mine is probed. Minesweeper indicates the presence of mines by revealing, in each probed square, the *number* of mines that are directly or diagonally adjacent. The goal is to probe every unmined square.

- Let $X_{i,j}$ be true iff square $[i, j]$ contains a mine. Write down the assertion that exactly two mines are adjacent to $[1,1]$ as a sentence involving some logical combination of $X_{i,j}$ propositions.
- Generalize your assertion from (a) by explaining how to construct a CNF sentence asserting that k of n neighbors contain mines.
- Explain precisely how an agent can use DPLL to prove that a given square does (or does not) contain a mine, ignoring the global constraint that there are exactly M mines in all.
- Suppose that the global constraint is constructed from your method from part (b). How does the number of clauses depend on M and N ? Suggest a way to modify DPLL so that the global constraint does not need to be represented explicitly.
- Are any conclusions derived by the method in part (c) invalidated when the global constraint is taken into account?
- Give examples of configurations of probe values that induce *long-range dependencies* such that the contents of a given unprobed square would give information about the contents of a far-distant square. (*Hint*: consider an $N \times 1$ board.)

7.23 How long does it take to prove $KB \models \alpha$ using DPLL when α is a literal *already contained in KB*? Explain.

7.24 Trace the behavior of DPLL on the knowledge base in Figure 7.16 when trying to prove Q , and compare this behavior with that of the forward-chaining algorithm.

7.25 Write a successor-state axiom for the *Locked* predicate, which applies to doors, assuming the only actions available are *Lock* and *Unlock*.

7.26 Section 7.7.1 provides some of the successor-state axioms required for the wumpus world. Write down axioms for all remaining fluent symbols.



7.27 Modify the HYBRID-WUMPUS-AGENT to use the 1-CNF logical state estimation method described on page 271. We noted on that page that such an agent will not be able to acquire, maintain, and use more complex beliefs such as the disjunction $P_{3,1} \vee P_{2,2}$. Suggest a method for overcoming this problem by defining additional proposition symbols, and try it out in the wumpus world. Does it improve the performance of the agent?

8

FIRST-ORDER LOGIC

In which we notice that the world is blessed with many objects, some of which are related to other objects, and in which we endeavor to reason about them.

FIRST-ORDER LOGIC

In Chapter 7, we showed how a knowledge-based agent could represent the world in which it operates and deduce what actions to take. We used propositional logic as our representation language because it sufficed to illustrate the basic concepts of logic and knowledge-based agents. Unfortunately, propositional logic is too puny a language to represent knowledge of complex environments in a concise way. In this chapter, we examine **first-order logic**,¹ which is sufficiently expressive to represent a good deal of our commonsense knowledge. It also either subsumes or forms the foundation of many other representation languages and has been studied intensively for many decades. We begin in Section 8.1 with a discussion of representation languages in general; Section 8.2 covers the syntax and semantics of first-order logic; Sections 8.3 and 8.4 illustrate the use of first-order logic for simple representations.

8.1 REPRESENTATION REVISITED

In this section, we discuss the nature of representation languages. Our discussion motivates the development of first-order logic, a much more expressive language than the propositional logic introduced in Chapter 7. We look at propositional logic and at other kinds of languages to understand what works and what fails. Our discussion will be cursory, compressing centuries of thought, trial, and error into a few paragraphs.

Programming languages (such as C++ or Java or Lisp) are by far the largest class of formal languages in common use. Programs themselves represent, in a direct sense, only computational processes. Data structures within programs can represent facts; for example, a program could use a 4×4 array to represent the contents of the wumpus world. Thus, the programming language statement $World[2,2] \leftarrow Pit$ is a fairly natural way to assert that there is a pit in square [2,2]. (Such representations might be considered *ad hoc*; database systems were developed precisely to provide a more general, domain-independent way to store and

¹ Also called **first-order predicate calculus**, sometimes abbreviated as **FOL** or **FOPC**.

retrieve facts.) What programming languages lack is any general mechanism for deriving facts from other facts; each update to a data structure is done by a domain-specific procedure whose details are derived by the programmer from his or her own knowledge of the domain. This procedural approach can be contrasted with the **declarative** nature of propositional logic, in which knowledge and inference are separate, and inference is entirely domain independent.

A second drawback of data structures in programs (and of databases, for that matter) is the lack of any easy way to say, for example, “There is a pit in [2,2] or [3,1]” or “If the wumpus is in [1,1] then he is not in [2,2].” Programs can store a single value for each variable, and some systems allow the value to be “unknown,” but they lack the expressiveness required to handle partial information.

Propositional logic is a declarative language because its semantics is based on a truth relation between sentences and possible worlds. It also has sufficient expressive power to deal with partial information, using disjunction and negation. Propositional logic has a third property that is desirable in representation languages, namely, **compositionality**. In a compositional language, the meaning of a sentence is a function of the meaning of its parts. For example, the meaning of “ $S_{1,4} \wedge S_{1,2}$ ” is related to the meanings of “ $S_{1,4}$ ” and “ $S_{1,2}$.” It would be very strange if “ $S_{1,4}$ ” meant that there is a stench in square [1,4] and “ $S_{1,2}$ ” meant that there is a stench in square [1,2], but “ $S_{1,4} \wedge S_{1,2}$ ” meant that France and Poland drew 1–1 in last week’s ice hockey qualifying match. Clearly, noncompositionality makes life much more difficult for the reasoning system.

As we saw in Chapter 7, however, propositional logic lacks the expressive power to *concisely* describe an environment with many objects. For example, we were forced to write a separate rule about breezes and pits for each square, such as

$$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1}).$$

In English, on the other hand, it seems easy enough to say, once and for all, “Squares adjacent to pits are breezy.” The syntax and semantics of English somehow make it possible to describe the environment concisely.

8.1.1 The language of thought

Natural languages (such as English or Spanish) are very expressive indeed. We managed to write almost this whole book in natural language, with only occasional lapses into other languages (including logic, mathematics, and the language of diagrams). There is a long tradition in linguistics and the philosophy of language that views natural language as a declarative knowledge representation language. If we could uncover the rules for natural language, we could use it in representation and reasoning systems and gain the benefit of the billions of pages that have been written in natural language.

The modern view of natural language is that it serves as a medium for **communication** rather than pure representation. When a speaker points and says, “Look!” the listener comes to know that, say, Superman has finally appeared over the rooftops. Yet we would not want to say that the sentence “Look!” represents that fact. Rather, the meaning of the sentence depends both on the sentence itself and on the **context** in which the sentence was spoken. Clearly, one could not store a sentence such as “Look!” in a knowledge base and expect to

AMBIGUITY

recover its meaning without also storing a representation of the context—which raises the question of how the context itself can be represented. Natural languages also suffer from **ambiguity**, a problem for a representation language. As Pinker (1995) puts it: “When people think about *spring*, surely they are not confused as to whether they are thinking about a season or something that goes *boing*—and if one word can correspond to two thoughts, thoughts can’t be words.”

The famous **Sapir–Whorf hypothesis** claims that our understanding of the world *is* strongly influenced by the language we speak. Whorf (1956) wrote “We cut nature up, organize it into concepts, and ascribe significances as we do, largely because we are parties to an agreement to organize it this way—an agreement that holds throughout our speech community and is codified in the patterns of our language.” It is certainly true that different speech communities divide up the world differently. The French have two words “chaise” and “fauteuil,” for a concept that English speakers cover with one: “chair.” But English speakers can easily recognize the category *fauteuil* and give it a name—roughly “open-arm chair”—so does language really make a difference? Whorf relied mainly on intuition and speculation, but in the intervening years we actually have real data from anthropological, psychological and neurological studies.

For example, can you remember which of the following two phrases formed the opening of Section 8.1?

“In this section, we discuss the nature of representation languages . . .”

“This section covers the topic of knowledge representation languages . . .”

Wanner (1974) did a similar experiment and found that subjects made the right choice at chance level—about 50% of the time—but remembered the content of what they read with better than 90% accuracy. This suggests that people process the words to form some kind of *nonverbal* representation.

More interesting is the case in which a concept is completely absent in a language. Speakers of the Australian aboriginal language Guugu Yimithirr have no words for relative directions, such as front, back, right, or left. Instead they use absolute directions, saying, for example, the equivalent of “I have a pain in my north arm.” This difference in language makes a difference in behavior: Guugu Yimithirr speakers are better at navigating in open terrain, while English speakers are better at placing the fork to the right of the plate.

Language also seems to influence thought through seemingly arbitrary grammatical features such as the gender of nouns. For example, “bridge” is masculine in Spanish and feminine in German. Boroditsky (2003) asked subjects to choose English adjectives to describe a photograph of a particular bridge. Spanish speakers chose *big*, *dangerous*, *strong*, and *towering*, whereas German speakers chose *beautiful*, *elegant*, *fragile*, and *slender*. Words can serve as anchor points that affect how we perceive the world. Loftus and Palmer (1974) showed experimental subjects a movie of an auto accident. Subjects who were asked “How fast were the cars going when they contacted each other?” reported an average of 32 mph, while subjects who were asked the question with the word “smashed” instead of “contacted” reported 41mph for the same cars in the same movie.

In a first-order logic reasoning system that uses CNF, we can see that the linguistic form “ $\neg(A \vee B)$ ” and “ $\neg A \wedge \neg B$ ” are the same because we can look inside the system and see that the two sentences are stored as the same canonical CNF form. Can we do that with the human brain? Until recently the answer was “no,” but now it is “maybe.” Mitchell *et al.* (2008) put subjects in an fMRI (functional magnetic resonance imaging) machine, showed them words such as “celery,” and imaged their brains. The researchers were then able to train a computer program to predict, from a brain image, what word the subject had been presented with. Given two choices (e.g., “celery” or “airplane”), the system predicts correctly 77% of the time. The system can even predict at above-chance levels for words it has never seen an fMRI image of before (by considering the images of related words) and for people it has never seen before (proving that fMRI reveals some level of common representation across people). This type of work is still in its infancy, but fMRI (and other imaging technology such as intracranial electrophysiology (Sahin *et al.*, 2009)) promises to give us much more concrete ideas of what human knowledge representations are like.

From the viewpoint of formal logic, representing the same knowledge in two different ways makes absolutely no difference; the same facts will be derivable from either representation. In practice, however, one representation might require fewer steps to derive a conclusion, meaning that a reasoner with limited resources could get to the conclusion using one representation but not the other. For *nondeductive* tasks such as learning from experience, outcomes are *necessarily* dependent on the form of the representations used. We show in Chapter 18 that when a learning program considers two possible theories of the world, both of which are consistent with all the data, the most common way of breaking the tie is to choose the most succinct theory—and that depends on the language used to represent theories. Thus, the influence of language on thought is unavoidable for any agent that does learning.

8.1.2 Combining the best of formal and natural languages

We can adopt the foundation of propositional logic—a declarative, compositional semantics that is context-independent and unambiguous—and build a more expressive logic on that foundation, borrowing representational ideas from natural language while avoiding its drawbacks. When we look at the syntax of natural language, the most obvious elements are nouns and noun phrases that refer to **objects** (squares, pits, wumpuses) and verbs and verb phrases that refer to **relations** among objects (is breezy, is adjacent to, shoots). Some of these relations are **functions**—relations in which there is only one “value” for a given “input.” It is easy to start listing examples of objects, relations, and functions:

OBJECT
RELATION
FUNCTION

- Objects: people, houses, numbers, theories, Ronald McDonald, colors, baseball games, wars, centuries . . .
- Relations: these can be unary relations or **properties** such as red, round, bogus, prime, multistoried . . . , or more general *n*-ary relations such as brother of, bigger than, inside, part of, has color, occurred after, owns, comes between, . . .
- Functions: father of, best friend, third inning of, one more than, beginning of . . .

PROPERTY

Indeed, almost any assertion can be thought of as referring to objects and properties or relations. Some examples follow:

- “One plus two equals three.”
Objects: one, two, three, one plus two; Relation: equals; Function: plus. (“One plus two” is a name for the object that is obtained by applying the function “plus” to the objects “one” and “two.” “Three” is another name for this object.)
- “Squares neighboring the wumpus are smelly.”
Objects: wumpus, squares; Property: smelly; Relation: neighboring.
- “Evil King John ruled England in 1200.”
Objects: John, England, 1200; Relation: ruled; Properties: evil, king.

The language of **first-order logic**, whose syntax and semantics we define in the next section, is built around objects and relations. It has been so important to mathematics, philosophy, and artificial intelligence precisely because those fields—and indeed, much of everyday human existence—can be usefully thought of as dealing with objects and the relations among them. First-order logic can also express facts about *some* or *all* of the objects in the universe. This enables one to represent general laws or rules, such as the statement “Squares neighboring the wumpus are smelly.”

ONTOLOGICAL
COMMITMENT

The primary difference between propositional and first-order logic lies in the **ontological commitment** made by each language—that is, what it assumes about the nature of *reality*. Mathematically, this commitment is expressed through the nature of the formal **models** with respect to which the truth of sentences is defined. For example, propositional logic assumes that there are facts that either hold or do not hold in the world. Each fact can be in one of two states: true or false, and each model assigns *true* or *false* to each proposition symbol (see Section 7.4.2).² First-order logic assumes more; namely, that the world consists of objects with certain relations among them that do or do not hold. The formal models are correspondingly more complicated than those for propositional logic. Special-purpose logics make still further ontological commitments; for example, **temporal logic** assumes that facts hold at particular *times* and that those times (which may be points or intervals) are ordered. Thus, special-purpose logics give certain kinds of objects (and the axioms about them) “first class” status within the logic, rather than simply defining them within the knowledge base. **Higher-order logic** views the relations and functions referred to by first-order logic as objects in themselves. This allows one to make assertions about *all* relations—for example, one could wish to define what it means for a relation to be transitive. Unlike most special-purpose logics, higher-order logic is strictly more expressive than first-order logic, in the sense that some sentences of higher-order logic cannot be expressed by any finite number of first-order logic sentences.

TEMPORAL LOGIC

HIGHER-ORDER
LOGIC

EPISTEMOLOGICAL
COMMITMENT

A logic can also be characterized by its **epistemological commitments**—the possible states of knowledge that it allows with respect to each fact. In both propositional and first-order logic, a sentence represents a fact and the agent either believes the sentence to be true, believes it to be false, or has no opinion. These logics therefore have three possible states of knowledge regarding any sentence. Systems using **probability theory**, on the other hand,

² In contrast, facts in **fuzzy logic** have a **degree of truth** between 0 and 1. For example, the sentence “Vienna is a large city” might be true in our world only to degree 0.6 in fuzzy logic.

can have any *degree of belief*, ranging from 0 (total disbelief) to 1 (total belief).³ For example, a probabilistic wumpus-world agent might believe that the wumpus is in [1,3] with probability 0.75. The ontological and epistemological commitments of five different logics are summarized in Figure 8.1.

Language	Ontological Commitment (What exists in the world)	Epistemological Commitment (What an agent believes about facts)
Propositional logic	facts	true/false/unknown
First-order logic	facts, objects, relations	true/false/unknown
Temporal logic	facts, objects, relations, times	true/false/unknown
Probability theory	facts	degree of belief $\in [0, 1]$
Fuzzy logic	facts with degree of truth $\in [0, 1]$	known interval value

Figure 8.1 Formal languages and their ontological and epistemological commitments.

In the next section, we will launch into the details of first-order logic. Just as a student of physics requires some familiarity with mathematics, a student of AI must develop a talent for working with logical notation. On the other hand, it is also important *not* to get too concerned with the *specifics* of logical notation—after all, there are dozens of different versions. The main things to keep hold of are how the language facilitates concise representations and how its semantics leads to sound reasoning procedures.

8.2 SYNTAX AND SEMANTICS OF FIRST-ORDER LOGIC

We begin this section by specifying more precisely the way in which the possible worlds of first-order logic reflect the ontological commitment to objects and relations. Then we introduce the various elements of the language, explaining their semantics as we go along.

8.2.1 Models for first-order logic

Recall from Chapter 7 that the models of a logical language are the formal structures that constitute the possible worlds under consideration. Each model links the vocabulary of the logical sentences to elements of the possible world, so that the truth of any sentence can be determined. Thus, models for propositional logic link proposition symbols to predefined truth values. Models for first-order logic are much more interesting. First, they have objects in them! The **domain** of a model is the set of objects or **domain elements** it contains. The domain is required to be *nonempty*—every possible world must contain at least one object. (See Exercise 8.7 for a discussion of empty worlds.) Mathematically speaking, it doesn’t matter *what* these objects are—all that matters is *how many* there are in each particular model—but for pedagogical purposes we’ll use a concrete example. Figure 8.2 shows a model with five

DOMAIN
DOMAIN ELEMENTS

³ It is important not to confuse the degree of belief in probability theory with the degree of truth in fuzzy logic. Indeed, some fuzzy systems allow uncertainty (degree of belief) about degrees of truth.

objects: Richard the Lionheart, King of England from 1189 to 1199; his younger brother, the evil King John, who ruled from 1199 to 1215; the left legs of Richard and John; and a crown.

TUPLE

The objects in the model may be *related* in various ways. In the figure, Richard and John are brothers. Formally speaking, a relation is just the set of **tuples** of objects that are related. (A tuple is a collection of objects arranged in a fixed order and is written with angle brackets surrounding the objects.) Thus, the brotherhood relation in this model is the set

$$\{ \langle \text{Richard the Lionheart, King John} \rangle, \langle \text{King John, Richard the Lionheart} \rangle \} . \quad (8.1)$$

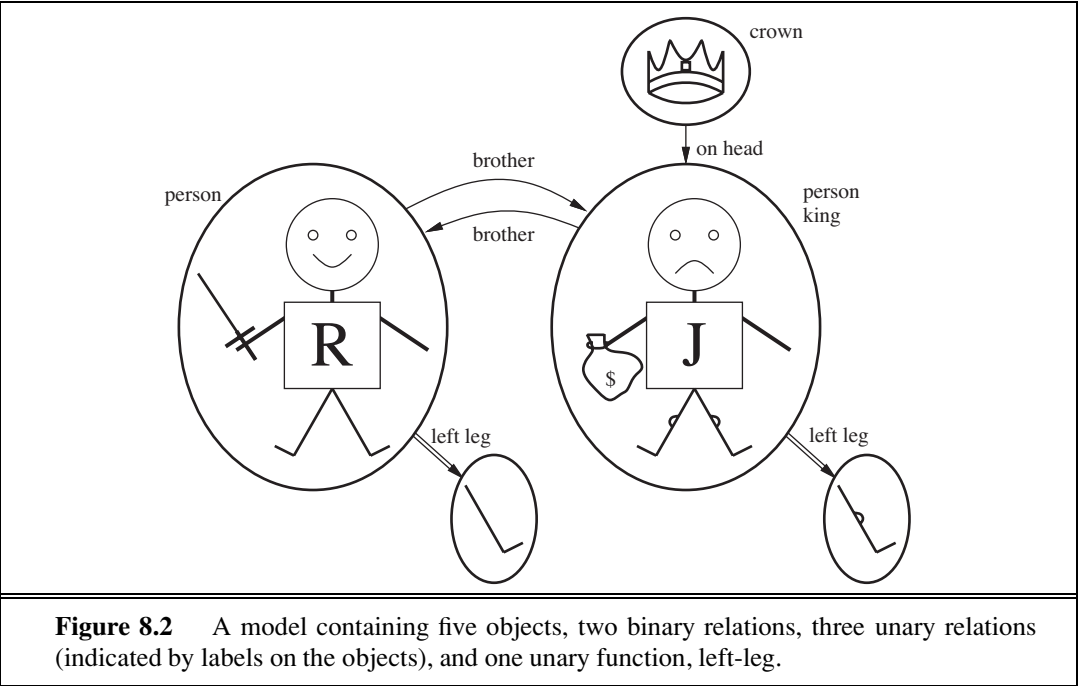
(Here we have named the objects in English, but you may, if you wish, mentally substitute the pictures for the names.) The crown is on King John’s head, so the “on head” relation contains just one tuple, $\langle \text{the crown, King John} \rangle$. The “brother” and “on head” relations are binary relations—that is, they relate pairs of objects. The model also contains unary relations, or properties: the “person” property is true of both Richard and John; the “king” property is true only of John (presumably because Richard is dead at this point); and the “crown” property is true only of the crown.

Certain kinds of relationships are best considered as functions, in that a given object must be related to exactly one object in this way. For example, each person has one left leg, so the model has a unary “left leg” function that includes the following mappings:

$$\begin{aligned} \langle \text{Richard the Lionheart} \rangle &\rightarrow \text{Richard’s left leg} \\ \langle \text{King John} \rangle &\rightarrow \text{John’s left leg} . \end{aligned} \quad (8.2)$$

TOTAL FUNCTIONS

Strictly speaking, models in first-order logic require **total functions**, that is, there must be a value for every input tuple. Thus, the crown must have a left leg and so must each of the left legs. There is a technical solution to this awkward problem involving an additional “invisible”



object that is the left leg of everything that has no left leg, including itself. Fortunately, as long as one makes no assertions about the left legs of things that have no left legs, these technicalities are of no import.

So far, we have described the elements that populate models for first-order logic. The other essential part of a model is the link between those elements and the vocabulary of the logical sentences, which we explain next.

8.2.2 Symbols and interpretations

We turn now to the syntax of first-order logic. The impatient reader can obtain a complete description from the formal grammar in Figure 8.3.

CONSTANT SYMBOL
PREDICATE SYMBOL
FUNCTION SYMBOL

The basic syntactic elements of first-order logic are the symbols that stand for objects, relations, and functions. The symbols, therefore, come in three kinds: **constant symbols**, which stand for objects; **predicate symbols**, which stand for relations; and **function symbols**, which stand for functions. We adopt the convention that these symbols will begin with uppercase letters. For example, we might use the constant symbols *Richard* and *John*; the predicate symbols *Brother*, *OnHead*, *Person*, *King*, and *Crown*; and the function symbol *LeftLeg*. As with proposition symbols, the choice of names is entirely up to the user. Each predicate and function symbol comes with an **arity** that fixes the number of arguments.

ARITY

INTERPRETATION

As in propositional logic, every model must provide the information required to determine if any given sentence is true or false. Thus, in addition to its objects, relations, and functions, each model includes an **interpretation** that specifies exactly which objects, relations and functions are referred to by the constant, predicate, and function symbols. One possible interpretation for our example—which a logician would call the **intended interpretation**—is as follows:

INTENDED
INTERPRETATION

- *Richard* refers to Richard the Lionheart and *John* refers to the evil King John.
- *Brother* refers to the brotherhood relation, that is, the set of tuples of objects given in Equation (8.1); *OnHead* refers to the “on head” relation that holds between the crown and King John; *Person*, *King*, and *Crown* refer to the sets of objects that are persons, kings, and crowns.
- *LeftLeg* refers to the “left leg” function, that is, the mapping given in Equation (8.2).

There are many other possible interpretations, of course. For example, one interpretation maps *Richard* to the crown and *John* to King John’s left leg. There are five objects in the model, so there are 25 possible interpretations just for the constant symbols *Richard* and *John*. Notice that not all the objects need have a name—for example, the intended interpretation does not name the crown or the legs. It is also possible for an object to have several names; there is an interpretation under which both *Richard* and *John* refer to the crown.⁴ If you find this possibility confusing, remember that, in propositional logic, it is perfectly possible to have a model in which *Cloudy* and *Sunny* are both true; it is the job of the knowledge base to rule out models that are inconsistent with our knowledge.

⁴ Later, in Section 8.2.8, we examine a semantics in which every object has exactly one name.

$$\begin{aligned}
 \text{Sentence} &\rightarrow \text{AtomicSentence} \mid \text{ComplexSentence} \\
 \text{AtomicSentence} &\rightarrow \text{Predicate} \mid \text{Predicate}(\text{Term}, \dots) \mid \text{Term} = \text{Term} \\
 \text{ComplexSentence} &\rightarrow (\text{Sentence}) \mid [\text{Sentence}] \\
 &\mid \neg \text{Sentence} \\
 &\mid \text{Sentence} \wedge \text{Sentence} \\
 &\mid \text{Sentence} \vee \text{Sentence} \\
 &\mid \text{Sentence} \Rightarrow \text{Sentence} \\
 &\mid \text{Sentence} \Leftrightarrow \text{Sentence} \\
 &\mid \text{Quantifier Variable}, \dots \text{Sentence} \\
 \\
 \text{Term} &\rightarrow \text{Function}(\text{Term}, \dots) \\
 &\mid \text{Constant} \\
 &\mid \text{Variable} \\
 \\
 \text{Quantifier} &\rightarrow \forall \mid \exists \\
 \text{Constant} &\rightarrow A \mid X_1 \mid \text{John} \mid \dots \\
 \text{Variable} &\rightarrow a \mid x \mid s \mid \dots \\
 \text{Predicate} &\rightarrow \text{True} \mid \text{False} \mid \text{After} \mid \text{Loves} \mid \text{Raining} \mid \dots \\
 \text{Function} &\rightarrow \text{Mother} \mid \text{LeftLeg} \mid \dots
 \end{aligned}$$

OPERATOR PRECEDENCE : $\neg, =, \wedge, \vee, \Rightarrow, \Leftrightarrow$

Figure 8.3 The syntax of first-order logic with equality, specified in Backus–Naur form (see page 1060 if you are not familiar with this notation). Operator precedences are specified, from highest to lowest. The precedence of quantifiers is such that a quantifier holds over everything to the right of it.

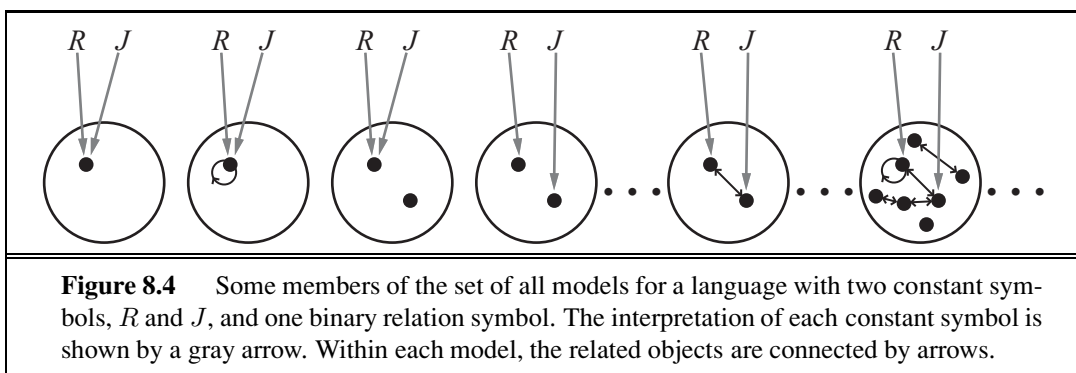


Figure 8.4 Some members of the set of all models for a language with two constant symbols, *R* and *J*, and one binary relation symbol. The interpretation of each constant symbol is shown by a gray arrow. Within each model, the related objects are connected by arrows.

In summary, a model in first-order logic consists of a set of objects and an interpretation that maps constant symbols to objects, predicate symbols to relations on those objects, and function symbols to functions on those objects. Just as with propositional logic, entailment, validity, and so on are defined in terms of *all possible models*. To get an idea of what the set of all possible models looks like, see Figure 8.4. It shows that models vary in how many objects they contain—from one up to infinity—and in the way the constant symbols map to objects. If there are two constant symbols and one object, then both symbols must refer to the same object; but this can still happen even with more objects. When there are more objects than constant symbols, some of the objects will have no names. Because the number of possible models is unbounded, checking entailment by the enumeration of all possible models is not feasible for first-order logic (unlike propositional logic). Even if the number of objects is restricted, the number of combinations can be very large. (See Exercise 8.5.) For the example in Figure 8.4, there are 137,506,194,466 models with six or fewer objects.

8.2.3 Terms

TERM

A **term** is a logical expression that refers to an object. Constant symbols are therefore terms, but it is not always convenient to have a distinct symbol to name every object. For example, in English we might use the expression “King John’s left leg” rather than giving a name to his leg. This is what function symbols are for: instead of using a constant symbol, we use *LeftLeg(John)*. In the general case, a complex term is formed by a function symbol followed by a parenthesized list of terms as arguments to the function symbol. It is important to remember that a complex term is just a complicated kind of name. It is not a “subroutine call” that “returns a value.” There is no *LeftLeg* subroutine that takes a person as input and returns a leg. We can reason about left legs (e.g., stating the general rule that everyone has one and then deducing that John must have one) without ever providing a definition of *LeftLeg*. This is something that cannot be done with subroutines in programming languages.⁵

The formal semantics of terms is straightforward. Consider a term $f(t_1, \dots, t_n)$. The function symbol f refers to some function in the model (call it F); the argument terms refer to objects in the domain (call them d_1, \dots, d_n); and the term as a whole refers to the object that is the value of the function F applied to d_1, \dots, d_n . For example, suppose the *LeftLeg* function symbol refers to the function shown in Equation (8.2) and *John* refers to King John, then *LeftLeg(John)* refers to King John’s left leg. In this way, the interpretation fixes the referent of every term.

8.2.4 Atomic sentences

Now that we have both terms for referring to objects and predicate symbols for referring to relations, we can put them together to make **atomic sentences** that state facts. An **atomic**

⁵ **λ -expressions** provide a useful notation in which new function symbols are constructed “on the fly.” For example, the function that squares its argument can be written as $(\lambda x \ x \times x)$ and can be applied to arguments just like any other function symbol. A λ -expression can also be defined and used as a predicate symbol. (See Chapter 22.) The lambda operator in Lisp plays exactly the same role. Notice that the use of λ in this way does *not* increase the formal expressive power of first-order logic, because any sentence that includes a λ -expression can be rewritten by “plugging in” its arguments to yield an equivalent sentence.

ATOMIC SENTENCE

ATOM

sentence (or **atom** for short) is formed from a predicate symbol optionally followed by a parenthesized list of terms, such as

Brother(Richard, John).

This states, under the intended interpretation given earlier, that Richard the Lionheart is the brother of King John.⁶ Atomic sentences can have complex terms as arguments. Thus,

Married(Father(Richard), Mother(John))

states that Richard the Lionheart's father is married to King John's mother (again, under a suitable interpretation).



*An atomic sentence is **true** in a given model if the relation referred to by the predicate symbol holds among the objects referred to by the arguments.*

8.2.5 Complex sentences

We can use **logical connectives** to construct more complex sentences, with the same syntax and semantics as in propositional calculus. Here are four sentences that are true in the model of Figure 8.2 under our intended interpretation:

$\neg \text{Brother}(\text{LeftLeg}(\text{Richard}), \text{John})$
 $\text{Brother}(\text{Richard}, \text{John}) \wedge \text{Brother}(\text{John}, \text{Richard})$
 $\text{King}(\text{Richard}) \vee \text{King}(\text{John})$
 $\neg \text{King}(\text{Richard}) \Rightarrow \text{King}(\text{John}) .$

8.2.6 Quantifiers

QUANTIFIER

Once we have a logic that allows objects, it is only natural to want to express properties of entire collections of objects, instead of enumerating the objects by name. **Quantifiers** let us do this. First-order logic contains two standard quantifiers, called *universal* and *existential*.

Universal quantification (\forall)

Recall the difficulty we had in Chapter 7 with the expression of general rules in propositional logic. Rules such as “Squares neighboring the wumpus are smelly” and “All kings are persons” are the bread and butter of first-order logic. We deal with the first of these in Section 8.3. The second rule, “All kings are persons,” is written in first-order logic as

$\forall x \text{ King}(x) \Rightarrow \text{Person}(x) .$

\forall is usually pronounced “For all . . .”. (Remember that the upside-down A stands for “all.”) Thus, the sentence says, “For all x , if x is a king, then x is a person.” The symbol x is called a **variable**. By convention, variables are lowercase letters. A variable is a term all by itself, and as such can also serve as the argument of a function—for example, *LeftLeg*(x). A term with no variables is called a **ground term**.

VARIABLE

GROUND TERM

Intuitively, the sentence $\forall x P$, where P is any logical expression, says that P is true for every object x . More precisely, $\forall x P$ is true in a given model if P is true in all possible **extended interpretations** constructed from the interpretation given in the model, where each

EXTENDED
INTERPRETATION

⁶ We usually follow the argument-ordering convention that $P(x, y)$ is read as “ x is a P of y .”

extended interpretation specifies a domain element to which x refers.

This sounds complicated, but it is really just a careful way of stating the intuitive meaning of universal quantification. Consider the model shown in Figure 8.2 and the intended interpretation that goes with it. We can extend the interpretation in five ways:

$x \rightarrow$ Richard the Lionheart,
 $x \rightarrow$ King John,
 $x \rightarrow$ Richard's left leg,
 $x \rightarrow$ John's left leg,
 $x \rightarrow$ the crown.

The universally quantified sentence $\forall x \text{ King}(x) \Rightarrow \text{Person}(x)$ is true in the original model if the sentence $\text{King}(x) \Rightarrow \text{Person}(x)$ is true under each of the five extended interpretations. That is, the universally quantified sentence is equivalent to asserting the following five sentences:

Richard the Lionheart is a king \Rightarrow Richard the Lionheart is a person.
 King John is a king \Rightarrow King John is a person.
 Richard's left leg is a king \Rightarrow Richard's left leg is a person.
 John's left leg is a king \Rightarrow John's left leg is a person.
 The crown is a king \Rightarrow the crown is a person.

Let us look carefully at this set of assertions. Since, in our model, King John is the only king, the second sentence asserts that he is a person, as we would hope. But what about the other four sentences, which appear to make claims about legs and crowns? Is that part of the meaning of “All kings are persons”? In fact, the other four assertions are true in the model, but make no claim whatsoever about the personhood qualifications of legs, crowns, or indeed Richard. This is because none of these objects is a king. Looking at the truth table for \Rightarrow (Figure 7.8 on page 246), we see that the implication is true whenever its premise is false—*regardless* of the truth of the conclusion. Thus, by asserting the universally quantified sentence, which is equivalent to asserting a whole list of individual implications, we end up asserting the conclusion of the rule just for those objects for whom the premise is true and saying nothing at all about those individuals for whom the premise is false. Thus, the truth-table definition of \Rightarrow turns out to be perfect for writing general rules with universal quantifiers.

A common mistake, made frequently even by diligent readers who have read this paragraph several times, is to use conjunction instead of implication. The sentence

$$\forall x \text{ King}(x) \wedge \text{Person}(x)$$

would be equivalent to asserting

Richard the Lionheart is a king \wedge Richard the Lionheart is a person,
 King John is a king \wedge King John is a person,
 Richard's left leg is a king \wedge Richard's left leg is a person,

and so on. Obviously, this does not capture what we want.

Existential quantification (\exists)

Universal quantification makes statements about every object. Similarly, we can make a statement about *some* object in the universe without naming it, by using an existential quantifier. To say, for example, that King John has a crown on his head, we write

$$\exists x \text{ Crown}(x) \wedge \text{OnHead}(x, \text{John}) .$$

$\exists x$ is pronounced “There exists an x such that . . .” or “For some x . . .”.

Intuitively, the sentence $\exists x P$ says that P is true for at least one object x . More precisely, $\exists x P$ is true in a given model if P is true in *at least one* extended interpretation that assigns x to a domain element. That is, at least one of the following is true:

Richard the Lionheart is a crown \wedge Richard the Lionheart is on John’s head;
 King John is a crown \wedge King John is on John’s head;
 Richard’s left leg is a crown \wedge Richard’s left leg is on John’s head;
 John’s left leg is a crown \wedge John’s left leg is on John’s head;
 The crown is a crown \wedge the crown is on John’s head.

The fifth assertion is true in the model, so the original existentially quantified sentence is true in the model. Notice that, by our definition, the sentence would also be true in a model in which King John was wearing two crowns. This is entirely consistent with the original sentence “King John has a crown on his head.”⁷

Just as \Rightarrow appears to be the natural connective to use with \forall , \wedge is the natural connective to use with \exists . Using \wedge as the main connective with \forall led to an overly strong statement in the example in the previous section; using \Rightarrow with \exists usually leads to a very weak statement, indeed. Consider the following sentence:

$$\exists x \text{ Crown}(x) \Rightarrow \text{OnHead}(x, \text{John}) .$$

On the surface, this might look like a reasonable rendition of our sentence. Applying the semantics, we see that the sentence says that at least one of the following assertions is true:

Richard the Lionheart is a crown \Rightarrow Richard the Lionheart is on John’s head;
 King John is a crown \Rightarrow King John is on John’s head;
 Richard’s left leg is a crown \Rightarrow Richard’s left leg is on John’s head;

and so on. Now an implication is true if both premise and conclusion are true, *or if its premise is false*. So if Richard the Lionheart is not a crown, then the first assertion is true and the existential is satisfied. So, an existentially quantified implication sentence is true whenever *any* object fails to satisfy the premise; hence such sentences really do not say much at all.

Nested quantifiers

We will often want to express more complex sentences using multiple quantifiers. The simplest case is where the quantifiers are of the same type. For example, “Brothers are siblings” can be written as

$$\forall x \forall y \text{ Brother}(x, y) \Rightarrow \text{Sibling}(x, y) .$$

⁷ There is a variant of the existential quantifier, usually written \exists^1 or $\exists!$, that means “There exists exactly one.” The same meaning can be expressed using equality statements.

Consecutive quantifiers of the same type can be written as one quantifier with several variables. For example, to say that siblinghood is a symmetric relationship, we can write

$$\forall x, y \text{ Sibling}(x, y) \Leftrightarrow \text{Sibling}(y, x) .$$

In other cases we will have mixtures. “Everybody loves somebody” means that for every person, there is someone that person loves:

$$\forall x \exists y \text{ Loves}(x, y) .$$

On the other hand, to say “There is someone who is loved by everyone,” we write

$$\exists y \forall x \text{ Loves}(x, y) .$$

The order of quantification is therefore very important. It becomes clearer if we insert parentheses. $\forall x (\exists y \text{ Loves}(x, y))$ says that *everyone* has a particular property, namely, the property that they love someone. On the other hand, $\exists y (\forall x \text{ Loves}(x, y))$ says that *someone* in the world has a particular property, namely the property of being loved by everybody.

Some confusion can arise when two quantifiers are used with the same variable name. Consider the sentence

$$\forall x (\text{Crown}(x) \vee (\exists x \text{ Brother}(\text{Richard}, x))) .$$

Here the x in $\text{Brother}(\text{Richard}, x)$ is *existentially* quantified. The rule is that the variable belongs to the innermost quantifier that mentions it; then it will not be subject to any other quantification. Another way to think of it is this: $\exists x \text{ Brother}(\text{Richard}, x)$ is a sentence about Richard (that he has a brother), not about x ; so putting a $\forall x$ outside it has no effect. It could equally well have been written $\exists z \text{ Brother}(\text{Richard}, z)$. Because this can be a source of confusion, we will always use different variable names with nested quantifiers.

Connections between \forall and \exists

The two quantifiers are actually intimately connected with each other, through negation. Asserting that everyone dislikes parsnips is the same as asserting there does not exist someone who likes them, and vice versa:

$$\forall x \neg \text{Likes}(x, \text{Parsnips}) \text{ is equivalent to } \neg \exists x \text{ Likes}(x, \text{Parsnips}) .$$

We can go one step further: “Everyone likes ice cream” means that there is no one who does not like ice cream:

$$\forall x \text{ Likes}(x, \text{IceCream}) \text{ is equivalent to } \neg \exists x \neg \text{Likes}(x, \text{IceCream}) .$$

Because \forall is really a conjunction over the universe of objects and \exists is a disjunction, it should not be surprising that they obey De Morgan’s rules. The De Morgan rules for quantified and unquantified sentences are as follows:

$$\begin{array}{ll} \forall x \neg P & \equiv \neg \exists x P & \neg(P \vee Q) & \equiv \neg P \wedge \neg Q \\ \neg \forall x P & \equiv \exists x \neg P & \neg(P \wedge Q) & \equiv \neg P \vee \neg Q \\ \forall x P & \equiv \neg \exists x \neg P & P \wedge Q & \equiv \neg(\neg P \vee \neg Q) \\ \exists x P & \equiv \neg \forall x \neg P & P \vee Q & \equiv \neg(\neg P \wedge \neg Q) . \end{array}$$

Thus, we do not really need both \forall and \exists , just as we do not really need both \wedge and \vee . Still, readability is more important than parsimony, so we will keep both of the quantifiers.

8.2.7 Equality

EQUALITY SYMBOL

First-order logic includes one more way to make atomic sentences, other than using a predicate and terms as described earlier. We can use the **equality symbol** to signify that two terms refer to the same object. For example,

$$\text{Father}(\text{John}) = \text{Henry}$$

says that the object referred to by $\text{Father}(\text{John})$ and the object referred to by Henry are the same. Because an interpretation fixes the referent of any term, determining the truth of an equality sentence is simply a matter of seeing that the referents of the two terms are the same object.

The equality symbol can be used to state facts about a given function, as we just did for the Father symbol. It can also be used with negation to insist that two terms are not the same object. To say that Richard has at least two brothers, we would write

$$\exists x, y \text{ Brother}(x, \text{Richard}) \wedge \text{Brother}(y, \text{Richard}) \wedge \neg(x = y) .$$

The sentence

$$\exists x, y \text{ Brother}(x, \text{Richard}) \wedge \text{Brother}(y, \text{Richard})$$

does not have the intended meaning. In particular, it is true in the model of Figure 8.2, where Richard has only one brother. To see this, consider the extended interpretation in which both x and y are assigned to King John. The addition of $\neg(x = y)$ rules out such models. The notation $x \neq y$ is sometimes used as an abbreviation for $\neg(x = y)$.

8.2.8 An alternative semantics?

Continuing the example from the previous section, suppose that we believe that Richard has two brothers, John and Geoffrey.⁸ Can we capture this state of affairs by asserting

$$\text{Brother}(\text{John}, \text{Richard}) \wedge \text{Brother}(\text{Geoffrey}, \text{Richard}) ? \quad (8.3)$$

Not quite. First, this assertion is true in a model where Richard has only one brother—we need to add $\text{John} \neq \text{Geoffrey}$. Second, the sentence doesn't rule out models in which Richard has many more brothers besides John and Geoffrey. Thus, the correct translation of "Richard's brothers are John and Geoffrey" is as follows:

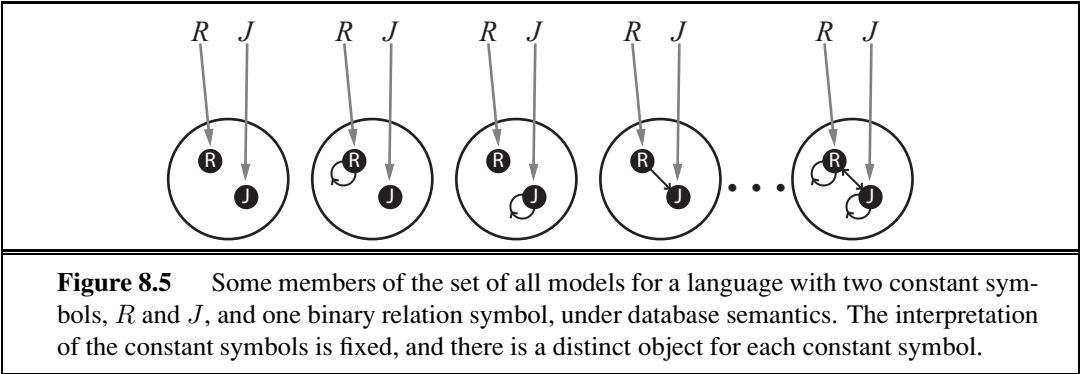
$$\begin{aligned} &\text{Brother}(\text{John}, \text{Richard}) \wedge \text{Brother}(\text{Geoffrey}, \text{Richard}) \wedge \text{John} \neq \text{Geoffrey} \\ &\wedge \forall x \text{ Brother}(x, \text{Richard}) \Rightarrow (x = \text{John} \vee x = \text{Geoffrey}) . \end{aligned}$$

For many purposes, this seems much more cumbersome than the corresponding natural-language expression. As a consequence, humans may make mistakes in translating their knowledge into first-order logic, resulting in unintuitive behaviors from logical reasoning systems that use the knowledge. Can we devise a semantics that allows a more straightforward logical expression?

One proposal that is very popular in database systems works as follows. First, we insist that every constant symbol refer to a distinct object—the so-called **unique-names assumption**. Second, we assume that atomic sentences not known to be true are in fact false—the **closed-world assumption**. Finally, we invoke **domain closure**, meaning that each model

UNIQUE-NAMES
ASSUMPTION
CLOSED-WORLD
ASSUMPTION
DOMAIN CLOSURE

⁸ Actually he had four, the others being William and Henry.



DATABASE
SEMANTICS

contains no more domain elements than those named by the constant symbols. Under the resulting semantics, which we call **database semantics** to distinguish it from the standard semantics of first-order logic, the sentence Equation (8.3) does indeed state that Richard’s two brothers are John and Geoffrey. Database semantics is also used in logic programming systems, as explained in Section 9.4.5.

It is instructive to consider the set of all possible models under database semantics for the same case as shown in Figure 8.4. Figure 8.5 shows some of the models, ranging from the model with no tuples satisfying the relation to the model with all tuples satisfying the relation. With two objects, there are four possible two-element tuples, so there are $2^4 = 16$ different subsets of tuples that can satisfy the relation. Thus, there are 16 possible models in all—a lot fewer than the infinitely many models for the standard first-order semantics. On the other hand, the database semantics requires definite knowledge of what the world contains.

This example brings up an important point: there is no one “correct” semantics for logic. The usefulness of any proposed semantics depends on how concise and intuitive it makes the expression of the kinds of knowledge we want to write down, and on how easy and natural it is to develop the corresponding rules of inference. Database semantics is most useful when we are certain about the identity of all the objects described in the knowledge base and when we have all the facts at hand; in other cases, it is quite awkward. For the rest of this chapter, we assume the standard semantics while noting instances in which this choice leads to cumbersome expressions.

8.3 USING FIRST-ORDER LOGIC

DOMAIN

Now that we have defined an expressive logical language, it is time to learn how to use it. The best way to do this is through examples. We have seen some simple sentences illustrating the various aspects of logical syntax; in this section, we provide more systematic representations of some simple **domains**. In knowledge representation, a domain is just some part of the world about which we wish to express some knowledge.

We begin with a brief description of the TELL/ASK interface for first-order knowledge bases. Then we look at the domains of family relationships, numbers, sets, and lists, and at

the wumpus world. The next section contains a more substantial example (electronic circuits) and Chapter 12 covers everything in the universe.

8.3.1 Assertions and queries in first-order logic

ASSERTION

Sentences are added to a knowledge base using TELL, exactly as in propositional logic. Such sentences are called **assertions**. For example, we can assert that John is a king, Richard is a person, and all kings are persons:

$$\begin{aligned} &\text{TELL}(KB, \text{King}(\text{John})) . \\ &\text{TELL}(KB, \text{Person}(\text{Richard})) . \\ &\text{TELL}(KB, \forall x \text{ King}(x) \Rightarrow \text{Person}(x)) . \end{aligned}$$

We can ask questions of the knowledge base using ASK. For example,

$$\text{ASK}(KB, \text{King}(\text{John}))$$

QUERY

returns *true*. Questions asked with ASK are called **queries** or **goals**. Generally speaking, any query that is logically entailed by the knowledge base should be answered affirmatively. For example, given the two preceding assertions, the query

GOAL

$$\text{ASK}(KB, \text{Person}(\text{John}))$$

should also return *true*. We can ask quantified queries, such as

$$\text{ASK}(KB, \exists x \text{ Person}(x)) .$$

The answer is *true*, but this is perhaps not as helpful as we would like. It is rather like answering “Can you tell me the time?” with “Yes.” If we want to know what value of x makes the sentence true, we will need a different function, ASK VARS, which we call with

$$\text{ASK VARS}(KB, \text{Person}(x))$$

SUBSTITUTION

BINDING LIST

and which yields a stream of answers. In this case there will be two answers: $\{x/\text{John}\}$ and $\{x/\text{Richard}\}$. Such an answer is called a **substitution** or **binding list**. ASK VARS is usually reserved for knowledge bases consisting solely of Horn clauses, because in such knowledge bases every way of making the query true will bind the variables to specific values. That is not the case with first-order logic; if KB has been told $\text{King}(\text{John}) \vee \text{King}(\text{Richard})$, then there is no binding to x for the query $\exists x \text{ King}(x)$, even though the query is true.

8.3.2 The kinship domain

The first example we consider is the domain of family relationships, or kinship. This domain includes facts such as “Elizabeth is the mother of Charles” and “Charles is the father of William” and rules such as “One’s grandmother is the mother of one’s parent.”

Clearly, the objects in our domain are people. We have two unary predicates, *Male* and *Female*. Kinship relations—parenthood, brotherhood, marriage, and so on—are represented by binary predicates: *Parent*, *Sibling*, *Brother*, *Sister*, *Child*, *Daughter*, *Son*, *Spouse*, *Wife*, *Husband*, *Grandparent*, *Grandchild*, *Cousin*, *Aunt*, and *Uncle*. We use functions for *Mother* and *Father*, because every person has exactly one of each of these (at least according to nature’s design).

We can go through each function and predicate, writing down what we know in terms of the other symbols. For example, one's mother is one's female parent:

$$\forall m, c \text{ } Mother(c) = m \Leftrightarrow Female(m) \wedge Parent(m, c) .$$

One's husband is one's male spouse:

$$\forall w, h \text{ } Husband(h, w) \Leftrightarrow Male(h) \wedge Spouse(h, w) .$$

Male and female are disjoint categories:

$$\forall x \text{ } Male(x) \Leftrightarrow \neg Female(x) .$$

Parent and child are inverse relations:

$$\forall p, c \text{ } Parent(p, c) \Leftrightarrow Child(c, p) .$$

A grandparent is a parent of one's parent:

$$\forall g, c \text{ } Grandparent(g, c) \Leftrightarrow \exists p \text{ } Parent(g, p) \wedge Parent(p, c) .$$

A sibling is another child of one's parents:

$$\forall x, y \text{ } Sibling(x, y) \Leftrightarrow x \neq y \wedge \exists p \text{ } Parent(p, x) \wedge Parent(p, y) .$$

We could go on for several more pages like this, and Exercise 8.14 asks you to do just that.

Each of these sentences can be viewed as an **axiom** of the kinship domain, as explained in Section 7.1. Axioms are commonly associated with purely mathematical domains—we will see some axioms for numbers shortly—but they are needed in all domains. They provide the basic factual information from which useful conclusions can be derived. Our kinship axioms are also **definitions**; they have the form $\forall x, y \text{ } P(x, y) \Leftrightarrow \dots$. The axioms define the *Mother* function and the *Husband*, *Male*, *Parent*, *Grandparent*, and *Sibling* predicates in terms of other predicates. Our definitions “bottom out” at a basic set of predicates (*Child*, *Spouse*, and *Female*) in terms of which the others are ultimately defined. This is a natural way in which to build up the representation of a domain, and it is analogous to the way in which software packages are built up by successive definitions of subroutines from primitive library functions. Notice that there is not necessarily a unique set of primitive predicates; we could equally well have used *Parent*, *Spouse*, and *Male*. In some domains, as we show, there is no clearly identifiable basic set.

DEFINITION

THEOREM

Not all logical sentences about a domain are axioms. Some are **theorems**—that is, they are entailed by the axioms. For example, consider the assertion that siblinghood is symmetric:

$$\forall x, y \text{ } Sibling(x, y) \Leftrightarrow Sibling(y, x) .$$

Is this an axiom or a theorem? In fact, it is a theorem that follows logically from the axiom that defines siblinghood. If we ASK the knowledge base this sentence, it should return *true*.

From a purely logical point of view, a knowledge base need contain only axioms and no theorems, because the theorems do not increase the set of conclusions that follow from the knowledge base. From a practical point of view, theorems are essential to reduce the computational cost of deriving new sentences. Without them, a reasoning system has to start from first principles every time, rather like a physicist having to rederive the rules of calculus for every new problem.

Not all axioms are definitions. Some provide more general information about certain predicates without constituting a definition. Indeed, some predicates have no complete definition because we do not know enough to characterize them fully. For example, there is no obvious definitive way to complete the sentence

$$\forall x \text{ Person}(x) \Leftrightarrow \dots$$

Fortunately, first-order logic allows us to make use of the *Person* predicate without completely defining it. Instead, we can write partial specifications of properties that every person has and properties that make something a person:

$$\begin{aligned} \forall x \text{ Person}(x) &\Rightarrow \dots \\ \forall x \dots &\Rightarrow \text{Person}(x). \end{aligned}$$

Axioms can also be “just plain facts,” such as *Male(Jim)* and *Spouse(Jim, Laura)*. Such facts form the descriptions of specific problem instances, enabling specific questions to be answered. The answers to these questions will then be theorems that follow from the axioms. Often, one finds that the expected answers are not forthcoming—for example, from *Spouse(Jim, Laura)* one expects (under the laws of many countries) to be able to infer $\neg \text{Spouse}(\text{George}, \text{Laura})$; but this does not follow from the axioms given earlier—even after we add $\text{Jim} \neq \text{George}$ as suggested in Section 8.2.8. This is a sign that an axiom is missing. Exercise 8.8 asks the reader to supply it.

8.3.3 Numbers, sets, and lists

NATURAL NUMBERS

PEANO AXIOMS

Numbers are perhaps the most vivid example of how a large theory can be built up from a tiny kernel of axioms. We describe here the theory of **natural numbers** or non-negative integers. We need a predicate *NatNum* that will be true of natural numbers; we need one constant symbol, 0; and we need one function symbol, *S* (successor). The **Peano axioms** define natural numbers and addition.⁹ Natural numbers are defined recursively:

$$\begin{aligned} \text{NatNum}(0) &. \\ \forall n \text{ NatNum}(n) &\Rightarrow \text{NatNum}(S(n)). \end{aligned}$$

That is, 0 is a natural number, and for every object *n*, if *n* is a natural number, then *S*(*n*) is a natural number. So the natural numbers are 0, *S*(0), *S*(*S*(0)), and so on. (After reading Section 8.2.8, you will notice that these axioms allow for other natural numbers besides the usual ones; see Exercise 8.12.) We also need axioms to constrain the successor function:

$$\begin{aligned} \forall n \ 0 &\neq S(n). \\ \forall m, n \ m &\neq n \Rightarrow S(m) \neq S(n). \end{aligned}$$

Now we can define addition in terms of the successor function:

$$\begin{aligned} \forall m \text{ NatNum}(m) &\Rightarrow +(0, m) = m. \\ \forall m, n \text{ NatNum}(m) \wedge \text{NatNum}(n) &\Rightarrow +(S(m), n) = S(+(m, n)). \end{aligned}$$

The first of these axioms says that adding 0 to any natural number *m* gives *m* itself. Notice the use of the binary function symbol “+” in the term $+(m, 0)$; in ordinary mathematics, the term would be written $m + 0$ using **infix** notation. (The notation we have used for first-order

INFIX

⁹ The Peano axioms also include the principle of induction, which is a sentence of second-order logic rather than of first-order logic. The importance of this distinction is explained in Chapter 9.

PREFIX

logic is called **prefix**.) To make our sentences about numbers easier to read, we allow the use of infix notation. We can also write $S(n)$ as $n + 1$, so the second axiom becomes

$$\forall m, n \text{ NatNum}(m) \wedge \text{NatNum}(n) \Rightarrow (m + 1) + n = (m + n) + 1 .$$

This axiom reduces addition to repeated application of the successor function.

SYNTACTIC SUGAR

The use of infix notation is an example of **syntactic sugar**, that is, an extension to or abbreviation of the standard syntax that does not change the semantics. Any sentence that uses sugar can be “desugared” to produce an equivalent sentence in ordinary first-order logic.

Once we have addition, it is straightforward to define multiplication as repeated addition, exponentiation as repeated multiplication, integer division and remainders, prime numbers, and so on. Thus, the whole of number theory (including cryptography) can be built up from one constant, one function, one predicate and four axioms.

SET

The domain of **sets** is also fundamental to mathematics as well as to commonsense reasoning. (In fact, it is possible to define number theory in terms of set theory.) We want to be able to represent individual sets, including the empty set. We need a way to build up sets by adding an element to a set or taking the union or intersection of two sets. We will want to know whether an element is a member of a set and we will want to distinguish sets from objects that are not sets.

We will use the normal vocabulary of set theory as syntactic sugar. The empty set is a constant written as $\{ \}$. There is one unary predicate, Set , which is true of sets. The binary predicates are $x \in s$ (x is a member of set s) and $s_1 \subseteq s_2$ (set s_1 is a subset, not necessarily proper, of set s_2). The binary functions are $s_1 \cap s_2$ (the intersection of two sets), $s_1 \cup s_2$ (the union of two sets), and $\{x|s\}$ (the set resulting from adjoining element x to set s). One possible set of axioms is as follows:

1. The only sets are the empty set and those made by adjoining something to a set:

$$\forall s \text{ Set}(s) \Leftrightarrow (s = \{ \}) \vee (\exists x, s_2 \text{ Set}(s_2) \wedge s = \{x|s_2\}) .$$

2. The empty set has no elements adjoined into it. In other words, there is no way to decompose $\{ \}$ into a smaller set and an element:

$$\neg \exists x, s \{x|s\} = \{ \} .$$

3. Adjoining an element already in the set has no effect:

$$\forall x, s \ x \in s \Leftrightarrow s = \{x|s\} .$$

4. The only members of a set are the elements that were adjoined into it. We express this recursively, saying that x is a member of s if and only if s is equal to some set s_2 adjoined with some element y , where either y is the same as x or x is a member of s_2 :

$$\forall x, s \ x \in s \Leftrightarrow \exists y, s_2 (s = \{y|s_2\} \wedge (x = y \vee x \in s_2)) .$$

5. A set is a subset of another set if and only if all of the first set's members are members of the second set:

$$\forall s_1, s_2 \ s_1 \subseteq s_2 \Leftrightarrow (\forall x \ x \in s_1 \Rightarrow x \in s_2) .$$

6. Two sets are equal if and only if each is a subset of the other:

$$\forall s_1, s_2 \ (s_1 = s_2) \Leftrightarrow (s_1 \subseteq s_2 \wedge s_2 \subseteq s_1) .$$

7. An object is in the intersection of two sets if and only if it is a member of both sets:

$$\forall x, s_1, s_2 \quad x \in (s_1 \cap s_2) \Leftrightarrow (x \in s_1 \wedge x \in s_2) .$$

8. An object is in the union of two sets if and only if it is a member of either set:

$$\forall x, s_1, s_2 \quad x \in (s_1 \cup s_2) \Leftrightarrow (x \in s_1 \vee x \in s_2) .$$

LIST

Lists are similar to sets. The differences are that lists are ordered and the same element can appear more than once in a list. We can use the vocabulary of Lisp for lists: *Nil* is the constant list with no elements; *Cons*, *Append*, *First*, and *Rest* are functions; and *Find* is the predicate that does for lists what *Member* does for sets. *List?* is a predicate that is true only of lists. As with sets, it is common to use syntactic sugar in logical sentences involving lists. The empty list is $[]$. The term $Cons(x, y)$, where y is a nonempty list, is written $[x|y]$. The term $Cons(x, Nil)$ (i.e., the list containing the element x) is written as $[x]$. A list of several elements, such as $[A, B, C]$, corresponds to the nested term $Cons(A, Cons(B, Cons(C, Nil)))$. Exercise 8.16 asks you to write out the axioms for lists.

8.3.4 The wumpus world

Some propositional logic axioms for the wumpus world were given in Chapter 7. The first-order axioms in this section are much more concise, capturing in a natural way exactly what we want to say.

Recall that the wumpus agent receives a percept vector with five elements. The corresponding first-order sentence stored in the knowledge base must include both the percept and the time at which it occurred; otherwise, the agent will get confused about when it saw what. We use integers for time steps. A typical percept sentence would be

$$Percept([Stench, Breeze, Glitter, None, None], 5) .$$

Here, *Percept* is a binary predicate, and *Stench* and so on are constants placed in a list. The actions in the wumpus world can be represented by logical terms:

$$Turn(Right), Turn(Left), Forward, Shoot, Grab, Climb .$$

To determine which is best, the agent program executes the query

$$ASKVARS(\exists a \text{ BestAction}(a, 5)) ,$$

which returns a binding list such as $\{a/Grab\}$. The agent program can then return *Grab* as the action to take. The raw percept data implies certain facts about the current state. For example:

$$\begin{aligned} \forall t, s, g, m, c \quad Percept([s, Breeze, g, m, c], t) &\Rightarrow Breeze(t) , \\ \forall t, s, b, m, c \quad Percept([s, b, Glitter, m, c], t) &\Rightarrow Glitter(t) , \end{aligned}$$

and so on. These rules exhibit a trivial form of the reasoning process called **perception**, which we study in depth in Chapter 24. Notice the quantification over time t . In propositional logic, we would need copies of each sentence for each time step.

Simple “reflex” behavior can also be implemented by quantified implication sentences. For example, we have

$$\forall t \quad Glitter(t) \Rightarrow BestAction(Grab, t) .$$

Given the percept and rules from the preceding paragraphs, this would yield the desired conclusion $BestAction(Grab, 5)$ —that is, *Grab* is the right thing to do.

We have represented the agent's inputs and outputs; now it is time to represent the environment itself. Let us begin with objects. Obvious candidates are squares, pits, and the wumpus. We could name each square— $Square_{1,2}$ and so on—but then the fact that $Square_{1,2}$ and $Square_{1,3}$ are adjacent would have to be an “extra” fact, and we would need one such fact for each pair of squares. It is better to use a complex term in which the row and column appear as integers; for example, we can simply use the list term $[1, 2]$. Adjacency of any two squares can be defined as

$$\forall x, y, a, b \text{ } Adjacent([x, y], [a, b]) \Leftrightarrow (x = a \wedge (y = b - 1 \vee y = b + 1)) \vee (y = b \wedge (x = a - 1 \vee x = a + 1)) .$$

We could name each pit, but this would be inappropriate for a different reason: there is no reason to distinguish among pits.¹⁰ It is simpler to use a unary predicate Pit that is true of squares containing pits. Finally, since there is exactly one wumpus, a constant $Wumpus$ is just as good as a unary predicate (and perhaps more dignified from the wumpus's viewpoint).

The agent's location changes over time, so we write $At(Agent, s, t)$ to mean that the agent is at square s at time t . We can fix the wumpus's location with $\forall t \text{ } At(Wumpus, [2, 2], t)$. We can then say that objects can only be at one location at a time:

$$\forall x, s_1, s_2, t \text{ } At(x, s_1, t) \wedge At(x, s_2, t) \Rightarrow s_1 = s_2 .$$

Given its current location, the agent can infer properties of the square from properties of its current percept. For example, if the agent is at a square and perceives a breeze, then that square is breezy:

$$\forall s, t \text{ } At(Agent, s, t) \wedge Breeze(t) \Rightarrow Breezy(s) .$$

It is useful to know that a *square* is breezy because we know that the pits cannot move about. Notice that $Breezy$ has no time argument.

Having discovered which places are breezy (or smelly) and, very important, *not* breezy (or *not* smelly), the agent can deduce where the pits are (and where the wumpus is). Whereas propositional logic necessitates a separate axiom for each square (see R_2 and R_3 on page 247) and would need a different set of axioms for each geographical layout of the world, first-order logic just needs one axiom:

$$\forall s \text{ } Breezy(s) \Leftrightarrow \exists r \text{ } Adjacent(r, s) \wedge Pit(r) . \quad (8.4)$$

Similarly, in first-order logic we can quantify over time, so we need just one successor-state axiom for each predicate, rather than a different copy for each time step. For example, the axiom for the arrow (Equation (7.2) on page 267) becomes

$$\forall t \text{ } HaveArrow(t + 1) \Leftrightarrow (HaveArrow(t) \wedge \neg Action(Shoot, t)) .$$

From these two example sentences, we can see that the first-order logic formulation is no less concise than the original English-language description given in Chapter 7. The reader

¹⁰ Similarly, most of us do not name each bird that flies overhead as it migrates to warmer regions in winter. An ornithologist wishing to study migration patterns, survival rates, and so on *does* name each bird, by means of a ring on its leg, because individual birds must be tracked.

is invited to construct analogous axioms for the agent's location and orientation; in these cases, the axioms quantify over both space and time. As in the case of propositional state estimation, an agent can use logical inference with axioms of this kind to keep track of aspects of the world that are not directly observed. Chapter 10 goes into more depth on the subject of first-order successor-state axioms and their uses for constructing plans.

8.4 KNOWLEDGE ENGINEERING IN FIRST-ORDER LOGIC

KNOWLEDGE ENGINEERING

The preceding section illustrated the use of first-order logic to represent knowledge in three simple domains. This section describes the general process of knowledge-base construction—a process called **knowledge engineering**. A knowledge engineer is someone who investigates a particular domain, learns what concepts are important in that domain, and creates a formal representation of the objects and relations in the domain. We illustrate the knowledge engineering process in an electronic circuit domain that should already be fairly familiar, so that we can concentrate on the representational issues involved. The approach we take is suitable for developing *special-purpose* knowledge bases whose domain is carefully circumscribed and whose range of queries is known in advance. *General-purpose* knowledge bases, which cover a broad range of human knowledge and are intended to support tasks such as natural language understanding, are discussed in Chapter 12.

8.4.1 The knowledge-engineering process

Knowledge engineering projects vary widely in content, scope, and difficulty, but all such projects include the following steps:

1. *Identify the task.* The knowledge engineer must delineate the range of questions that the knowledge base will support and the kinds of facts that will be available for each specific problem instance. For example, does the wumpus knowledge base need to be able to choose actions or is it required to answer questions only about the contents of the environment? Will the sensor facts include the current location? The task will determine what knowledge must be represented in order to connect problem instances to answers. This step is analogous to the PEAS process for designing agents in Chapter 2.
2. *Assemble the relevant knowledge.* The knowledge engineer might already be an expert in the domain, or might need to work with real experts to extract what they know—a process called **knowledge acquisition**. At this stage, the knowledge is not represented formally. The idea is to understand the scope of the knowledge base, as determined by the task, and to understand how the domain actually works.

KNOWLEDGE ACQUISITION

For the wumpus world, which is defined by an artificial set of rules, the relevant knowledge is easy to identify. (Notice, however, that the definition of adjacency was not supplied explicitly in the wumpus-world rules.) For real domains, the issue of relevance can be quite difficult—for example, a system for simulating VLSI designs might or might not need to take into account stray capacitances and skin effects.

ONTOLOGY

3. *Decide on a vocabulary of predicates, functions, and constants.* That is, translate the important domain-level concepts into logic-level names. This involves many questions of knowledge-engineering *style*. Like programming style, this can have a significant impact on the eventual success of the project. For example, should pits be represented by objects or by a unary predicate on squares? Should the agent's orientation be a function or a predicate? Should the wumpus's location depend on time? Once the choices have been made, the result is a vocabulary that is known as the **ontology** of the domain. The word *ontology* means a particular theory of the nature of being or existence. The ontology determines what kinds of things exist, but does not determine their specific properties and interrelationships.
4. *Encode general knowledge about the domain.* The knowledge engineer writes down the axioms for all the vocabulary terms. This pins down (to the extent possible) the meaning of the terms, enabling the expert to check the content. Often, this step reveals misconceptions or gaps in the vocabulary that must be fixed by returning to step 3 and iterating through the process.
5. *Encode a description of the specific problem instance.* If the ontology is well thought out, this step will be easy. It will involve writing simple atomic sentences about instances of concepts that are already part of the ontology. For a logical agent, problem instances are supplied by the sensors, whereas a "disembodied" knowledge base is supplied with additional sentences in the same way that traditional programs are supplied with input data.
6. *Pose queries to the inference procedure and get answers.* This is where the reward is: we can let the inference procedure operate on the axioms and problem-specific facts to derive the facts we are interested in knowing. Thus, we avoid the need for writing an application-specific solution algorithm.
7. *Debug the knowledge base.* Alas, the answers to queries will seldom be correct on the first try. More precisely, the answers will be correct *for the knowledge base as written*, assuming that the inference procedure is sound, but they will not be the ones that the user is expecting. For example, if an axiom is missing, some queries will not be answerable from the knowledge base. A considerable debugging process could ensue. Missing axioms or axioms that are too weak can be easily identified by noticing places where the chain of reasoning stops unexpectedly. For example, if the knowledge base includes a diagnostic rule (see Exercise 8.13) for finding the wumpus,

$$\forall s \text{ Smelly}(s) \Rightarrow \text{Adjacent}(\text{Home}(\text{Wumpus}), s),$$

instead of the biconditional, then the agent will never be able to prove the *absence* of wumpuses. Incorrect axioms can be identified because they are false statements about the world. For example, the sentence

$$\forall x \text{ NumOfLegs}(x, 4) \Rightarrow \text{Mammal}(x)$$

is false for reptiles, amphibians, and, more importantly, tables. *The falsehood of this sentence can be determined independently of the rest of the knowledge base.* In contrast,



a typical error in a program looks like this:

```
offset = position + 1.
```

It is impossible to tell whether this statement is correct without looking at the rest of the program to see whether, for example, `offset` is used to refer to the current position, or to one beyond the current position, or whether the value of `position` is changed by another statement and so `offset` should also be changed again.

To understand this seven-step process better, we now apply it to an extended example—the domain of electronic circuits.

8.4.2 The electronic circuits domain

We will develop an ontology and knowledge base that allow us to reason about digital circuits of the kind shown in Figure 8.6. We follow the seven-step process for knowledge engineering.

Identify the task

There are many reasoning tasks associated with digital circuits. At the highest level, one analyzes the circuit's functionality. For example, does the circuit in Figure 8.6 actually add properly? If all the inputs are high, what is the output of gate A2? Questions about the circuit's structure are also interesting. For example, what are all the gates connected to the first input terminal? Does the circuit contain feedback loops? These will be our tasks in this section. There are more detailed levels of analysis, including those related to timing delays, circuit area, power consumption, production cost, and so on. Each of these levels would require additional knowledge.

Assemble the relevant knowledge

What do we know about digital circuits? For our purposes, they are composed of wires and gates. Signals flow along wires to the input terminals of gates, and each gate produces a

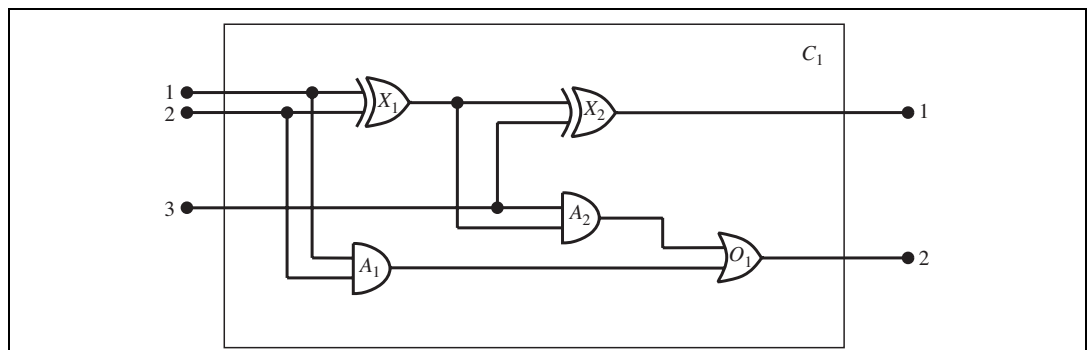


Figure 8.6 A digital circuit C1, purporting to be a one-bit full adder. The first two inputs are the two bits to be added, and the third input is a carry bit. The first output is the sum, and the second output is a carry bit for the next adder. The circuit contains two XOR gates, two AND gates, and one OR gate.

signal on the output terminal that flows along another wire. To determine what these signals will be, we need to know how the gates transform their input signals. There are four types of gates: AND, OR, and XOR gates have two input terminals, and NOT gates have one. All gates have one output terminal. Circuits, like gates, have input and output terminals.

To reason about functionality and connectivity, we do not need to talk about the wires themselves, the paths they take, or the junctions where they come together. All that matters is the connections between terminals—we can say that one output terminal is connected to another input terminal without having to say what actually connects them. Other factors such as the size, shape, color, or cost of the various components are irrelevant to our analysis.

If our purpose were something other than verifying designs at the gate level, the ontology would be different. For example, if we were interested in debugging faulty circuits, then it would probably be a good idea to include the wires in the ontology, because a faulty wire can corrupt the signal flowing along it. For resolving timing faults, we would need to include gate delays. If we were interested in designing a product that would be profitable, then the cost of the circuit and its speed relative to other products on the market would be important.

Decide on a vocabulary

We now know that we want to talk about circuits, terminals, signals, and gates. The next step is to choose functions, predicates, and constants to represent them. First, we need to be able to distinguish gates from each other and from other objects. Each gate is represented as an object named by a constant, about which we assert that it is a gate with, say, $Gate(X_1)$. The behavior of each gate is determined by its type: one of the constants AND , OR , XOR , or NOT . Because a gate has exactly one type, a function is appropriate: $Type(X_1) = XOR$. Circuits, like gates, are identified by a predicate: $Circuit(C_1)$.

Next we consider terminals, which are identified by the predicate $Terminal(x)$. A gate or circuit can have one or more input terminals and one or more output terminals. We use the function $In(1, X_1)$ to denote the first input terminal for gate X_1 . A similar function Out is used for output terminals. The function $Arity(c, i, j)$ says that circuit c has i input and j output terminals. The connectivity between gates can be represented by a 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, $On(t)$, 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 C_1 ?” We therefore introduce as objects two signal values, 1 and 0, and a function $Signal(t)$ that denotes the signal value for the terminal t .

Encode general knowledge of the domain

One sign that we have a good ontology is that we require only a few general rules, which can be stated clearly and concisely. These are all the axioms we will need:

1. If two terminals are connected, then they have the same signal:

$$\forall t_1, t_2 \quad Terminal(t_1) \wedge Terminal(t_2) \wedge Connected(t_1, t_2) \Rightarrow Signal(t_1) = Signal(t_2) .$$

2. The signal at every terminal is either 1 or 0:

$$\forall t \text{ Terminal}(t) \Rightarrow \text{Signal}(t) = 1 \vee \text{Signal}(t) = 0 .$$
3. Connected is commutative:

$$\forall t_1, t_2 \text{ Connected}(t_1, t_2) \Leftrightarrow \text{Connected}(t_2, t_1) .$$
4. There are four types of gates:

$$\forall g \text{ Gate}(g) \wedge k = \text{Type}(g) \Rightarrow k = \text{AND} \vee k = \text{OR} \vee k = \text{XOR} \vee k = \text{NOT} .$$
5. An AND gate's output is 0 if and only if any of its inputs is 0:

$$\forall g \text{ Gate}(g) \wedge \text{Type}(g) = \text{AND} \Rightarrow$$

$$\text{Signal}(\text{Out}(1, g)) = 0 \Leftrightarrow \exists n \text{ Signal}(\text{In}(n, g)) = 0 .$$
6. An OR gate's output is 1 if and only if any of its inputs is 1:

$$\forall g \text{ Gate}(g) \wedge \text{Type}(g) = \text{OR} \Rightarrow$$

$$\text{Signal}(\text{Out}(1, g)) = 1 \Leftrightarrow \exists n \text{ Signal}(\text{In}(n, g)) = 1 .$$
7. An XOR gate's output is 1 if and only if its inputs are different:

$$\forall g \text{ Gate}(g) \wedge \text{Type}(g) = \text{XOR} \Rightarrow$$

$$\text{Signal}(\text{Out}(1, g)) = 1 \Leftrightarrow \text{Signal}(\text{In}(1, g)) \neq \text{Signal}(\text{In}(2, g)) .$$
8. A NOT gate's output is different from its input:

$$\forall g \text{ Gate}(g) \wedge (\text{Type}(g) = \text{NOT}) \Rightarrow$$

$$\text{Signal}(\text{Out}(1, g)) \neq \text{Signal}(\text{In}(1, g)) .$$
9. The gates (except for NOT) have two inputs and one output.

$$\forall g \text{ Gate}(g) \wedge \text{Type}(g) \neq \text{NOT} \Rightarrow \text{Arity}(g, 1, 1) .$$

$$\forall g \text{ Gate}(g) \wedge k = \text{Type}(g) \wedge (k = \text{AND} \vee k = \text{OR} \vee k = \text{XOR}) \Rightarrow$$

$$\text{Arity}(g, 2, 1)$$
10. A circuit has terminals, up to its input and output arity, and nothing beyond its arity:

$$\forall c, i, j \text{ Circuit}(c) \wedge \text{Arity}(c, i, j) \Rightarrow$$

$$\forall n (n \leq i \Rightarrow \text{Terminal}(\text{In}(c, n))) \wedge (n > i \Rightarrow \text{In}(c, n) = \text{Nothing}) \wedge$$

$$\forall n (n \leq j \Rightarrow \text{Terminal}(\text{Out}(c, n))) \wedge (n > j \Rightarrow \text{Out}(c, n) = \text{Nothing})$$
11. Gates, terminals, signals, gate types, and *Nothing* are all distinct.

$$\forall g, t \text{ Gate}(g) \wedge \text{Terminal}(t) \Rightarrow$$

$$g \neq t \neq 1 \neq 0 \neq \text{OR} \neq \text{AND} \neq \text{XOR} \neq \text{NOT} \neq \text{Nothing} .$$
12. Gates are circuits.

$$\forall g \text{ Gate}(g) \Rightarrow \text{Circuit}(g)$$

Encode the specific problem instance

The circuit shown in Figure 8.6 is encoded as circuit C_1 with the following description. First, we categorize the circuit and its component gates:

$$\begin{aligned} &\text{Circuit}(C_1) \wedge \text{Arity}(C_1, 3, 2) \\ &\text{Gate}(X_1) \wedge \text{Type}(X_1) = \text{XOR} \\ &\text{Gate}(X_2) \wedge \text{Type}(X_2) = \text{XOR} \\ &\text{Gate}(A_1) \wedge \text{Type}(A_1) = \text{AND} \\ &\text{Gate}(A_2) \wedge \text{Type}(A_2) = \text{AND} \\ &\text{Gate}(O_1) \wedge \text{Type}(O_1) = \text{OR} . \end{aligned}$$

Then, we show the connections between them:

$$\begin{array}{ll}
 \text{Connected}(\text{Out}(1, X_1), \text{In}(1, X_2)) & \text{Connected}(\text{In}(1, C_1), \text{In}(1, X_1)) \\
 \text{Connected}(\text{Out}(1, X_1), \text{In}(2, A_2)) & \text{Connected}(\text{In}(1, C_1), \text{In}(1, A_1)) \\
 \text{Connected}(\text{Out}(1, A_2), \text{In}(1, O_1)) & \text{Connected}(\text{In}(2, C_1), \text{In}(2, X_1)) \\
 \text{Connected}(\text{Out}(1, A_1), \text{In}(2, O_1)) & \text{Connected}(\text{In}(2, C_1), \text{In}(2, A_1)) \\
 \text{Connected}(\text{Out}(1, X_2), \text{Out}(1, C_1)) & \text{Connected}(\text{In}(3, C_1), \text{In}(2, X_2)) \\
 \text{Connected}(\text{Out}(1, O_1), \text{Out}(2, C_1)) & \text{Connected}(\text{In}(3, C_1), \text{In}(1, A_2)) .
 \end{array}$$

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?

$$\begin{array}{l}
 \exists i_1, i_2, i_3 \text{ Signal}(\text{In}(1, C_1)) = i_1 \wedge \text{Signal}(\text{In}(2, C_1)) = i_2 \wedge \text{Signal}(\text{In}(3, C_1)) = i_3 \\
 \wedge \text{Signal}(\text{Out}(1, C_1)) = 0 \wedge \text{Signal}(\text{Out}(2, C_1)) = 1 .
 \end{array}$$

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. ASK VARS will give us 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?

$$\begin{array}{l}
 \exists i_1, i_2, i_3, o_1, o_2 \text{ Signal}(\text{In}(1, C_1)) = i_1 \wedge \text{Signal}(\text{In}(2, C_1)) = i_2 \\
 \wedge \text{Signal}(\text{In}(3, C_1)) = i_3 \wedge \text{Signal}(\text{Out}(1, C_1)) = o_1 \wedge \text{Signal}(\text{Out}(2, C_1)) = o_2 .
 \end{array}$$

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 8.26.) 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.

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 fail to read Section 8.2.8 and hence forget to assert that $1 \neq 0$. 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

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

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 :

$$\text{Signal}(\text{Out}(1, X_1)) = 1 \Leftrightarrow \text{Signal}(\text{In}(1, X_1)) \neq \text{Signal}(\text{In}(2, X_1)) .$$

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

$$\text{Signal}(\text{Out}(1, X_1)) = 1 \Leftrightarrow 1 \neq 0 .$$

Now the problem is apparent: the system is unable to infer that $\text{Signal}(\text{Out}(1, X_1)) = 1$, 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.
- The syntax of first-order logic builds on that of propositional logic. It adds terms to represent objects, and has universal and existential quantifiers to construct assertions about all or some of the possible values of the quantified variables.
- A **possible world**, or **model**, for first-order logic includes a set of objects and an **interpretation** that maps constant symbols to objects, predicate symbols to relations among objects, and function symbols to functions on objects.
- An atomic sentence is true just when the relation named by the predicate holds between the objects named by the terms. **Extended interpretations**, which map quantifier variables to objects in the model, define the truth of quantified sentences.
- 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.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Although Aristotle's logic deals with generalizations over objects, it fell far short of the expressive power of first-order logic. A major barrier to its further development was its concentration on one-place predicates to the exclusion of many-place relational predicates. 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, 2004).

True first-order logic dates from the introduction of quantifiers in Gottlob Frege's (1879) *Begriffsschrift* ("Concept Writing" or "Conceptual Notation"). Peirce (1883) also developed first-order logic independently of Frege, although slightly later. Frege's ability to nest quantifiers was a big step forward, but he used an awkward notation. 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).

Leopold Löwenheim (1915) gave a systematic treatment of model theory for first-order logic, including the first proper treatment of the equality symbol. 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 University. 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 12.

Practical applications built with first-order logic include a system for evaluating the manufacturing requirements for electronic products (Mannion, 2002), a system for reasoning about policies for file access and digital rights management (Halpern and Weissman, 2008), and a system for the automated composition of Web services (McIlraith and Zeng, 2001).

Reactions to the Whorf hypothesis (Whorf, 1956) and the problem of language and thought in general, appear in several recent books (Gumperz and Levinson, 1996; Bowerman and Levinson, 2001; Pinker, 2003; Gentner and Goldin-Meadow, 2003). The "theory" theory (Gopnik and Glymour, 2002; Tenenbaum *et al.*, 2007) views children's learning about the world as analogous to the construction of scientific theories. Just as the predictions of a machine learning algorithm depend strongly on the vocabulary supplied to it, so will the child's formulation of theories depend on the linguistic environment in which learning occurs.

There are a number of good introductory texts on first-order logic, including some by leading figures in the history of logic: Alfred Tarski (1941), Alonzo Church (1956), and W.V. Quine (1982) (which 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, as do Huth and Ryan (2004), who concentrate on program verification. Barwise and Etchemendy (2002) take an approach similar to the one used here. Smullyan (1995) presents results concisely, using the tableau format. 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) is both a solid introduction to logic and the first systematic treatment of logical agents with percepts and actions, and there are two good handbooks: van Bentham and ter Meulen (1997) and Robinson and Voronkov (2001). The journal of record for the field of pure mathematical logic is the *Journal of Symbolic Logic*, whereas the *Journal of Applied Logic* deals with concerns closer to those of artificial intelligence.

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—it represents facts with a map language. 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 model, 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 models for a domain with D elements. Don't worry about eliminating redundant combinations.

8.6 Which of the following are valid (necessarily true) sentences?

- a. $(\exists x \ x = x) \Rightarrow (\forall y \ \exists z \ y = z)$.
- b. $\forall x \ P(x) \vee \neg P(x)$.
- c. $\forall x \ \text{Smart}(x) \vee (x = x)$.

8.7 Consider a version of the semantics for first-order logic in which models with empty domains are allowed. Give at least two examples of sentences that are valid according to the

standard semantics but not according to the new semantics. Discuss which outcome makes more intuitive sense for your examples.

8.8 Does the fact $\neg \text{Spouse}(\text{George}, \text{Laura})$ follow from the facts $\text{Jim} \neq \text{George}$ and $\text{Spouse}(\text{Jim}, \text{Laura})$? If so, give a proof; if not, supply additional axioms as needed. What happens if we use Spouse as a unary function symbol instead of a binary predicate?

8.9 This exercise uses the function MapColor and predicates $\text{In}(x, y)$, $\text{Borders}(x, y)$, and $\text{Country}(x)$, whose arguments are geographical regions, along with constant symbols for various regions. In each of the following we give an English sentence and a number of candidate logical expressions. For each of the logical expressions, state whether it (1) correctly expresses the English sentence; (2) is syntactically invalid and therefore meaningless; or (3) is syntactically valid but does not express the meaning of the English sentence.

a. Paris and Marseilles are both in France.

- (i) $\text{In}(\text{Paris} \wedge \text{Marseilles}, \text{France})$.
- (ii) $\text{In}(\text{Paris}, \text{France}) \wedge \text{In}(\text{Marseilles}, \text{France})$.
- (iii) $\text{In}(\text{Paris}, \text{France}) \vee \text{In}(\text{Marseilles}, \text{France})$.

b. There is a country that borders both Iraq and Pakistan.

- (i) $\exists c \text{ Country}(c) \wedge \text{Border}(c, \text{Iraq}) \wedge \text{Border}(c, \text{Pakistan})$.
- (ii) $\exists c \text{ Country}(c) \Rightarrow [\text{Border}(c, \text{Iraq}) \wedge \text{Border}(c, \text{Pakistan})]$.
- (iii) $[\exists c \text{ Country}(c)] \Rightarrow [\text{Border}(c, \text{Iraq}) \wedge \text{Border}(c, \text{Pakistan})]$.
- (iv) $\exists c \text{ Border}(\text{Country}(c), \text{Iraq} \wedge \text{Pakistan})$.

c. All countries that border Ecuador are in South America.

- (i) $\forall c \text{ Country}(c) \wedge \text{Border}(c, \text{Ecuador}) \Rightarrow \text{In}(c, \text{SouthAmerica})$.
- (ii) $\forall c \text{ Country}(c) \Rightarrow [\text{Border}(c, \text{Ecuador}) \Rightarrow \text{In}(c, \text{SouthAmerica})]$.
- (iii) $\forall c [\text{Country}(c) \Rightarrow \text{Border}(c, \text{Ecuador})] \Rightarrow \text{In}(c, \text{SouthAmerica})$.
- (iv) $\forall c \text{ Country}(c) \wedge \text{Border}(c, \text{Ecuador}) \wedge \text{In}(c, \text{SouthAmerica})$.

d. No region in South America borders any region in Europe.

- (i) $\neg[\exists c, d \text{ In}(c, \text{SouthAmerica}) \wedge \text{In}(d, \text{Europe}) \wedge \text{Borders}(c, d)]$.
- (ii) $\forall c, d [\text{In}(c, \text{SouthAmerica}) \wedge \text{In}(d, \text{Europe})] \Rightarrow \neg \text{Borders}(c, d)$.
- (iii) $\neg \forall c \text{ In}(c, \text{SouthAmerica}) \Rightarrow \exists d \text{ In}(d, \text{Europe}) \wedge \neg \text{Borders}(c, d)$.
- (iv) $\forall c \text{ In}(c, \text{SouthAmerica}) \Rightarrow \forall d \text{ In}(d, \text{Europe}) \Rightarrow \neg \text{Borders}(c, d)$.

e. No two adjacent countries have the same map color.

- (i) $\forall x, y \neg \text{Country}(x) \vee \neg \text{Country}(y) \vee \neg \text{Borders}(x, y) \vee \neg (\text{MapColor}(x) = \text{MapColor}(y))$.
- (ii) $\forall x, y (\text{Country}(x) \wedge \text{Country}(y) \wedge \text{Borders}(x, y) \wedge \neg(x = y)) \Rightarrow \neg (\text{MapColor}(x) = \text{MapColor}(y))$.
- (iii) $\forall x, y \text{ Country}(x) \wedge \text{Country}(y) \wedge \text{Borders}(x, y) \wedge \neg (\text{MapColor}(x) = \text{MapColor}(y))$.
- (iv) $\forall x, y (\text{Country}(x) \wedge \text{Country}(y) \wedge \text{Borders}(x, y)) \Rightarrow \text{MapColor}(x \neq y)$.

8.10 Consider a vocabulary with the following symbols:

Occupation(p, o): Predicate. Person p has occupation o .

Customer($p1, p2$): Predicate. Person $p1$ is a customer of person $p2$.

Boss($p1, p2$): Predicate. Person $p1$ is a boss of person $p2$.

Doctor, Surgeon, Lawyer, Actor: Constants denoting occupations.

Emily, Joe: Constants denoting people.

Use these symbols to write the following assertions in first-order logic:

- a. Emily is either a surgeon or a lawyer.
- b. Joe is an actor, but he also holds another job.
- c. All surgeons are doctors.
- d. Joe does not have a lawyer (i.e., is not a customer of any lawyer).
- e. Emily has a boss who is a lawyer.
- f. There exists a lawyer all of whose customers are doctors.
- g. Every surgeon has a lawyer.

8.11 Complete the following exercises about logical sentences:

- a. Translate into *good, natural* English (no x s or y s!):

$$\begin{aligned} \forall x, y, l \text{ } \textit{SpeaksLanguage}(x, l) \wedge \textit{SpeaksLanguage}(y, l) \\ \Rightarrow \textit{Understands}(x, y) \wedge \textit{Understands}(y, x). \end{aligned}$$

- b. Explain why this sentence is entailed by the sentence

$$\begin{aligned} \forall x, y, l \text{ } \textit{SpeaksLanguage}(x, l) \wedge \textit{SpeaksLanguage}(y, l) \\ \Rightarrow \textit{Understands}(x, y). \end{aligned}$$

- c. Translate into first-order logic the following sentences:

- (i) Understanding leads to friendship.
- (ii) Friendship is transitive.

Remember to define all predicates, functions, and constants you use.

8.12 Rewrite the first two Peano axioms in Section 8.3.3 as a single axiom that defines *NatNum*(x) so as to exclude the possibility of natural numbers except for those generated by the successor function.

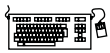
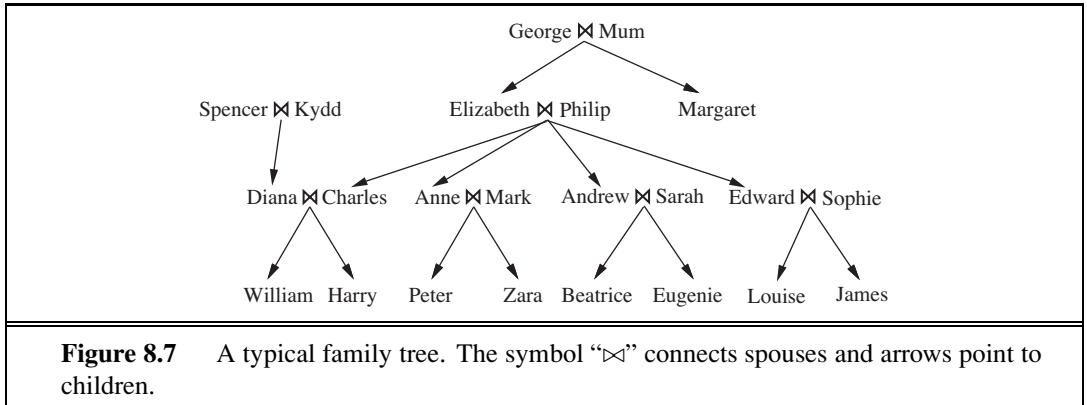
8.13 Equation (8.4) on page 306 defines the conditions under which a square is breezy. Here we consider two other ways to describe this aspect of the wumpus world.

DIAGNOSTIC RULE

- a. We can write **diagnostic rules** leading from observed effects to hidden causes. For finding pits, the obvious diagnostic rules say that if a square is breezy, some adjacent square must contain a pit; and if a square is not breezy, then no adjacent square contains a pit. Write these two rules in first-order logic and show that their conjunction is logically equivalent to Equation (8.4).

CAUSAL RULE

- b. We can write **causal rules** leading from cause to effect. One obvious causal rule is that a pit causes all adjacent squares to be breezy. Write this rule in first-order logic, explain why it is incomplete compared to Equation (8.4), and supply the missing axiom.



8.14 Write axioms describing the predicates *Grandchild*, *Greatgrandparent*, *Ancestor*, *Brother*, *Sister*, *Daughter*, *Son*, *FirstCousin*, *BrotherInLaw*, *SisterInLaw*, *Aunt*, and *Uncle*. Find out the proper definition of *m*th 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 8.7. 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, Zara’s great-grandparents, and Eugenie’s ancestors.

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

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

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

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

$$\forall x, y \quad \text{Adjacent}([x, y], [x + 1, y]) \wedge \text{Adjacent}([x, y], [x, y + 1]) .$$

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

8.19 Assuming predicates *Parent*(*p*, *q*) and *Female*(*p*) and constants *Joan* and *Kevin*, with the obvious meanings, express each of the following sentences in first-order logic. (You may use the abbreviation \exists^1 to mean “there exists exactly one.”)

- Joan has a daughter (possibly more than one, and possibly sons as well).
- Joan has exactly one daughter (but may have sons as well).
- Joan has exactly one child, a daughter.
- Joan and Kevin have exactly one child together.
- Joan has at least one child with Kevin, and no children with anyone else.

8.20 Arithmetic assertions can be written in first-order logic with the predicate symbol $<$, the function symbols $+$ and \times , and the constant symbols 0 and 1. Additional predicates can also be defined with biconditionals.

- Represent the property “ x is an even number.”
- Represent the property “ x is prime.”
- Goldbach’s conjecture is the conjecture (unproven as yet) that every even number is equal to the sum of two primes. Represent this conjecture as a logical sentence.

8.21 In Chapter 6, we used equality to indicate the relation between a variable and its value. For instance, we wrote $WA = red$ to mean that Western Australia is colored red. Representing this in first-order logic, we must write more verbosely $ColorOf(WA) = red$. What incorrect inference could be drawn if we wrote sentences such as $WA = red$ directly as logical assertions?

8.22 Write in first-order logic the assertion that every key and at least one of every pair of socks will eventually be lost forever, using only the following vocabulary: $Key(x)$, x is a key; $Sock(x)$, x is a sock; $Pair(x, y)$, x and y are a pair; Now , the current time; $Before(t_1, t_2)$, time t_1 comes before time t_2 ; $Lost(x, t)$, object x is lost at time t .

8.23 For each of the following sentences in English, decide if the accompanying first-order logic sentence is a good translation. If not, explain why not and correct it. (Some sentences may have more than one error!)

- No two people have the same social security number.

$$\neg \exists x, y, n \text{ Person}(x) \wedge \text{Person}(y) \Rightarrow [\text{HasSS}\#(x, n) \wedge \text{HasSS}\#(y, n)].$$

- John’s social security number is the same as Mary’s.

$$\exists n \text{ HasSS}\#(\text{John}, n) \wedge \text{HasSS}\#(\text{Mary}, n).$$

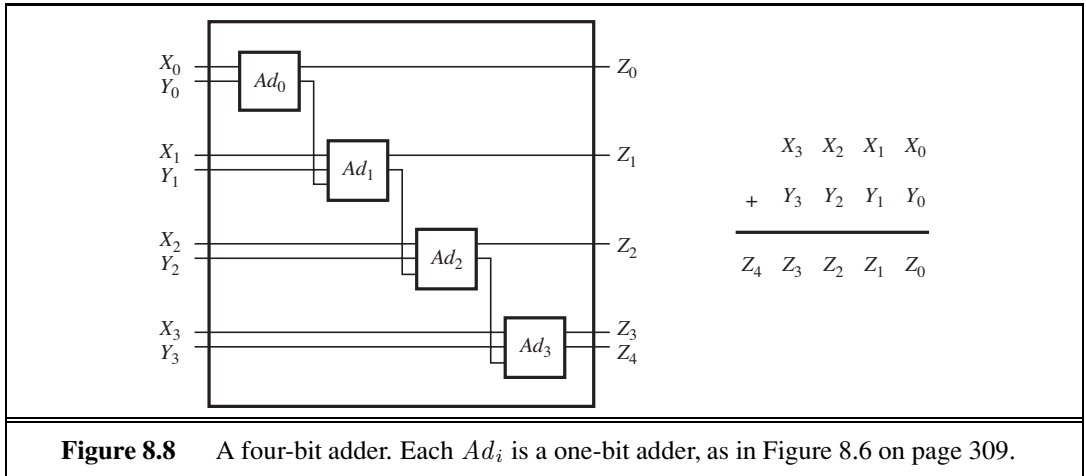
- Everyone’s social security number has nine digits.

$$\forall x, n \text{ Person}(x) \Rightarrow [\text{HasSS}\#(x, n) \wedge \text{Digits}(n, 9)].$$

- Rewrite each of the above (uncorrected) sentences using a function symbol $SS\#$ instead of the predicate $\text{HasSS}\#$.

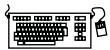
8.24 Represent the following sentences in first-order logic, using a consistent vocabulary (which you must define):

- Some students took French in spring 2001.
- Every student who takes French passes it.
- Only one student took Greek in spring 2001.
- The best score in Greek is always higher than the best score in French.
- Every person who buys a policy is smart.
- No person buys an expensive policy.
- 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.
- l. All Greeks speak the same language. (Use $Speaks(x, l)$ to mean that person x speaks language l .)

8.25 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?”



8.26 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 8.8, and pose the queries needed to verify that it is in fact correct.

8.27 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.

8.28 Consider a first-order logical knowledge base that describes worlds containing people, songs, albums (e.g., “Meet the Beatles”) and disks (i.e., particular physical instances of CDs). The vocabulary contains the following symbols:

$CopyOf(d, a)$: Predicate. Disk d is a copy of album a .

$Owns(p, d)$: Predicate. Person p owns disk d .

$Sings(p, s, a)$: Album a includes a recording of song s sung by person p .

$Wrote(p, s)$: Person p wrote song s .

$McCartney, Gershwin, BHoliday, Joe, EleanorRigby, TheManILove, Revolver$: Constants with the obvious meanings.

Express the following statements in first-order logic:

- a. Gershwin wrote “The Man I Love.”
- b. Gershwin did not write “Eleanor Rigby.”
- c. Either Gershwin or McCartney wrote “The Man I Love.”
- d. Joe has written at least one song.
- e. Joe owns a copy of *Revolver*.
- f. Every song that McCartney sings on *Revolver* was written by McCartney.
- g. Gershwin did not write any of the songs on *Revolver*.
- h. Every song that Gershwin wrote has been recorded on some album. (Possibly different songs are recorded on different albums.)
- i. There is a single album that contains every song that Joe has written.
- j. Joe owns a copy of an album that has Billie Holiday singing “The Man I Love.”
- k. Joe owns a copy of every album that has a song sung by McCartney. (Of course, each different album is instantiated in a different physical CD.)
- l. Joe owns a copy of every album on which all the songs are sung by Billie Holiday.

9

INFERENCE IN FIRST-ORDER LOGIC

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

Chapter 7 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. Section 9.1 introduces inference rules for quantifiers and shows how to reduce first-order inference to propositional inference, albeit at potentially 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. Forward and backward chaining can be very efficient, but are applicable only to knowledge bases that can be expressed as sets of Horn clauses. General first-order sentences require resolution-based **theorem proving**, which is described in Section 9.5.

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.

9.1.1 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:

$$\forall x \text{ King}(x) \wedge \text{Greedy}(x) \Rightarrow \text{Evil}(x) .$$

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
GROUND TERM

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 $SUBST(\theta, \alpha)$ denote the result of applying the substitution θ to the sentence α . Then the rule is written

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

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

In the rule for **Existential Instantiation**, the variable is replaced by a single *new constant symbol*. The formal statement is as follows: for any sentence α , variable v , and constant symbol k that does not appear elsewhere in the knowledge base,

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

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 applying the existential instantiation rule just gives a name to that object. Of course, 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

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, we no longer need $\exists x \ Kill(x, Victim)$ once we have added the sentence $Kill(Murderer, Victim)$. 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

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.

9.1.2 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 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

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

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

$$\begin{aligned} &\text{King}(\text{John}) \wedge \text{Greedy}(\text{John}) \Rightarrow \text{Evil}(\text{John}) \\ &\text{King}(\text{Richard}) \wedge \text{Greedy}(\text{Richard}) \Rightarrow \text{Evil}(\text{Richard}) , \end{aligned}$$

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

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 $\text{Father}(\text{Father}(\text{Father}(\text{John})))$ 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 ($\text{Father}(\text{Richard})$ and $\text{Father}(\text{John})$), 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

$$\begin{aligned} &\forall x \text{ } King(x) \wedge Greedy(x) \Rightarrow Evil(x) \\ &King(John) \\ &Greedy(John) \end{aligned}$$

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

9.2.1 A first-order inference rule

The inference that John is evil—that is, that $\{x/John\}$ solves the query $Evil(x)$ —works like this: to use the rule that greedy kings are evil, 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 each of the conjuncts of 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 $\theta = \{x/John\}$ 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:

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

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 to find a substitution both for the variables in the implication sentence and for the variables in the sentences that are in the knowledge base. 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.

This inference process can be captured as a single inference rule that we call **Generalized Modus Ponens**:² 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 ,

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

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:

$$\begin{array}{ll} p_1' \text{ is } King(John) & p_1 \text{ is } King(x) \\ p_2' \text{ is } Greedy(y) & p_2 \text{ is } Greedy(x) \\ \theta \text{ is } \{x/John, y/John\} & q \text{ is } Evil(x) \\ \text{SUBST}(\theta, q) \text{ is } Evil(John). \end{array}$$

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 ,

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

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

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

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

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

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 ground (variable-free) propositional logic to first-order logic. We will see in the rest of this 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 that are required to allow particular inferences to proceed.

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

UNIFICATION

UNIFIER

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

Let us look at some examples of how UNIFY should behave. Suppose we have a query $\text{AskVars}(\text{Knows}(\text{John}, x))$: whom does John know? Answers to this query can be found

² Generalized Modus Ponens is more general than Modus Ponens (page 249) in the sense that the known facts and the premise of the implication need match only up to a substitution, rather than exactly. On the other hand, Modus Ponens allows any sentence α as the premise, rather than just a conjunction of atomic sentences.

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:

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

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 x_{17} (a new variable name) without changing its meaning. Now the unification will work:

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

Exercise 9.12 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 and substitution of variables. (For example, $\{x/John\}$ and $\{y/John\}$ are considered equivalent, as are $\{x/John, y/John\}$ and $\{x/John, y/x\}$.) In this case it is $\{y/John, x/z\}$.

An algorithm for computing most general unifiers is shown in Figure 9.1. The process is 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. For example, $S(x)$ can’t unify with $S(S(x))$. 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.

9.2.3 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

STANDARDIZING
APART

MOST GENERAL
UNIFIER

OCCUR CHECK

```

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

  if  $\theta = \text{failure}$  then return failure
  else if  $x = y$  then return  $\theta$ 
  else if VARIABLE?( $x$ ) then return UNIFY-VAR( $x, y, \theta$ )
  else if VARIABLE?( $y$ ) then return UNIFY-VAR( $y, x, \theta$ )
  else if COMPOUND?( $x$ ) and COMPOUND?( $y$ ) then
    return UNIFY( $x$ .ARGS,  $y$ .ARGS, UNIFY( $x$ .OP,  $y$ .OP,  $\theta$ ))
  else if LIST?( $x$ ) and LIST?( $y$ ) then
    return UNIFY( $x$ .REST,  $y$ .REST, UNIFY( $x$ .FIRST,  $y$ .FIRST,  $\theta$ ))
  else return failure

```

```

function UNIFY-VAR( $var, x, \theta$ ) returns a substitution

  if  $\{var/val\} \in \theta$  then return UNIFY( $val, x, \theta$ )
  else if  $\{x/val\} \in \theta$  then return UNIFY( $var, val, \theta$ )
  else if OCCUR-CHECK?( $var, x$ ) then return failure
  else return add  $\{var/x\}$  to  $\theta$ 

```

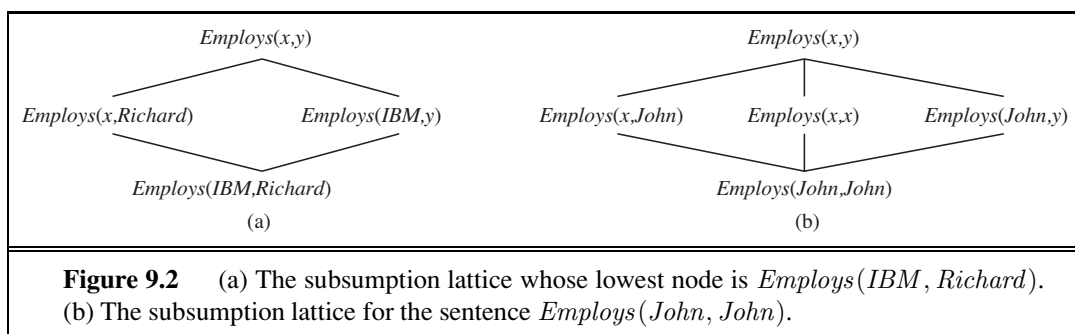
Figure 9.1 The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution θ that is the argument to UNIFY is built up along the way and is used to make sure that later comparisons are consistent with bindings that were established earlier.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

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 one long list and unify each query against every element of 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; it 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 for efficient access.

Predicate indexing is useful when there are many predicate symbols but only a few clauses for each symbol. Sometimes, however, a predicate has many clauses. 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 $Employs(IBM, y)$, 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 $Employs(IBM, Richard)$, the queries are

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

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, however, 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 commercial databases, where facts number in the billions, the problem has been the subject of intensive study and technology development..

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. 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 forward chaining can be implemented very efficiently.

9.3.1 First-order definite clauses

First-order definite clauses closely resemble propositional definite clauses (page 256): 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:

$$\begin{aligned} &King(x) \wedge Greedy(x) \Rightarrow Evil(x) . \\ &King(John) . \\ &Greedy(y) . \end{aligned}$$

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.) 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 $\exists x \text{ Owns}(\text{Nono}, x) \wedge \text{Missile}(x)$ is transformed into two definite clauses by Existential Instantiation, introducing a new constant M_1 :

$$\text{Owns}(\text{Nono}, M_1) \quad (9.4)$$

$$\text{Missile}(M_1) \quad (9.5)$$

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

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

We will also need to know that missiles are weapons:

$$\text{Missile}(x) \Rightarrow \text{Weapon}(x) \quad (9.7)$$

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

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

“West, who is American . . .”:

$$\text{American}(\text{West}) . \quad (9.9)$$

“The country Nono, an enemy of America . . .”:

$$\text{Enemy}(\text{Nono}, \text{America}) . \quad (9.10)$$

DATALOG

This knowledge base contains no function symbols and is therefore an instance of the class of **Datalog** knowledge bases. Datalog is a language that is restricted to first-order definite clauses with no function symbols. Datalog gets its name because it can represent the type of statements typically made in relational databases. We will see that the absence of function symbols makes inference much easier.

9.3.2 A simple forward-chaining algorithm

RENAMING

The first forward-chaining algorithm we consider is a simple one, 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, $\text{Likes}(x, \text{IceCream})$ and $\text{Likes}(y, \text{IceCream})$ are renamings of each other because they differ only in the choice of x or y ; their meanings are identical: everyone likes ice cream.

We 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 $\text{Sells}(\text{West}, M_1, \text{Nono})$ is added.
 Rule (9.7) is satisfied with $\{x/M_1\}$, and $\text{Weapon}(M_1)$ is added.
 Rule (9.8) is satisfied with $\{x/\text{Nono}\}$, and $\text{Hostile}(\text{Nono})$ is added.
- On the second iteration, rule (9.3) is satisfied with $\{x/\text{West}, y/M_1, z/\text{Nono}\}$, and $\text{Criminal}(\text{West})$ 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 258); 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

```

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

  repeat until  $new$  is empty
     $new \leftarrow \{ \}$ 
    for each  $rule$  in  $KB$  do
       $(p_1 \wedge \dots \wedge p_n \Rightarrow q) \leftarrow \text{STANDARDIZE-VARIABLES}(rule)$ 
      for each  $\theta$  such that  $\text{SUBST}(\theta, p_1 \wedge \dots \wedge p_n) = \text{SUBST}(\theta, p'_1 \wedge \dots \wedge p'_n)$ 
        for some  $p'_1, \dots, p'_n$  in  $KB$ 
           $q' \leftarrow \text{SUBST}(\theta, q)$ 
          if  $q'$  does not unify with some sentence already in  $KB$  or  $new$  then
            add  $q'$  to  $new$ 
             $\phi \leftarrow \text{UNIFY}(q', \alpha)$ 
            if  $\phi$  is not fail then return  $\phi$ 
    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 . The function `STANDARDIZE-VARIABLES` replaces all variables in its arguments with new ones that have not been used before.

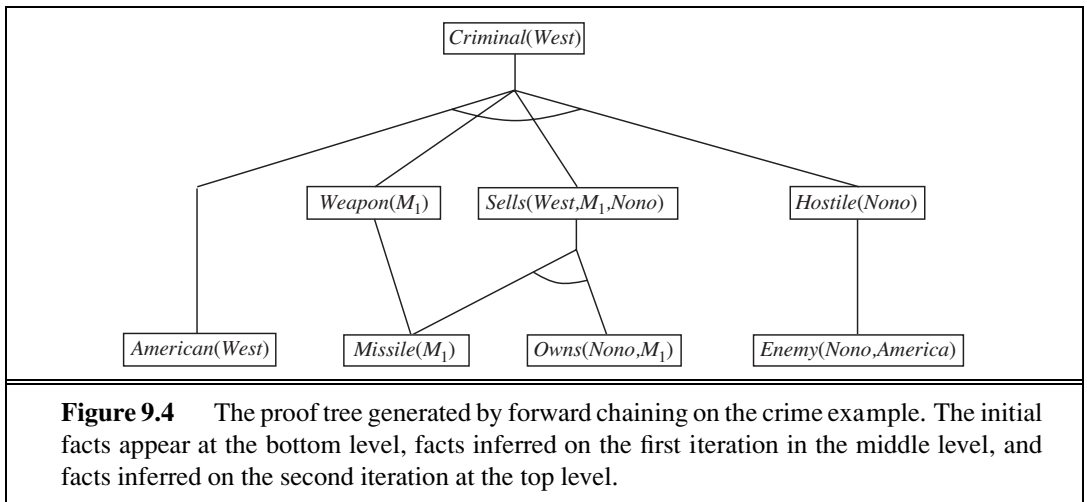


Figure 9.4 The proof tree generated by forward chaining on the crime example. The initial facts appear at the bottom level, facts inferred on the first iteration in the middle level, and facts inferred on the second iteration at the top level.

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 258.) 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

$$\begin{aligned} & \text{NatNum}(0) \\ & \forall n \text{ NatNum}(n) \Rightarrow \text{NatNum}(S(n)) , \end{aligned}$$

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

9.3.3 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 inefficiency. 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 address each of these issues 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

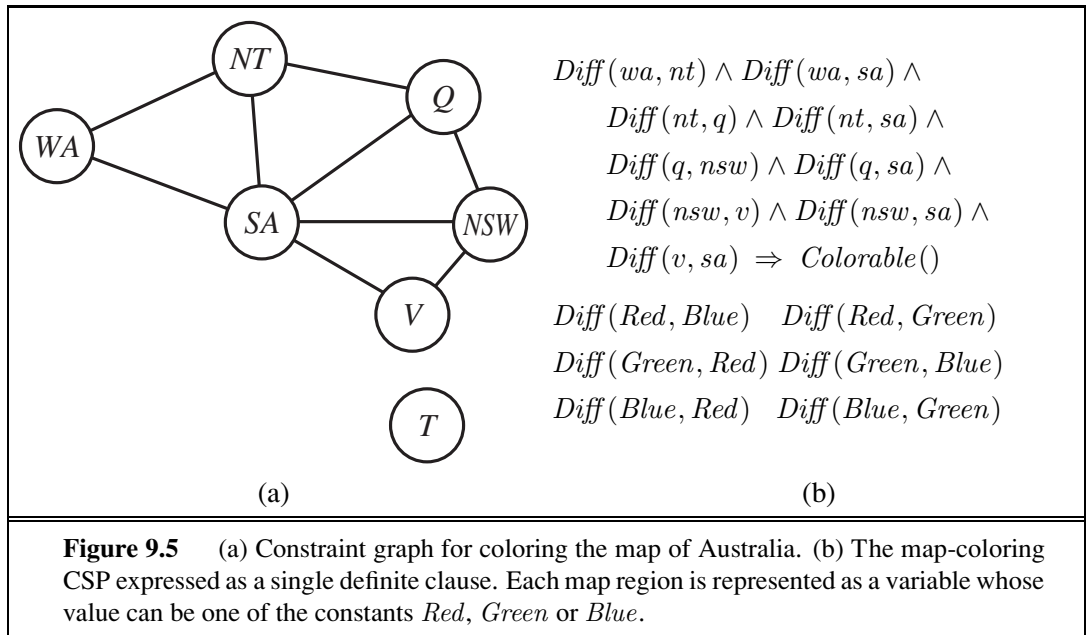
$$\text{Missile}(x) \Rightarrow \text{Weapon}(x) .$$

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

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

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 **minimum-remaining-values** (MRV) heuristic used for CSPs in Chapter 6 would suggest ordering the conjuncts to look for missiles first if fewer missiles than objects 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 6.1, 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 3-SAT 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 knowledge base. It is easy to show that the data complexity of forward chaining is polynomial.
- 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 6 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

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

which corresponds to the reduced CSP shown in Figure 6.12 on page 224. Algorithms for solving tree-structured CSPs can be applied directly to the problem of rule matching.

- We can try to eliminate redundant rule-matching attempts in the forward-chaining algorithm, as described next.

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

$$\text{Missile}(x) \Rightarrow \text{Weapon}(x)$$

matches against $\text{Missile}(M_1)$ (again), and of course the conclusion $\text{Weapon}(M_1)$ 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 repeatedly constructing partial matches 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

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

and the fact $\text{American}(\text{West})$. 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.

The **rete** algorithm³ was the first to address this problem. The algorithm preprocesses the set of rules in the knowledge base to construct a sort of dataflow network in which each

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



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
SYSTEM

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

DEDUCTIVE
DATABASES

MAGIC SET

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, as in PL-FC-ENTAILS? (page 258). A third approach has emerged in the field of **deductive databases**, which are large-scale databases, like relational databases, but which use forward chaining as the standard inference tool rather than SQL queries. 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 :

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

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

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 for definite clauses. 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 also see that backward chaining has some disadvantages compared with forward chaining, and we look at ways to overcome them. Finally, we look at the close connection between logic programming and constraint satisfaction problems.

9.4.1 A backward-chaining algorithm

Figure 9.6 shows a backward-chaining algorithm for definite clauses. $\text{FOL-BC-ASK}(KB, \text{goal})$ will be proved if the knowledge base contains a clause of the form $\text{lhs} \Rightarrow \text{goal}$, where *lhs* (left-hand side) is a list of conjuncts. An atomic fact like *American(West)* is considered as a clause whose *lhs* is the empty list. Now a query that contains variables might be proved in multiple ways. For example, the query *Person(x)* could be proved with the substitution $\{x/\text{John}\}$ as well as with $\{x/\text{Richard}\}$. So we implement FOL-BC-ASK as a **generator**—a function that returns multiple times, each time giving one possible result.

GENERATOR

Backward chaining is a kind of AND/OR search—the OR part because the goal query can be proved by any rule in the knowledge base, and the AND part because all the conjuncts in the *lhs* of a clause must be proved. FOL-BC-OR works by fetching all clauses that might unify with the goal, standardizing the variables in the clause to be brand-new variables, and then, if the *rhs* of the clause does indeed unify with the goal, proving every conjunct in the *lhs*, using FOL-BC-AND. That function in turn works by proving each of the conjuncts in turn, keeping track of the accumulated substitution as we go. Figure 9.7 is the proof tree for deriving *Criminal(West)* from sentences (9.3) through (9.10).

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 show how backward chaining is used in logic programming systems.


```

function FOL-BC-ASK(KB, query) returns a generator of substitutions
  return FOL-BC-OR(KB, query, { })



---


generator FOL-BC-OR(KB, goal,  $\theta$ ) yields a substitution
  for each rule (lhs  $\Rightarrow$  rhs) in FETCH-RULES-FOR-GOAL(KB, goal) do
    (lhs, rhs)  $\leftarrow$  STANDARDIZE-VARIABLES((lhs, rhs))
    for each  $\theta'$  in FOL-BC-AND(KB, lhs, UNIFY(rhs, goal,  $\theta$ )) do
      yield  $\theta'$ 



---


generator FOL-BC-AND(KB, goals,  $\theta$ ) yields a substitution
  if  $\theta = \text{failure}$  then return
  else if LENGTH(goals) = 0 then yield  $\theta$ 
  else do
    first, rest  $\leftarrow$  FIRST(goals), REST(goals)
    for each  $\theta'$  in FOL-BC-OR(KB, SUBST( $\theta$ , first),  $\theta$ ) do
      for each  $\theta''$  in FOL-BC-AND(KB, rest,  $\theta'$ ) do
        yield  $\theta''$ 

```

Figure 9.6 A simple backward-chaining algorithm for first-order knowledge bases.

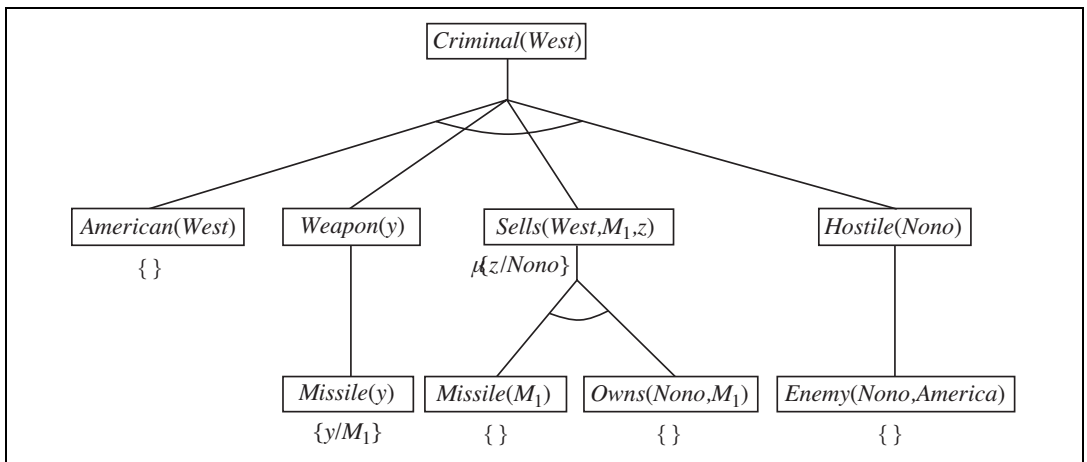


Figure 9.7 Proof tree constructed by backward chaining to prove that West is a criminal. The tree should be read depth first, left to right. To prove *Criminal*(*West*), we have to prove the four conjuncts below it. Some of these are in the knowledge base, and others require further backward chaining. 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*.

9.4.2 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,

$$\text{Algorithm} = \text{Logic} + \text{Control}.$$

PROLOG

Prolog is the most widely used logic programming language. 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—the opposite of our convention for logic. Commas separate conjuncts in a clause, and the clause is written “backwards” from what we are used to; instead of $A \wedge B \Rightarrow C$ in Prolog we have $C :- A, B$. Here is a typical example:

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

The notation $[E|L]$ denotes a list whose first element is E and whose rest is L . 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 . In most high-level languages we can write a similar recursive function that describes how to append two lists. The Prolog definition is actually much more powerful, however, because it describes a *relation* that holds among three arguments, rather than a *function* computed from two arguments. For example, we can ask the query `append(X,Y,[1,2])`: what two lists can be appended to give $[1,2]$? We get back the solutions

```
X=[ ]      Y=[1,2];
X=[1]     Y=[2];
X=[1,2]   Y=[ ]
```

The execution of Prolog programs is done through 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:

- Prolog uses the database semantics of Section 8.2.8 rather than first-order semantics, and this is apparent in its treatment of equality and negation (see Section 9.4.5).
- 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 confusing results—for example, if facts are asserted in a branch of the proof tree that eventually fails.
- The **occur check** is omitted from Prolog’s unification algorithm. This means that some unsound inferences can be made; these are almost never a problem in practice.
- Prolog uses depth-first backward-chaining search with no checks for infinite recursion. This makes it very fast when given the right set of axioms, but incomplete when given the wrong ones.

Prolog’s design represents a compromise between declarativeness and execution efficiency—inasmuch as efficiency was understood at the time Prolog was designed.

9.4.3 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, our implementation had to explicitly manage the iteration over possible results generated by each of the subfunctions. Prolog interpreters have a global data structure, a stack of **choice points**, to keep track of the multiple possibilities that we considered in FOL-BC-OR. This global stack is more efficient, and it makes debugging easier, because the debugger can move up and down the stack.

Second, our simple implementation of FOL-BC-ASK spends a good deal of time generating substitutions. Instead of explicitly constructing substitutions, Prolog has logic variables that 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.

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

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, x, z ← NEW-VARIABLE(), NEW-VARIABLE(), 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 used so far. The procedure **CALL**(*continuation*) continues execution with the specified continuation.

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).)

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 implementers 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. Several points are 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**.
- 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 and the second argument unifies with the third, 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.

OPEN-CODE

CONTINUATION

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.

OR-PARALLELISM

AND-PARALLELISM

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

9.4.4 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

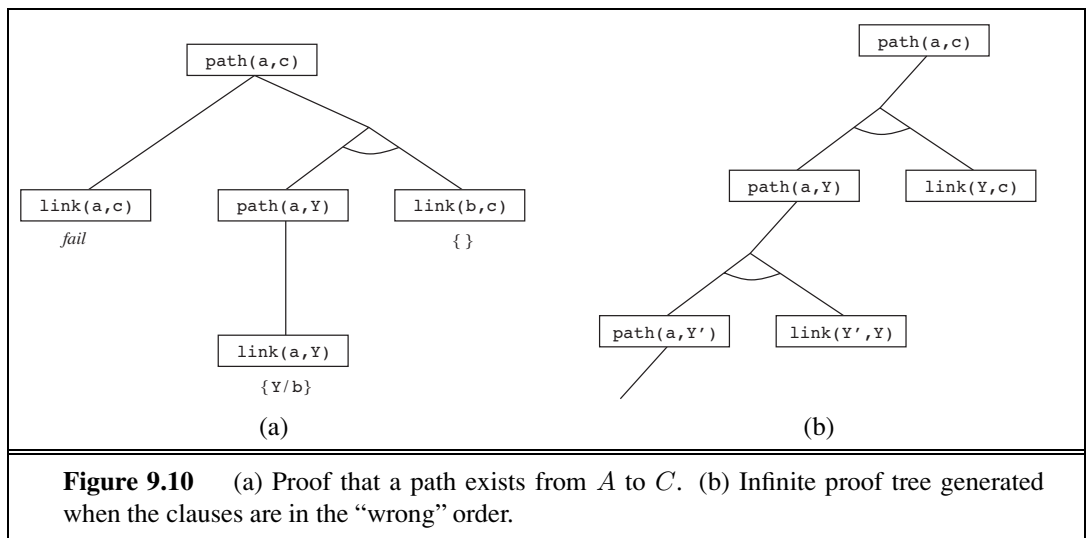
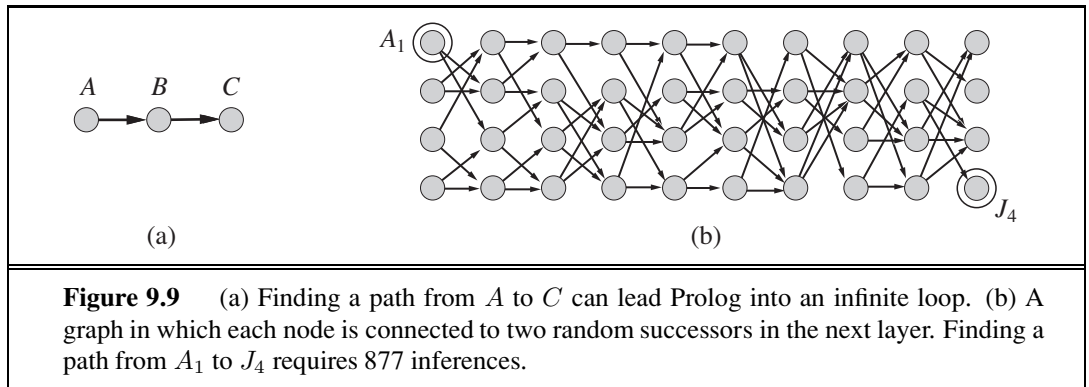
```
path(X,Z) :- path(X,Y), link(Y,Z).
path(X,Z) :- link(X,Z).
```

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 knowledge bases, which means that the programmer need worry less about infinite loops. (It is still possible to get an infinite loop with predicates like `father(X, Y)` that refer to a potentially unbounded number of objects.)

9.4.5 Database semantics of Prolog

Prolog uses database semantics, as discussed in Section 8.2.8. The unique names assumption says that every Prolog constant and every ground term refers to a distinct object, and the closed world assumption says that the only sentences that are true are those that are entailed

by the knowledge base. There is no way to assert that a sentence is false in Prolog. This makes Prolog less expressive than first-order logic, but it is part of what makes Prolog more efficient and more concise. Consider the following Prolog assertions about some course offerings:

$$\text{Course}(CS, 101), \text{Course}(CS, 102), \text{Course}(CS, 106), \text{Course}(EE, 101). \quad (9.11)$$

Under the unique names assumption, *CS* and *EE* are different (as are 101, 102, and 106), so this means that there are four distinct courses. Under the closed-world assumption there are no other courses, so there are exactly four courses. But if these were assertions in FOL rather than in Prolog, then all we could say is that there are somewhere between one and infinity courses. That's because the assertions (in FOL) 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. If we wanted to translate Equation (9.11) into FOL, we would get this:

$$\begin{aligned} \text{Course}(d, n) \quad \Leftrightarrow \quad & (d = CS \wedge n = 101) \vee (d = CS \wedge n = 102) \\ & \vee (d = CS \wedge n = 106) \vee (d = EE \wedge n = 101). \end{aligned} \quad (9.12)$$

COMPLETION

This is called the **completion** of Equation (9.11). It expresses in FOL the idea that there are at most four courses. To express in FOL the idea that there are at least four courses, we need to write the completion of the equality predicate:

$$\begin{aligned} x = y \quad \Leftrightarrow \quad & (x = CS \wedge y = CS) \vee (x = EE \wedge y = EE) \vee (x = 101 \wedge y = 101) \\ & \vee (x = 102 \wedge y = 102) \vee (x = 106 \wedge y = 106). \end{aligned}$$

The completion is useful for understanding database semantics, but for practical purposes, if your problem can be described with database semantics, it is more efficient to reason with Prolog or some other database semantics system, rather than translating into FOL and reasoning with a full FOL theorem prover.

9.4.6 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 6.5.

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.

Consider the following example. We define `triangle(X, Y, Z)` as a predicate that holds if the three arguments are numbers that 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)`, it succeeds. 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; we can't compare an unbound value to 0.

Constraint logic programming (CLP) allows variables to be *constrained* rather than *bound*. A CLP solution 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 $7 \geq Z \geq 1$. Standard logic programs are just a special case of CLP in which the solution constraints must be equality constraints—that is, bindings.

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 6, including heuristic conjunct ordering, backjumping, cutset conditioning, and so on. CLP systems therefore combine elements of constraint satisfaction algorithms, logic programming, and deductive databases.

Several systems that allow the programmer more control over the search order for inference have been defined. 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.

9.5 RESOLUTION

The last of our three families of logical systems is based on **resolution**. We saw on page 250 that propositional resolution using refutation is a complete inference procedure for propositional logic. In this section, we describe how to extend resolution to first-order logic.

9.5.1 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

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

becomes, in CNF,

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



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.

⁶ A clause can also be represented as an implication with a conjunction of atoms in the premise and a disjunction of atoms in the conclusion (Exercise 7.13). This is called **implicative normal form** or **Kowalski form** (especially when written with a right-to-left implication symbol (Kowalski, 1979)) and is often much easier to read.

The procedure for conversion to CNF is similar to the propositional case, which we saw on page 253. The principal difference arises from the need to eliminate existential quantifiers. We illustrate the procedure by translating the sentence “Everyone who loves all animals is loved by someone,” or

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

The steps are as follows:

- **Eliminate implications:**

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

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

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

Our sentence goes through the following transformations:

$$\begin{aligned} & \forall x [\exists y \neg(\neg \text{Animal}(y) \vee \text{Loves}(x, y))] \vee [\exists y \text{ Loves}(y, x)] . \\ & \forall x [\exists y \neg \neg \text{Animal}(y) \wedge \neg \text{Loves}(x, y)] \vee [\exists y \text{ Loves}(y, x)] . \\ & \forall x [\exists y \text{ Animal}(y) \wedge \neg \text{Loves}(x, y)] \vee [\exists y \text{ Loves}(y, x)] . \end{aligned}$$

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 $(\exists x P(x)) \vee (\exists x Q(x))$ 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

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

- **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. However, we can’t apply Existential Instantiation to our sentence above because it doesn’t match the pattern $\exists v \alpha$; only parts of the sentence match the pattern. If we blindly apply the rule to the two matching parts we get

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

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 and z :

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

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.

SKOLEMIZATION

SKOLEM FUNCTION

- **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:

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

- **Distribute \vee over \wedge :**

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

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 $G(z)$ refers to someone who might love x .) Fortunately, humans seldom need look at CNF sentences—the translation process is easily automated.

9.5.2 The resolution inference rule

The resolution rule for first-order clauses is simply a lifted version of the propositional resolution rule given on page 253. 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

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

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

This rule is called 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.

9.5.3 Example proofs

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

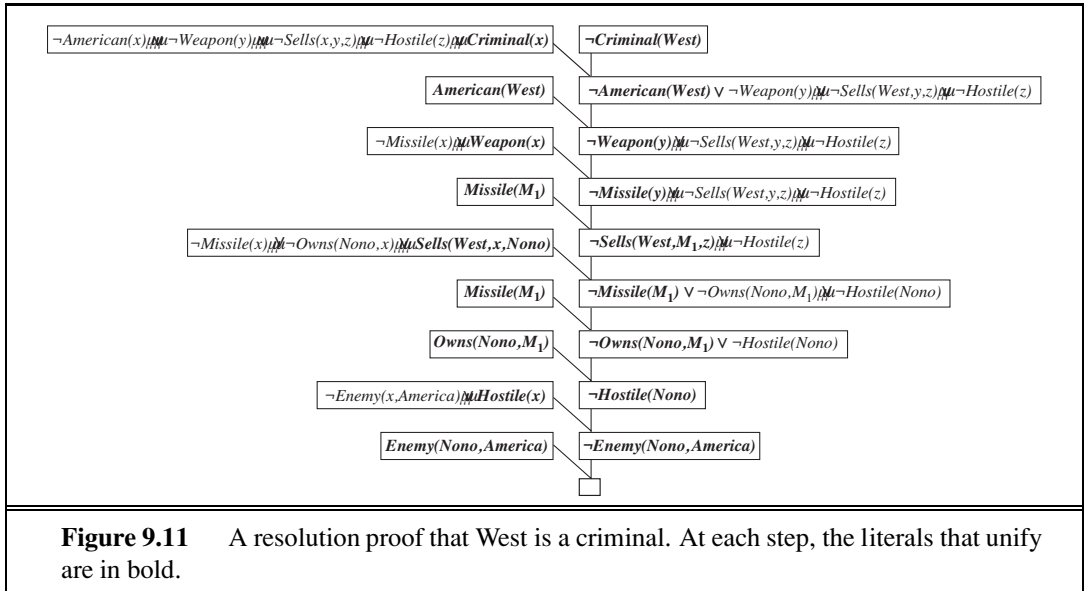


Figure 7.12, so we need not repeat it here. Instead, we give two example proofs. The first is the crime example from Section 9.3. The sentences in CNF are

$$\begin{aligned}
 &\neg American(x) \vee \neg Weapon(y) \vee \neg Sells(x, y, z) \vee \neg Hostile(z) \vee Criminal(x) \\
 &\neg Missile(x) \vee \neg Owns(Nono, x) \vee Sells(West, x, Nono) \\
 &\neg Enemy(x, America) \vee Hostile(x) \\
 &\neg Missile(x) \vee Weapon(x) \\
 &Owns(Nono, M_1) \\
 &American(West) \\
 &Missile(M_1) \\
 &Enemy(Nono, America) .
 \end{aligned}$$

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 choose 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 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. $\forall x [\forall y \text{ Animal}(y) \Rightarrow \text{Loves}(x, y)] \Rightarrow [\exists y \text{ Loves}(y, x)]$
- B. $\forall x [\exists z \text{ Animal}(z) \wedge \text{Kills}(x, z)] \Rightarrow [\forall y \neg \text{Loves}(y, x)]$
- C. $\forall x \text{ Animal}(x) \Rightarrow \text{Loves}(\text{Jack}, x)$
- D. $\text{Kills}(\text{Jack}, \text{Tuna}) \vee \text{Kills}(\text{Curiosity}, \text{Tuna})$
- E. $\text{Cat}(\text{Tuna})$
- F. $\forall x \text{ Cat}(x) \Rightarrow \text{Animal}(x)$
- \neg G. $\neg \text{Kills}(\text{Curiosity}, \text{Tuna})$

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

- A1. $\text{Animal}(F(x)) \vee \text{Loves}(G(x), x)$
- A2. $\neg \text{Loves}(x, F(x)) \vee \text{Loves}(G(x), x)$
- B. $\neg \text{Loves}(y, x) \vee \neg \text{Animal}(z) \vee \neg \text{Kills}(x, z)$
- C. $\neg \text{Animal}(x) \vee \text{Loves}(\text{Jack}, x)$
- D. $\text{Kills}(\text{Jack}, \text{Tuna}) \vee \text{Kills}(\text{Curiosity}, \text{Tuna})$
- E. $\text{Cat}(\text{Tuna})$
- F. $\neg \text{Cat}(x) \vee \text{Animal}(x)$
- \neg G. $\neg \text{Kills}(\text{Curiosity}, \text{Tuna})$

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.

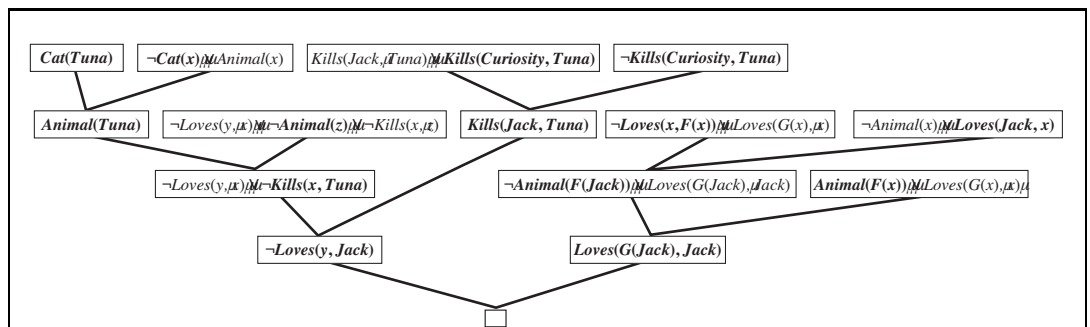


Figure 9.12 A resolution proof that Curiosity killed the cat. Notice the use of factoring in the derivation of the clause $\text{Loves}(G(\text{Jack}), \text{Jack})$. Notice also in the upper right, the unification of $\text{Loves}(x, F(x))$ and $\text{Loves}(\text{Jack}, x)$ can only succeed after the variables have been standardized apart.

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 more work to obtain the answer. The goal is $\exists w \text{ Kills}(w, \text{Tuna})$, which, when negated, becomes $\neg \text{Kills}(w, \text{Tuna})$ in CNF. Repeating the proof in Figure 9.12 with the new negated goal, we obtain a similar proof tree, but with the substitution $\{w/\text{Curiosity}\}$ 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, $\neg \text{Kills}(w, \text{Tuna})$ resolves with $\text{Kills}(\text{Jack}, \text{Tuna}) \vee \text{Kills}(\text{Curiosity}, \text{Tuna})$ to give $\text{Kills}(\text{Jack}, \text{Tuna})$, which resolves again with $\neg \text{Kills}(w, \text{Tuna})$ 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 $\neg \text{Kills}(w, \text{Tuna}) \vee \text{Answer}(w)$. 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 $\text{Answer}(\text{Curiosity})$. The nonconstructive proof would generate the clause $\text{Answer}(\text{Curiosity}) \vee \text{Answer}(\text{Jack})$, which does not constitute an answer.

ANSWER LITERAL

9.5.4 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

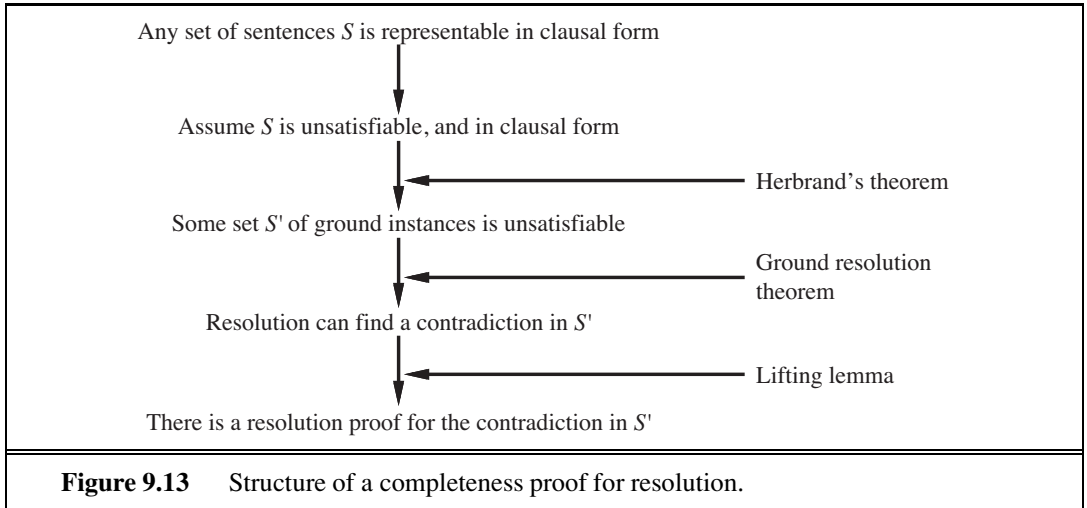
We show that resolution is **refutation-complete**, which means that *if* a set of sentences is unsatisfiable, then resolution will always be able to derive a contradiction. Resolution cannot be used to generate all logical consequences of a set of sentences, but it can be used to establish that a given sentence is entailed by the set of sentences. Hence, it can be used to find all answers to a given question, $Q(x)$, by proving that $KB \wedge \neg Q(x)$ is unsatisfiable.

We 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 Robinson’s original proof with some simplifications from Genesereth and Nilsson (1987). The basic structure of the proof (Figure 9.13) is 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.



To carry out the first step, we 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 constructable from the following:

- The function symbols in S , if any.
- 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:

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

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

$$\begin{aligned} &\{\neg P(A, F(A, A)) \vee \neg Q(A, A) \vee R(A, B), \\ &\quad \neg P(B, F(B, A)) \vee \neg Q(B, A) \vee R(B, B), \\ &\quad \neg P(F(A, A), F(F(A, A), A)) \vee \neg Q(F(A, A), A) \vee R(F(A, A), B), \\ &\quad \neg P(F(A, B), F(F(A, B), A)) \vee \neg Q(F(A, B), A) \vee R(F(A, B), B), \dots\} \end{aligned}$$

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 255) 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, Kurt 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 stated this lemma:

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.

9.5.5 Equality

None of the inference methods described so far in this chapter handle an assertion of the form $x = y$. Three distinct approaches 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:

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

Given these sentences, a standard inference procedure such as resolution can perform tasks requiring equality reasoning, such as solving mathematical equations. However, these axioms will generate a lot of conclusions, most of them not helpful to a proof. So there has been a search for more efficient ways of handling equality. One alternative is to add inference rules rather than axioms. The simplest rule, **demodulation**, takes a unit clause $x = y$ and some clause α that contains the term x , and yields a new clause formed by substituting y for x within α . It works if the term within α unifies with x ; it need not be exactly equal to x . Note that demodulation is directional; given $x = y$, the x always gets replaced with y , never vice versa. That means that demodulation can be used for simplifying expressions using demodulators such as $x + 0 = x$ or $x^1 = x$. As another example, given

$$\begin{aligned}
 &Father(Father(x)) = PaternalGrandfather(x) \\
 &Birthdate(Father(Father(Bella)), 1926)
 \end{aligned}$$

we can conclude by demodulation

$$Birthdate(PaternalGrandfather(Bella), 1926) .$$

More formally, we have

DEMODULATION

- **Demodulation:** For any terms x , y , and z , where z appears somewhere in literal m_i and where $\text{UNIFY}(x, z) = \theta$,

$$\frac{x = y, \quad m_1 \vee \cdots \vee m_n}{\text{SUB}(\text{SUBST}(\theta, x), \text{SUBST}(\theta, y), m_1 \vee \cdots \vee m_n)} .$$

where SUBST is the usual substitution of a binding list, and $\text{SUB}(x, y, m)$ means to replace x with y everywhere that x occurs within m .

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 z appears somewhere in literal m_i , and where $\text{UNIFY}(x, z) = \theta$,

$$\frac{\ell_1 \vee \cdots \vee \ell_k \vee x = y, \quad m_1 \vee \cdots \vee m_n}{\text{SUB}(\text{SUBST}(\theta, x), \text{SUBST}(\theta, y), \text{SUBST}(\theta, \ell_1 \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_n))} .$$

For example, from

$$P(F(x, B), x) \vee Q(x) \quad \text{and} \quad F(A, y) = y \vee R(y)$$

we have $\theta = \text{UNIFY}(F(A, y), F(x, B)) = \{x/A, y/B\}$, and we can conclude by paramodulation the sentence

$$P(B, A) \vee Q(A) \vee R(B) .$$

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 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 CLP systems described in Section 9.4.

EQUATIONAL
UNIFICATION

9.5.6 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

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 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 clauses. Unit resolution proofs on Horn clauses resemble forward chaining.

The OTTER theorem prover (Organized Techniques for Theorem-proving and Effective Research, McCune, 1992), 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.

SET OF SUPPORT

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. For example, we can insist that every resolution step involve at least one element of a special set of clauses—the *set of support*. The resolvent is then added 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 is complete. For example, one can 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 goal-directed proof trees that are often easy for humans to understand.

INPUT RESOLUTION

Input resolution: In this 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 on page 348 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 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.

LINEAR RESOLUTION

SUBSUMPTION

Subsumption: The subsumption method eliminates all sentences that are subsumed by (that is, 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.

Practical uses of resolution theorem provers

SYNTHESIS

VERIFICATION

Theorem provers can be applied to the problems involved in the **synthesis** and **verification** of both hardware and software. 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 hardware, the axioms describe the interactions between signals and circuit elements. (See Section 8.4.2 on page 309 for an example.) Logical reasoners designed specially for verification have been able to verify entire CPUs, including their timing properties (Srivasa and Bickford, 1990). The AURA theorem prover has been applied to design circuits that are more compact than any previous design (Wojciechowski and Wojcik, 1983).

DEDUCTIVE
SYNTHESIS

In the case of software, reasoning about programs is quite similar to reasoning about actions, as in Chapter 7: axioms describe the preconditions and effects of each statement. 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 Herbert Simon (1963). The idea is to constructively prove a theorem to the effect that “there exists a program p satisfying a certain specification.” 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, such as scientific computing code, is also an active area of research.

Similar techniques are now being applied to software verification 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). The RSA public key encryption algorithm and the Boyer–Moore string-matching algorithm have been verified this way (Boyer and Moore, 1984).

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 (**universal instantiation** and **existential instantiation**) to **propositionalize** the inference problem. Typically, this approach is slow, unless the domain is small.
- The use of **unification** to identify appropriate substitutions for variables eliminates the instantiation step in first-order proofs, making the process more efficient in many cases.
- 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** knowledge bases 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 and runs in polynomial time.
- Backward chaining is used in **logic programming systems**, 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**.
- Prolog, unlike first-order logic, uses a closed world with the unique names assumption and negation as failure. These make Prolog a more practical programming language, but bring it further from pure logic.
- 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. One of the most important issues is dealing with equality; we showed how **demodulation** and **paramodulation** can be used.
- 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

Gottlob Frege, who developed full first-order logic in 1879, based his system of inference on a collection of valid schemas plus a single inference rule, Modus Ponens. 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). Oddly enough, it was Skolem who introduced the Herbrand universe (Skolem, 1928).

Herbrand's theorem (Herbrand, 1930) has played a vital role in the development of automated reasoning. Herbrand is also 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 understandable fashion.

Abraham Robinson proposed that an automated reasoner could be built using propositionalization and Herbrand's theorem, and Paul Gilmore (1960) wrote the first program. Davis and Putnam (1960) introduced the propositionalization method of Section 9.1. Prawitz (1960) developed the key idea of letting the quest for propositional inconsistency drive the search, and generating terms from the Herbrand universe only when they were necessary to establish propositional inconsistency. After further development by other researchers, this idea led J. A. Robinson (no relation) to develop resolution (Robinson, 1965).

In AI, resolution was adopted for question-answering systems by Cordell Green and Bertram Raphael (1968). 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. 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), incorporating the efficient **rete** match process (Forgy, 1982), was used for applications such as the R1 expert system for minicomputer configuration (McDermott, 1982).

RETE

The SOAR cognitive architecture (Laird *et al.*, 1987; Laird, 2008) was designed to handle very large rule sets—up to a million rules (Doorenbos, 1994). Example applications of SOAR include controlling simulated fighter aircraft (Jones *et al.*, 1998), airspace management (Taylor *et al.*, 2007), AI characters for computer games (Wintermute *et al.*, 2007), and training tools for soldiers (Wray and Jones, 2005).

The field of **deductive databases** began with a workshop in Toulouse in 1977 that brought together experts in logical inference and database systems (Gallaire and Minker, 1978). Influential work by Chandra and Harel (1980) and Ullman (1985) led to the adoption of Datalog as a standard language for deductive databases. 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. Current work includes the idea of integrating multiple databases into a consistent dataspace (Halevy, 2007).

Backward chaining for logical inference appeared first in Hewitt's PLANNER language (1969). Meanwhile, in 1972, 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 Robert Kowalski, working

with Colmerauer; see Kowalski (1988) and Colmerauer and Roussel (1993) for a historical overview. 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 Prolog programs can be competitive with C programs in terms of speed.

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 work on constraint logic programming was done by Jaffar and Lassez (1987). Jaffar *et al.* (1992) developed the CLP(R) system for handling real-valued constraints. There are now commercial products for solving large-scale configuration and optimization problems with constraint programming; one of the best known is ILOG (Junker, 2003). Answer set programming (Gelfond, 2008) extends Prolog, allowing disjunction and negation.

Texts on logic programming and Prolog, including Shoham (1994), Bratko (2001), Clocksin (2003), and Clocksin and Mellish (2003). Prior to 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 Gelemter's Geometry Theorem Prover (Gelemter, 1959) used heuristic search methods combined with diagrams for pruning false subgoals and was able to prove some quite intricate results in Euclidean geometry. 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). Jouannaud and Kirchner (1991) survey equational unification from a term-rewriting perspective. An overview of unification is given by Baader and Snyder (2001).

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.

A *Computational Logic* (Boyer and Moore, 1979) is the basic reference on the Boyer-Moore theorem prover. Stickel (1992) covers the Prolog Technology Theorem Prover (PTTP), which combines the advantages of Prolog compilation with the completeness of model elimination. SETHEO (Letz *et al.*, 1992) is another widely used theorem prover based on this approach. LEANTAP (Beckert and Posegga, 1995) is an efficient theorem prover implemented in only 25 lines of Prolog. Weidenbach (2001) describes SPASS, one of the strongest current theorem provers. The most successful theorem prover in recent annual competitions has been VAMPIRE (Riazanov and Voronkov, 2002). The COQ system (Bertot *et al.*, 2004) and the E

equational solver (Schulz, 2004) have also proven to be valuable tools for proving correctness. Theorem provers have been used to automatically synthesize and verify software for controlling spacecraft (Denney *et al.*, 2006), including NASA's new Orion capsule (Lowry, 2008). The design of the FM9001 32-bit microprocessor was proved correct by the NQTHM system (Hunt and Brock, 1992). The Conference on Automated Deduction (CADE) runs an annual contest for automated theorem provers. From 2002 through 2008, the most successful system has been VAMPIRE (Riazanov and Voronkov, 2002). Wiedijk (2003) compares the strength of 15 mathematical provers. TPTP (Thousands of Problems for Theorem Provers) is a library of theorem-proving problems, useful for comparing the performance of systems (Sutcliffe and Suttner, 1998; Sutcliffe *et al.*, 2006).

Theorem provers have come up with novel mathematical results that eluded human mathematicians for decades, as detailed in the book *Automated Reasoning and the Discovery of Missing Elegant Proofs* (Wos and Pieper, 2003). 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) was used by Natarajan Shankar to give the first fully rigorous formal proof of Gödel's Incompleteness Theorem (Shankar, 1986). The NUPRL system proved Girard's paradox (Howe, 1987) and Higman's Lemma (Murthy and Russell, 1990). In 1933, Herbert Robbins proposed a simple set of axioms—the **Robbins algebra**—that appeared to define Boolean algebra, but no proof could be found (despite serious work by Alfred Tarski and others). On October 10, 1996, after eight days of computation, EQP (a version of OTTER) found a proof (McCune, 1997).

Many early papers in mathematical logic are to be found in *From Frege to Gödel: A Source Book in Mathematical Logic* (van Heijenoort, 1967). 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 Duffy (1991), Wos *et al.* (1992), Bibel (1993), and Kaufmann *et al.* (2000). The principal journal for theorem proving is the *Journal of Automated Reasoning*; the main conferences are the annual Conference on Automated Deduction (CADE) and the International Joint Conference on Automated Reasoning (IJCAR). The *Handbook of Automated Reasoning* (Robinson and Voronkov, 2001) collects papers in the field. MacKenzie's *Mechanizing Proof* (2004) covers the history and technology of theorem proving for the popular audience.

ROBBINS ALGEBRA

EXERCISES

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

9.2 From $Likes(Jerry, IceCream)$ it seems reasonable to infer $\exists x \text{ Likes}(x, IceCream)$. 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.

EXISTENTIAL
INTRODUCTION

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

- a. $\text{AsHighAs}(\text{Everest}, \text{Everest})$.
- b. $\text{AsHighAs}(\text{Kilimanjaro}, \text{Everest})$.
- c. $\text{AsHighAs}(\text{Kilimanjaro}, \text{Everest}) \wedge \text{AsHighAs}(\text{BenNevis}, \text{Everest})$
(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. $\text{Older}(\text{Father}(y), y), \text{Older}(\text{Father}(x), \text{John})$.
- d. $\text{Knows}(\text{Father}(y), y), \text{Knows}(x, x)$.

9.5 Consider the subsumption lattices shown in Figure 9.2 (page 329).

- a. Construct the lattice for the sentence $\text{Employs}(\text{Mother}(\text{John}), \text{Father}(\text{Richard}))$.
- b. Construct the lattice for the sentence $\text{Employs}(\text{IBM}, y)$ (“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 $\text{Employs}(x, \text{Father}(x))$.

9.6 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.

9.7 These questions concern issues with substitution and Skolemization.

- a. Given the premise $\forall x \exists y P(x, y)$, it is not valid to conclude that $\exists q P(q, q)$. Give an example of a predicate P where the first is true but the second is false.
- b. Suppose that an inference engine is incorrectly written with the occurs check omitted, so that it allows a literal like $P(x, F(x))$ to be unified with $P(q, q)$. (As mentioned, most standard implementations of Prolog actually do allow this.) Show that such an inference engine will allow the conclusion $\exists y P(q, q)$ to be inferred from the premise $\forall x \exists y P(x, y)$.

- c. Suppose that a procedure that converts first-order logic to clausal form incorrectly Skolemizes $\forall x \exists y P(x, y)$ to $P(x, Sk0)$ —that is, it replaces y by a Skolem constant rather than by a Skolem function of x . Show that an inference engine that uses such a procedure will likewise allow $\exists q P(q, q)$ to be inferred from the premise $\forall x \exists y P(x, y)$.
- d. A common error among students is to suppose that, in unification, one is allowed to substitute a term for a Skolem constant instead of for a variable. For instance, they will say that the formulas $P(Sk1)$ and $P(A)$ can be unified under the substitution $\{Sk1/A\}$. Give an example where this leads to an invalid inference.

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 Suppose you are given the following axioms:

1. $0 \leq 3$.
2. $7 \leq 9$.
3. $\forall x \quad x \leq x$.
4. $\forall x \quad x \leq x + 0$.
5. $\forall x \quad x + 0 \leq x$.
6. $\forall x, y \quad x + y \leq y + x$.
7. $\forall w, x, y, z \quad w \leq y \wedge x \leq z \Rightarrow w + x \leq y + z$.
8. $\forall x, y, z \quad x \leq y \wedge y \leq z \Rightarrow x \leq z$.

- a. Give a backward-chaining proof of the sentence $7 \leq 3 + 9$. (Be sure, of course, to use only the axioms given here, not anything else you may know about arithmetic.) Show only the steps that leads to success, not the irrelevant steps.
- b. Give a forward-chaining proof of the sentence $7 \leq 3 + 9$. Again, show only the steps that lead to success.

9.10 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 (Section 8.3.2 on page 301) 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?

9.11 Suppose we put into a logical knowledge base 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 following indexing schemes S1–S5 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.
- **Q1**: $\text{Age}(443\text{-}44\text{-}4321, x)$
- **Q2**: $\text{ResidesIn}(x, \text{Houston})$
- **Q3**: $\text{Mother}(x, y)$
- **Q4**: $\text{Age}(x, 34) \wedge \text{ResidesIn}(x, \text{TinyTownUSA})$

9.12 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 in which one part unifies with another.)

9.13 In this exercise, use the sentences you wrote in Exercise 9.6 to answer a question by using a backward-chaining algorithm.

- a. Draw the proof tree generated by an exhaustive backward-chaining algorithm for the query $\exists h \text{ Horse}(h)$, 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*: See Smith *et al.* (1986).)

9.14 Trace the execution of the backward-chaining algorithm in Figure 9.6 (page 338) when it is applied to solve the crime problem (page 330). Show the sequence of values taken on by the *goals* variable, and arrange them into a tree.

9.15 The following Prolog code defines a predicate *P*. (Remember that uppercase terms are variables, not constants, in Prolog.)

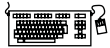
```
P(X, [X|Y]) .
P(X, [Y|Z]) :- P(X, Z) .
```

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



9.16 This exercise looks 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.



9.17 This exercise looks 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 . Rewrite rules are a key component of equational reasoning systems. 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 write `primitive(0)` to say that 0 is a primitive term.

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

9.18 This exercise considers 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?

9.19 Suppose a knowledge base contains just the following first-order Horn clauses:

$$\begin{aligned} & \text{Ancestor}(\text{Mother}(x), x) \\ & \text{Ancestor}(x, y) \wedge \text{Ancestor}(y, z) \Rightarrow \text{Ancestor}(x, z) \end{aligned}$$

Consider a forward chaining algorithm that, on the j th iteration, terminates if the KB contains a sentence that unifies with the query, else adds to the KB every atomic sentence that can be inferred from the sentences already in the KB after iteration $j - 1$.

- For each of the following queries, say whether the algorithm will (1) give an answer (if so, write down that answer); or (2) terminate with no answer; or (3) never terminate.
 - $\text{Ancestor}(\text{Mother}(y), \text{John})$
 - $\text{Ancestor}(\text{Mother}(\text{Mother}(y)), \text{John})$
 - $\text{Ancestor}(\text{Mother}(\text{Mother}(\text{Mother}(y))), \text{Mother}(y))$
 - $\text{Ancestor}(\text{Mother}(\text{John}), \text{Mother}(\text{Mother}(\text{John})))$
- Can a resolution algorithm prove the sentence $\neg \text{Ancestor}(\text{John}, \text{John})$ from the original knowledge base? Explain how, or why not.
- Suppose we add the assertion that $\neg(\text{Mother}(x) = x)$ and augment the resolution algorithm with inference rules for equality. Now what is the answer to (b)?

9.20 Let \mathcal{L} be the first-order language with a single predicate $S(p, q)$, meaning “ p shaves q .” Assume a domain of people.

- a. Consider the sentence “There exists a person P who shaves every one who does not shave themselves, and only people that do not shave themselves.” Express this in \mathcal{L} .
- b. Convert the sentence in (a) to clausal form.
- c. Construct a resolution proof to show that the clauses in (b) are inherently inconsistent. (Note: you do not need any additional axioms.)

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

9.22 Construct an example of two clauses that can be resolved together in two different ways giving two different outcomes.

9.23 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.

9.24 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 $0, 1, 2, \dots, \infty$ 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).

9.25 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.

9.26 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 CLASSICAL PLANNING

In which we see how an agent can take advantage of the structure of a problem to construct complex plans of action.

We have defined AI as the study of rational action, which means that **planning**—devising a plan of action to achieve one’s goals—is a critical part of AI. We have seen two examples of planning agents so far: the search-based problem-solving agent of Chapter 3 and the hybrid logical agent of Chapter 7. In this chapter we introduce a representation for planning problems that scales up to problems that could not be handled by those earlier approaches.

Section 10.1 develops an expressive yet carefully constrained language for representing planning problems. Section 10.2 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 domain-independent heuristics were constructed for constraint satisfaction problems in Chapter 6.) Section 10.3 shows how a data structure called the planning graph can make the search for a plan more efficient. We then describe a few of the other approaches to planning, and conclude by comparing the various approaches.

This chapter covers fully observable, deterministic, static environments with single agents. Chapters 11 and 17 cover partially observable, stochastic, dynamic environments with multiple agents.

10.1 DEFINITION OF CLASSICAL PLANNING

The problem-solving agent of Chapter 3 can find sequences of actions that result in a goal state. But it deals with atomic representations of states and thus needs good domain-specific heuristics to perform well. The hybrid propositional logical agent of Chapter 7 can find plans without domain-specific heuristics because it uses domain-independent heuristics based on the logical structure of the problem. But it relies on ground (variable-free) propositional inference, which means that it may be swamped when there are many actions and states. For example, in the wumpus world, the simple action of moving a step forward had to be repeated for all four agent orientations, T time steps, and n^2 current locations.

PDDL

In response to this, planning researchers have settled on a **factored representation**—one in which a state of the world is represented by a collection of variables. We use a language called **PDDL**, the Planning Domain Definition Language, that allows us to express all $4Tn^2$ actions with one action schema. There have been several versions of PDDL; we select a simple version and alter its syntax to be consistent with the rest of the book.¹ We now show how PDDL describes the four things we need to define a search problem: the initial state, the actions that are available in a state, the result of applying an action, and the goal test.

SET SEMANTICS

Each **state** is represented as a conjunction of fluents that are ground, functionless atoms. For example, $Poor \wedge Unknown$ might represent the state of a hapless agent, and a state in a package delivery problem might be $At(Truck_1, Melbourne) \wedge At(Truck_2, Sydney)$. **Database semantics** is used: the closed-world assumption means that any fluents that are not mentioned are false, and the unique names assumption means that $Truck_1$ and $Truck_2$ are distinct. The following fluents are *not* allowed in a state: $At(x, y)$ (because it is non-ground), $\neg Poor$ (because it is a negation), and $At(Father(Fred), Sydney)$ (because it uses a function symbol). The representation of states is carefully designed so that a state can be treated either as a conjunction of fluents, which can be manipulated by logical inference, or as a *set* of fluents, which can be manipulated with set operations. The **set semantics** is sometimes easier to deal with.

ACTION SCHEMA

Actions are described by a set of action schemas that implicitly define the $ACTIONS(s)$ and $RESULT(s, a)$ functions needed to do a problem-solving search. We saw in Chapter 7 that any system for action description needs to solve the frame problem—to say what changes and what stays the same as the result of the action. Classical planning concentrates on problems where most actions leave most things unchanged. Think of a world consisting of a bunch of objects on a flat surface. The action of nudging an object causes that object to change its location by a vector Δ . A concise description of the action should mention only Δ ; it shouldn't have to mention all the objects that stay in place. PDDL does that by specifying the result of an action in terms of what changes; everything that stays the same is left unmentioned.

A set of ground (variable-free) actions can be represented by a single **action schema**. The schema is a **lifted** representation—it lifts the level of reasoning from propositional logic to a restricted subset of first-order logic. For example, here is an action schema for flying a plane from one location to another:

$$\begin{aligned} &Action(Fly(p, from, to), \\ &\quad PRECOND: At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to) \\ &\quad EFFECT: \neg At(p, from) \wedge At(p, to)) \end{aligned}$$

PRECONDITION

EFFECT

The schema consists of the action name, a list of all the variables used in the schema, a **precondition** and an **effect**. Although we haven't said yet how the action schema converts into logical sentences, think of the variables as being universally quantified. We are free to choose whatever values we want to instantiate the variables. For example, here is one ground

¹ PDDL was derived from the original STRIPS planning language (Fikes and Nilsson, 1971), which is slightly more restricted than PDDL: STRIPS preconditions and goals cannot contain negative literals.

action that results from substituting values for all the variables:

$$\begin{aligned} & \text{Action}(\text{Fly}(P_1, \text{SFO}, \text{JFK}), \\ & \quad \text{PRECOND: } \text{At}(P_1, \text{SFO}) \wedge \text{Plane}(P_1) \wedge \text{Airport}(\text{SFO}) \wedge \text{Airport}(\text{JFK}) \\ & \quad \text{EFFECT: } \neg \text{At}(P_1, \text{SFO}) \wedge \text{At}(P_1, \text{JFK})) \end{aligned}$$

The precondition and effect of an action are each conjunctions of literals (positive or negated atomic sentences). The precondition defines the states in which the action can be executed, and the effect defines the result of executing the action. An action a can be executed in state s if s entails the precondition of a . Entailment can also be expressed with the set semantics: $s \models q$ iff every positive literal in q is in s and every negated literal in q is not. In formal notation we say

$$(a \in \text{ACTIONS}(s)) \Leftrightarrow s \models \text{PRECOND}(a),$$

where any variables in a are universally quantified. For example,

$$\begin{aligned} & \forall p, \text{from}, \text{to} \quad (\text{Fly}(p, \text{from}, \text{to}) \in \text{ACTIONS}(s)) \Leftrightarrow \\ & \quad s \models (\text{At}(p, \text{from}) \wedge \text{Plane}(p) \wedge \text{Airport}(\text{from}) \wedge \text{Airport}(\text{to})) \end{aligned}$$

APPLICABLE

We say that action a is **applicable** in state s if the preconditions are satisfied by s . When an action schema a contains variables, it may have multiple applicable instantiations. For example, with the initial state defined in Figure 10.1, the *Fly* action can be instantiated as *Fly*($P_1, \text{SFO}, \text{JFK}$) or as *Fly*($P_2, \text{JFK}, \text{SFO}$), both of which are applicable in the initial state. If an action a has v variables, then, in a domain with k unique names of objects, it takes $O(v^k)$ time in the worst case to find the applicable ground actions.

PROPOSITIONALIZE

Sometimes we want to **propositionalize** a PDDL problem—replace each action schema with a set of ground actions and then use a propositional solver such as SATPLAN to find a solution. However, this is impractical when v and k are large.

DELETE LIST

ADD LIST

The **result** of executing action a in state s is defined as a state s' which is represented by the set of fluents formed by starting with s , removing the fluents that appear as negative literals in the action's effects (what we call the **delete list** or $\text{DEL}(a)$), and adding the fluents that are positive literals in the action's effects (what we call the **add list** or $\text{ADD}(a)$):

$$\text{RESULT}(s, a) = (s - \text{DEL}(a)) \cup \text{ADD}(a). \quad (10.1)$$

For example, with the action *Fly*($P_1, \text{SFO}, \text{JFK}$), we would remove $\text{At}(P_1, \text{SFO})$ and add $\text{At}(P_1, \text{JFK})$. It is a requirement of action schemas that any variable in the effect must also appear in the precondition. That way, when the precondition is matched against the state s , all the variables will be bound, and $\text{RESULT}(s, a)$ will therefore have only ground atoms. In other words, ground states are closed under the **RESULT** operation.

Also note that the fluents do not explicitly refer to time, as they did in Chapter 7. There we needed superscripts for time, and successor-state axioms of the form

$$F^{t+1} \Leftrightarrow \text{ActionCauses}F^t \vee (F^t \wedge \neg \text{ActionCausesNot}F^t).$$

In PDDL the times and states are implicit in the action schemas: the precondition always refers to time t and the effect to time $t + 1$.

A set of action schemas serves as a definition of a planning *domain*. A specific *problem* within the domain is defined with the addition of an initial state and a goal. The **initial**

```

Init(At(C1, SFO) ∧ At(C2, JFK) ∧ At(P1, SFO) ∧ At(P2, JFK)
    ∧ Cargo(C1) ∧ Cargo(C2) ∧ Plane(P1) ∧ Plane(P2)
    ∧ Airport(JFK) ∧ Airport(SFO))
Goal(At(C1, JFK) ∧ At(C2, SFO))
Action(Load(c, p, a),
    PRECOND: At(c, a) ∧ At(p, a) ∧ Cargo(c) ∧ Plane(p) ∧ Airport(a)
    EFFECT: ¬ At(c, a) ∧ In(c, p))
Action(Unload(c, p, a),
    PRECOND: In(c, p) ∧ At(p, a) ∧ Cargo(c) ∧ Plane(p) ∧ Airport(a)
    EFFECT: At(c, a) ∧ ¬ In(c, p))
Action(Fly(p, from, to),
    PRECOND: At(p, from) ∧ Plane(p) ∧ Airport(from) ∧ Airport(to)
    EFFECT: ¬ At(p, from) ∧ At(p, to))

```

Figure 10.1 A PDDL description of an air cargo transportation planning problem.

INITIAL STATE

GOAL

state is a conjunction of ground atoms. (As with all states, the closed-world assumption is used, which means that any atoms that are not mentioned are false.) The **goal** is just like a precondition: a conjunction of literals (positive or negative) that may contain variables, such as *At*(*p*, *SFO*) ∧ *Plane*(*p*). Any variables are treated as existentially quantified, so this goal is to have *any* plane at SFO. The problem is solved when we can find a sequence of actions that end in a state *s* that entails the goal. For example, the state *Rich* ∧ *Famous* ∧ *Miserable* entails the goal *Rich* ∧ *Famous*, and the state *Plane*(*Plane*₁) ∧ *At*(*Plane*₁, *SFO*) entails the goal *At*(*p*, *SFO*) ∧ *Plane*(*p*).

Now we have defined planning as a search problem: we have an initial state, an ACTIONS function, a RESULT function, and a goal test. We'll look at some example problems before investigating efficient search algorithms.

10.1.1 Example: Air cargo transport

Figure 10.1 shows an air cargo transport problem involving loading and unloading cargo 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 some care must be taken to make sure the *At* predicates are maintained properly. When a plane flies from one airport to another, all the cargo inside the plane goes with it. In first-order logic it would be easy to quantify over all objects that are inside the plane. But basic PDDL does not have a universal quantifier, so we need a different solution. The approach we use is to say that a piece of cargo ceases to be *At* anywhere when it is *In* a plane; the cargo only becomes *At* the new airport when it is unloaded. So *At* really means “available for use at a given location.” The following plan is a solution to the problem:

```

[Load(C1, P1, SFO), Fly(P1, SFO, JFK), Unload(C1, P1, JFK),
 Load(C2, P2, JFK), Fly(P2, JFK, SFO), Unload(C2, P2, SFO)] .

```


Finally, there is the problem of spurious actions such as $Fly(P_1, JFK, JFK)$, which should be a no-op, but which has contradictory effects (according to the definition, the effect would include $At(P_1, JFK) \wedge \neg At(P_1, JFK)$). It is common to ignore such problems, because they seldom cause incorrect plans to be produced. The correct approach is to add inequality preconditions saying that the *from* and *to* airports must be different; see another example of this in Figure 10.3.

10.1.2 Example: The spare tire problem

Consider the problem of changing a flat tire (Figure 10.2). 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 an 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 parked in a particularly bad neighborhood, so that the effect of leaving it overnight is that the tires disappear. A solution to the problem is $[Remove(Flat, Axle), Remove(Spare, Trunk), PutOn(Spare, Axle)]$.

```

Init(Tire(Flat)  $\wedge$  Tire(Spare)  $\wedge$  At(Flat, Axle)  $\wedge$  At(Spare, Trunk))
Goal(At(Spare, Axle))
Action(Remove(obj, loc),
  PRECOND: At(obj, loc)
  EFFECT:  $\neg At(obj, loc) \wedge At(obj, Ground)$ )
Action(PutOn(t, Axle),
  PRECOND: Tire(t)  $\wedge At(t, Ground) \wedge \neg At(Flat, Axle)$ 
  EFFECT:  $\neg At(t, Ground) \wedge At(t, Axle)$ )
Action(LeaveOvernight,
  PRECOND:
  EFFECT:  $\neg At(Spare, Ground) \wedge \neg At(Spare, Axle) \wedge \neg At(Spare, Trunk)$ 
          $\wedge \neg At(Flat, Ground) \wedge \neg At(Flat, Axle) \wedge \neg At(Flat, Trunk)$ )

```

Figure 10.2 The simple spare tire problem.

10.1.3 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.² 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

² The blocks world used in planning research is much simpler than SHRDLU's version, shown on page 20.

```

Init( $On(A, Table) \wedge On(B, Table) \wedge On(C, A)$ 
 $\wedge Block(A) \wedge Block(B) \wedge Block(C) \wedge Clear(B) \wedge Clear(C)$ )
Goal( $On(A, B) \wedge On(B, C)$ )
Action( $Move(b, x, y)$ ,
  PRECOND:  $On(b, x) \wedge Clear(b) \wedge Clear(y) \wedge Block(b) \wedge Block(y) \wedge$ 
 $(b \neq x) \wedge (b \neq y) \wedge (x \neq y)$ ,
  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 10.3 A planning problem in the blocks world: building a three-block tower. One solution is the sequence $[MoveToTable(C, A), Move(B, Table, C), Move(A, Table, B)]$.

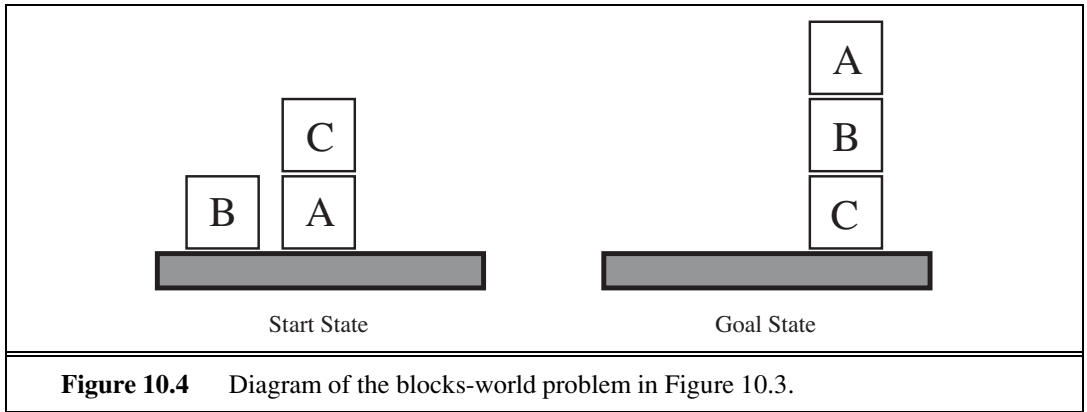


Figure 10.4 Diagram of the blocks-world problem in Figure 10.3.

of what other blocks. For example, a goal might be to get block A on B and block B on C (see Figure 10.4).

We 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, $\forall x \neg On(x, b)$. Basic PDDL does not allow quantifiers, so instead we introduce a predicate $Clear(x)$ that is true when nothing is on x . (The complete problem description is in Figure 10.3.)

The action $Move$ moves a block b from x to y if both b and y are clear. After the move is made, b is still clear but y is not. A first attempt at the $Move$ schema 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 does not maintain $Clear$ properly when x or y is the table. When x is the $Table$, 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

for us 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:

$$\begin{aligned} & \text{Action}(\text{MoveToTable}(b, x), \\ & \quad \text{PRECOND: } On(b, x) \wedge \text{Clear}(b), \\ & \quad \text{EFFECT: } On(b, \text{Table}) \wedge \text{Clear}(x) \wedge \neg On(b, x)) . \end{aligned}$$

Second, we take the interpretation of $\text{Clear}(x)$ to be “there is a clear space on x to hold a block.” Under this interpretation, $\text{Clear}(\text{Table})$ will always be true. The only problem is that nothing prevents the planner from using $\text{Move}(b, x, \text{Table})$ instead of $\text{MoveToTable}(b, x)$. 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 $\text{Block}(b) \wedge \text{Block}(y)$ to the precondition of Move .

10.1.4 The complexity of classical planning

In this subsection we consider the theoretical complexity of planning and distinguish two decision problems. **PlanSAT** is the question of whether there exists any plan that solves a planning problem. **Bounded PlanSAT** asks whether there is a solution of length k or less; this can be used to find an optimal plan.

The first result is that both decision problems are decidable for classical planning. The proof follows from the fact that the number of states is finite. But if we add function symbols to the language, then the number of states becomes infinite, and PlanSAT becomes only semidecidable: an algorithm exists that will terminate with the correct answer for any solvable problem, but may not terminate on unsolvable problems. The Bounded PlanSAT problem remains decidable even in the presence of function symbols. For proofs of the assertions in this section, see Ghallab *et al.* (2004).

Both PlanSAT and Bounded PlanSAT are in the complexity class PSPACE, a class that is larger (and hence more difficult) than NP and refers to problems that can be solved by a deterministic Turing machine with a polynomial amount of space. Even if we make some rather severe restrictions, the problems remain quite difficult. For example, if we disallow negative effects, both problems are still NP-hard. However, if we also disallow negative preconditions, PlanSAT reduces to the class P.

These worst-case results may seem discouraging. We can take solace in the fact that agents are usually not asked to find plans for arbitrary worst-case problem instances, but rather are asked for plans in specific domains (such as blocks-world problems with n blocks), which can be much easier than the theoretical worst case. For many domains (including the blocks world and the air cargo world), Bounded PlanSAT is NP-complete while PlanSAT is in P; in other words, optimal planning is usually hard, but sub-optimal planning is sometimes easy. To do well on easier-than-worst-case problems, we will need good search heuristics. That’s the true advantage of the classical planning formalism: it has facilitated the development of very accurate domain-independent heuristics, whereas systems based on successor-state axioms in first-order logic have had less success in coming up with good heuristics.

PlanSAT

Bounded PlanSAT

10.2 ALGORITHMS FOR PLANNING AS STATE-SPACE SEARCH

Now we turn our attention to planning algorithms. We saw how the description of a planning problem defines a search problem: we can search from the initial state through the space of states, looking for a goal. One of the nice advantages of the declarative representation of action schemas is that we can also search backward from the goal, looking for the initial state. Figure 10.5 compares forward and backward searches.

10.2.1 Forward (progression) state-space search

Now that we have shown how a planning problem maps into a search problem, we can solve planning problems with any of the heuristic search algorithms from Chapter 3 or a local search algorithm from Chapter 4 (provided we keep track of the actions used to reach the goal). From the earliest days of planning research (around 1961) until 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.

First, forward search is prone to exploring irrelevant actions. Consider the noble task of buying a copy of *AI: A Modern Approach* from an online bookseller. Suppose there is an

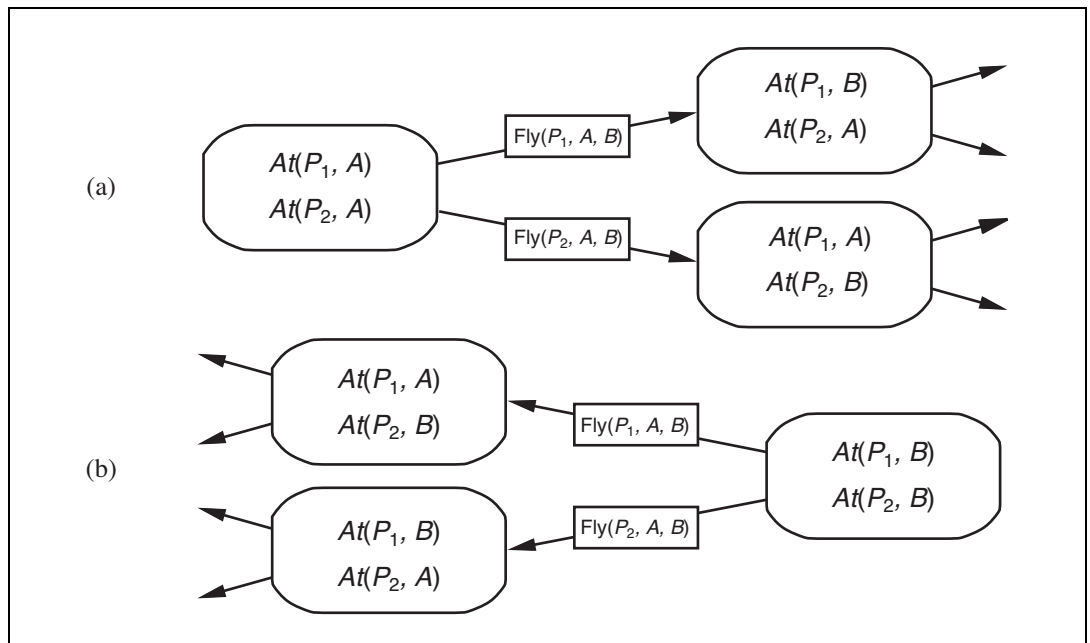


Figure 10.5 Two approaches to searching for a plan. (a) Forward (progression) search through the space of states, starting in the initial state and using the problem's actions to search forward for a member of the set of goal states. (b) Backward (regression) search through sets of relevant states, starting at the set of states representing the goal and using the inverse of the actions to search backward for the initial state.

action schema $Buy(isbn)$ with effect $Own(isbn)$. ISBNs are 10 digits, so this action schema represents 10 billion ground actions. An uninformed forward-search algorithm would have to start enumerating these 10 billion actions to find one that leads to the goal.

Second, planning problems often have large state spaces. 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. 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). So in any state there is a minimum of 450 actions (when all the packages are at airports with no planes) and a maximum of 10,450 (when all packages and planes are at the same airport). On average, let's say there are about 2000 possible actions per state, so the search graph up to the depth of the obvious solution has about 2000^{41} nodes.

Clearly, even this relatively small problem instance is hopeless without an accurate heuristic. Although many real-world applications of planning have relied on domain-specific heuristics, it turns out (as we see in Section 10.2.3) that strong domain-independent heuristics can be derived automatically; that is what makes forward search feasible.

10.2.2 Backward (regression) relevant-states search

In regression search we start at the goal and apply the actions backward until we find a sequence of steps that reaches the initial state. It is called **relevant-states** search because we only consider actions that are relevant to the goal (or current state). As in belief-state search (Section 4.4), there is a *set* of relevant states to consider at each step, not just a single state.

We start with the goal, which is a conjunction of literals forming a description of a set of states—for example, the goal $\neg Poor \wedge Famous$ describes those states in which *Poor* is false, *Famous* is true, and any other fluent can have any value. If there are n ground fluents in a domain, then there are 2^n ground states (each fluent can be true or false), but 3^n descriptions of sets of goal states (each fluent can be positive, negative, or not mentioned).

In general, backward search works only when we know how to regress from a state description to the predecessor state description. For example, it is hard to search backwards for a solution to the n -queens problem because there is no easy way to describe the states that are one move away from the goal. Happily, the PDDL representation was designed to make it easy to regress actions—if a domain can be expressed in PDDL, then we can do regression search on it. Given a ground goal description g and a ground action a , the regression from g over a gives us a state description g' defined by

$$g' = (g - \text{ADD}(a)) \cup \text{Precond}(a) .$$

That is, the effects that were added by the action need not have been true before, and also the preconditions must have held before, or else the action could not have been executed. Note that $\text{DEL}(a)$ does not appear in the formula; that's because while we know the fluents in $\text{DEL}(a)$ are no longer true after the action, we don't know whether or not they were true before, so there's nothing to be said about them.

To get the full advantage of backward search, we need to deal with partially uninstantiated actions and states, not just ground ones. For example, suppose the goal is to deliver a specific piece of cargo to SFO: $At(C_2, SFO)$. That suggests the action $Unload(C_2, p', SFO)$:

$Action(Unload(C_2, p', SFO),$
 PRECOND: $In(C_2, p') \wedge At(p', SFO) \wedge Cargo(C_2) \wedge Plane(p') \wedge Airport(SFO)$
 EFFECT: $At(C_2, SFO) \wedge \neg In(C_2, p')$.

(Note that we have **standardized** variable names (changing p to p' in this case) so that there will be no confusion between variable names if we happen to use the same action schema twice in a plan. The same approach was used in Chapter 9 for first-order logical inference.) This represents unloading the package from an *unspecified* plane at SFO; any plane will do, but we need not say which one now. We can take advantage of the power of first-order representations: a single description summarizes the possibility of using *any* of the planes by implicitly quantifying over p' . The regressed state description is

$$g' = In(C_2, p') \wedge At(p', SFO) \wedge Cargo(C_2) \wedge Plane(p') \wedge Airport(SFO) .$$

The final issue is deciding which actions are candidates to regress over. In the forward direction we chose actions that were **applicable**—those actions that could be the next step in the plan. In backward search we want actions that are **relevant**—those actions that could be the *last* step in a plan leading up to the current goal state.

RELEVANCE

For an action to be relevant to a goal it obviously must contribute to the goal: at least one of the action's effects (either positive or negative) must unify with an element of the goal. What is less obvious is that the action must not have any effect (positive or negative) that negates an element of the goal. Now, if the goal is $A \wedge B \wedge C$ and an action has the effect $A \wedge B \wedge \neg C$ then there is a colloquial sense in which that action is very relevant to the goal—it gets us two-thirds of the way there. But it is not relevant in the technical sense defined here, because this action could not be the *final* step of a solution—we would always need at least one more step to achieve C .

Given the goal $At(C_2, SFO)$, several instantiations of $Unload$ are relevant: we could choose any specific plane to unload from, or we could leave the plane unspecified by using the action $Unload(C_2, p', SFO)$. We can reduce the branching factor without ruling out any solutions by always using the action formed by substituting the most general unifier into the (standardized) action schema.

As another example, consider the goal $Own(0136042597)$, given an initial state with 10 billion ISBNs, and the single action schema

$$A = Action(Buy(i), PRECOND: ISBN(i), EFFECT: Own(i)) .$$

As we mentioned before, forward search without a heuristic would have to start enumerating the 10 billion ground Buy actions. But with backward search, we would unify the goal $Own(0136042597)$ with the (standardized) effect $Own(i')$, yielding the substitution $\theta = \{i'/0136042597\}$. Then we would regress over the action $Subst(\theta, A')$ to yield the predecessor state description $ISBN(0136042597)$. This is part of, and thus entailed by, the initial state, so we are done.

We can make this more formal. Assume a goal description g which contains a goal literal g_i and an action schema A that is standardized to produce A' . If A' has an effect literal e'_j where $\text{Unify}(g_i, e'_j) = \theta$ and where we define $a' = \text{SUBST}(\theta, A')$ and if there is no effect in a' that is the negation of a literal in g , then a' is a relevant action towards g .

Backward search keeps the branching factor lower than forward search, for most problem domains. However, the fact that backward search uses state sets rather than individual states makes it harder to come up with good heuristics. That is the main reason why the majority of current systems favor forward search.

10.2.3 Heuristics for planning

Neither forward nor backward search is efficient without a good heuristic function. Recall from Chapter 3 that a heuristic function $h(s)$ estimates the distance from a state s to the goal and that if we can derive an **admissible** heuristic for this distance—one that does not overestimate—then we can use A^* search to find optimal solutions. An admissible heuristic can be derived by defining a **relaxed problem** that is easier to solve. The exact cost of a solution to this easier problem then becomes the heuristic for the original problem.

By definition, there is no way to analyze an atomic state, and thus it requires some ingenuity by a human analyst to define good domain-specific heuristics for search problems with atomic states. Planning uses a factored representation for states and action schemas. That makes it possible to define good domain-independent heuristics and for programs to automatically apply a good domain-independent heuristic for a given problem.

Think of a search problem as a graph where the nodes are states and the edges are actions. The problem is to find a path connecting the initial state to a goal state. There are two ways we can relax this problem to make it easier: by adding more edges to the graph, making it strictly easier to find a path, or by grouping multiple nodes together, forming an abstraction of the state space that has fewer states, and thus is easier to search.

We look first at heuristics that add edges to the graph. For example, the **ignore preconditions heuristic** drops all preconditions from actions. Every action becomes applicable in every state, and any single goal fluent can be achieved in one step (if there is an applicable action—if not, the problem is impossible). This almost implies that the number of steps required to solve the relaxed problem is the number of unsatisfied goals—almost but not quite, because (1) some action may achieve multiple goals and (2) some actions may undo the effects of others. For many problems an accurate heuristic is obtained by considering (1) and ignoring (2). First, we relax the actions by removing all preconditions and all effects except those that are literals in the goal. Then, we count the minimum number of actions required such that the union of those actions' effects satisfies the goal. This is an instance of the **set-cover problem**. There is one minor irritation: the set-cover problem is NP-hard. Fortunately a simple greedy algorithm is guaranteed to return a set covering whose size is within a factor of $\log n$ of the true minimum covering, where n is the number of literals in the goal. Unfortunately, the greedy algorithm loses the guarantee of admissibility.

It is also possible to ignore only *selected* preconditions of actions. Consider the sliding-block puzzle (8-puzzle or 15-puzzle) from Section 3.2. We could encode this as a planning

IGNORE
PRECONDITIONS
HEURISTIC

SET-COVER
PROBLEM

problem involving tiles with a single schema *Slide*:

$Action(Slide(t, s_1, s_2),$

PRECOND: $On(t, s_1) \wedge Tile(t) \wedge Blank(s_2) \wedge Adjacent(s_1, s_2)$

EFFECT: $On(t, s_2) \wedge Blank(s_1) \wedge \neg On(t, s_1) \wedge \neg Blank(s_2)$)

As we saw in Section 3.6, if we remove the preconditions $Blank(s_2) \wedge Adjacent(s_1, s_2)$ then any tile can move in one action to any space and we get the number-of-misplaced-tiles heuristic. If we remove $Blank(s_2)$ then we get the Manhattan-distance heuristic. It is easy to see how these heuristics could be derived automatically from the action schema description. The ease of manipulating the schemas is the great advantage of the factored representation of planning problems, as compared with the atomic representation of search problems.

IGNORE DELETE
LISTS

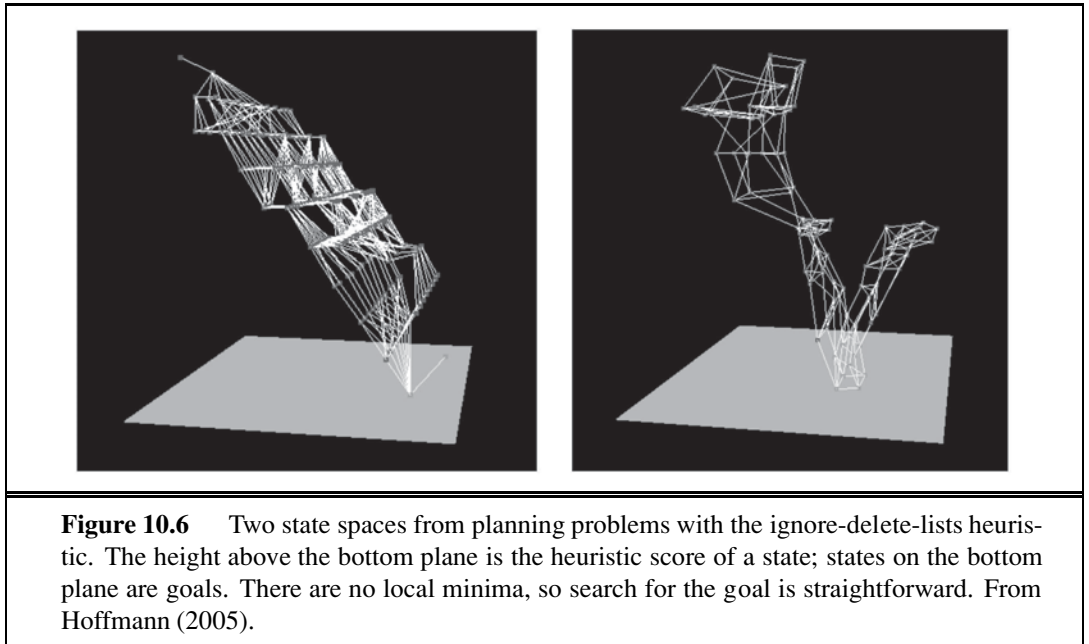
Another possibility is the **ignore delete lists** heuristic. Assume for a moment that all goals and preconditions contain only positive literals³ We want to create a relaxed version of the original problem that will be easier to solve, and where the length of the solution will serve as a good heuristic. We can do that by removing the delete lists from all actions (i.e., removing all negative literals from effects). That makes it possible to make monotonic progress towards the goal—no action will ever undo progress made by another action. It turns out it is still NP-hard to find the optimal solution to this relaxed problem, but an approximate solution can be found in polynomial time by hill-climbing. Figure 10.6 diagrams part of the state space for two planning problems using the ignore-delete-lists heuristic. The dots represent states and the edges actions, and the height of each dot above the bottom plane represents the heuristic value. States on the bottom plane are solutions. In both these problems, there is a wide path to the goal. There are no dead ends, so no need for backtracking; a simple hillclimbing search will easily find a solution to these problems (although it may not be an optimal solution).

STATE ABSTRACTION

The relaxed problems leave us with a simplified—but still expensive—planning problem just to calculate the value of the heuristic function. Many planning problems have 10^{100} states or more, and relaxing the *actions* does nothing to reduce the number of states. Therefore, we now look at relaxations that decrease the number of states by forming a **state abstraction**—a many-to-one mapping from states in the ground representation of the problem to the abstract representation.

The easiest form of state abstraction is to ignore some fluents. For example, consider an air cargo problem with 10 airports, 50 planes, and 200 pieces of cargo. Each plane can be at one of 10 airports and each package can be either in one of the planes or unloaded at one of the airports. So there are $50^{10} \times 200^{50+10} \approx 10^{155}$ states. Now consider a particular problem in that domain in which it happens that all the packages are at just 5 of the airports, and all packages at a given airport have the same destination. Then a useful abstraction of the problem is to drop all the *At* fluents except for the ones involving one plane and one package at each of the 5 airports. Now there are only $5^{10} \times 5^{5+10} \approx 10^{17}$ states. A solution in this abstract state space will be shorter than a solution in the original space (and thus will be an admissible heuristic), and the abstract solution is easy to extend to a solution to the original problem (by adding additional *Load* and *Unload* actions).

³ Many problems are written with this convention. For problems that aren't, replace every negative literal $\neg P$ in a goal or precondition with a new positive literal, P' .



DECOMPOSITION
SUBGOAL
INDEPENDENCE

A key idea in defining heuristics is **decomposition**: dividing a problem into parts, solving each part independently, and then combining the parts. The **subgoal independence** assumption is that 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.

Suppose the goal is a set of fluents G , which we divide into disjoint subsets G_1, \dots, G_n . We then find plans P_1, \dots, P_n that solve the respective subgoals. What is an estimate of the cost of the plan for achieving all of G ? We can think of each $\text{Cost}(P_i)$ as a heuristic estimate, and we know that if we combine estimates by taking their maximum value, we always get an admissible heuristic. So $\max_i \text{COST}(P_i)$ is admissible, and sometimes it is exactly correct: it could be that P_1 serendipitously achieves all the G_i . But in most cases, in practice the estimate is too low. Could we sum the costs instead? For many problems that is a reasonable estimate, but it is not admissible. The best case is when we can determine that G_i and G_j are **independent**. If the effects of P_i leave all the preconditions and goals of P_j unchanged, then the estimate $\text{COST}(P_i) + \text{COST}(P_j)$ is admissible, and more accurate than the max estimate. We show in Section 10.3.1 that planning graphs can help provide better heuristic estimates.

It is clear that there is great potential for cutting down the search space by forming abstractions. The trick is choosing the right abstractions and using them in a way that makes the total cost—defining an abstraction, doing an abstract search, and mapping the abstraction back to the original problem—less than the cost of solving the original problem. The tech-

niques of **pattern databases** from Section 3.6.3 can be useful, because the cost of creating the pattern database can be amortized over multiple problem instances.

An example of a system that makes use of effective heuristics is FF, or FASTFORWARD (Hoffmann, 2005), a forward state-space searcher that uses the ignore-delete-lists heuristic, estimating the heuristic with the help of a planning graph (see Section 10.3). FF then uses hill-climbing search (modified to keep track of the plan) with the heuristic to find a solution. When it hits a plateau or local maximum—when no action leads to a state with better heuristic score—then FF uses iterative deepening search until it finds a state that is better, or it gives up and restarts hill-climbing.

10.3 PLANNING GRAPHS

PLANNING GRAPH

All of the heuristics we have suggested 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 search for a solution over the space formed by the planning graph, using an algorithm called GRAPHPLAN.

A planning problem asks if we can reach a goal state from the initial state. Suppose we are given a tree of all possible actions from the initial state to successor states, and their successors, and so on. If we indexed this tree appropriately, we could answer the planning question “can we reach state G from state S_0 ” immediately, just by looking it up. Of course, the tree is of exponential size, so this approach is impractical. A planning graph is polynomial-size approximation to this tree that can be constructed quickly. The planning graph can’t answer definitively whether G is reachable from S_0 , but it can *estimate* how many steps it takes to reach G . The estimate is always correct when it reports the goal is not reachable, and it never overestimates the number of steps, so it is an admissible heuristic.

LEVEL

A planning graph is a directed graph organized into **levels**: first a level S_0 for the initial state, consisting of nodes representing each fluent that holds in S_0 ; then a level A_0 consisting of nodes for each ground action that might be applicable in S_0 ; then alternating levels S_i followed by A_i ; until we reach a termination condition (to be discussed later).

Roughly speaking, S_i contains all the literals that *could* hold at time i , depending on the actions executed at preceding time steps. If it is possible that either P or $\neg P$ could hold, then both will be represented in S_i . Also roughly speaking, A_i contains all the actions that *could* have their preconditions satisfied at time i . We say “roughly speaking” because the planning graph records only a restricted subset of the possible negative interactions among actions; therefore, a literal might show up at level S_j when actually it could not be true until a later level, if at all. (A literal will never show up too late.) Despite the possible error, the level j at which a literal first appears is a good estimate of how difficult it is to achieve the literal from the initial state.

Planning graphs work only for propositional planning problems—ones with no variables. As we mentioned on page 368, it is straightforward to propositionalize a set of ac-

depending on the choice of actions in A_0 , either, but not both, could be the result. In other words, S_1 represents a belief state: a set of possible states. The members of this set are all subsets of the literals such that there is no mutex link between any members of the subset.

We continue in this way, alternating between state level S_i and action level A_i until we reach a point where two consecutive levels are identical. At this point, we say that the graph has **leveled off**. The graph in Figure 10.8 levels off at S_2 .

LEVELED OFF

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 that two actions cannot both be executed at the same level. 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.

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.

A planning graph is polynomial in the size of the planning problem. For a planning problem with l literals and a actions, each S_i has no more than l nodes and l^2 mutex links, and each A_i has no more than $a + l$ nodes (including the no-ops), $(a + l)^2$ mutex links, and $2(al + l)$ precondition and effect links. Thus, an entire graph with n levels has a size of $O(n(a + l)^2)$. The time to build the graph has the same complexity.

10.3.1 Planning graphs for heuristic estimation

A planning graph, once constructed, is a rich source of information about the problem. First, if any goal literal fails to appear in the final level of the graph, then the problem is unsolvable. Second, we can estimate the cost of achieving any goal literal g_i from state s as the level at which g_i first appears in the planning graph constructed from initial state s . We call this the

LEVEL COST

level cost of g_i . In Figure 10.8, $Have(Cake)$ has level cost 0 and $Eaten(Cake)$ has level cost 1. It is easy to show (Exercise 10.10) that these estimates are admissible for the individual goals. The estimate might not always be accurate, however, because planning graphs allow several actions at each level, whereas the heuristic counts just the level and not the number of actions. 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 nonpersistence actions. Level costs extracted from serial graphs are often quite reasonable estimates of actual costs.

SERIAL PLANNING GRAPH

MAX-LEVEL

To estimate the cost of a *conjunction* of goals, there are three simple approaches. The **max-level** heuristic simply takes the maximum level cost of any of the goals; this is admissible, but not necessarily accurate.

LEVEL SUM

The **level sum** heuristic, following the subgoal independence assumption, returns the sum of the level costs of the goals; this can be inadmissible but works well in practice for problems that are largely decomposable. It is much more accurate than the number-of-unsatisfied-goals heuristic from Section 10.2. For our problem, the level-sum heuristic estimate for the conjunctive goal $Have(Cake) \wedge Eaten(Cake)$ will be $0 + 1 = 1$, whereas the correct answer is 2, achieved by the plan $[Eat(Cake), Bake(Cake)]$. That doesn't seem so bad. A more serious error is that if $Bake(Cake)$ were not in the set of actions, then the estimate would still be 1, when in fact the conjunctive goal would be impossible.

SET-LEVEL

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 is admissible, it dominates the max-level heuristic, and it works extremely well on tasks in which there is a good deal of interaction among subplans. It is not perfect, of course; for example, it ignores interactions among three or more literals.

As a tool for generating accurate heuristics, we can view the planning graph as a relaxed problem that is efficiently solvable. 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, 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 a 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 page 211.)

One example of an unsolvable problem that cannot be recognized as such by a planning graph is the blocks-world problem where the goal is to get block A on B , B on C , and C on A . This is an impossible goal; a tower with the bottom on top of the top. But a planning graph

cannot detect the impossibility, because any two of the three subgoals are achievable. There are no mutexes between any pair of literals, only between the three as a whole. To detect that this problem is impossible, we would have to search over the planning graph.

10.3.2 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 10.9) repeatedly adds a level to a planning graph with EXPAND-GRAPH. Once all the goals show up as non-mutex in the graph, GRAPHPLAN calls EXTRACT-SOLUTION to search for a plan that solves the problem. If that fails, it expands another level and tries again, terminating with failure when there is no reason to go on.

```

function GRAPHPLAN(problem) returns solution or failure
  graph  $\leftarrow$  INITIAL-PLANNING-GRAPH(problem)
  goals  $\leftarrow$  CONJUNCTS(problem.GOAL)
  nogoods  $\leftarrow$  an empty hash table
  for tl = 0 to  $\infty$  do
    if goals all non-mutex in  $S_t$  of graph then
      solution  $\leftarrow$  EXTRACT-SOLUTION(graph, goals, NUMLEVELS(graph), nogoods)
      if solution  $\neq$  failure then return solution
    if graph and nogoods have both leveled off then return failure
    graph  $\leftarrow$  EXPAND-GRAPH(graph, problem)

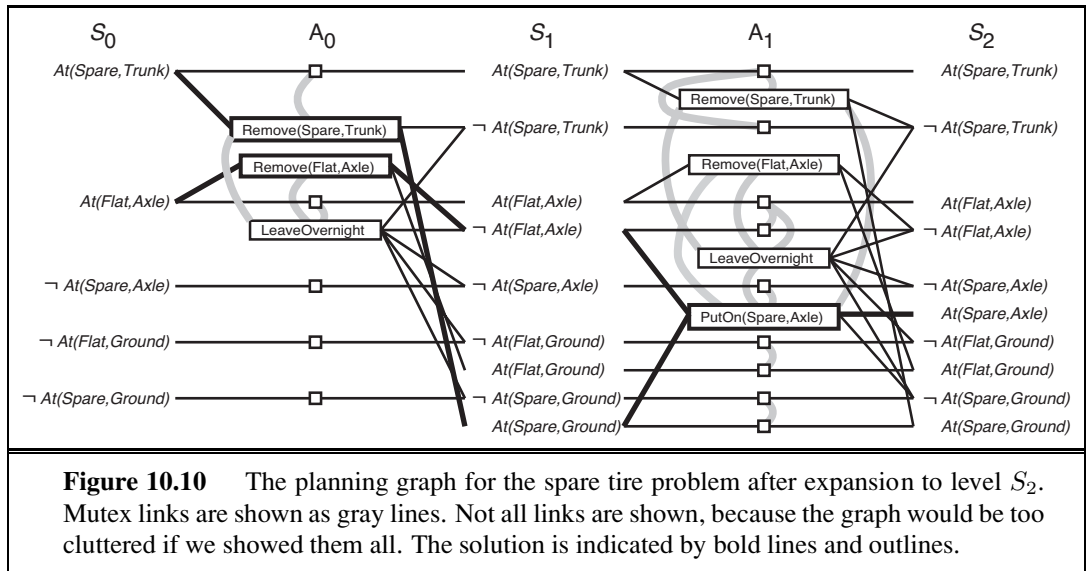
```

Figure 10.9 The GRAPHPLAN algorithm. GRAPHPLAN calls EXPAND-GRAPH to add a level until either a solution is found by EXTRACT-SOLUTION, or no solution is possible.

Let us now trace the operation of GRAPHPLAN on the spare tire problem from page 370. The graph is shown in Figure 10.10. The first line of GRAPHPLAN initializes the planning graph to a one-level (S_0) graph representing the initial state. The positive fluents from the problem description's initial state are shown, as are the relevant negative fluents. Not shown are the unchanging positive literals (such as $Tire(Spare)$) and the irrelevant negative literals. The goal $At(Spare, Axle)$ 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 into A_0 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.

$At(Spare, Axle)$ is still not present in S_1 , so again we do not call EXTRACT-SOLUTION. We call EXPAND-GRAPH again, adding A_1 and S_1 and giving us the planning graph shown in Figure 10.10. 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. We can formulate EXTRACT-SOLUTION as a Boolean constraint satisfaction problem (CSP) where the variables are the actions at each level, the values for each variable are *in* or *out* of the plan, and the constraints are the mutexes and the need to satisfy each goal and precondition.

Alternatively, we can define EXTRACT-SOLUTION as a backward 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 .

In the case where EXTRACT-SOLUTION fails to find a solution for a set of goals at a given level, we record the $(level, goals)$ pair as a **no-good**, just as we did in constraint learning for CSPs (page 220). Whenever EXTRACT-SOLUTION is called again with the same level and goals, we can find the recorded no-good and immediately return failure rather than searching again. We see shortly that no-goods are also used in the termination test.

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, prefer actions with easier preconditions. That is, choose an action such that the sum (or maximum) of the level costs of its preconditions is smallest.

10.3.3 Termination of GRAPHPLAN

So far, we have skated over the question of termination. Here we show that GRAPHPLAN will in fact terminate and return failure when there is no solution.

The first thing to understand is why we can't stop expanding the graph as soon as it has leveled off. Consider an air cargo domain with one plane and n pieces of cargo at airport A , all of which have airport B as their destination. In this version of the problem, only one piece of cargo can fit in the plane at a time. The graph will level off at level 4, reflecting the fact that for any single piece of cargo, we can load it, fly it, and unload it at the destination in three steps. But that does not mean that a solution can be extracted from the graph at level 4; in fact a solution will require $4n - 1$ steps: for each piece of cargo we load, fly, and unload, and for all but the last piece we need to fly back to airport A to get the next piece.

How long do we have to keep expanding after the graph has leveled off? If the function EXTRACT-SOLUTION fails to find a solution, then there must have been at least one set of goals that were not achievable and were marked as a no-good. So if it is possible that there might be fewer no-goods in the next level, then we should continue. As soon as the graph itself and the no-goods have both leveled off, with no solution found, we can terminate with failure because there is no possibility of a subsequent change that could add a solution.

Now all we have to do is prove that the graph and the no-goods will always level off. The key to this proof is 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 the monotonic increase of literals; 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 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.

- *No-goods decrease monotonically*: If a set of goals is not achievable at a given level, then they are not achievable in any *previous* level. The proof is by contradiction: if they were achievable at some previous level, then we could just add persistence actions to make them achievable at a subsequent level.

Because the actions and literals increase monotonically and because there are only a finite number of actions and literals, there must come a level that has the same number of actions and literals as the previous level. Because mutexes and no-goods decrease, and because there can never be fewer than zero mutexes or no-goods, there must come a level that has the same number of mutexes and no-goods as the previous level. Once a graph has reached this state, then if one of the goals is missing or is mutex with another goal, then we can stop the GRAPHPLAN algorithm and return failure. That concludes a sketch of the proof; for more details see Ghallab *et al.* (2004).

Year	Track	Winning Systems (approaches)
2008	Optimal	GAMER (model checking, bidirectional search)
2008	Satisficing	LAMA (fast downward search with FF heuristic)
2006	Optimal	SATPLAN, MAXPLAN (Boolean satisfiability)
2006	Satisficing	SGPLAN (forward search; partitions into independent subproblems)
2004	Optimal	SATPLAN (Boolean satisfiability)
2004	Satisficing	FAST DIAGONALLY DOWNWARD (forward search with causal graph)
2002	Automated	LPG (local search, planning graphs converted to CSPs)
2002	Hand-coded	TLPLAN (temporal action logic with control rules for forward search)
2000	Automated	FF (forward search)
2000	Hand-coded	TALPLANNER (temporal action logic with control rules for forward search)
1998	Automated	IPP (planning graphs); HSP (forward search)

Figure 10.11 Some of the top-performing systems in the International Planning Competition. Each year there are various tracks: “Optimal” means the planners must produce the shortest possible plan, while “Satisficing” means nonoptimal solutions are accepted. “Hand-coded” means domain-specific heuristics are allowed; “Automated” means they are not.

10.4 OTHER CLASSICAL PLANNING APPROACHES

Currently the most popular and effective approaches to fully automated planning are:

- Translating to a Boolean satisfiability (SAT) problem
- Forward state-space search with carefully crafted heuristics (Section 10.2)
- Search using a planning graph (Section 10.3)

These three approaches are not the only ones tried in the 40-year history of automated planning. Figure 10.11 shows some of the top systems in the International Planning Competitions, which have been held every even year since 1998. In this section we first describe the translation to a satisfiability problem and then describe three other influential approaches: planning as first-order logical deduction; as constraint satisfaction; and as plan refinement.

10.4.1 Classical planning as Boolean satisfiability

In Section 7.7.4 we saw how SATPLAN solves planning problems that are expressed in propositional logic. Here we show how to translate a PDDL description into a form that can be processed by SATPLAN. The translation is a series of straightforward steps:

- Propositionalize the actions: replace each action schema with a set of ground actions formed by substituting constants for each of the variables. These ground actions are not part of the translation, but will be used in subsequent steps.
- Define the initial state: assert F^0 for every fluent F in the problem’s initial state, and $\neg F$ for every fluent not mentioned in the initial state.
- Propositionalize the goal: for every variable in the goal, replace the literals that contain the variable with a disjunction over constants. For example, the goal of having block A

on another block, $On(A, x) \wedge Block(x)$ in a world with objects A, B and C , would be replaced by the goal

$$(On(A, A) \wedge Block(A)) \vee (On(A, B) \wedge Block(B)) \vee (On(A, C) \wedge Block(C)).$$

- Add successor-state axioms: For each fluent F , add an axiom of the form

$$F^{t+1} \Leftrightarrow ActionCausesF^t \vee (F^t \wedge \neg ActionCausesNotF^t),$$

where $ActionCausesF$ is a disjunction of all the ground actions that have F in their add list, and $ActionCausesNotF$ is a disjunction of all the ground actions that have F in their delete list.

- Add precondition axioms: For each ground action A , add the axiom $A^t \Rightarrow PRE(A)^t$, that is, if an action is taken at time t , then the preconditions must have been true.
- Add action exclusion axioms: say that every action is distinct from every other action.

The resulting translation is in the form that we can hand to SATPLAN to find a solution.

10.4.2 Planning as first-order logical deduction: Situation calculus

PDDL is a language that carefully balances the expressiveness of the language with the complexity of the algorithms that operate on it. But some problems remain difficult to express in PDDL. For example, we can't express the goal "move all the cargo from A to B regardless of how many pieces of cargo there are" in PDDL, but we can do it in first-order logic, using a universal quantifier. Likewise, first-order logic can concisely express global constraints such as "no more than four robots can be in the same place at the same time." PDDL can only say this with repetitious preconditions on every possible action that involves a move.

The propositional logic representation of planning problems also has limitations, such as the fact that the notion of time is tied directly to fluents. For example, $South^2$ means "the agent is facing south at time 2." With that representation, there is no way to say "the agent would be facing south at time 2 if it executed a right turn at time 1; otherwise it would be facing east." First-order logic lets us get around this limitation by replacing the notion of linear time with a notion of branching *situations*, using a representation called **situation calculus** that works like this:

- The initial state is called a **situation**. If s is a situation and a is an action, then $RESULT(s, a)$ is also a situation. There are no other situations. Thus, a situation corresponds to a sequence, or history, of actions. You can also think of a situation as the result of applying the actions, but note that two situations are the same only if their start and actions are the same: $(RESULT(s, a) = RESULT(s', a')) \Leftrightarrow (s = s' \wedge a = a')$. Some examples of actions and situations are shown in Figure 10.12.
- A function or relation that can vary from one situation to the next is a **fluent**. By convention, the situation s is always the last argument to the fluent, for example $At(x, l, s)$ is a relational fluent that is true when object x is at location l in situation s , and $Location$ is a functional fluent such that $Location(x, s) = l$ holds in the same situations as $At(x, l, s)$.
- Each action's preconditions are described with a **possibility axiom** that says when the action can be taken. It has the form $\Phi(s) \Rightarrow Poss(a, s)$ where $\Phi(s)$ is some formula

SITUATION
CALCULUS

SITUATION

POSSIBILITY AXIOM

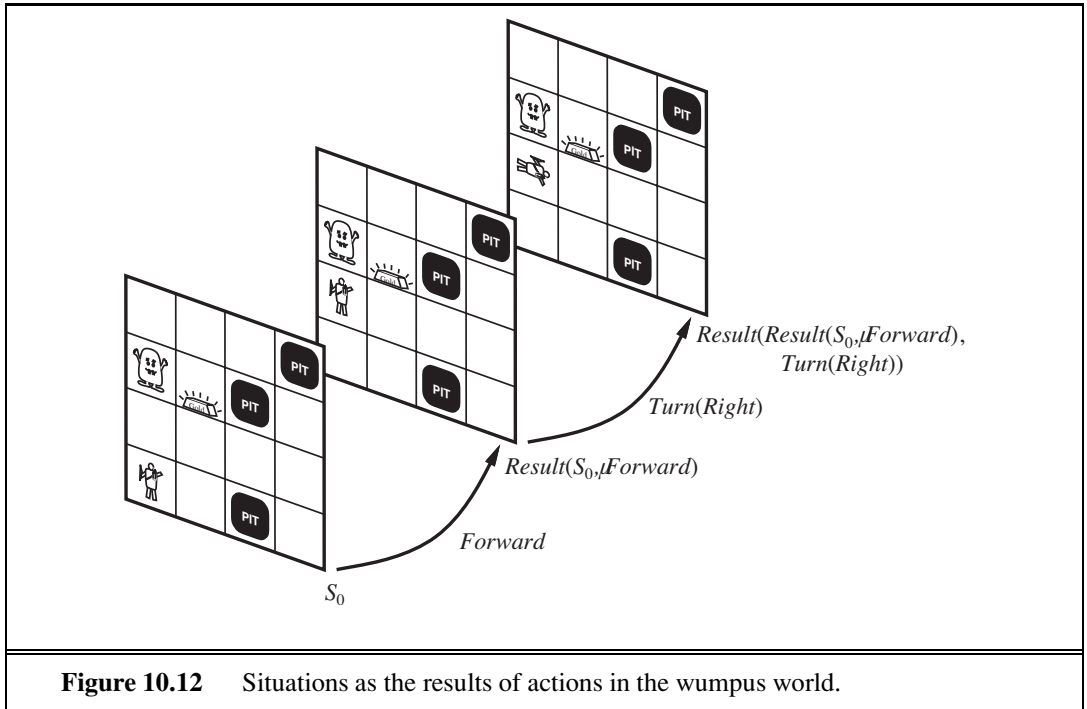


Figure 10.12 Situations as the results of actions in the wumpus world.

involving s that describes the preconditions. An example from the wumpus world says that it is possible to shoot if the agent is alive and has an arrow:

$$Alive(Agent, s) \wedge Have(Agent, Arrow, s) \Rightarrow Poss(Shoot, s)$$

- Each fluent is described with a **successor-state axiom** that says what happens to the fluent, depending on what action is taken. This is similar to the approach we took for propositional logic. The axiom has the form

$$\begin{aligned} \text{Action is possible} &\Rightarrow \\ &(\text{Fluent is true in result state} \Leftrightarrow \text{Action's effect made it true} \\ &\quad \vee \text{It was true before and action left it alone}) . \end{aligned}$$

For example, the axiom for the relational fluent *Holding* says that the agent is holding some gold g after executing a possible action if and only if the action was a *Grab* of g or if the agent was already holding g and the action was not releasing it:

$$\begin{aligned} Poss(a, s) &\Rightarrow \\ &(\text{Holding}(Agent, g, \text{Result}(a, s)) \Leftrightarrow \\ &\quad a = \text{Grab}(g) \vee (\text{Holding}(Agent, g, s) \wedge a \neq \text{Release}(g))) . \end{aligned}$$

- We need **unique action axioms** so that the agent can deduce that, for example, $a \neq \text{Release}(g)$. For each distinct pair of action names A_i and A_j we have an axiom that says the actions are different:

$$A_i(x, \dots) \neq A_j(y, \dots)$$

and for each action name A_i we have an axiom that says two uses of that action name are equal if and only if all their arguments are equal:

$$A_i(x_1, \dots, x_n) = A_i(y_1, \dots, y_n) \Leftrightarrow x_1 = y_1 \wedge \dots \wedge x_n = y_n.$$

- A solution is a situation (and hence a sequence of actions) that satisfies the goal.

Work in situation calculus has done a lot to define the formal semantics of planning and to open up new areas of investigation. But so far there have not been any practical large-scale planning programs based on logical deduction over the situation calculus. This is in part because of the difficulty of doing efficient inference in FOL, but is mainly because the field has not yet developed effective heuristics for planning with situation calculus.

10.4.3 Planning as constraint satisfaction

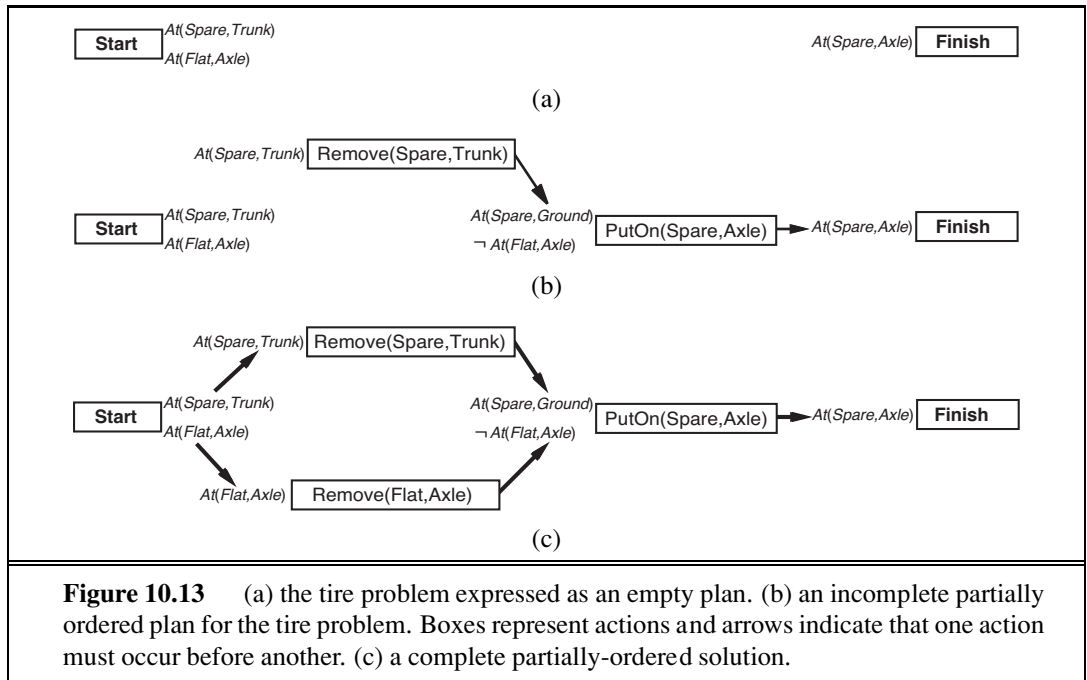
We have seen that constraint satisfaction has a lot in common with Boolean satisfiability, and we have seen that CSP techniques are effective for scheduling problems, so it is not surprising that it is possible to encode a bounded planning problem (i.e., the problem of finding a plan of length k) as a constraint satisfaction problem (CSP). The encoding is similar to the encoding to a SAT problem (Section 10.4.1), with one important simplification: at each time step we need only a single variable, $Action^t$, whose domain is the set of possible actions. We no longer need one variable for every action, and we don't need the action exclusion axioms. It is also possible to encode a planning graph into a CSP. This is the approach taken by GP-CSP (Do and Kambhampati, 2003).

10.4.4 Planning as refinement of partially ordered plans

All the approaches we have seen so far construct *totally ordered* plans consisting of a strictly linear sequences of actions. This representation ignores the fact that many subproblems are independent. A solution to an air cargo problem consists of a totally ordered sequence of actions, yet if 30 packages are being loaded onto one plane in one airport and 50 packages are being loaded onto another at another airport, it seems pointless to come up with a strict linear ordering of 80 load actions; the two subsets of actions should be thought of independently.

An alternative is to represent plans as *partially ordered* structures: a plan is a set of actions and a set of constraints of the form $Before(a_i, a_j)$ saying that one action occurs before another. In the bottom of Figure 10.13, we see a partially ordered plan that is a solution to the spare tire problem. Actions are boxes and ordering constraints are arrows. Note that $Remove(Spare, Trunk)$ and $Remove(Flat, Axle)$ can be done in either order as long as they are both completed before the $PutOn(Spare, Axle)$ action.

Partially ordered plans are created by a *search through the space of plans* rather than through the state space. We start with the empty plan consisting of just the initial state and the goal, with no actions in between, as in the top of Figure 10.13. The search procedure then looks for a **flaw** in the plan, and makes an addition to the plan to correct the flaw (or if no correction can be made, the search backtracks and tries something else). A flaw is anything that keeps the partial plan from being a solution. For example, one flaw in the empty plan is that no action achieves $At(Spare, Axle)$. One way to correct the flaw is to insert into the plan



the action $PutOn(Spare, Axle)$. Of course that introduces some new flaws: the preconditions of the new action are not achieved. The search keeps adding to the plan (backtracking if necessary) until all flaws are resolved, as in the bottom of Figure 10.13. At every step, we make the **least commitment** possible to fix the flaw. For example, in adding the action $Remove(Spare, Trunk)$ we need to commit to having it occur before $PutOn(Spare, Axle)$, but we make no other commitment that places it before or after other actions. If there were a variable in the action schema that could be left unbound, we would do so.

LEAST COMMITMENT

In the 1980s and 90s, partial-order planning was seen as the best way to handle planning problems with independent subproblems—after all, it was the only approach that explicitly represents independent branches of a plan. On the other hand, it has the disadvantage of not having an explicit representation of states in the state-transition model. That makes some computations cumbersome. By 2000, forward-search planners had developed excellent heuristics that allowed them to efficiently discover the independent subproblems that partial-order planning was designed for. As a result, partial-order planners are not competitive on fully automated classical planning problems.

However, partial-order planning remains an important part of the field. For some specific tasks, such as operations scheduling, partial-order planning with domain specific heuristics is the technology of choice. Many of these systems use libraries of high-level plans, as described in Section 11.2. Partial-order planning is also often used in domains where it is important for humans to understand the plans. Operational plans for spacecraft and Mars rovers are generated by partial-order planners and are then checked by human operators before being uploaded to the vehicles for execution. The plan refinement approach makes it easier for the humans to understand what the planning algorithms are doing and verify that they are correct.

10.5 ANALYSIS OF PLANNING APPROACHES

Planning combines the two major areas of AI we have covered so far: *search* and *logic*. 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 both to improvements in performance amounting to several orders of magnitude in the last decade and to 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 n propositions in a domain, then there are 2^n states. As we have seen, planning is PSPACE-hard. Against such pessimism, the identification of independent subproblems can be a powerful weapon. In the best case—full decomposability of the problem—we get an exponential speedup. Decomposability is destroyed, however, by negative interactions between actions. GRAPHPLAN records mutexes to point out where the difficult interactions are. SATPLAN represents a similar range of mutex relations, but does so by using the general CNF form rather than a specific data structure. Forward search addresses the problem heuristically by trying to find patterns (subsets of propositions) that cover the independent subproblems. Since this approach is heuristic, it can work even when the subproblems are not completely independent.

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*, as in Figure 10.4 on page 371), 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 without backtracking (although it might not always find the shortest plan).

As a more complex example, for the Remote Agent planner that 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.

Planners such as GRAPHPLAN, SATPLAN, and FF have moved the field of planning forward, by raising the level of performance of planning systems, by clarifying the representational and combinatorial issues involved, and by the development of useful heuristics. However, there is a question of how far these techniques will scale. It seems likely that further progress on larger problems cannot rely only on factored and propositional representations, and will require some kind of synthesis of first-order and hierarchical representations with the efficient heuristics currently in use.

SERIALIZABLE
SUBGOAL

10.6 SUMMARY

In this chapter, we defined the problem of planning in deterministic, fully observable, static environments. We described the PDDL representation for planning problems and several algorithmic approaches for solving them. The points to remember:

- Planning systems are problem-solving algorithms that operate on explicit propositional or relational 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.
- PDDL, the Planning Domain Definition Language, describes the initial and goal states as conjunctions of literals, and actions in terms of their preconditions and effects.
- State-space search can operate in the forward direction (**progression**) or the backward direction (**regression**). Effective heuristics can be derived by subgoal independence assumptions and by various relaxations of the planning problem.
- 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 (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.
- Other approaches include first-order deduction over situation calculus axioms; encoding a planning problem as a Boolean satisfiability problem or as a constraint satisfaction problem; and explicitly searching through the space of partially ordered plans.
- 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. Bylander (1992) shows simple STRIPS planning to be PSPACE-complete. Fikes and Nilsson (1993) give a historical retrospective on the STRIPS project and its relationship to more recent planning efforts.

The representation language used by STRIPS has been far more influential than its algorithmic approach; what we call the “classical” language is close to what STRIPS used.

The Action Description Language, or ADL (Pednault, 1986), relaxed some of the STRIPS restrictions 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 planning problems and has been used as the standard language for the International Planning Competition since 1998. There have been several extensions; the most recent version, PDDL 3.0, includes plan constraints and preferences (Gerevini and Long, 2005).

LINEAR PLANNING

Planners in the early 1970s generally considered totally ordered action sequences. Problem decomposition was achieved by computing a subplan for each subgoal and then 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 10.7), 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 with logic programming: WARPLAN is only 100 lines of code, a small fraction of the size of comparable planners of the time.

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.

Partial-order planning dominated the next 20 years of research, yet the first clear formal exposition was TWEAK (Chapman, 1987), a planner that was simple enough to allow proofs of completeness and intractability (NP-hardness and undecidability) of various planning problems. Chapman's work led to a straightforward description of a complete partial-order planner (McAllester and Rosenblitt, 1991), then to the widely distributed implementations SNLP (Soderland and Weld, 1991) and UCPOP (Penberthy and Weld, 1992). Partial-order planning fell out of favor in the late 1990s as faster methods emerged. Nguyen and Kambhampati (2001) suggest that a reconsideration is merited: with accurate heuristics derived from a planning graph, their REPOP planner scales up much better than GRAPHPLAN in parallelizable domains and is competitive with the fastest state-space planners.

The resurgence of interest in state-space planning was pioneered by Drew McDermott's UNPOP program (1996), which was the first to suggest the ignore-delete-list heuristic. 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; Haslum *et al.*, 2005; Haslum, 2006) were the first to make

state-space search practical for large planning problems. HSP searches in the forward direction while HSPR (Bonet and Geffner, 1999) searches backward. The most successful state-space searcher to date is FF (Hoffmann, 2001; Hoffmann and Nebel, 2001; Hoffmann, 2005), winner of the AIPS 2000 planning competition. FASTDOWNWARD (Helmert, 2006) is a forward state-space search planner that preprocesses the action schemas into an alternative representation which makes some of the constraints more explicit. FASTDOWNWARD (Helmert and Richter, 2004; Helmert, 2006) won the 2004 planning competition, and LAMA (Richter and Westphal, 2008), a planner based on FASTDOWNWARD with improved heuristics, won the 2008 competition.

Bylander (1994) and Ghallab *et al.* (2004) discuss the computational complexity of several variants of the planning problem. Helmert (2003) proves complexity bounds for many of the standard benchmark problems, and Hoffmann (2005) analyzes the search space of the ignore-delete-list heuristic. Heuristics for the set-covering problem are discussed by Caprara *et al.* (1995) for scheduling operations of the Italian railway. Edelkamp (2009) and Haslum *et al.* (2007) describe how to construct pattern databases for planning heuristics. As we mentioned in Chapter 3, Felner *et al.* (2004) show encouraging results using pattern databases for sliding blocks puzzles, which can be thought of as a planning domain, but Hoffmann *et al.* (2006) show some limitations of abstraction for classical planning problems.

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 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) thoroughly analyze heuristics derived from planning graphs. Our discussion of planning graphs is based partly on this work and on lecture notes and articles by Subbarao Kambhampati (Bryce and Kambhampati, 2007). 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, 2003), searched planning graphs using a local search technique inspired by WALKSAT.

The situation calculus approach to planning was introduced by John McCarthy (1963). The version we show here was proposed by Ray Reiter (1991, 2001).

Kautz *et al.* (1996) investigated various ways to propositionalize action schemas, 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). CPLAN, a planner based on constraint satisfaction, was described by van Beek and Chen (1999).

Most recently, there has been interest in the representation of plans as **binary decision diagrams**, compact data structures for Boolean expressions 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.

Readings in Planning (Allen *et al.*, 1990) is a comprehensive anthology of early work in the field. Weld (1994, 1999) provides two excellent surveys of planning algorithms of the 1990s. 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. *Automated Planning* (Ghallab *et al.*, 2004) is an excellent textbook on all aspects of planning. LaValle's text *Planning Algorithms* (2006) covers both classical and stochastic planning, with extensive coverage of robot motion planning.

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, the International Workshop on Planning and Scheduling for Space, and the European Conference on Planning.

EXERCISES

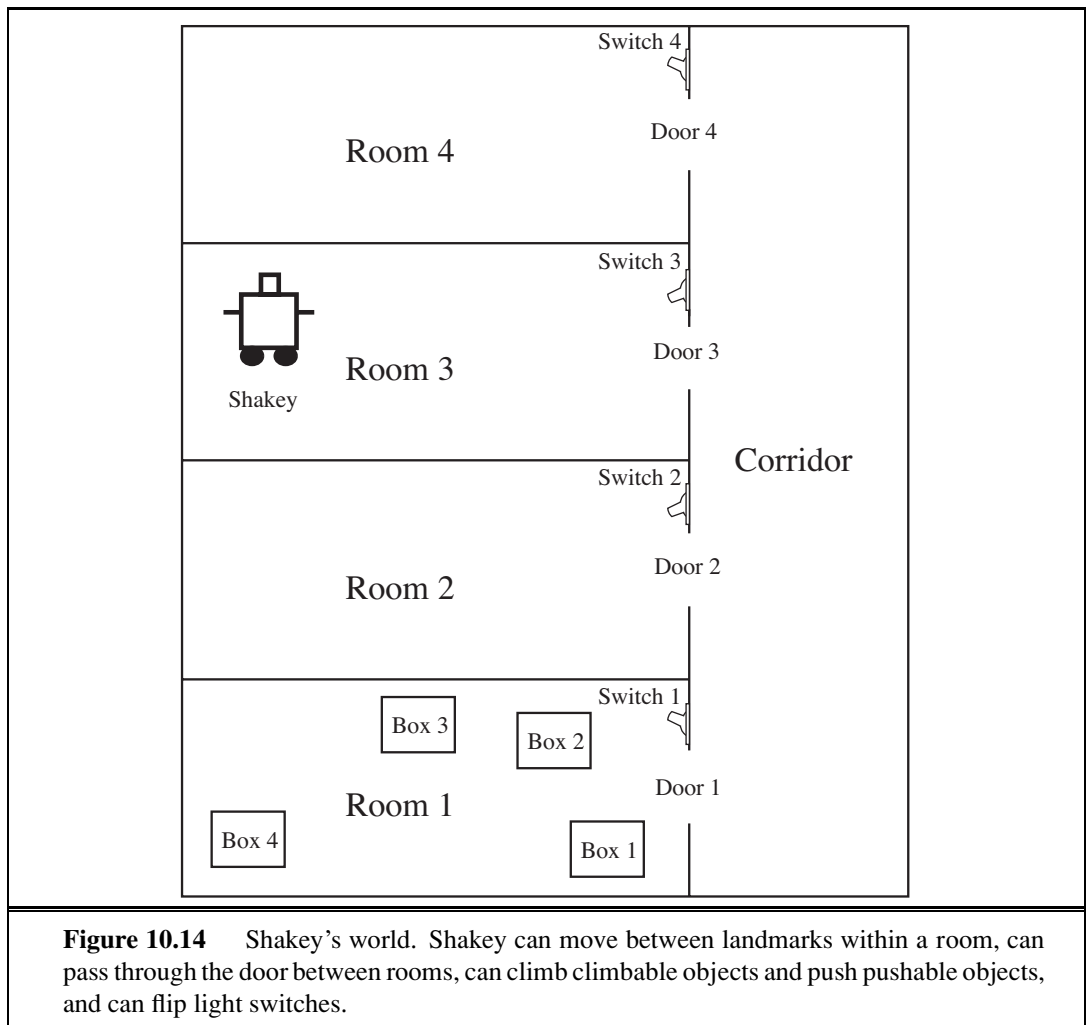
10.1 Describe the differences and similarities between problem solving and planning.

10.2 Given the action schemas and initial state from Figure 10.1, what are all the applicable concrete instances of $Fly(p, from, to)$ in the state described by

$$At(P_1, JFK) \wedge At(P_2, SFO) \wedge Plane(P_1) \wedge Plane(P_2) \\ \wedge Airport(JFK) \wedge Airport(SFO) ?$$

10.3 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. The result of a *Grasp* is that the monkey holds 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 the six action schemas.
- 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 classical planning system?
- d. Your schema for pushing is probably incorrect, because if the object is too heavy, its position will remain the same when the *Push* schema is applied. Fix your action schema to account for heavy objects.

10.4 The original STRIPS planner was designed to control Shakey the robot. Figure 10.14 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 could not climb on a box or toggle a switch, but the planner was capable of finding and printing out plans that were beyond the robot's abilities. Shakey's six actions are the following:

- $Go(x, y, r)$, which requires that Shakey be *At* x and that x and y are locations *In* the same room r . 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: $Push(b, x, y, r)$. You will need the predicate *Box* and constants for the boxes.
- Climb onto a box from position x : $ClimbUp(x, b)$; climb down from a box to position x : $ClimbDown(b, x)$. We will need the predicate *On* and the constant *Floor*.
- Turn a light switch on or off: $TurnOn(s, b)$; $TurnOff(s, b)$. To turn a light on or off, Shakey must be on top of a box at the light switch's location.

Write PDDL sentences for Shakey's six actions and the initial state from Figure 10.14. Construct a plan for Shakey to get Box_2 into $Room_2$.

10.5 A finite Turing machine has a finite one-dimensional tape of cells, each cell containing one of a finite number of symbols. One cell has a read and write head above it. There is a finite set of states the machine can be in, one of which is the accept state. At each time step, depending on the symbol on the cell under the head and the machine's current state, there are a set of actions we can choose from. Each action involves writing a symbol to the cell under the head, transitioning the machine to a state, and optionally moving the head left or right. The mapping that determines which actions are allowed is the Turing machine's program. Your goal is to control the machine into the accept state.

Represent the Turing machine acceptance problem as a planning problem. If you can do this, it demonstrates that determining whether a planning problem has a solution is at least as hard as the Turing acceptance problem, which is PSPACE-hard.

10.6 Explain why dropping negative effects from every action schema in a planning problem results in a relaxed problem.

10.7 Figure 10.4 (page 371) shows a blocks-world problem that is 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 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.

SUSSMAN ANOMALY

10.8 Prove that backward search with PDDL problems is complete.

10.9 Construct levels 0, 1, and 2 of the planning graph for the problem in Figure 10.1.

10.10 Prove the following assertions about planning graphs:

- a. A literal that does not appear in the final level of the graph cannot be achieved.

- b. The level cost of a literal in a serial graph is no greater than the actual cost of an optimal plan for achieving it.

10.11 The set-level heuristic (see page 382) uses a planning graph to estimate the cost of achieving a conjunctive goal from the current state. What relaxed problem is the set-level heuristic the solution to?

10.12 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?

10.13 We contrasted forward and backward state-space searchers 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.

10.14 Up to now we have assumed that the plans we create always make sure that an action's preconditions are satisfied. Let us now investigate what propositional successor-state axioms such as $HaveArrow^{t+1} \Leftrightarrow (HaveArrow^t \wedge \neg Shoot^t)$ 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 satisfied.
- b. Consider a plan p that contains the actions required to achieve a goal but also includes illegal actions. Is it the case that

$$initial\ state \wedge successor\text{-}state\ axioms \wedge p \models goal?$$

- c. With first-order successor-state axioms in situation calculus, is it possible to prove that a plan containing illegal actions will achieve the goal?

10.15 Consider how to translate a set of action schemas into the successor-state axioms of situation calculus.

- a. Consider the schema for $Fly(p, from, to)$. Write a logical definition for the predicate $Poss(Fly(p, from, to), s)$, which is true if the preconditions for $Fly(p, from, to)$ are satisfied in situation s .
- b. 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.

- c. 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.
- d. Finally, develop a general and precisely specified procedure for carrying out the translation from a set of action schemas to a set of successor-state axioms.

10.16 In the SATPLAN algorithm in Figure 7.22 (page 272), 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 $g^0 \vee g^1 \vee \dots \vee g^{T_{\max}}$.

- 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.

11 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 used in the real world for planning and scheduling the operations of spacecraft, factories, and military campaigns are more complex; they extend both the representation language and the way the planner interacts with the environment. This chapter shows how. Section 11.1 extends the classical language for planning to talk about actions with durations and resource constraints. Section 11.2 describes methods for constructing plans that are organized hierarchically. This allows human experts to communicate to the planner what they know about how to solve the problem. Hierarchy also lends itself to efficient plan construction because the planner can solve a problem at an abstract level before delving into details. Section 11.3 presents agent architectures that can handle uncertain environments and interleave deliberation with execution, and gives some examples of real-world systems. Section 11.4 shows how to plan when the environment contains other agents.

11.1 TIME, SCHEDULES, AND RESOURCES

The classical planning representation talks about *what to do*, and in *what order*, but the representation cannot talk about time: *how long* an action takes and *when* it occurs. For example, the planners of Chapter 10 could produce a schedule for an airline that says which planes are assigned to which flights, but we really need to know departure and arrival times as well. This is the subject matter of **scheduling**. The real world also imposes many **resource constraints**; for example, an airline has a limited number of staff—and staff who are on one flight cannot be on another at the same time. This section covers methods for representing and solving planning problems that include temporal and resource constraints.

The approach we take in this section is “plan first, schedule later”: that is, we divide the overall problem into a *planning* phase in which actions are selected, with some ordering constraints, 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.

JOB

DURATION

CONSUMABLE

REUSABLE

MAKESPAN

$$\begin{aligned} &Jobs(\{AddEngine1 \prec AddWheels1 \prec Inspect1\}, \\ &\quad \{AddEngine2 \prec AddWheels2 \prec Inspect2\}) \\ &Resources(EngineHoists(1), WheelStations(1), Inspectors(2), LugNuts(500)) \\ &Action(AddEngine1, DURATION:30, \\ &\quad USE:EngineHoists(1)) \\ &Action(AddEngine2, DURATION:60, \\ &\quad USE:EngineHoists(1)) \\ &Action(AddWheels1, DURATION:30, \\ &\quad CONSUME:LugNuts(20), USE:WheelStations(1)) \\ &Action(AddWheels2, DURATION:15, \\ &\quad CONSUME:LugNuts(20), USE:WheelStations(1)) \\ &Action(Inspect_i, DURATION:10, \\ &\quad USE:Inspectors(1)) \end{aligned}$$

Figure 11.1 A job-shop scheduling problem for assembling two cars, with resource constraints. The notation $A \prec B$ means that action A must precede action B .

This approach is common in real-world manufacturing and logistical settings, where the planning phase is often performed by human experts. The automated methods of Chapter 10 can also be used for the planning phase, provided that they produce plans with just the minimal ordering constraints required for correctness. GRAPHPLAN (Section 10.3), SATPLAN (Section 10.4.1), and partial-order planners (Section 10.4.4) can do this; search-based methods (Section 10.2) produce totally ordered plans, but these can easily be converted to plans with minimal ordering constraints.

11.1.1 Representing temporal and resource constraints

A typical **job-shop scheduling problem**, as first introduced in Section 6.1.2, consists of a set of **jobs**, each of which consists a collection of **actions** with ordering constraints among them. Each action has a **duration** and a set of resource constraints required by the action. Each constraint specifies a *type* of resource (e.g., bolts, wrenches, or pilots), the number of that resource required, and whether that resource is **consumable** (e.g., the bolts are no longer available for use) or **reusable** (e.g., a pilot is occupied during a flight but is available again when the flight is over). Resources can also be *produced* by actions with negative consumption, including manufacturing, growing, and resupply actions. A solution to a job-shop scheduling problem must specify the start times for each action and must satisfy all the temporal ordering constraints and resource constraints. As with search and planning problems, solutions can be evaluated according to a cost function; this can be quite complicated, with nonlinear resource costs, time-dependent delay costs, and so on. For simplicity, we assume that the cost function is just the total duration of the plan, which is called the **makespan**.

Figure 11.1 shows a simple example: a problem involving the assembly of two cars. The problem consists of two jobs, each of the form $[AddEngine, AddWheels, Inspect]$. Then the

Resources statement declares that there are four types of resources, and gives the number of each type available at the start: 1 engine hoist, 1 wheel station, 2 inspectors, and 500 lug nuts. The action schemas give the duration and resource needs of each action. The lug nuts are *consumed* as wheels are added to the car, whereas the other resources are “borrowed” at the start of an action and released at the action’s end.

AGGREGATION

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 proposed schedule 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 actions.

11.1.2 Solving scheduling problems

CRITICAL PATH
METHOD

We begin by considering just the temporal scheduling problem, ignoring resource constraints. To minimize makespan (plan duration), we must find the earliest start times for all the actions consistent with the ordering constraints supplied with the problem. It is helpful to view these ordering constraints as a directed graph relating the actions, as shown in Figure 11.2. We can apply the **critical path method** (CPM) to this graph to determine the possible start and end times of each action. A **path** through a graph representing 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 11.2.)

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. Actions that are off the critical path have 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 11.2 that the whole plan will take 85 minutes, that each action in the top job has 15 minutes of slack, and that each action on the critical path has no slack (by definition). Together the *ES* and *LS* times for all the actions constitute a **schedule** for the problem.

SLACK

SCHEDULE

The following formulas serve as a definition for *ES* and *LS* and also as the outline of a dynamic-programming algorithm to compute them. *A* and *B* are actions, and $A \prec B$ means that *A* comes before *B*:

$$\begin{aligned}
 ES(Start) &= 0 \\
 ES(B) &= \max_{A \prec B} ES(A) + Duration(A) \\
 LS(Finish) &= ES(Finish) \\
 LS(A) &= \min_{B \succ A} LS(B) - Duration(A) .
 \end{aligned}$$

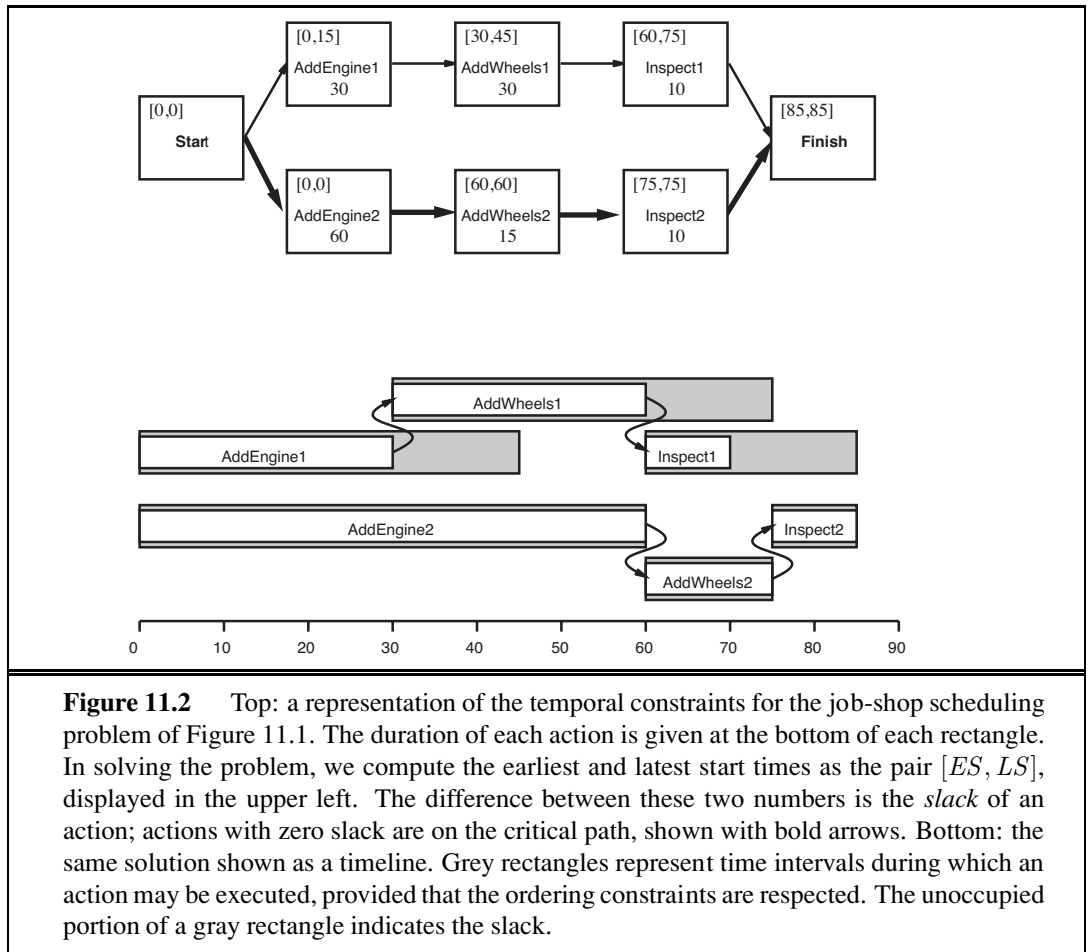
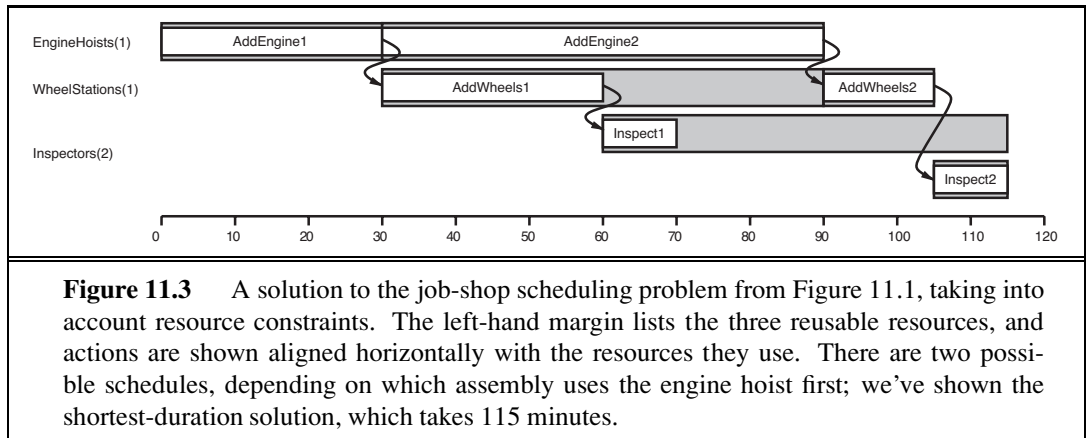


Figure 11.2 Top: a representation of the temporal constraints for the job-shop scheduling problem of Figure 11.1. The duration of each action is given at the bottom of each rectangle. In solving the problem, we compute the earliest and latest start times as the pair $[ES, LS]$, displayed in the upper 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. Bottom: the same solution 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 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 backward from the *Finish* action.

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, finding a minimum-duration schedule, given a partial ordering on the actions and no resource constraints, is quite easy.

Mathematically speaking, critical-path problems are easy to solve because they are defined as a *conjunction* of *linear* inequalities on the start and end times. When we introduce resource constraints, the resulting constraints on start and end times become more complicated. For example, the *AddEngine* actions, which begin at the same time in Figure 11.2,



require the same *EngineHoist* and so cannot overlap. The “cannot overlap” constraint is a *disjunction* of two linear inequalities, one for each possible ordering. The introduction of disjunctions turns out to make scheduling with resource constraints NP-hard.

Figure 11.3 shows the solution with the fastest completion time, 115 minutes. This is 30 minutes longer than the 85 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.

The complexity of scheduling with resource constraints is often seen in practice as well as in theory. A challenge problem posed in 1963—to find the optimal schedule for a 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 Chapters 3 and 4. One simple but popular heuristic is the **minimum slack** algorithm: on each iteration, schedule for the earliest possible start whichever unscheduled action has all its predecessors scheduled and has the least slack; then update the *ES* and *LS* times for each affected action and repeat. The heuristic resembles the minimum-remaining-values (MRV) 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 11.3.

Up to this point, we have assumed that the set of actions and ordering constraints is fixed. Under these assumptions, every scheduling problem can be solved by a nonoverlapping sequence that avoids all resource conflicts, provided that each action is feasible by itself. If a scheduling problem is proving very difficult, however, it may not be a good idea to solve it this way—it may be better to reconsider the actions and constraints, in case that leads to a much easier scheduling problem. Thus, 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 10 can be augmented to handle this information. For example, partial-order planners can detect resource constraint violations in much the same way they detect conflicts with causal links. Heuristics can be devised to estimate the total completion time of a plan. This is currently an active area of research.

11.2 HIERARCHICAL PLANNING

The problem-solving and planning methods of the preceding chapters all operate with a fixed set of atomic actions. Actions can be strung together into sequences or branching networks; state-of-the-art algorithms can generate solutions containing thousands of actions.

For plans executed by the human brain, atomic actions are muscle activations. In very round numbers, we have about 10^3 muscles to activate (639, by some counts, but many of them have multiple subunits); we can modulate their activation perhaps 10 times per second; and we are alive and awake for about 10^9 seconds in all. Thus, a human life contains about 10^{13} actions, give or take one or two orders of magnitude. Even if we restrict ourselves to planning over much shorter time horizons—for example, a two-week vacation in Hawaii—a detailed motor plan would contain around 10^{10} actions. This is a lot more than 1000.

To bridge this gap, AI systems will probably have to do what humans appear to do: plan at higher levels of abstraction. A reasonable plan for the Hawaii vacation might be “Go to San Francisco airport; take Hawaiian Airlines flight 11 to Honolulu; do vacation stuff for two weeks; take Hawaiian Airlines flight 12 back to San Francisco; go home.” Given such a plan, the action “Go to San Francisco airport” can be viewed as a planning task in itself, with a solution such as “Drive to the long-term parking lot; park; take the shuttle to the terminal.” Each of these actions, in turn, can be decomposed further, until we reach the level of actions that can be executed without deliberation to generate the required motor control sequences.

In this example, we see that planning can occur both before and during the execution of the plan; for example, one would probably defer the problem of planning a route from a parking spot in long-term parking to the shuttle bus stop until a particular parking spot has been found during execution. Thus, that particular action will remain at an abstract level prior to the execution phase. We defer discussion of this topic until Section 11.3. Here, we concentrate on the aspect of **hierarchical decomposition**, an idea that pervades almost all attempts to manage complexity. For example, 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 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.

11.2.1 High-level actions

The basic formalism we adopt to understand hierarchical decomposition comes from the area of **hierarchical task networks** or HTN planning. As in classical planning (Chapter 10), we assume full observability and determinism and the availability of a set of actions, now called **primitive actions**, with standard precondition–effect schemas. The key additional concept is the **high-level action** or HLA—for example, the action “Go to San Francisco airport” in the

HIERARCHICAL
DECOMPOSITION

HIERARCHICAL TASK
NETWORK

PRIMITIVE ACTION

HIGH-LEVEL ACTION

```

Refinement(Go(Home, SFO),
  STEPS: [Drive(Home, SFOLongTermParking),
         Shuttle(SFOLongTermParking, SFO)] )
Refinement(Go(Home, SFO),
  STEPS: [Taxi(Home, SFO)] )

```

```

Refinement(Navigate([a, b], [x, y]),
  PRECOND:  $a = x \wedge b = y$ 
  STEPS: [] )
Refinement(Navigate([a, b], [x, y]),
  PRECOND: Connected([a, b], [a - 1, b])
  STEPS: [Left, Navigate([a - 1, b], [x, y])] )
Refinement(Navigate([a, b], [x, y]),
  PRECOND: Connected([a, b], [a + 1, b])
  STEPS: [Right, Navigate([a + 1, b], [x, y])] )
...

```

Figure 11.4 Definitions of possible refinements for two high-level actions: going to San Francisco airport and navigating in the vacuum world. In the latter case, note the recursive nature of the refinements and the use of preconditions.

REFINEMENT

example given earlier. Each HLA has one or more possible **refinements**, into a sequence¹ of actions, each of which may be an HLA or a primitive action (which has no refinements by definition). For example, the action “Go to San Francisco airport,” represented formally as *Go*(*Home*, *SFO*), might have two possible refinements, as shown in Figure 11.4. The same figure shows a **recursive** refinement for navigation in the vacuum world: to get to a destination, take a step, and then go to the destination.

These examples show that high-level actions and their refinements embody knowledge about *how to do things*. For instance, the refinements for *Go*(*Home*, *SFO*) say that to get to the airport you can drive or take a taxi; buying milk, sitting down, and moving the knight to e4 are not to be considered.

IMPLEMENTATION

An HLA refinement that contains only primitive actions is called an **implementation** of the HLA. For example, in the vacuum world, the sequences [*Right*, *Right*, *Down*] and [*Down*, *Right*, *Right*] both implement the HLA *Navigate*([1, 3], [3, 2]). An implementation of a high-level plan (a sequence of HLAs) is the concatenation of implementations of each HLA in the sequence. Given the precondition–effect definitions of each primitive action, it is straightforward to determine whether any given implementation of a high-level plan achieves the goal. We can say, then, that *a high-level plan achieves the goal from a given state if at least one of its implementations achieves the goal from that state*. The “at least one” in this definition is crucial—not *all* implementations need to achieve the goal, because the agent gets



¹ HTN planners often allow refinement into partially ordered plans, and they allow the refinements of two different HLAs in a plan to *share* actions. We omit these important complications in the interest of understanding the basic concepts of hierarchical planning.

to decide which implementation it will execute. Thus, the set of possible implementations in HTN planning—each of which may have a different outcome—is not the same as the set of possible outcomes in nondeterministic planning. There, we required that a plan work for *all* outcomes because the agent doesn't get to choose the outcome; nature does.

The simplest case is an HLA that has exactly one implementation. In that case, we can compute the preconditions and effects of the HLA from those of the implementation (see Exercise 11.3) and then treat the HLA exactly as if it were a primitive action itself. It can be shown that the right collection of HLAs can result in the time complexity of blind search dropping from exponential in the solution depth to linear in the solution depth, although devising such a collection of HLAs may be a nontrivial task in itself. When HLAs have multiple possible implementations, there are two options: one is to search among the implementations for one that works, as in Section 11.2.2; the other is to reason directly about the HLAs—despite the multiplicity of implementations—as explained in Section 11.2.3. The latter method enables the derivation of provably correct abstract plans, without the need to consider their implementations.

11.2.2 Searching for primitive solutions

HTN planning is often formulated with a single “top level” action called *Act*, where the aim is to find an implementation of *Act* that achieves the goal. This approach is entirely general. For example, classical planning problems can be defined as follows: for each primitive action a_i , provide one refinement of *Act* with steps $[a_i, Act]$. That creates a recursive definition of *Act* that lets us add actions. But we need some way to stop the recursion; we do that by providing one more refinement for *Act*, one with an empty list of steps and with a precondition equal to the goal of the problem. This says that if the goal is already achieved, then the right implementation is to do nothing.

The approach leads to a simple algorithm: repeatedly choose an HLA in the current plan and replace it with one of its refinements, until the plan achieves the goal. One possible implementation based on breadth-first tree search is shown in Figure 11.5. Plans are considered in order of depth of nesting of the refinements, rather than number of primitive steps. It is straightforward to design a graph-search version of the algorithm as well as depth-first and iterative deepening versions.

In essence, this form of hierarchical search explores the space of sequences that conform to the knowledge contained in the HLA library about how things are to be done. A great deal of knowledge can be encoded, not just in the action sequences specified in each refinement but also in the preconditions for the refinements. For some domains, HTN planners have been able to generate huge plans with very little search. 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 tens of millions of steps. Another important aspect of HTN plans is that they are, by definition, hierarchically structured; usually this makes them easy for humans to understand.

```

function HIERARCHICAL-SEARCH(problem, hierarchy) returns a solution, or failure
  frontier  $\leftarrow$  a FIFO queue with [Act] as the only element
  loop do
    if EMPTY?(frontier) then return failure
    plan  $\leftarrow$  POP(frontier) /* chooses the shallowest plan in frontier */
    hla  $\leftarrow$  the first HLA in plan, or null if none
    prefix, suffix  $\leftarrow$  the action subsequences before and after hla in plan
    outcome  $\leftarrow$  RESULT(problem.INITIAL-STATE, prefix)
    if hla is null then /* so plan is primitive and outcome is its result */
      if outcome satisfies problem.GOAL then return plan
    else for each sequence in REFINEMENTS(hla, outcome, hierarchy) do
      frontier  $\leftarrow$  INSERT(APPEND(prefix, sequence, suffix), frontier)

```

Figure 11.5 A breadth-first implementation of hierarchical forward planning search. The initial plan supplied to the algorithm is [*Act*]. The REFINEMENTS function returns a set of action sequences, one for each refinement of the HLA whose preconditions are satisfied by the specified state, *outcome*.

The computational benefits of hierarchical search can be seen by examining an idealized case. Suppose that a planning problem has a solution with d primitive actions. For a nonhierarchical, forward state-space planner with b allowable actions at each state, the cost is $O(b^d)$, as explained in Chapter 3. For an HTN planner, let us suppose a very regular refinement structure: each nonprimitive action has r possible refinements, each into k actions at the next lower level. We want to know how many different refinement trees there are with this structure. Now, if there are d actions at the primitive level, then the number of levels below the root is $\log_k d$, so the number of internal refinement nodes is $1 + k + k^2 + \dots + k^{\log_k d - 1} = (d - 1)/(k - 1)$. Each internal node has r possible refinements, so $r^{(d-1)/(k-1)}$ possible regular decomposition trees could be constructed. Examining this formula, we see that keeping r small and k large can result in huge savings: essentially we are taking the k th root of the nonhierarchical cost, if b and r are comparable. Small r and large k means a library of HLAs with a small number of refinements each yielding a long action sequence (that nonetheless allows us to solve any problem). This is not always possible: long action sequences that are usable across a wide range of problems are extremely precious.

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.

11.2.3 Searching for abstract solutions

The hierarchical search algorithm in the preceding section refines HLAs all the way to primitive action sequences to determine if a plan is workable. This contradicts common sense: one should be able to determine that the two-HLA high-level plan

$[Drive(Home, SFO_{LongTermParking}), Shuttle(SFO_{LongTermParking}, SFO)]$

gets one to the airport without having to determine a precise route, choice of parking spot, and so on. The solution seems obvious: write precondition–effect descriptions of the HLAs, just as we write down what the primitive actions do. From the descriptions, it ought to be easy to prove that the high-level plan achieves the goal. This is the holy grail, so to speak, of hierarchical planning because if we derive a high-level plan that provably achieves the goal, working in a small search space of high-level actions, then we can commit to that plan and work on the problem of refining each step of the plan. This gives us the exponential reduction we seek. For this to work, it has to be the case that every high-level plan that “claims” to achieve the goal (by virtue of the descriptions of its steps) does in fact achieve the goal in the sense defined earlier: it must have at least one implementation that does achieve the goal. This property has been called the **downward refinement property** for HLA descriptions.

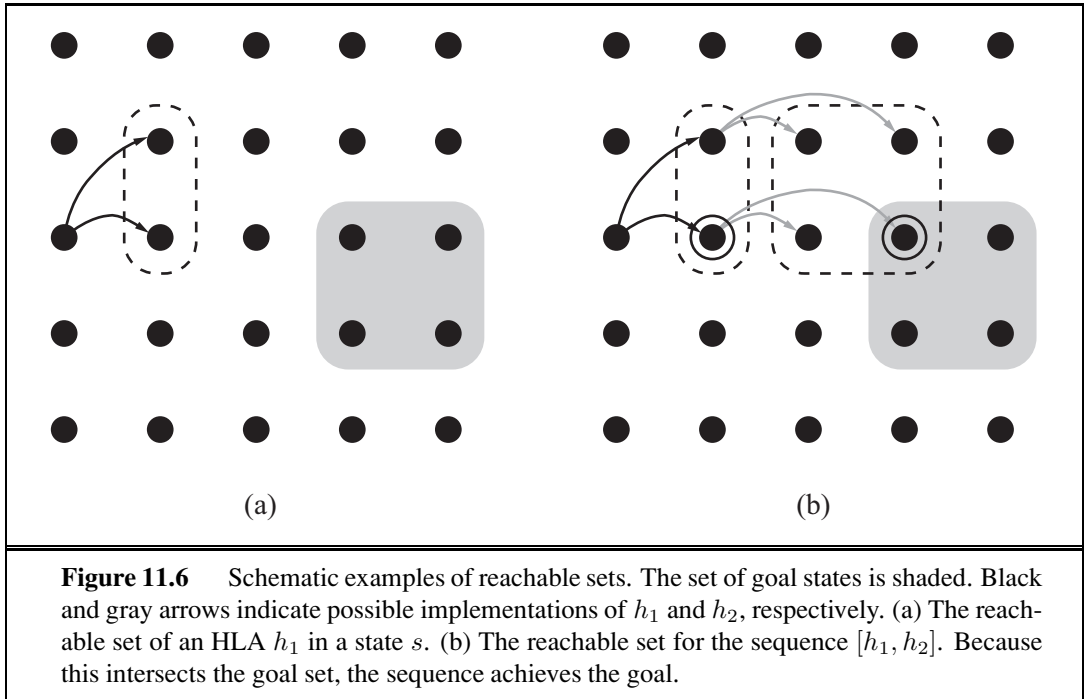
DOWNWARD
REFINEMENT
PROPERTY

Writing HLA descriptions that satisfy the downward refinement property is, in principle, easy: as long as the descriptions are *true*, then any high-level plan that claims to achieve the goal must in fact do so—otherwise, the descriptions are making some false claim about what the HLAs do. We have already seen how to write true descriptions for HLAs that have exactly one implementation (Exercise 11.3); a problem arises when the HLA has *multiple* implementations. How can we describe the effects of an action that can be implemented in many different ways?

One safe answer (at least for problems where all preconditions and goals are positive) is to include only the positive effects that are achieved by *every* implementation of the HLA and the negative effects of *any* implementation. Then the downward refinement property would be satisfied. Unfortunately, this semantics for HLAs is much too conservative. Consider again the HLA $Go(Home, SFO)$, which has two refinements, and suppose, for the sake of argument, a simple world in which one can always drive to the airport and park, but taking a taxi requires *Cash* as a precondition. In that case, $Go(Home, SFO)$ doesn’t always get you to the airport. In particular, it fails if *Cash* is false, and so we cannot assert $At(Agent, SFO)$ as an effect of the HLA. This makes no sense, however; if the agent didn’t have *Cash*, it would drive itself. Requiring that an effect hold for *every* implementation is equivalent to assuming that *someone else*—an adversary—will choose the implementation. It treats the HLA’s multiple outcomes exactly as if the HLA were a **nondeterministic** action, as in Section 4.3. For our case, the agent itself will choose the implementation.

The programming languages community has coined the term **demonic nondeterminism** for the case where an adversary makes the choices, contrasting this with **angelic nonde-**

DEMONIC
NONDETERMINISM



ANGELIC
NONDETERMINISM
ANGELIC SEMANTICS
REACHABLE SET

terminism, where the agent itself makes the choices. We borrow this term to define **angelic semantics** for HLA descriptions. The basic concept required for understanding angelic semantics is the **reachable set** of an HLA: given a state s , the reachable set for an HLA h , written as $\text{REACH}(s, h)$, is the set of states reachable by any of the HLA's implementations. The key idea is that the agent can choose *which* element of the reachable set it ends up in when it executes the HLA; thus, an HLA with multiple refinements is more “powerful” than the same HLA with fewer refinements. We can also define the reachable set of a sequences of HLAs. For example, the reachable set of a sequence $[h_1, h_2]$ is the union of all the reachable sets obtained by applying h_2 in each state in the reachable set of h_1 :

$$\text{REACH}(s, [h_1, h_2]) = \bigcup_{s' \in \text{REACH}(s, h_1)} \text{REACH}(s', h_2).$$

Given these definitions, a high-level plan—a sequence of HLAs—achieves the goal if its reachable set *intersects* the set of goal states. (Compare this to the much stronger condition for demonic semantics, where every member of the reachable set has to be a goal state.) Conversely, if the reachable set doesn't intersect the goal, then the plan definitely doesn't work. Figure 11.6 illustrates these ideas.

The notion of reachable sets yields a straightforward algorithm: search among high-level plans, looking for one whose reachable set intersects the goal; once that happens, the algorithm can *commit* to that abstract plan, knowing that it works, and focus on refining the plan further. We will come back to the algorithmic issues later; first, we consider the question of how the effects of an HLA—the reachable set for each possible initial state—are represented. As with the classical action schemas of Chapter 10, we represent the *changes*

made to each fluent. Think of a fluent as a state variable. A primitive action can *add* or *delete* a variable or leave it *unchanged*. (With conditional effects (see Section 11.3.1) there is a fourth possibility: flipping a variable to its opposite.)

An HLA under angelic semantics can do more: it can *control* the value of a variable, setting it to true or false depending on which implementation is chosen. In fact, an HLA can have nine different effects on a variable: if the variable starts out true, it can always keep it true, always make it false, or have a choice; if the variable starts out false, it can always keep it false, always make it true, or have a choice; and the three choices for each case can be combined arbitrarily, making nine. Notationally, this is a bit challenging. We'll use the \sim symbol to mean “possibly, if the agent so chooses.” Thus, an effect $\tilde{+}A$ means “possibly add A ,” that is, either leave A unchanged or make it true. Similarly, $\tilde{-}A$ means “possibly delete A ” and $\tilde{\pm}A$ means “possibly add or delete A .” For example, the HLA $Go(Home, SFO)$, with the two refinements shown in Figure 11.4, possibly deletes *Cash* (if the agent decides to take a taxi), so it should have the effect $\tilde{-}Cash$. Thus, we see that the descriptions of HLAs are *derivable*, in principle, from the descriptions of their refinements—in fact, this is required if we want true HLA descriptions, such that the downward refinement property holds. Now, suppose we have the following schemas for the HLAs h_1 and h_2 :

$$\begin{aligned} Action(h_1, \text{PRECOND: } \neg A, \text{EFFECT: } A \wedge \tilde{-}B) , \\ Action(h_2, \text{PRECOND: } \neg B, \text{EFFECT: } \tilde{+}A \wedge \tilde{\pm}C) . \end{aligned}$$

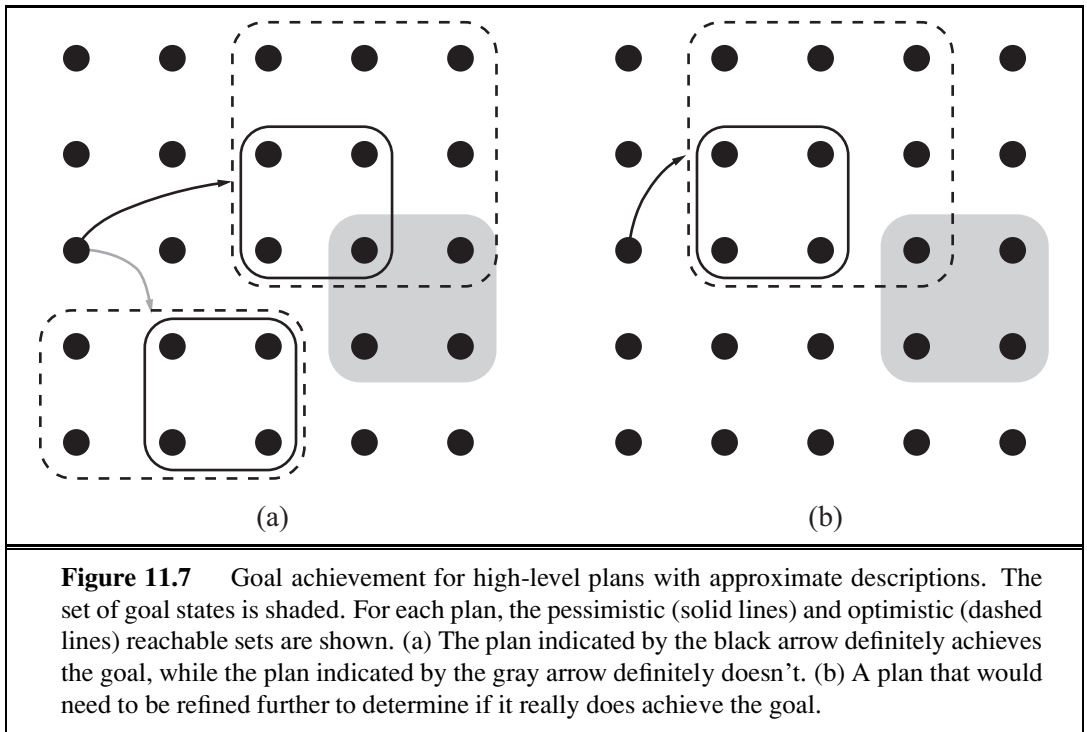
That is, h_1 adds A and possibly deletes B , while h_2 possibly adds A and has full control over C . Now, if only B is true in the initial state and the goal is $A \wedge C$ then the sequence $[h_1, h_2]$ achieves the goal: we choose an implementation of h_1 that makes B false, then choose an implementation of h_2 that leaves A true and makes C true.

The preceding discussion assumes that the effects of an HLA—the reachable set for any given initial state—can be described exactly by describing the effect on each variable. It would be nice if this were always true, but in many cases we can only approximate the effects because an HLA may have infinitely many implementations and may produce arbitrarily wiggly reachable sets—rather like the wiggly-belief-state problem illustrated in Figure 7.21 on page 271. For example, we said that $Go(Home, SFO)$ possibly deletes *Cash*; it also possibly adds $At(Car, SFOLongTermParking)$; but it cannot do both—in fact, it must do exactly one. As with belief states, we may need to write *approximate* descriptions. We will use two kinds of approximation: an **optimistic description** $REACH^+(s, h)$ of an HLA h may overstate the reachable set, while a **pessimistic description** $REACH^-(s, h)$ may understate the reachable set. Thus, we have

$$REACH^-(s, h) \subseteq REACH(s, h) \subseteq REACH^+(s, h) .$$

For example, an optimistic description of $Go(Home, SFO)$ says that it possibly deletes *Cash* and possibly adds $At(Car, SFOLongTermParking)$. Another good example arises in the 8-puzzle, half of whose states are unreachable from any given state (see Exercise 3.4 on page 113): the optimistic description of Act might well include the whole state space, since the exact reachable set is quite wiggly.

With approximate descriptions, the test for whether a plan achieves the goal needs to be modified slightly. If the optimistic reachable set for the plan doesn't intersect the goal,



then the plan doesn't work; if the pessimistic reachable set intersects the goal, then the plan does work (Figure 11.7(a)). With exact descriptions, a plan either works or it doesn't, but with approximate descriptions, there is a middle ground: if the optimistic set intersects the goal but the pessimistic set doesn't, then we cannot tell if the plan works (Figure 11.7(b)). When this circumstance arises, the uncertainty can be resolved by refining the plan. This is a very common situation in human reasoning. For example, in planning the aforementioned two-week Hawaii vacation, one might propose to spend two days on each of seven islands. Prudence would indicate that this ambitious plan needs to be refined by adding details of inter-island transportation.

An algorithm for hierarchical planning with approximate angelic descriptions is shown in Figure 11.8. For simplicity, we have kept to the same overall scheme used previously in Figure 11.5, that is, a breadth-first search in the space of refinements. As just explained, the algorithm can detect plans that will and won't work by checking the intersections of the optimistic and pessimistic reachable sets with the goal. (The details of how to compute the reachable sets of a plan, given approximate descriptions of each step, are covered in Exercise 11.5.) When a workable abstract plan is found, the algorithm *decomposes* the original problem into subproblems, one for each step of the plan. The initial state and goal for each subproblem are obtained by regressing a guaranteed-reachable goal state through the action schemas for each step of the plan. (See Section 10.2.2 for a discussion of how regression works.) Figure 11.6(b) illustrates the basic idea: the right-hand circled state is the guaranteed-reachable goal state, and the left-hand circled state is the intermediate goal obtained by regressing the

```

function ANGELIC-SEARCH(problem, hierarchy, initialPlan) returns solution or fail
  frontier  $\leftarrow$  a FIFO queue with initialPlan as the only element
  loop do
    if EMPTY?(frontier) then return fail
    plan  $\leftarrow$  POP(frontier) /* chooses the shallowest node in frontier */
    if REACH+(problem.INITIAL-STATE, plan) intersects problem.GOAL then
      if plan is primitive then return plan /* REACH+ is exact for primitive plans */
      guaranteed  $\leftarrow$  REACH-(problem.INITIAL-STATE, plan)  $\cap$  problem.GOAL
      if guaranteed  $\neq \{\}$  and MAKING-PROGRESS(plan, initialPlan) then
        finalState  $\leftarrow$  any element of guaranteed
        return DECOMPOSE(hierarchy, problem.INITIAL-STATE, plan, finalState)
      hla  $\leftarrow$  some HLA in plan
      prefix, suffix  $\leftarrow$  the action subsequences before and after hla in plan
      for each sequence in REFINEMENTS(hla, outcome, hierarchy) do
        frontier  $\leftarrow$  INSERT(APPEND(prefix, sequence, suffix), frontier)



---


function DECOMPOSE(hierarchy, s0, plan, sf) returns a solution
  solution  $\leftarrow$  an empty plan
  while plan is not empty do
    action  $\leftarrow$  REMOVE-LAST(plan)
    si  $\leftarrow$  a state in REACH-(s0, plan) such that sf  $\in$  REACH-(si, action)
    problem  $\leftarrow$  a problem with INITIAL-STATE = si and GOAL = sf
    solution  $\leftarrow$  APPEND(ANGELIC-SEARCH(problem, hierarchy, action), solution)
    sf  $\leftarrow$  si
  return solution

```

Figure 11.8 A hierarchical planning algorithm that uses angelic semantics to identify and commit to high-level plans that work while avoiding high-level plans that don't. The predicate MAKING-PROGRESS checks to make sure that we aren't stuck in an infinite regression of refinements. At top level, call ANGELIC-SEARCH with $[Act]$ as the *initialPlan*.

goal through the final action.

The ability to commit to or reject high-level plans can give ANGELIC-SEARCH a significant computational advantage over HIERARCHICAL-SEARCH, which in turn may have a large advantage over plain old BREADTH-FIRST-SEARCH. Consider, for example, cleaning up a large vacuum world consisting of rectangular rooms connected by narrow corridors. It makes sense to have an HLA for *Navigate* (as shown in Figure 11.4) and one for *CleanWholeRoom*. (Cleaning the room could be implemented with the repeated application of another HLA to clean each row.) Since there are five actions in this domain, the cost for BREADTH-FIRST-SEARCH grows as 5^d , where d is the length of the shortest solution (roughly twice the total number of squares); the algorithm cannot manage even two 2×2 rooms. HIERARCHICAL-SEARCH is more efficient, but still suffers from exponential growth because it tries all ways of cleaning that are consistent with the hierarchy. ANGELIC-SEARCH scales approximately linearly in the number of squares—it commits to a good high-level se-

quence and prunes away the other options. Notice that cleaning a set of rooms by cleaning each room in turn is hardly rocket science: it is easy for humans precisely because of the hierarchical structure of the task. When we consider how difficult humans find it to solve small puzzles such as the 8-puzzle, it seems likely that the human capacity for solving complex problems derives to a great extent from their skill in abstracting and decomposing the problem to eliminate combinatorics.

The angelic approach can be extended to find least-cost solutions by generalizing the notion of reachable set. Instead of a state being reachable or not, it has a cost for the most efficient way to get there. (The cost is ∞ for unreachable states.) The optimistic and pessimistic descriptions bound these costs. In this way, angelic search can find provably optimal abstract plans without considering their implementations. The same approach can be used to obtain effective **hierarchical lookahead** algorithms for online search, in the style of LRTA* (page 152). In some ways, such algorithms mirror aspects of human deliberation in tasks such as planning a vacation to Hawaii—consideration of alternatives is done initially at an abstract level over long time scales; some parts of the plan are left quite abstract until execution time, such as how to spend two lazy days on Molokai, while others parts are planned in detail, such as the flights to be taken and lodging to be reserved—without these refinements, there is no guarantee that the plan would be feasible.

HIERARCHICAL
LOOKAHEAD

11.3 PLANNING AND ACTING IN NONDETERMINISTIC DOMAINS

In this section we extend planning to handle partially observable, nondeterministic, and unknown environments. Chapter 4 extended search in similar ways, and the methods here are also similar: **sensorless planning** (also known as **conformant planning**) for environments with no observations; **contingency planning** for partially observable and nondeterministic environments; and **online planning** and **replanning** for unknown environments.

While the basic concepts are the same as in Chapter 4, there are also significant differences. These arise because planners deal with factored representations rather than atomic representations. This affects the way we represent the agent's capability for action and observation and the way we represent **belief states**—the sets of possible physical states the agent might be in—for unobservable and partially observable environments. We can also take advantage of many of the domain-independent methods given in Chapter 10 for calculating search heuristics.

Consider this problem: given a chair and a table, the goal is to have them match—have the same color. In the initial state we have two cans of paint, but the colors of the paint and the furniture are unknown. Only the table is initially in the agent's field of view:

$$\begin{aligned} &Init(Object(Table) \wedge Object(Chair) \wedge Can(C_1) \wedge Can(C_2) \wedge InView(Table)) \\ &Goal(Color(Chair, c) \wedge Color(Table, c)) \end{aligned}$$

There are two actions: removing the lid from a paint can and painting an object using the paint from an open can. The action schemas are straightforward, with one exception: we now allow preconditions and effects to contain variables that are not part of the action's variable

list. That is, $Paint(x, can)$ does not mention the variable c , representing the color of the paint in the can. In the fully observable case, this is not allowed—we would have to name the action $Paint(x, can, c)$. But in the partially observable case, we might or might not know what color is in the can. (The variable c is universally quantified, just like all the other variables in an action schema.)

$$\begin{aligned} &Action(RemoveLid(can), \\ &\quad PRECOND: Can(can) \\ &\quad EFFECT: Open(can)) \\ &Action(Paint(x, can), \\ &\quad PRECOND: Object(x) \wedge Can(can) \wedge Color(can, c) \wedge Open(can) \\ &\quad EFFECT: Color(x, c)) \end{aligned}$$

To solve a partially observable problem, the agent will have to reason about the percepts it will obtain when it is executing the plan. The percept will be supplied by the agent's sensors when it is actually acting, but when it is planning it will need a model of its sensors. In Chapter 4, this model was given by a function, $PERCEPT(s)$. For planning, we augment PDDL with a new type of schema, the **percept schema**:

PERCEPT SCHEMA

$$\begin{aligned} &Percept(Color(x, c), \\ &\quad PRECOND: Object(x) \wedge InView(x)) \\ &Percept(Color(can, c), \\ &\quad PRECOND: Can(can) \wedge InView(can) \wedge Open(can)) \end{aligned}$$

The first schema says that whenever an object is in view, the agent will perceive the color of the object (that is, for the object x , the agent will learn the truth value of $Color(x, c)$ for all c). The second schema says that if an open can is in view, then the agent perceives the color of the paint in the can. Because there are no exogenous events in this world, the color of an object will remain the same, even if it is not being perceived, until the agent performs an action to change the object's color. Of course, the agent will need an action that causes objects (one at a time) to come into view:

$$\begin{aligned} &Action(LookAt(x), \\ &\quad PRECOND: InView(y) \wedge (x \neq y) \\ &\quad EFFECT: InView(x) \wedge \neg InView(y)) \end{aligned}$$

For a fully observable environment, we would have a *Percept* axiom with no preconditions for each fluent. A sensorless agent, on the other hand, has no *Percept* axioms at all. Note that even a sensorless agent can solve the painting problem. One 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).

A contingent planning agent with sensors can generate a better plan. First, look at the table and chair to obtain their colors; if they are already the same then the plan is done. If not, look at the paint cans; if the paint in a can is the same color as one piece of furniture, then apply that paint to the other piece. Otherwise, paint both pieces with any color.

Finally, an online planning agent might generate a contingent plan with fewer branches at first—perhaps ignoring the possibility that no cans match any of the furniture—and deal

with problems when they arise by replanning. It could also deal with incorrectness of its action schemas. Whereas a contingent planner simply assumes that the effects of an action always succeed—that painting the chair does the job—a replanning agent would check the result and make an additional plan to fix any unexpected failure, such as an unpainted area or the original color showing through.

In the real world, agents use a combination of approaches. Car manufacturers sell spare tires and air bags, which are physical embodiments of contingent plan branches designed to handle punctures or crashes. On the other hand, most car drivers never consider these possibilities; when a problem arises they respond as replanning agents. In general, agents plan only for contingencies that have important consequences and a nonnegligible chance of happening. Thus, a car driver contemplating a trip across the Sahara desert should make explicit contingency plans for breakdowns, whereas a trip to the supermarket requires less advance planning. We next look at each of the three approaches in more detail.

11.3.1 Sensorless planning

Section 4.4.1 (page 138) introduced the basic idea of searching in belief-state space to find a solution for sensorless problems. Conversion of a sensorless planning problem to a belief-state planning problem works much the same way as it did in Section 4.4.1; the main differences are that the underlying physical transition model is represented by a collection of action schemas and the belief state can be represented by a logical formula instead of an explicitly enumerated set of states. For simplicity, we assume that the underlying planning problem is deterministic.

The initial belief state for the sensorless painting problem can ignore *InView* fluents because the agent has no sensors. Furthermore, we take as given the unchanging facts $Object(Table) \wedge Object(Chair) \wedge Can(C_1) \wedge Can(C_2)$ because these hold in every belief state. The agent doesn't know the colors of the cans or the objects, or whether the cans are open or closed, but it does know that objects and cans have colors: $\forall x \exists c Color(x, c)$. After Skolemizing, (see Section 9.5), we obtain the initial belief state:

$$b_0 = Color(x, C(x)) .$$

In classical planning, where the **closed-world assumption** is made, we would assume that any fluent not mentioned in a state is false, but in sensorless (and partially observable) planning we have to switch to an **open-world assumption** in which states contain both positive and negative fluents, and if a fluent does not appear, its value is unknown. Thus, the belief state corresponds exactly to the set of possible worlds that satisfy the formula. Given this initial belief state, the following action sequence is a solution:

$$[RemoveLid(Can_1), Paint(Chair, Can_1), Paint(Table, Can_1)] .$$

We now show how to progress the belief state through the action sequence to show that the final belief state satisfies the goal.

First, note that in a given belief state b , the agent can consider any action whose preconditions are satisfied by b . (The other actions cannot be used because the transition model doesn't define the effects of actions whose preconditions might be unsatisfied.) According

to Equation (4.4) (page 139), the general formula for updating the belief state b given an applicable action a in a deterministic world is as follows:

$$b' = \text{RESULT}(b, a) = \{s' : s' = \text{RESULT}_P(s, a) \text{ and } s \in b\}$$

where RESULT_P defines the physical transition model. For the time being, we assume that the initial belief state is always a conjunction of literals, that is, a 1-CNF formula. To construct the new belief state b' , we must consider what happens to each literal ℓ in each physical state s in b when action a is applied. For literals whose truth value is already known in b , the truth value in b' is computed from the current value and the add list and delete list of the action. (For example, if ℓ is in the delete list of the action, then $\neg\ell$ is added to b' .) What about a literal whose truth value is unknown in b ? There are three cases:

1. If the action adds ℓ , then ℓ will be true in b' regardless of its initial value.
2. If the action deletes ℓ , then ℓ will be false in b' regardless of its initial value.
3. If the action does not affect ℓ , then ℓ will retain its initial value (which is unknown) and will not appear in b' .

Hence, we see that the calculation of b' is almost identical to the observable case, which was specified by Equation (10.1) on page 368:

$$b' = \text{RESULT}(b, a) = (b - \text{DEL}(a)) \cup \text{ADD}(a) .$$

We cannot quite use the set semantics because (1) we must make sure that b' does not contain both ℓ and $\neg\ell$, and (2) atoms may contain unbound variables. But it is still the case that $\text{RESULT}(b, a)$ is computed by starting with b , setting any atom that appears in $\text{DEL}(a)$ to false, and setting any atom that appears in $\text{ADD}(a)$ to true. For example, if we apply $\text{RemoveLid}(\text{Can}_1)$ to the initial belief state b_0 , we get

$$b_1 = \text{Color}(x, C(x)) \wedge \text{Open}(\text{Can}_1) .$$

When we apply the action $\text{Paint}(\text{Chair}, \text{Can}_1)$, the precondition $\text{Color}(\text{Can}_1, c)$ is satisfied by the known literal $\text{Color}(x, C(x))$ with binding $\{x/\text{Can}_1, c/C(\text{Can}_1)\}$ and the new belief state is

$$b_2 = \text{Color}(x, C(x)) \wedge \text{Open}(\text{Can}_1) \wedge \text{Color}(\text{Chair}, C(\text{Can}_1)) .$$

Finally, we apply the action $\text{Paint}(\text{Table}, \text{Can}_1)$ to obtain

$$b_3 = \text{Color}(x, C(x)) \wedge \text{Open}(\text{Can}_1) \wedge \text{Color}(\text{Chair}, C(\text{Can}_1)) \\ \wedge \text{Color}(\text{Table}, C(\text{Can}_1)) .$$

The final belief state satisfies the goal, $\text{Color}(\text{Table}, c) \wedge \text{Color}(\text{Chair}, c)$, with the variable c bound to $C(\text{Can}_1)$.



The preceding analysis of the update rule has shown a very important fact: *the family of belief states defined as conjunctions of literals is closed under updates defined by PDDL action schemas*. That is, if the belief state starts as a conjunction of literals, then any update will yield a conjunction of literals. That means that in a world with n fluents, any belief state can be represented by a conjunction of size $O(n)$. This is a very comforting result, considering that there are 2^n states in the world. It says we can compactly represent all the subsets of those 2^n states that we will ever need. Moreover, the process of checking for belief

states that are subsets or supersets of previously visited belief states is also easy, at least in the propositional case.

The fly in the ointment of this pleasant picture is that it only works for action schemas that have the *same effects* for all states in which their preconditions are satisfied. It is this property that enables the preservation of the 1-CNF belief-state representation. As soon as the effect can depend on the state, dependencies are introduced between fluents and the 1-CNF property is lost. Consider, for example, the simple vacuum world defined in Section 3.2.1. Let the fluents be *AtL* and *AtR* for the location of the robot and *CleanL* and *CleanR* for the state of the squares. According to the definition of the problem, the *Suck* action has no precondition—it can always be done. The difficulty is that its effect depends on the robot’s location: when the robot is *AtL*, the result is *CleanL*, but when it is *AtR*, the result is *CleanR*. For such actions, our action schemas will need something new: a **conditional effect**. These have the syntax “**when** *condition*: *effect*,” where *condition* is a logical formula to be compared against the current state, and *effect* is a formula describing the resulting state. For the vacuum world, we have

Action(*Suck*,
EFFECT:**when** *AtL*: *CleanL* \wedge **when** *AtR*: *CleanR*) .

When applied to the initial belief state *True*, the resulting belief state is $(AtL \wedge CleanL) \vee (AtR \wedge CleanR)$, which is no longer in 1-CNF. (This transition can be seen in Figure 4.14 on page 141.) In general, conditional effects can induce arbitrary dependencies among the fluents in a belief state, leading to belief states of exponential size in the worst case.

It is important to understand the difference between preconditions and conditional effects. *All* conditional effects whose conditions are satisfied have their effects applied to generate the resulting state; if none are satisfied, then the resulting state is unchanged. On the other hand, if a *precondition* is unsatisfied, then the action is inapplicable and the resulting state is undefined. From the point of view of sensorless planning, it is better to have conditional effects than an inapplicable action. For example, we could split *Suck* into two actions with unconditional effects as follows:

Action(*SuckL*,
PRECOND:*AtL*; EFFECT:*CleanL*)
Action(*SuckR*,
PRECOND:*AtR*; EFFECT:*CleanR*) .

Now we have only unconditional schemas, so the belief states all remain in 1-CNF; unfortunately, we cannot determine the applicability of *SuckL* and *SuckR* in the initial belief state.

It seems inevitable, then, that nontrivial problems will involve wiggly belief states, just like those encountered when we considered the problem of state estimation for the wumpus world (see Figure 7.21 on page 271). The solution suggested then was to use a **conservative approximation** to the exact belief state; for example, the belief state can remain in 1-CNF if it contains all literals whose truth values can be determined and treats all other literals as unknown. While this approach is *sound*, in that it never generates an incorrect plan, it is *incomplete* because it may be unable to find solutions to problems that necessarily involve interactions among literals. To give a trivial example, if the goal is for the robot to be on

a clean square, then $[Suck]$ is a solution but a sensorless agent that insists on 1-CNF belief states will not find it.

Perhaps a better solution is to look for action sequences that keep the belief state as simple as possible. For example, in the sensorless vacuum world, the action sequence $[Right, Suck, Left, Suck]$ generates the following sequence of belief states:

$$\begin{aligned} b_0 &= True \\ b_1 &= AtR \\ b_2 &= AtR \wedge CleanR \\ b_3 &= AtL \wedge CleanR \\ b_4 &= AtL \wedge CleanR \wedge CleanL \end{aligned}$$

That is, the agent *can* solve the problem while retaining a 1-CNF belief state, even though some sequences (e.g., those beginning with *Suck*) go outside 1-CNF. The general lesson is not lost on humans: we are always performing little actions (checking the time, patting our pockets to make sure we have the car keys, reading street signs as we navigate through a city) to eliminate uncertainty and keep our belief state manageable.

There is another, quite different approach to the problem of unmanageably wiggly belief states: don't bother computing them at all. Suppose the initial belief state is b_0 and we would like to know the belief state resulting from the action sequence $[a_1, \dots, a_m]$. Instead of computing it explicitly, just represent it as " b_0 then $[a_1, \dots, a_m]$." This is a lazy but unambiguous representation of the belief state, and it's quite concise— $O(n + m)$ where n is the size of the initial belief state (assumed to be in 1-CNF) and m is the maximum length of an action sequence. As a belief-state representation, it suffers from one drawback, however: determining whether the goal is satisfied, or an action is applicable, may require a lot of computation.

The computation can be implemented as an entailment test: if A_m represents the collection of successor-state axioms required to define occurrences of the actions a_1, \dots, a_m —as explained for SATPLAN in Section 10.4.1—and G_m asserts that the goal is true after m steps, then the plan achieves the goal if $b_0 \wedge A_m \models G_m$, that is, if $b_0 \wedge A_m \wedge \neg G_m$ is unsatisfiable. Given a modern SAT solver, it may be possible to do this much more quickly than computing the full belief state. For example, if none of the actions in the sequence has a particular goal fluent in its add list, the solver will detect this immediately. It also helps if partial results about the belief state—for example, fluents known to be true or false—are cached to simplify subsequent computations.

The final piece of the sensorless planning puzzle is a heuristic function to guide the search. The meaning of the heuristic function is the same as for classical planning: an estimate (perhaps admissible) of the cost of achieving the goal from the given belief state. With belief states, we have one additional fact: solving any subset of a belief state is necessarily easier than solving the belief state:

$$\text{if } b_1 \subseteq b_2 \text{ then } h^*(b_1) \leq h^*(b_2) .$$

Hence, any admissible heuristic computed for a subset is admissible for the belief state itself. The most obvious candidates are the singleton subsets, that is, individual physical states. We

can take any random collection of states s_1, \dots, s_N that are in the belief state b , apply any admissible heuristic h from Chapter 10, and return

$$H(b) = \max\{h(s_1), \dots, h(s_N)\}$$

as the heuristic estimate for solving b . We could also use a planning graph directly on b itself: if it is a conjunction of literals (1-CNF), simply set those literals to be the initial state layer of the graph. If b is not in 1-CNF, it may be possible to find sets of literals that together entail b . For example, if b is in disjunctive normal form (DNF), each term of the DNF formula is a conjunction of literals that entails b and can form the initial layer of a planning graph. As before, we can take the maximum of the heuristics obtained from each set of literals. We can also use inadmissible heuristics such as the ignore-delete-lists heuristic (page 377), which seems to work quite well in practice.

11.3.2 Contingent planning

We saw in Chapter 4 that contingent planning—the generation of plans with conditional branching based on percepts—is appropriate for environments with partial observability, non-determinism, or both. For the partially observable painting problem with the percept axioms given earlier, one possible contingent solution is as follows:

```
[LookAt(Table), LookAt(Chair),
  if Color(Table, c) ∧ Color(Chair, c) then NoOp
  else [RemoveLid(Can1), LookAt(Can1), RemoveLid(Can2), LookAt(Can2),
    if Color(Table, c) ∧ Color(can, c) then Paint(Chair, can)
    else if Color(Chair, c) ∧ Color(can, c) then Paint(Table, can)
    else [Paint(Chair, Can1), Paint(Table, Can1)]]]
```

Variables in this plan should be considered existentially quantified; the second line says that if there exists some color c that is the color of the table and the chair, then the agent need not do anything to achieve the goal. When executing this plan, a contingent-planning agent can maintain its belief state as a logical formula and evaluate each branch condition by determining if the belief state entails the condition formula or its negation. (It is up to the contingent-planning algorithm to make sure that the agent will never end up in a belief state where the condition formula's truth value is unknown.) Note that with first-order conditions, the formula may be satisfied in more than one way; for example, the condition $\text{Color}(\text{Table}, c) \wedge \text{Color}(\text{can}, c)$ might be satisfied by $\{\text{can}/\text{Can}_1\}$ and by $\{\text{can}/\text{Can}_2\}$ if both cans are the same color as the table. In that case, the agent can choose any satisfying substitution to apply to the rest of the plan.

As shown in Section 4.4.2, calculating the new belief state after an action and subsequent percept is done in two stages. The first stage calculates the belief state after the action, just as for the sensorless agent:

$$\hat{b} = (b - \text{DEL}(a)) \cup \text{ADD}(a)$$

where, as before, we have assumed a belief state represented as a conjunction of literals. The second stage is a little trickier. Suppose that percept literals p_1, \dots, p_k are received. One might think that we simply need to add these into the belief state; in fact, we can also infer

that the preconditions for sensing are satisfied. Now, if a percept p has exactly one percept axiom, $Percept(p, PRECOND:c)$, where c is a conjunction of literals, then those literals can be thrown into the belief state along with p . On the other hand, if p has more than one percept axiom whose preconditions might hold according to the predicted belief state \hat{b} , then we have to add in the *disjunction* of the preconditions. Obviously, this takes the belief state outside 1-CNF and brings up the same complications as conditional effects, with much the same classes of solutions.

Given a mechanism for computing exact or approximate belief states, we can generate contingent plans with an extension of the AND–OR forward search over belief states used in Section 4.4. Actions with nondeterministic effects—which are defined simply by using a disjunction in the EFFECT of the action schema—can be accommodated with minor changes to the belief-state update calculation and no change to the search algorithm.² For the heuristic function, many of the methods suggested for sensorless planning are also applicable in the partially observable, nondeterministic case.

11.3.3 Online replanning

Imagine watching a spot-welding robot in a car plant. The robot’s fast, accurate motions are repeated over and over again as each car passes down the line. Although technically impressive, the robot probably does not seem at all *intelligent* because the motion is a fixed, preprogrammed sequence; the robot obviously doesn’t “know what it’s doing” in any meaningful sense. Now suppose that a poorly attached door falls off the car just as the robot is about to apply a spot-weld. The robot quickly replaces its welding actuator with a gripper, picks up the door, checks it for scratches, reattaches it to the car, sends an email to the floor supervisor, switches back to the welding actuator, and resumes its work. All of a sudden, the robot’s behavior seems *purposive* rather than rote; we assume it results not from a vast, precomputed contingent plan but from an online replanning process—which means that the robot *does* need to know what it’s trying to do.

Replanning presupposes some form of **execution monitoring** to determine the need for a new plan. One such need arises when a contingent planning agent gets tired of planning for every little contingency, such as whether the sky might fall on its head.³ Some branches of a partially constructed contingent plan can simply say *Replan*; if such a branch is reached during execution, the agent reverts to planning mode. As we mentioned earlier, the decision as to how much of the problem to solve in advance and how much to leave to replanning is one that involves tradeoffs among possible events with different costs and probabilities of occurring. Nobody wants to have their car break down in the middle of the Sahara desert and only then think about having enough water.

EXECUTION
MONITORING

² If cyclic solutions are required for a nondeterministic problem, AND–OR search must be generalized to a loop version such as LAO* (Hansen and Zilberstein, 2001).

³ In 1954, a Mrs. Hodges of Alabama was hit by meteorite that crashed through her roof. In 1992, a piece of the Mbale meteorite hit a small boy on the head; fortunately, its descent was slowed by banana leaves (Jenniskens *et al.*, 1994). And in 2009, a German boy claimed to have been hit in the hand by a pea-sized meteorite. No serious injuries resulted from any of these incidents, suggesting that the need for preplanning against such contingencies is sometimes overstated.

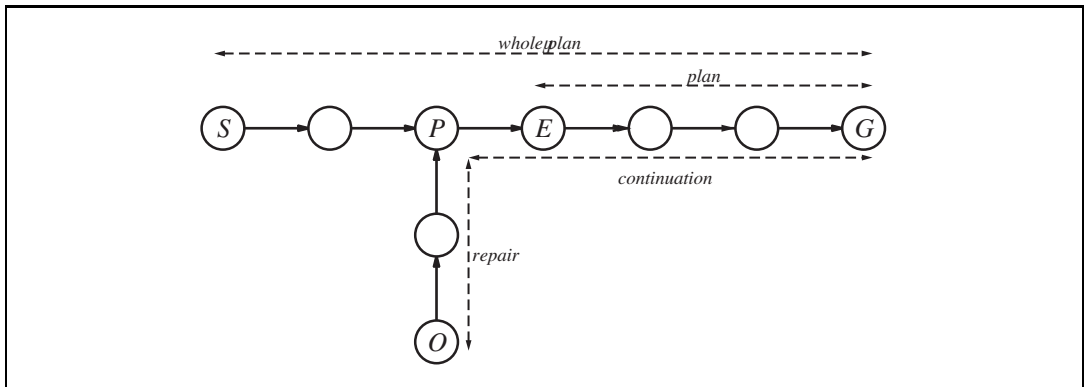


Figure 11.9 Before execution, the planner comes up with a plan, here called *whole plan*, to get from *S* to *G*. The agent executes steps of the plan until it expects to be in state *E*, but observes it is actually in *O*. The agent then replans for the minimal repair plus continuation to reach *G*.

MISSING
PRECONDITION

MISSING EFFECT
MISSING STATE
VARIABLE

EXOGENOUS EVENT

Replanning may also be needed if the agent's model of the world is incorrect. The model for an action may have a **missing precondition**—for example, the agent may not know that removing the lid of a paint can often requires a screwdriver; the model may have a **missing effect**—for example, painting an object may get paint on the floor as well; or the model may have a **missing state variable**—for example, the model given earlier has no notion of the amount of paint in a can, of how its actions affect this amount, or of the need for the amount to be nonzero. The model may also lack provision for **exogenous events** such as someone knocking over the paint can. Exogenous events can also include changes in the goal, such as the addition of the requirement that the table and chair not be painted black. Without the ability to monitor and replan, an agent's behavior is likely to be extremely fragile if it relies on absolute correctness of its model.

The online agent has a choice of how carefully to monitor the environment. We distinguish three levels:

ACTION MONITORING

- **Action monitoring:** before executing an action, the agent verifies that all the preconditions still hold.

PLAN MONITORING

- **Plan monitoring:** before executing an action, the agent verifies that the remaining plan will still succeed.

GOAL MONITORING

- **Goal monitoring:** before executing an action, the agent checks to see if there is a better set of goals it could be trying to achieve.

In Figure 11.9 we see a schematic of action monitoring. The agent keeps track of both its original plan, *wholeplan*, and the part of the plan that has not been executed yet, which is denoted by *plan*. After executing the first few steps of the plan, the agent expects to be in state *E*. But the agent observes it is actually in state *O*. It then needs to repair the plan by finding some point *P* on the original plan that it can get back to. (It may be that *P* is the goal state, *G*.) The agent tries to minimize the total cost of the plan: the repair part (from *O* to *P*) plus the continuation (from *P* to *G*).

Now let's return to the example problem of achieving a chair and table of matching color. Suppose the agent comes up with this plan:

```
[LookAt(Table), LookAt(Chair),
  if Color(Table, c) ∧ Color(Chair, c) then NoOp
  else [RemoveLid(Can1), LookAt(Can1),
    if Color(Table, c) ∧ Color(Can1, c) then Paint(Chair, Can1)
    else REPLAN]] .
```

Now the agent is ready to execute the plan. Suppose the agent observes that the table and can of paint are white and the chair is black. It then executes *Paint(Chair, Can₁)*. At this point a classical planner would declare victory; the plan has been executed. But an online execution monitoring agent needs to check the preconditions of the remaining empty plan—that the table and chair are the same color. Suppose the agent perceives that they do not have the same color—in fact, the chair is now a mottled gray because the black paint is showing through. 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(Chair, Can₁)* action, so the agent chooses the empty sequence for *repair* and makes its *plan* be the same [*Paint*] 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 chair 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. Note also that the original plan need not cover every contingency. If the agent reaches the step marked REPLAN, it can then generate a new plan (perhaps involving *Can₂*).

Action monitoring is a simple method of execution monitoring, but it can sometimes lead to less than intelligent behavior. For example, suppose there is no black or white paint, and the agent constructs a plan to solve the painting problem by painting both the chair and table red. Suppose that there is only enough red paint for the chair. With action monitoring, the agent would go ahead and paint the chair red, then notice that it is out of paint and cannot paint the table, at which point it would replan a repair—perhaps painting both chair and table green. A plan-monitoring agent can detect failure whenever the current state is such that the remaining plan no longer works. Thus, it would not waste time painting the chair red. 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.⁴ Plan monitoring also allows for **serendipity**—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.

It is straightforward to modify a planning algorithm so that each action in the plan is annotated with the action's preconditions, thus enabling action monitoring. It is slightly

⁴ Plan monitoring means that finally, after 424 pages, we have an agent that is smarter than a dung beetle (see page 39). A plan-monitoring agent would notice that the dung ball was missing from its grasp and would replan to get another ball and plug its hole.

more complex to enable plan monitoring. Partial-order and planning-graph planners have the advantage that they have already built up structures that contain the relations necessary for plan monitoring. Augmenting state-space planners with the necessary annotations can be done by careful bookkeeping as the goal fluents are regressed through the plan.

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?” then the answer is no, because the agent could inadvertently arrive at a dead end from which there is no repair. For example, the vacuum agent might have a faulty model of itself and not know that its batteries can run out. Once they do, it cannot repair any plans. If we rule out dead ends—assume that there exists a plan to reach the goal from *any* state in the environment—and 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.

Trouble occurs when an action is actually not nondeterministic, but rather depends on some precondition that the agent does not know about. For example, sometimes a paint can may be empty, so painting from that can has no effect. No amount of retrying is going to change this.⁵ One solution is to choose randomly from among the set of possible repair plans, rather than to try the same one each time. In this case, the repair plan of opening another can might work. A better approach is to **learn** a better model. Every prediction failure is an opportunity for learning; an agent should be able to modify its model of the world to accord with its percepts. From then on, the replanner will be able to come up with a repair that gets at the root problem, rather than relying on luck to choose a good repair. This kind of learning is described in Chapters 18 and 19.

11.4 MULTIAGENT PLANNING

So far, we have assumed that only one agent is doing the sensing, planning, and acting. When there are multiple agents in the environment, each agent faces a **multiagent planning problem** in which it tries to achieve its own goals with the help or hindrance of others.

Between the purely single-agent and truly multiagent cases is a wide spectrum of problems that exhibit various degrees of decomposition of the monolithic agent. An agent with multiple effectors that can operate concurrently—for example, a human who can type and speak at the same time—needs to do **multieffector planning** to manage each effector while handling positive and negative interactions among the effectors. When the effectors are physically decoupled into detached units—as in a fleet of delivery robots in a factory—multieffector planning becomes **multibody planning**. A multibody problem is still a “standard” single-agent problem as long as the relevant sensor information collected by each body can be pooled—either centrally or within each body—to form a common estimate of the world state that then informs the execution of the overall plan; in this case, the multiple bodies act as a single body. When communication constraints make this impossible, we have

MULTIAGENT
PLANNING PROBLEM

MULTIEFFECTOR
PLANNING

MULTIBODY
PLANNING

⁵ Futile repetition of a plan repair is exactly the behavior exhibited by the sphex wasp (page 39).

DECENTRALIZED
PLANNING

what is sometimes called a **decentralized planning** problem; this is perhaps a misnomer, because the planning phase is centralized but the execution phase is at least partially decoupled. In this case, the subplan constructed for each body may need to include explicit communicative actions with other bodies. For example, multiple reconnaissance robots covering a wide area may often be out of radio contact with each other and should share their findings during times when communication is feasible.

COORDINATION

When a single entity is doing the planning, there is really only one goal, which all the bodies necessarily share. When the bodies are distinct agents that do their own planning, they may still share identical goals; for example, two human tennis players who form a doubles team share the goal of winning the match. Even with shared goals, however, the multibody and multiagent cases are quite different. In a multibody robotic doubles team, a single plan dictates which body will go where on the court and which body will hit the ball. In a multiagent doubles team, on the other hand, each agent decides what to do; without some method for **coordination**, both agents may decide to cover the same part of the court and each may leave the ball for the other to hit.

The clearest case of a multiagent problem, of course, is when the agents have different goals. In tennis, the goals of two opposing teams are in direct conflict, leading to the zero-sum situation of Chapter 5. Spectators could be viewed as agents if their support or disdain is a significant factor and can be influenced by the players' conduct; otherwise, they can be treated as an aspect of nature—just like the weather—that is assumed to be indifferent to the players' intentions.⁶

INCENTIVE

Finally, some systems are a mixture of centralized and multiagent planning. For example, a delivery company may do centralized, offline planning for the routes of its trucks and planes each day, but leave some aspects open for autonomous decisions by drivers and pilots who can respond individually to traffic and weather situations. Also, the goals of the company and its employees are brought into alignment, to some extent, by the payment of **incentives** (salaries and bonuses)—a sure sign that this is a true multiagent system.

The issues involved in multiagent planning can be divided roughly into two sets. The first, covered in Section 11.4.1, involves issues of representing and planning for multiple simultaneous actions; these issues occur in all settings from multieffector to multiagent planning. The second, covered in Section 11.4.2, involves issues of cooperation, coordination, and competition arising in true multiagent settings.

11.4.1 Planning with multiple simultaneous actions

MULTIACTOR
ACTOR

For the time being, we will treat the multieffector, multibody, and multiagent settings in the same way, labeling them generically as **multiactor** settings, using the generic term **actor** to cover effectors, bodies, and agents. The goal of this section is to work out how to define transition models, correct plans, and efficient planning algorithms for the multiactor setting. A correct plan is one that, if executed by the actors, achieves the goal. (In the true multiagent setting, of course, the agents may not agree to execute any particular plan, but at least they

⁶ We apologize to residents of the United Kingdom, where the mere act of contemplating a game of tennis guarantees rain.

```

Actors( $A, B$ )
Init( $At(A, LeftBaseline) \wedge At(B, RightNet) \wedge$ 
     $Approaching(Ball, RightBaseline) \wedge Partner(A, B) \wedge Partner(B, A)$ 
     $Goal(Returned(Ball) \wedge (At(a, RightNet) \vee At(a, LeftNet))$ 
     $Action(Hit(actor, Ball),$ 
         $PRECOND: Approaching(Ball, loc) \wedge At(actor, loc)$ 
         $EFFECT: Returned(Ball))$ 
     $Action(Go(actor, to),$ 
         $PRECOND: At(actor, loc) \wedge to \neq loc,$ 
         $EFFECT: At(actor, to) \wedge \neg At(actor, loc))$ 

```

Figure 11.10 The doubles tennis problem. Two actors A and B are playing together and can be in one of four locations: *LeftBaseline*, *RightBaseline*, *LeftNet*, and *RightNet*. The ball can be returned only if a player is in the right place. Note that each action must include the actor as an argument.

will know what plans *would* work if they *did* agree to execute them.) For simplicity, we assume perfect **synchronization**: each action takes the same amount of time and actions at each point in the joint plan are simultaneous.

We begin with the transition model; for the deterministic case, this is the function $RESULT(s, a)$. In the single-agent setting, there might be b different choices for the action; b can be quite large, especially for first-order representations with many objects to act on, but action schemas provide a concise representation nonetheless. In the multiactor setting with n actors, the single action a is replaced by a **joint action** $\langle a_1, \dots, a_n \rangle$, where a_i is the action taken by the i th actor. Immediately, we see two problems: first, we have to describe the transition model for b^n different joint actions; second, we have a joint planning problem with a branching factor of b^n .

Having put the actors together into a multiactor system with a huge branching factor, the principal focus of research on multiactor planning has been to *decouple* the actors to the extent possible, so that the complexity of the problem grows linearly with n rather than exponentially. If the actors have no interaction with one another—for example, n actors each playing a game of solitaire—then we can simply solve n separate problems. If the actors are **loosely coupled**, can we attain something close to this exponential improvement? This is, of course, a central question in many areas of AI. We have seen it explicitly in the context of CSPs, where “tree like” constraint graphs yielded efficient solution methods (see page 225), as well as in the context of disjoint pattern databases (page 106) and additive heuristics for planning (page 378).

The standard approach to loosely coupled problems is to pretend the problems are completely decoupled and then fix up the interactions. For the transition model, this means writing action schemas as if the actors acted independently. Let’s see how this works for the doubles tennis problem. Let’s suppose that at one point in the game, the team has the goal of returning the ball that has been hit to them and ensuring that at least one of them is covering the net.

A first pass at a multiactor definition might look like Figure 11.10. With this definition, it is easy to see that the following **joint plan** works:

JOINT PLAN

PLAN 1:

$A : [Go(A, RightBaseline), Hit(A, Ball)]$
 $B : [NoOp(B), NoOp(B)]$.

Problems arise, however, when a plan has both agents hitting the ball at the same time. In the real world, this won't work, but the action schema for *Hit* says that the ball will be returned successfully. Technically, the difficulty is that preconditions constrain the *state* in which an action can be executed successfully, but do not constrain other actions that might mess it up. We solve this by augmenting action schemas with one new feature: a **concurrent action list** stating which actions must or must not be executed concurrently. For example, the *Hit* action could be described as follows:

CONCURRENT ACTION LIST

$Action(Hit(a, Ball),$
 $CONCURRENT: b \neq a \Rightarrow \neg Hit(b, Ball)$
 $PRECOND: Approaching(Ball, loc) \wedge At(a, loc)$
 $EFFECT: Returned(Ball))$.

In other words, the *Hit* action has its stated effect only if no other *Hit* action by another agent occurs at the same time. (In the SATPLAN approach, this would be handled by a partial **action exclusion axiom**.) For some actions, the desired effect is achieved *only* when another action occurs concurrently. For example, two agents are needed to carry a cooler full of beverages to the tennis court:

$Action(Carry(a, cooler, here, there),$
 $CONCURRENT: b \neq a \wedge Carry(b, cooler, here, there)$
 $PRECOND: At(a, here) \wedge At(cooler, here) \wedge Cooler(cooler)$
 $EFFECT: At(a, there) \wedge At(cooler, there) \wedge \neg At(a, here) \wedge \neg At(cooler, here))$.

With these kinds of action schemas, any of the planning algorithms described in Chapter 10 can be adapted with only minor modifications to generate multiactor plans. To the extent that the coupling among subplans is loose—meaning that concurrency constraints come into play only rarely during plan search—one would expect the various heuristics derived for single-agent planning to also be effective in the multiactor context. 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.

11.4.2 Planning with multiple agents: Cooperation and coordination

Now let us consider the true multiagent setting in which each agent makes its own plan. To start with, let us assume that the goals and knowledge base are shared. One might think that this reduces to the multibody case—each agent simply computes the joint solution and executes its own part of that solution. Alas, the “*the*” in “*the joint solution*” is misleading. For our doubles team, more than one joint solution exists:

PLAN 2:

$A : [Go(A, LeftNet), NoOp(A)]$
 $B : [Go(B, RightBaseline), Hit(B, Ball)]$.

If both agents can agree on either plan 1 or plan 2, the goal will be achieved. But 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 both try to hit the ball. The agents may realize this, but how can they coordinate to make sure they agree on the plan?

CONVENTION

One option is to adopt a **convention** before engaging in joint activity. A convention is any constraint on the selection of joint plans. For example, the convention “stick to your side of the court” would rule out plan 1, causing the doubles partners to select plan 2. Drivers on a road face the problem of not colliding with each other; this is (partially) solved by adopting the convention “stay on the right side of the road” in most countries; the alternative, “stay on the left side,” works equally well as long as all agents in an environment agree. 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 a community all speaks the same language. When conventions are widespread, they are called **social laws**.

SOCIAL LAWS

In the absence of a convention, agents can use **communication** to achieve common knowledge of a feasible joint plan. For example, a 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. 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. Note that communication can work as well with competitive agents as with cooperative ones.

PLAN RECOGNITION

Conventions can also arise through evolutionary processes. For example, seed-eating harvester ants are social creatures that evolved from the less social wasps. Colonies of ants execute very elaborate joint plans without any centralized control—the queen’s job is to reproduce, not to do centralized planning—and with very limited computation, communication, and memory capabilities in each ant (Gordon, 2000, 2007). The colony has many roles, including interior workers, patrollers, and foragers. Each ant chooses to perform a role according to the local conditions it observes. For example, foragers travel away from the nest, search for a seed, and when they find one, bring it back immediately. Thus, the rate at which foragers return to the nest is an approximation of the availability of food today. When the rate is high, other ants abandon their current role and take on the role of scavenger. The ants appear to have a convention on the importance of roles—foraging is the most important—and ants will easily switch into the more important roles, but not into the less important. There is some learning mechanism: a colony learns to make more successful and prudent actions over the course of its decades-long life, even though individual ants live only about a year.

One final example of cooperative multiagent behavior appears in the flocking behavior of birds. We can obtain a reasonable simulation of a flock if each bird agent (sometimes called a **boïd**) observes the positions of its nearest neighbors and then chooses the heading and acceleration that maximizes the weighted sum of these three components:

BOID

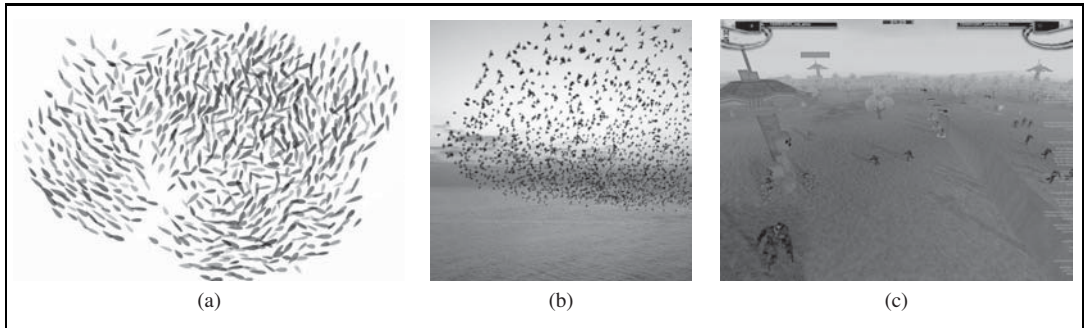


Figure 11.11 (a) A simulated flock of birds, using Reynolds's boids model. Image courtesy Giuseppe Randazzo, novastructura.net. (b) An actual flock of starlings. Image by Eduardo (pastaboy sleeps on flickr). (c) Two competitive teams of agents attempting to capture the towers in the NERO game. Image courtesy Risto Miikkulainen.

1. Cohesion: a positive score for getting closer to the average position of the neighbors
2. Separation: a negative score for getting too close to any one neighbor
3. Alignment: a positive score for getting closer to the average heading of the neighbors

EMERGENT
BEHAVIOR

If all the boids execute this policy, the flock exhibits the **emergent behavior** of flying as a pseudorigid body with roughly constant density that does not disperse over time, and that occasionally makes sudden swooping motions. You can see a still images in Figure 11.11(a) and compare it to an actual flock in (b). As with ants, there is no need for each agent to possess a joint plan that models the actions of other agents.

The most difficult multiagent problems involve both cooperation with members of one's own team and competition against members of opposing teams, all without centralized control. We see this in games such as robotic soccer or the NERO game shown in Figure 11.11(c), in which two teams of software agents compete to capture the control towers. As yet, methods for efficient planning in these kinds of environments—for example, taking advantage of loose coupling—are in their infancy.

11.5 SUMMARY

This chapter has addressed some of the complications of planning and acting in the real world. The main points:

- 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 **high-level actions** (HLAs) that can be implemented in various ways by lower-level action sequences. The effects of HLAs can be defined with **angelic semantics**, allowing provably correct high-level plans to be derived without consideration of lower-level implementations. HTN methods can 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.
- **Contingent 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 conformant and contingent plans can be constructed by search in the space of **belief states**. Efficient representation or computation of belief states is a key problem.
- An **online planning agent** uses execution monitoring and splices in repairs as needed to recover from unexpected situations, which can be due to nondeterministic actions, exogenous events, or incorrect models of the environment.
- **Multiagent** planning is necessary when there are other agents in the environment with which to cooperate or compete. Joint plans can be constructed, but must be augmented with some form of coordination if two agents are to agree on which joint plan to execute.
- This chapter extends classic planning to cover nondeterministic environments (where outcomes of actions are uncertain), but it is not the last word on planning. Chapter 17 describes techniques for stochastic environments (in which outcomes of actions have probabilities associated with them): Markov decision processes, partially observable Markov decision processes, and game theory. In Chapter 21 we show that reinforcement learning allows an agent to learn how to behave from past successes and failures.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Planning with time constraints was first dealt with by DEVISER (Vere, 1983). The representation of time in plans was addressed by Allen (1984) and 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.

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).

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 (Desctotte 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. We chose to present planning and scheduling as two separate problems; (Cushing *et al.*, 2007) show that this can lead to incompleteness on certain problems. 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. Space applications have driven the development of algorithms for resource allocations; see Laborie (2003) and Muscettola (2002). The literature on scheduling is presented in a classic survey article (Lawler *et al.*, 1993), a recent book (Pinedo, 2008), and an edited handbook (Blazewicz *et al.*, 2007).

MACROPS

ABSTRACTION
HIERARCHY

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. Our presentation of HLAs and angelic semantics is due to Marthi *et al.* (2007, 2008). Kambhampati *et al.* (1998) have proposed an approach in which decompositions are just another form of plan refinement, similar to the refinements for non-hierarchical partial-order planning.

CASE-BASED
PLANNING

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.

Early planners lacked conditionals and loops, but some could use coercion to form conformant plans. Sacerdoti's NOAH solved the "keys and boxes" problem, a planning challenge problem in which the planner knows little about the initial state, using coercion. 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.

Goldman and Boddy (1996) introduced the term **conformant planning**, 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, there are three main approaches to conformant planning. The first two use heuristic search in belief-state space: HSCP (Bertoli *et al.*, 2001a) uses binary decision diagrams (BDDs) to represent belief states, whereas Hoffmann and Brafman (2006) adopt the lazy approach of computing precondition and goal tests on demand using a SAT solver. The third approach, championed primarily by Jussi Rintanen (2007), formulates the entire sensorless planning problem as a quantified Boolean formula (QBF) and solves it using a general-purpose QBF solver. Current conformant planners are five orders of magnitude faster than CGP. The winner of the 2006 conformant-planning track at the International Planning Competition was T_0 (Palacios and Geffner, 2007), which uses heuristic search in belief-state space while keeping the belief-state representation simple by defining derived literals that cover conditional effects. Bryce and Kambhampati (2007) discuss how a planning graph can be generalized to generate good heuristics for conformant and contingent planning.

There has been some confusion in the literature between the terms "conditional" and "contingent" planning. Following Majercik and Littman (2003), we use "conditional" to mean a plan (or action) that has different effects depending on the actual state of the world, and "contingent" to mean a plan in which the agent can choose different actions depending on the results of sensing. The problem of contingent planning received more attention after the publication of Drew McDermott's (1978a) influential article, *Planning and Acting*.

The contingent-planning approach described in the chapter is based on Hoffmann and Brafman (2005), and was influenced by 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 MBP (Model-Based Planner), which uses binary decision diagrams to do conformant and contingent planning.

In retrospect, it is now possible to see how the major classical planning algorithms led to extended versions for uncertain domains. Fast-forward heuristic search through state space led to forward search in belief space (Bonet and Geffner, 2000; Hoffmann and Brafman, 2005); SATPLAN led to stochastic SATPLAN (Majercik and Littman, 2003) and to planning with quantified Boolean logic (Rintanen, 2007); partial order planning led to UWL (Etzioni *et al.*, 1992) and CNLP (Peot and Smith, 1992); GRAPHPLAN led to Sensory Graphplan or SGP (Weld *et al.*, 1998).

The first online planner with execution monitoring was PLANEX (Fikes *et al.*, 1972), which worked with the STRIPS planner to control the robot Shakey. 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. 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, job-shop scheduling for an Australian beer factory, and planning the construction of multistory buildings (Kartam and Levitt, 1990).

REACTIVE PLANNING

In the mid-1980s, pessimism about the slow run times of planning systems led to the proposal of reflex agents called **reactive planning** systems (Brooks, 1986; Agre and Chapman, 1987). PENG (Agre and Chapman, 1987) could play a (fully observable) video game by using Boolean circuits combined with a “visual” representation of current goals and the agent’s internal state. “Universal plans” (Schoppers, 1987, 1989) 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 (see Chapter 17). A universal plan (or a policy) contains a mapping from any state to the action that should be taken in that state. Koenig (2001) surveys online planning techniques, under the name *Agent-Centered Search*.

POLICY

Multiagent planning has leaped in popularity in recent years, although it does have a long history. Konolige (1982) formalizes multiagent planning in first-order logic, while Pednault (1986) gives 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). Boutilier and Brafman (2001) show how to adapt partial-order planning to a multiactor setting. Brafman and Domshlak (2008) devise a multiactor planning algorithm whose complexity grows only linearly with the number of actors, provided that the degree of coupling (measured partly by the **tree width** of the graph of interactions among agents) is bounded. Petrik and Zilberstein (2009) show that an approach based on bilinear programming outperforms the cover-set approach we outlined in the chapter.

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 dishonesty. Stone (2000) shows how agents can cooperate as teammates in the competitive, dynamic, partially observable environment of robotic soccer. In a later article, Stone (2003) analyzes two competitive multiagent environments—RoboCup, a robotic soccer competition, and TAC, the auction-based Trading Agents Competition—and finds that the computational intractability of our current theoretically well-founded approaches has led to many multiagent systems being designed by *ad hoc* methods.

In his highly influential *Society of Mind* theory, Marvin Minsky (1986, 2007) proposes that human minds are constructed from an ensemble of agents. Livnat and Pippenger (2006) prove that, for the problem of optimal path-finding, and given a limitation on the total amount of computing resources, the best architecture for an agent is an ensemble of subagents, each of which tries to optimize its own objective, and all of which are in conflict with one another.

The boid model on page 429 is due to Reynolds (1987), who won an Academy Award for its application to swarms of penguins in *Batman Returns*. The NERO game and the methods for learning strategies are described by Bryant and Miikkulainen (2007).

Recent book on multiagent systems include those by Weiss (2000a), Young (2004), Vlassis (2008), and Shoham and Leyton-Brown (2009). There is an annual conference on autonomous agents and multiagent systems (AAMAS).

EXERCISES

11.1 The goals we have considered so far all ask the planner to make the world satisfy the goal at just one time step. Not all goals can be expressed this way: you do not achieve the goal of suspending a chandelier above the ground by throwing it in the air. More seriously, you wouldn't want your spacecraft life-support system to supply oxygen one day but not the next. A *maintenance goal* is achieved when the agent's plan causes a condition to hold continuously from a given state onward. Describe how to extend the formalism of this chapter to support maintenance goals.

11.2 You have a number of trucks with which to deliver a set of packages. Each package starts at some location on a grid map, and has a destination somewhere else. Each truck is directly controlled by moving forward and turning. Construct a hierarchy of high-level actions for this problem. What knowledge about the solution does your hierarchy encode?

11.3 Suppose that a high-level action has exactly one implementation as a sequence of primitive actions. Give an algorithm for computing its preconditions and effects, given the complete refinement hierarchy and schemas for the primitive actions.

11.4 Suppose that the optimistic reachable set of a high-level plan is a superset of the goal set; can anything be concluded about whether the plan achieves the goal? What if the pessimistic reachable set doesn't intersect the goal set? Explain.

11.5 Write an algorithm that takes an initial state (specified by a set of propositional literals) and a sequence of HLAs (each defined by preconditions and angelic specifications of optimistic and pessimistic reachable sets) and computes optimistic and pessimistic descriptions of the reachable set of the sequence.

11.6 In Figure 11.2 we showed how to describe actions in a scheduling problem by using separate fields for DURATION, USE, and CONSUME. Now suppose we wanted to combine scheduling with nondeterministic planning, which requires nondeterministic and conditional effects. Consider each of the three fields and explain if they should remain separate fields, or if they should become effects of the action. Give an example for each of the three.

11.7 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 changes the contents of a memory location, and 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 action 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 by using a temporary variable.

11.8 Suppose the *Flip* action always changes the truth value of variable *L*. Show how to define its effects by using an action schema with conditional effects. Show that, despite the use of conditional effects, a 1-CNF belief state representation remains in 1-CNF after a *Flip*.

11.9 In the blocks world we were forced to introduce two action schemas, *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.

11.10 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.

11.11 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.

11.12 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.

12 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, thoughts, and shopping.

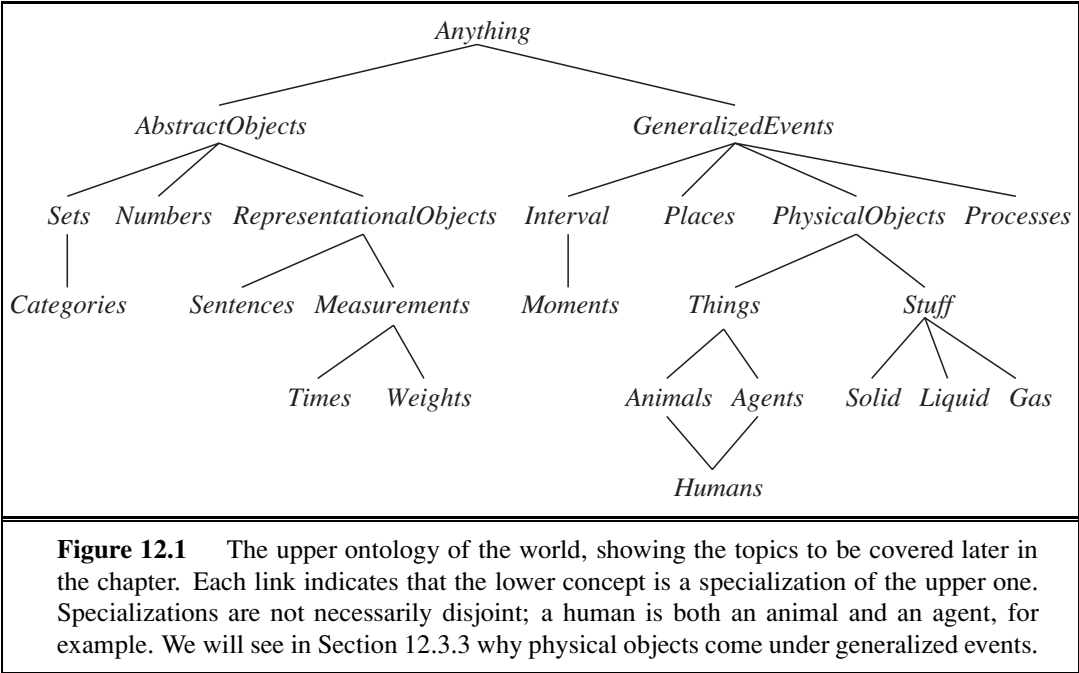
The previous chapters described the technology for knowledge-based agents: the syntax, semantics, and proof theory of propositional and first-order logic, and the implementation of agents that use these logics. In this chapter we address the question of what *content* to put into such an agent's knowledge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

Section 12.1 introduces the idea of a general ontology, which organizes everything in the world into a hierarchy of categories. Section 12.2 covers the basic categories of objects, substances, and measures; Section 12.3 covers events, and Section 12.4 discusses knowledge about beliefs. We then return to consider the technology for reasoning with this content: Section 12.5 discusses reasoning systems designed for efficient inference with categories, and Section 12.6 discusses reasoning with default information. Section 12.7 brings all the knowledge together in the context of an Internet shopping environment.

12.1 ONTOLOGICAL ENGINEERING

In “toy” domains, the choice of representation is not that important; many choices will work. Complex domains such as shopping on the Internet or driving a car in traffic require more general and flexible representations. This chapter shows how to create these representations, concentrating on general concepts—such as *Events*, *Time*, *Physical Objects*, and *Beliefs*—that occur in many different domains. Representing these abstract concepts is sometimes called **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 of different types of objects—robots, televisions, books, or whatever—can be filled in later. This is analogous to the way that designers of an object-oriented programming framework (such as the Java Swing graphical framework) define general concepts like *Window*, expecting users to



UPPER ONTOLOGY

use these to define more specific concepts like *SpreadsheetWindow*. 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 12.1.

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, although certain aspects of the real world are hard to capture in FOL. The principal difficulty is that most 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 rules 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 delay the discussion of exceptions until Section 12.5 of this chapter, and the more general topic of reasoning with uncertainty until Chapter 13.

Of what use is an upper ontology? Consider the ontology for circuits in Section 8.4.2. It makes many simplifying assumptions: time is omitted completely; signals are fixed and do not propagate; the structure of the circuit remains constant. A more general ontology would consider signals at particular times, and would 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, 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 represent where the wires are on the board.

If we look at the wumpus world, similar considerations apply. Although we do represent time, it has a 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 biological taxonomy to help the agent predict the behavior of cave-dwellers 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 “Maybe.” In this section, we present one general-purpose ontology that synthesizes ideas from those centuries. Two major characteristics of general-purpose ontologies 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 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.

We should say up front that the enterprise of general ontological engineering has so far had only limited success. None of the top AI applications (as listed in Chapter 1) make use of a shared ontology—they all use special-purpose knowledge engineering. Social/political considerations can make it difficult for competing parties to agree on an ontology. As Tom Gruber (2004) says, “Every ontology is a treaty—a social agreement—among people with some common motive in sharing.” When competing concerns outweigh the motivation for sharing, there can be no common ontology. Those ontologies that do exist have been created along four routes:

1. By a team of trained ontologist/logicians, who architect the ontology and write axioms. The CYC system was mostly built this way (Lenat and Guha, 1990).
2. By importing categories, attributes, and values from an existing database or databases. DBPEDIA was built by importing structured facts from Wikipedia (Bizer *et al.*, 2007).
3. By parsing text documents and extracting information from them. TEXTRUNNER was built by reading a large corpus of Web pages (Banko and Etzioni, 2008).
4. By enticing unskilled amateurs to enter commonsense knowledge. The OPENMIND system was built by volunteers who proposed facts in English (Singh *et al.*, 2002; Chklovski and Gil, 2005).

12.2 CATEGORIES AND OBJECTS

CATEGORY



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 would normally 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 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 and yellow mottled skin, one-foot diameter, ovoid shape, red flesh, black seeds, and presence in the fruit aisle, one can infer that an object is a watermelon; from this, one infers that it would be useful for fruit salad.

REIFICATION

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**¹ 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** of $Balls$. We will use subcategory, subclass, and subset interchangeably.

SUBCATEGORY

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 can infer 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. The largest such taxonomy organizes about 10 million living and extinct species, many of them beetles,² into a single hierarchy; 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. Here are some types of facts, with examples of each:

- An object is a member of a category.
 $BB_9 \in Basketballs$
- A category is a subclass of another category.
 $Basketballs \subset Balls$
- All members of a category have some properties.
 $(x \in Basketballs) \Rightarrow Spherical(x)$

¹ Turning a proposition into an object is called **reification**, from the Latin word *res*, or thing. John McCarthy proposed the term “thingification,” but it never caught on.

² The famous biologist J. B. S. Haldane deduced “An inordinate fondness for beetles” on the part of the Creator.

- Members of a category can be recognized by some properties.
 $Orange(x) \wedge Round(x) \wedge Diameter(x) = 9.5'' \wedge x \in Balls \Rightarrow x \in Basketballs$
- A category as a whole has some properties.
 $Dogs \in DomesticatedSpecies$

Notice that because *Dogs* is a category and is a member of *DomesticatedSpecies*, the latter must be a category of categories. Of course there are exceptions to many of the above rules (punctured basketballs are not spherical); we deal with these exceptions later.

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 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

EXHAUSTIVE
DECOMPOSITION
PARTITION

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

(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 on page 443, strict logical definitions for categories are neither always possible nor always necessary.

12.2.1 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,

$$\begin{aligned} \text{PartOf}(x, y) \wedge \text{PartOf}(y, z) &\Rightarrow \text{PartOf}(x, z) . \\ \text{PartOf}(x, x) & . \end{aligned}$$

Therefore, we can conclude $\text{PartOf}(\text{Bucharest}, \text{Earth})$.

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:

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

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 12.5.2, we describe 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 12.8.) 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

$$\text{BunchOf}(\{\text{Apple}_1, \text{Apple}_2, \text{Apple}_3\})$$

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 $\text{BunchOf}(\{x\}) = x$. Furthermore, $\text{BunchOf}(\text{Apples})$ 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 $\text{BunchOf}(s)$:

$$\forall x \quad x \in s \Rightarrow \text{PartOf}(x, \text{BunchOf}(s)) .$$

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

$$\forall y \quad [\forall x \quad x \in s \Rightarrow \text{PartOf}(x, y)] \Rightarrow \text{PartOf}(\text{BunchOf}(s), y) .$$

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.

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 yellow or 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 fuzzy 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:

$$\text{Typical}(c) \subseteq c.$$

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


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

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). He used the example of *games* to show that members of a category shared “family resemblances” rather than necessary and sufficient characteristics: what strict definition encompasses chess, tag, solitaire, and dodgeball?

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.

12.2.2 Measurements

MEASURE

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. We represent the length with a **units function** that takes a number as argument. (An alternative scheme is explored in Exercise 12.9.) If the line segment is called L_1 , we can write

UNITS FUNCTION

$$\text{Length}(L_1) = \text{Inches}(1.5) = \text{Centimeters}(3.81) .$$

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

$$\text{Centimeters}(2.54 \times d) = \text{Inches}(d) .$$

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:

$$\text{Diameter}(\text{Basketball}_{12}) = \text{Inches}(9.5) .$$

$$\text{ListPrice}(\text{Basketball}_{12}) = \$ (19) .$$

$$d \in \text{Days} \Rightarrow \text{Duration}(d) = \text{Hours}(24) .$$

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 $\text{Inches}(0)$ and $\text{Centimeters}(0)$ refer to the same zero length, they are not identical to other zero measures, such as $\text{Seconds}(0)$.

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:

$$e_1 \in \text{Exercises} \wedge e_2 \in \text{Exercises} \wedge \text{Wrote}(\text{Norvig}, e_1) \wedge \text{Wrote}(\text{Russell}, e_2) \Rightarrow \\ \text{Difficulty}(e_1) > \text{Difficulty}(e_2) .$$

$$e_1 \in \text{Exercises} \wedge e_2 \in \text{Exercises} \wedge \text{Difficulty}(e_1) > \text{Difficulty}(e_2) \Rightarrow \\ \text{ExpectedScore}(e_1) < \text{ExpectedScore}(e_2) .$$

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.

12.2.3 Objects: Things and stuff

INDIVIDUATION
STUFF

The real world can be seen as consisting of primitive objects (e.g., atomic 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).

COUNT NOUNS
MASS NOUN

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 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. Here we describe just one; the others are covered in the historical notes section.

To represent *stuff* properly, we begin with the obvious. We 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 one 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 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 omit for now, any part of a butter-object is also a butter-object:

$$b \in Butter \wedge PartOf(p, b) \Rightarrow p \in Butter .$$

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

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

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*. Note that the category *PoundOfButter*, which includes as members all butter-objects weighing one pound, is not a kind of *stuff*. 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: some properties are **intrinsic**: they belong to the very substance of the object, rather than to the object as a whole. When you cut an instance of *stuff* in half, the two pieces retain the intrinsic properties—things like density, boiling point, flavor, color, ownership, and so on. On the other hand, their **extrinsic** properties—weight, length, shape, and so on—are not retained under subdivision. A category 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.

12.3 EVENTS

EVENT CALCULUS

In Section 10.4.2, we showed how situation calculus represents actions and their effects. Situation calculus is limited in its applicability: it was designed to describe a world in which actions are discrete, instantaneous, and happen one at a time. Consider a continuous action, such as filling a bathtub. Situation calculus can say that the tub is empty before the action and full when the action is done, but it can't talk about what happens *during* the action. It also can't describe two actions happening at the same time—such as brushing one's teeth while waiting for the tub to fill. To handle such cases we introduce an alternative formalism known as **event calculus**, which is based on points of time rather than on situations.³

Event calculus reifies fluents and events. The fluent $At(Shankar, Berkeley)$ is an object that refers to the fact of Shankar being in Berkeley, but does not by itself say anything about whether it is true. To assert that a fluent is actually true at some point in time we use the predicate T , as in $T(At(Shankar, Berkeley), t)$.

Events are described as instances of event categories.⁴ The event E_1 of Shankar flying from San Francisco to Washington, D.C. is described as

$$E_1 \in Flyings \wedge Flyer(E_1, Shankar) \wedge Origin(E_1, SF) \wedge Destination(E_1, DC).$$

If this is too verbose, we can define an alternative three-argument version of the category of flying events and say

$$E_1 \in Flyings(Shankar, SF, DC).$$

We then use $Happens(E_1, i)$ to say that the event E_1 took place over the time interval i , and we say the same thing in functional form with $Extent(E_1) = i$. We represent time intervals by a (start, end) pair of times; that is, $i = (t_1, t_2)$ is the time interval that starts at t_1 and ends at t_2 . The complete set of predicates for one version of the event calculus is

$T(f, t)$	Fluent f is true at time t
$Happens(e, i)$	Event e happens over the time interval i
$Initiates(e, f, t)$	Event e causes fluent f to start to hold at time t
$Terminates(e, f, t)$	Event e causes fluent f to cease to hold at time t
$Clipped(f, i)$	Fluent f ceases to be true at some point during time interval i
$Restored(f, i)$	Fluent f becomes true sometime during time interval i

We assume a distinguished event, $Start$, that describes the initial state by saying which fluents are initiated or terminated at the start time. We define T by saying that a fluent holds at a point in time if the fluent was initiated by an event at some time in the past and was not made false (clipped) by an intervening event. A fluent does not hold if it was terminated by an event and

³ The terms “event” and “action” may be used interchangeably. Informally, “action” connotes an agent while “event” connotes the possibility of agentless actions.

⁴ Some versions of event calculus do not distinguish event categories from instances of the categories.

not made true (restored) by another event. Formally, the axioms are:

$$\begin{aligned} & \text{Happens}(e, (t_1, t_2)) \wedge \text{Initiates}(e, f, t_1) \wedge \neg \text{Clipped}(f, (t_1, t)) \wedge t_1 < t \Rightarrow \\ & \quad T(f, t) \\ & \text{Happens}(e, (t_1, t_2)) \wedge \text{Terminates}(e, f, t_1) \wedge \neg \text{Restored}(f, (t_1, t)) \wedge t_1 < t \Rightarrow \\ & \quad \neg T(f, t) \end{aligned}$$

where *Clipped* and *Restored* are defined by

$$\begin{aligned} \text{Clipped}(f, (t_1, t_2)) & \Leftrightarrow \\ & \exists e, t, t_3 \text{ Happens}(e, (t, t_3)) \wedge t_1 \leq t < t_2 \wedge \text{Terminates}(e, f, t) \\ \text{Restored}(f, (t_1, t_2)) & \Leftrightarrow \\ & \exists e, t, t_3 \text{ Happens}(e, (t, t_3)) \wedge t_1 \leq t < t_2 \wedge \text{Initiates}(e, f, t) \end{aligned}$$

It is convenient to extend *T* to work over intervals as well as time points; a fluent holds over an interval if it holds on every point within the interval:

$$T(f, (t_1, t_2)) \Leftrightarrow [\forall t (t_1 \leq t < t_2) \Rightarrow T(f, t)]$$

Fluents and actions are defined with domain-specific axioms that are similar to successor-state axioms. For example, we can say that the only way a wumpus-world agent gets an arrow is at the start, and the only way to use up an arrow is to shoot it:

$$\begin{aligned} \text{Initiates}(e, \text{HaveArrow}(a), t) & \Leftrightarrow e = \text{Start} \\ \text{Terminates}(e, \text{HaveArrow}(a), t) & \Leftrightarrow e \in \text{Shootings}(a) \end{aligned}$$

By reifying events we make it possible to add any amount of arbitrary information about them. For example, we can say that Shankar's flight was bumpy with *Bumpy*(*E*₁). In an ontology where events are *n*-ary predicates, there would be no way to add extra information like this; moving to an *n* + 1-ary predicate isn't a scalable solution.

We can extend event calculus to make it possible to represent simultaneous events (such as two people being necessary to ride a seesaw), exogenous events (such as the wind blowing and changing the location of an object), continuous events (such as the level of water in the bathtub continuously rising) and other complications.

12.3.1 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 something different—it would not be a trip from San Francisco to Washington, but instead a trip from San Francisco to somewhere over Kansas. On the other hand, the category of events denoted by *Flyings* 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 bag of peanuts), that event is still a member of *Flyings*. 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 process *e* that happens over an interval also happens over any subinterval:

$$(e \in \text{Processes}) \wedge \text{Happens}(e, (t_1, t_4)) \wedge (t_1 < t_2 < t_3 < t_4) \Rightarrow \text{Happens}(e, (t_2, t_3)).$$

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

TEMPORAL
SUBSTANCE

SPATIAL SUBSTANCE

12.3.2 Time intervals

Event calculus opens us up to the possibility of talking about time, and time intervals. We will consider two kinds of time intervals: moments and extended intervals. The distinction is that only moments have zero duration:

$$\begin{aligned} & \text{Partition}(\{\text{Moments}, \text{ExtendedIntervals}\}, \text{Intervals}) \\ & i \in \text{Moments} \Leftrightarrow \text{Duration}(i) = \text{Seconds}(0) . \end{aligned}$$

Next we invent a time scale and associate points on that scale with moments, giving us absolute times. The time scale is arbitrary; we measure it in seconds and say that the moment at midnight (GMT) on January 1, 1900, has time 0. The functions *Begin* 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.

$$\begin{aligned} \text{Interval}(i) & \Rightarrow \text{Duration}(i) = (\text{Time}(\text{End}(i)) - \text{Time}(\text{Begin}(i))) . \\ \text{Time}(\text{Begin}(\text{AD1900})) & = \text{Seconds}(0) . \\ \text{Time}(\text{Begin}(\text{AD2001})) & = \text{Seconds}(3187324800) . \\ \text{Time}(\text{End}(\text{AD2001})) & = \text{Seconds}(3218860800) . \\ \text{Duration}(\text{AD2001}) & = \text{Seconds}(31536000) . \end{aligned}$$

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:

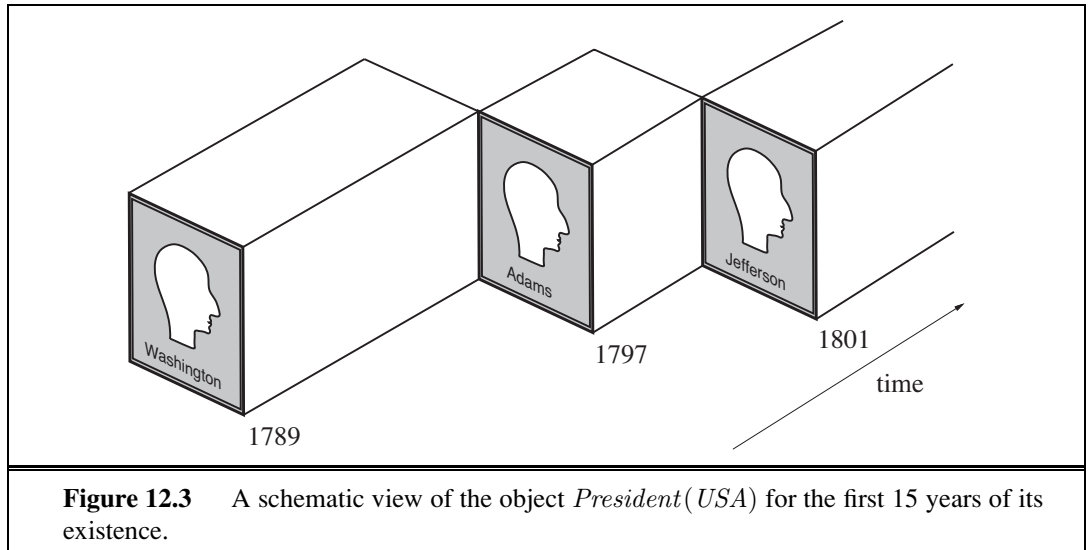
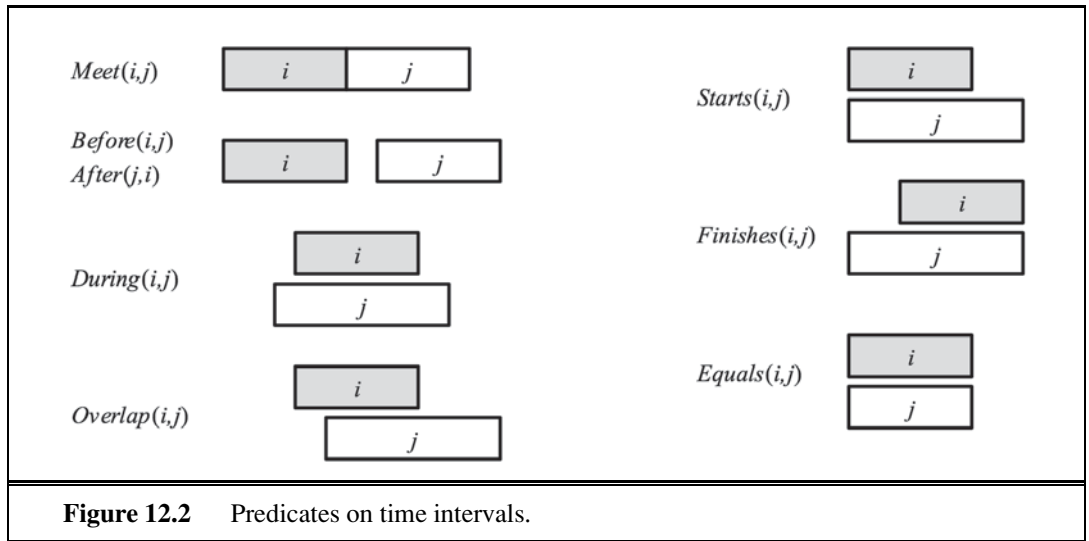
$$\begin{aligned} \text{Time}(\text{Begin}(\text{AD2001})) & = \text{Date}(0, 0, 0, 1, \text{Jan}, 2001) \\ \text{Date}(0, 20, 21, 24, 1, 1995) & = \text{Seconds}(3000000000) . \end{aligned}$$

Two intervals *Meet* if the end time of the first equals the start time of the second. The complete set of interval relations, as proposed by Allen (1983), is shown graphically in Figure 12.2 and logically below:

$$\begin{aligned} \text{Meet}(i, j) & \Leftrightarrow \text{End}(i) = \text{Begin}(j) \\ \text{Before}(i, j) & \Leftrightarrow \text{End}(i) < \text{Begin}(j) \\ \text{After}(j, i) & \Leftrightarrow \text{Before}(i, j) \\ \text{During}(i, j) & \Leftrightarrow \text{Begin}(j) < \text{Begin}(i) < \text{End}(i) < \text{End}(j) \\ \text{Overlap}(i, j) & \Leftrightarrow \text{Begin}(i) < \text{Begin}(j) < \text{End}(i) < \text{End}(j) \\ \text{Begins}(i, j) & \Leftrightarrow \text{Begin}(i) = \text{Begin}(j) \\ \text{Finishes}(i, j) & \Leftrightarrow \text{End}(i) = \text{End}(j) \\ \text{Equals}(i, j) & \Leftrightarrow \text{Begin}(i) = \text{Begin}(j) \wedge \text{End}(i) = \text{End}(j) \end{aligned}$$

These all have their intuitive meaning, with the exception of *Overlap*: we tend to think of overlap as symmetric (if *i* overlaps *j* then *j* overlaps *i*), but in this definition, *Overlap*(*i*, *j*) only holds if *i* begins before *j*. To say that the reign of Elizabeth II immediately followed that of George VI, and the reign of Elvis overlapped with the 1950s, we can write the following:

$$\begin{aligned} & \text{Meets}(\text{ReignOf}(\text{George VI}), \text{ReignOf}(\text{Elizabeth II})) . \\ & \text{Overlap}(\text{Fifties}, \text{ReignOf}(\text{Elvis})) . \\ & \text{Begin}(\text{Fifties}) = \text{Begin}(\text{AD1950}) . \\ & \text{End}(\text{Fifties}) = \text{End}(\text{AD1959}) . \end{aligned}$$



12.3.3 Fluents and objects

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, such as $Population(USA)$. A 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 1797, John Adams from 1797 to 1801, and so on, as in Figure 12.3. To say that George Washington was president throughout 1790, we can write

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

We use the function symbol *Equals* rather than the standard logical predicate $=$, because we cannot have a predicate as an argument to T , and because 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 identity is between the subevents of each object that are defined by the period 1790.

12.4 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. Knowledge about one's own knowledge and reasoning processes is useful for controlling inference. For example, suppose Alice asks "what is the square root of 1764" and Bob replies "I don't know." If Alice insists "think harder," Bob should realize that with some more thought, this question can in fact be answered. On the other hand, if the question were "Is your mother sitting down right now?" then Bob should realize that thinking harder is unlikely to help. Knowledge about the knowledge of other agents is also important; Bob should realize that his mother knows whether she is sitting or not, and that asking her would be a way to find out.

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 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. We will be happy just to be able to conclude that mother knows whether or not she is sitting.

We begin with the **propositional attitudes** that an agent can have toward mental objects: attitudes such as *Believes*, *Knows*, *Wants*, *Intends*, and *Informs*. The difficulty is that these attitudes do not behave like "normal" predicates. For example, suppose we try to assert that Lois knows that Superman can fly:

$$Knows(Lois, CanFly(Superman)) .$$

One minor issue with this is that we normally think of $CanFly(Superman)$ as a sentence, but here it appears as a term. That issue can be patched up just by reifying $CanFly(Superman)$; making it a fluent. A more serious problem is that, if it is true that Superman is Clark Kent, then we must conclude that Lois knows that Clark can fly:

$$\begin{aligned} (Superman = Clark) \wedge Knows(Lois, CanFly(Superman)) \\ \models Knows(Lois, CanFly(Clark)) . \end{aligned}$$

This is a consequence of the fact that equality reasoning is built into logic. Normally that is a good thing; if our agent knows that $2 + 2 = 4$ and $4 < 5$, then we want our agent to know

REFERENTIAL
TRANSPARENCY

that $2 + 2 < 5$. This property is called **referential transparency**—it doesn't matter what term a logic uses to refer to an object, what matters is the object that the term names. But for propositional attitudes like *believes* and *knows*, we would like to have referential opacity—the terms used *do* matter, because not all agents know which terms are co-referential.

MODAL LOGIC

Modal logic is designed to address this problem. Regular logic is concerned with a single modality, the modality of truth, allowing us to express “ P is true.” Modal logic includes special modal operators that take sentences (rather than terms) as arguments. For example, “ A knows P ” is represented with the notation $\mathbf{K}_A P$, where \mathbf{K} is the modal operator for knowledge. It takes two arguments, an agent (written as the subscript) and a sentence. The syntax of modal logic is the same as first-order logic, except that sentences can also be formed with modal operators.

POSSIBLE WORLD
ACCESSIBILITY
RELATIONS

The semantics of modal logic is more complicated. In first-order logic a **model** contains a set of objects and an interpretation that maps each name to the appropriate object, relation, or function. In modal logic we want to be able to consider both the possibility that Superman's secret identity is Clark and that it isn't. Therefore, we will need a more complicated model, one that consists of a collection of **possible worlds** rather than just one true world. The worlds are connected in a graph by **accessibility relations**, one relation for each modal operator. We say that world w_1 is accessible from world w_0 with respect to the modal operator \mathbf{K}_A if everything in w_1 is consistent with what A knows in w_0 , and we write this as $\text{Acc}(\mathbf{K}_A, w_0, w_1)$. In diagrams such as Figure 12.4 we show accessibility as an arrow between possible worlds. As an example, in the real world, Bucharest is the capital of Romania, but for an agent that did not know that, other possible worlds are accessible, including ones where the capital of Romania is Sibiu or Sofia. Presumably a world where $2 + 2 = 5$ would not be accessible to any agent.

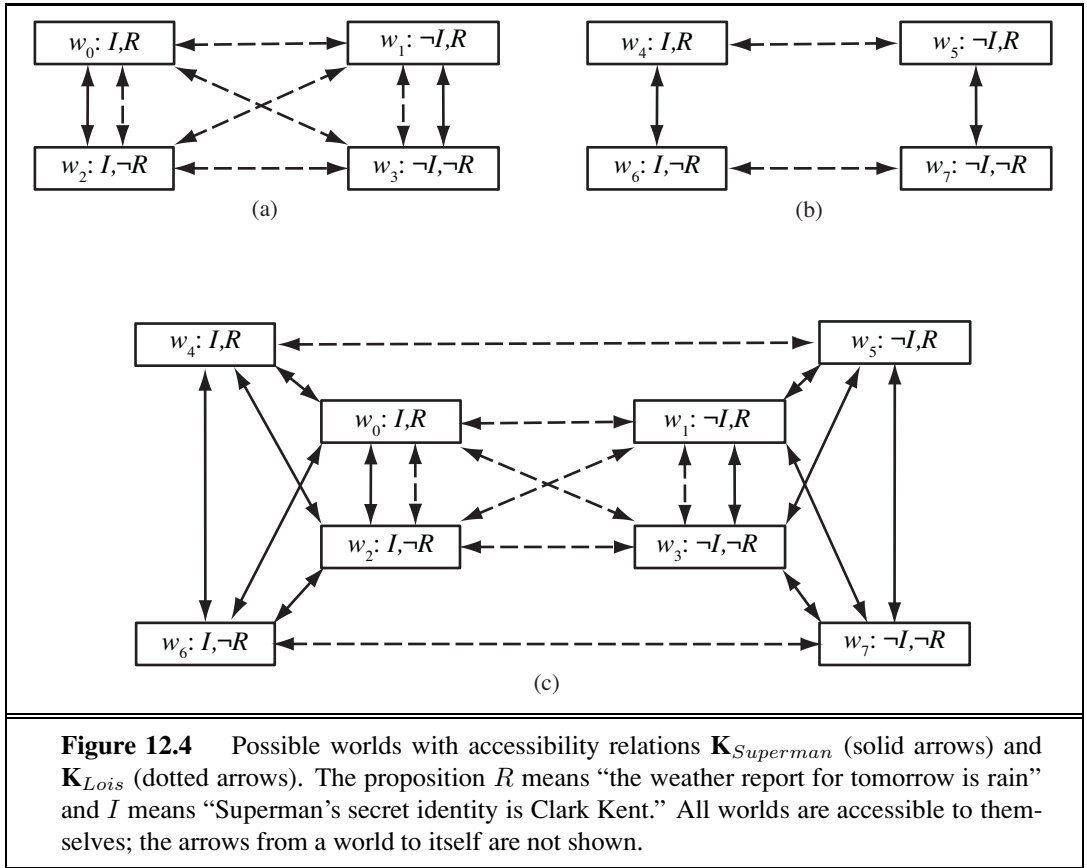
In general, a knowledge atom $\mathbf{K}_A P$ is true in world w if and only if P is true in every world accessible from w . The truth of more complex sentences is derived by recursive application of this rule and the normal rules of first-order logic. That means that modal logic can be used to reason about nested knowledge sentences: what one agent knows about another agent's knowledge. For example, we can say that, even though Lois doesn't know whether Superman's secret identity is Clark Kent, she does know that Clark knows:

$$\mathbf{K}_{\text{Lois}}[\mathbf{K}_{\text{Clark}} \text{Identity}(\text{Superman}, \text{Clark}) \vee \mathbf{K}_{\text{Clark}} \neg \text{Identity}(\text{Superman}, \text{Clark})]$$

Figure 12.4 shows some possible worlds for this domain, with accessibility relations for Lois and Superman.

In the TOP-LEFT diagram, it is common knowledge that Superman knows his own identity, and neither he nor Lois has seen the weather report. So in w_0 the worlds w_0 and w_2 are accessible to Superman; maybe rain is predicted, maybe not. For Lois all four worlds are accessible from each other; she doesn't know anything about the report or if Clark is Superman. But she does know that Superman knows whether he is Clark, because in every world that is accessible to Lois, either Superman knows I , or he knows $\neg I$. Lois does not know which is the case, but either way she knows Superman knows.

In the TOP-RIGHT diagram it is common knowledge that Lois has seen the weather report. So in w_4 she knows rain is predicted and in w_6 she knows rain is not predicted.



Superman does not know the report, but he knows that Lois knows, because in every world that is accessible to him, either she knows R or she knows $\neg R$.

In the BOTTOM diagram we represent the scenario where it is common knowledge that Superman knows his identity, and Lois might or might not have seen the weather report. We represent this by combining the two top scenarios, and adding arrows to show that Superman does not know which scenario actually holds. Lois does know, so we don’t need to add any arrows for her. In w_0 Superman still knows I but not R , and now he does not know whether Lois knows R . From what Superman knows, he might be in w_0 or w_2 , in which case Lois knows R . From what Superman knows, he might be in w_4 , in which case she knows R , or w_6 , in which case she knows $\neg R$.

There are an infinite number of possible worlds, so the trick is to introduce just the ones you need to represent what you are trying to model. A new possible world is needed to talk about different possible facts (e.g., rain is predicted or not), or to talk about different states of knowledge (e.g., does Lois know that rain is predicted). That means two possible worlds, such as w_4 and w_0 in Figure 12.4, might have the same base facts about the world, but differ in their accessibility relations, and therefore in facts about knowledge.

Modal logic solves some tricky issues with the interplay of quantifiers and knowledge. The English sentence “Bond knows that someone is a spy” is ambiguous. The first reading is

that there is a particular someone who Bond knows is a spy; we can write this as

$$\exists x \mathbf{K}_{Bond} Spy(x),$$

which in modal logic means that there is an x that, in all accessible worlds, Bond knows to be a spy. The second reading is that Bond just knows that there is at least one spy:

$$\mathbf{K}_{Bond} \exists x Spy(x).$$

The modal logic interpretation is that in each accessible world there is an x that is a spy, but it need not be the same x in each world.

Now that we have a modal operator for knowledge, we can write axioms for it. First, we can say that agents are able to draw deductions; if an agent knows P and knows that P implies Q , then the agent knows Q :

$$(\mathbf{K}_a P \wedge \mathbf{K}_a (P \Rightarrow Q)) \Rightarrow \mathbf{K}_a Q.$$

From this (and a few other rules about logical identities) we can establish that $\mathbf{K}_A (P \vee \neg P)$ is a tautology; every agent knows every proposition P is either true or false. On the other hand, $(\mathbf{K}_A P) \vee (\mathbf{K}_A \neg P)$ is not a tautology; in general, there will be lots of propositions that an agent does not know to be true and does not know to be false.

It is said (going back to Plato) that knowledge is justified true belief. That is, if it is true, if you believe it, and if you have an unassailably good reason, then you know it. That means that if you know something, it must be true, and we have the axiom:

$$\mathbf{K}_a P \Rightarrow P.$$

Furthermore, logical agents should be able to introspect on their own knowledge. If they know something, then they know that they know it:

$$\mathbf{K}_a P \Rightarrow \mathbf{K}_a (\mathbf{K}_a P).$$

We can define similar axioms for belief (often denoted by \mathbf{B}) and other modalities. However, one problem with the modal logic approach is that it assumes **logical omniscience** on the part of agents. That is, if an agent knows a set of axioms, then it knows all consequences of those axioms. This is on shaky ground even for the somewhat abstract notion of knowledge, but it seems even worse for belief, because belief has more connotation of referring to things that are physically represented in the agent, not just potentially derivable. There have been attempts to define a form of limited rationality for agents; to say that agents believe those assertions that can be derived with the application of no more than k reasoning steps, or no more than s seconds of computation. These attempts have been generally unsatisfactory.

LOGICAL
OMNISCIENCE

12.5 REASONING SYSTEMS FOR CATEGORIES

Categories are the primary building blocks of large-scale knowledge representation schemes. 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.

12.5.1 Semantic networks

EXISTENTIAL
GRAPHS

In 1909, Charles S. Peirce proposed a graphical notation of nodes and edges 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 links. For example, Figure 12.5 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.⁵

For this reason, we have used a special notation—the double-boxed link—in Figure 12.5. This link asserts that

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

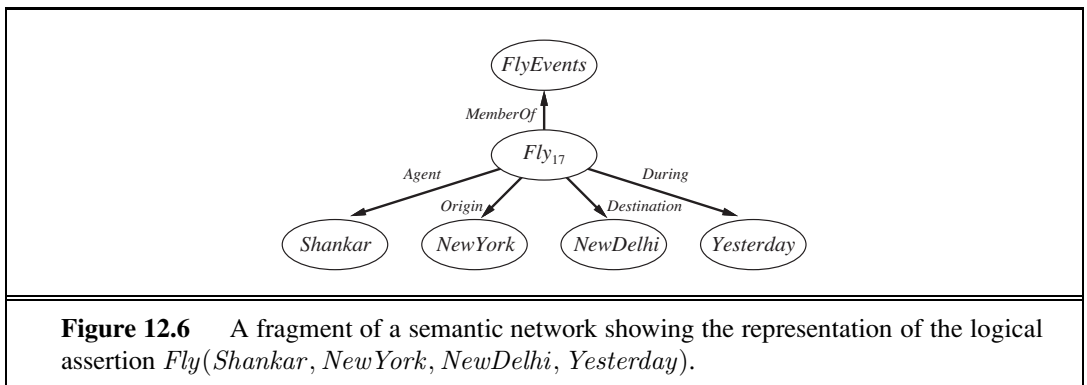
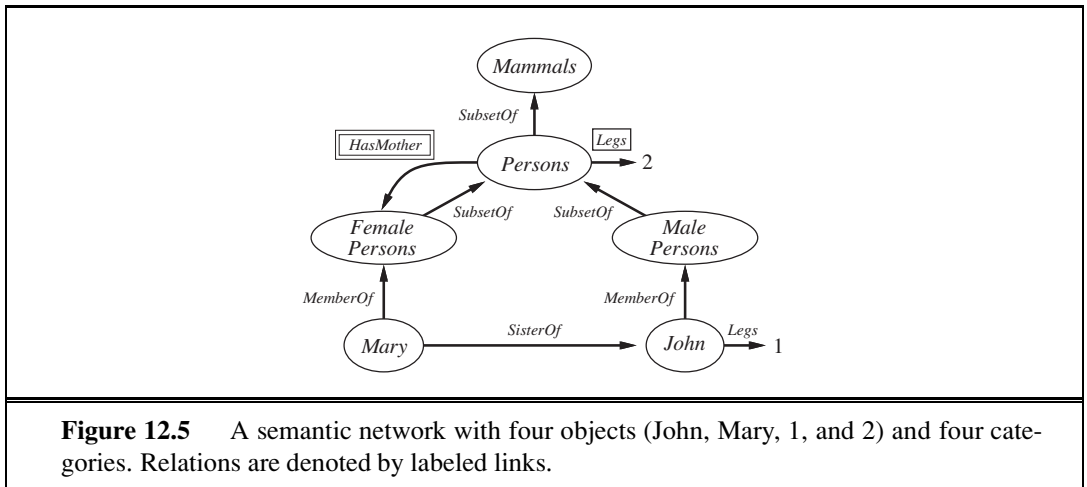
We might also want to assert that persons have two legs—that is,

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

As before, we need to be careful not to assert that a category has legs; the single-boxed link in Figure 12.5 is used to assert properties of every member of a category.

The semantic network notation makes it convenient to perform **inheritance** reasoning of the kind introduced in Section 12.2. 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

⁵ 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 12.22 for more on these issues.



mechanism, compared with logical theorem proving, has been one of the main attractions of semantic networks.

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 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 12.6.

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 belonging to an appropriate event category. Figure 12.6 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.

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 univer-

sally 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 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—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 VALUE

One of the most important aspects of semantic networks is their ability to represent **default values** for categories. Examining Figure 12.5 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 12.6 goes into more depth on this issue and on default reasoning in general.

12.5.2 Description logics

DESCRIPTION LOGIC

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.

$ \begin{aligned} \text{Concept} &\rightarrow \mathbf{Thing} \mid \text{ConceptName} \\ &\mid \mathbf{And}(\text{Concept}, \dots) \\ &\mid \mathbf{All}(\text{RoleName}, \text{Concept}) \\ &\mid \mathbf{AtLeast}(\text{Integer}, \text{RoleName}) \\ &\mid \mathbf{AtMost}(\text{Integer}, \text{RoleName}) \\ &\mid \mathbf{Fills}(\text{RoleName}, \text{IndividualName}, \dots) \\ &\mid \mathbf{SameAs}(\text{Path}, \text{Path}) \\ &\mid \mathbf{OneOf}(\text{IndividualName}, \dots) \\ \text{Path} &\rightarrow [\text{RoleName}, \dots] \end{aligned} $
--

Figure 12.7 The syntax of descriptions in a subset of the CLASSIC language.

The CLASSIC language (Borgida *et al.*, 1989) is a typical description logic. The syntax of CLASSIC descriptions is shown in Figure 12.7.⁶ For example, to say that bachelors are unmarried adult males we would write

$$\text{Bachelor} = \text{And}(\text{Unmarried}, \text{Adult}, \text{Male}) .$$

The equivalent in first-order logic would be

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

Notice that the description logic has an algebra of operations on predicates, which of course we can't do in first-order logic. Any description in CLASSIC can be translated into an equivalent first-order sentence, 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

$$\begin{aligned}
&\text{And}(\text{Man}, \text{AtLeast}(3, \text{Son}), \text{AtMost}(2, \text{Daughter}), \\
&\quad \text{All}(\text{Son}, \text{And}(\text{Unemployed}, \text{Married}, \text{All}(\text{Spouse}, \text{Doctor}))), \\
&\quad \text{All}(\text{Daughter}, \text{And}(\text{Professor}, \text{Fills}(\text{Department}, \text{Physics}, \text{Math})))) .
\end{aligned}$$

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.⁷

⁶ 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.

⁷ CLASSIC provides efficient subsumption testing in practice, but the worst-case run time is exponential.

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. 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. 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.

12.6 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.

12.6.1 Circumscription and default logic

We have seen two examples of reasoning processes that violate the **monotonicity** property of logic that was proved in Chapter 7.⁸ In this chapter we saw that a property inherited by all members of a category in a semantic network could be overridden by more specific information for a subcategory. In Section 9.4.5, we saw that under the closed-world assumption, if a proposition α is not mentioned in KB then $KB \models \neg\alpha$, but $KB \wedge \alpha \models \alpha$.

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. 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
LOGIC

⁸ 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 $KB \wedge \beta \models \alpha$.

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, 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.⁹ 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:

$$\begin{aligned} & Republican(Nixon) \wedge Quaker(Nixon) . \\ & Republican(x) \wedge \neg Abnormal_2(x) \Rightarrow \neg Pacifist(x) . \\ & Quaker(x) \wedge \neg Abnormal_3(x) \Rightarrow Pacifist(x) . \end{aligned}$$

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 was 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

$$P : J_1, \dots, J_n / C$$

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

⁹ 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 closed-world assumption and definite-clause KBs, because the fixed point reached by forward chaining on definite-clause KBs is the unique minimal model. See page 258 for more on this point.

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:

$$\begin{aligned} & \text{Republican}(\text{Nixon}) \wedge \text{Quaker}(\text{Nixon}) . \\ & \text{Republican}(x) : \neg \text{Pacifist}(x) / \neg \text{Pacifist}(x) . \\ & \text{Quaker}(x) : \text{Pacifist}(x) / \text{Pacifist}(x) . \end{aligned}$$

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. 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, and the *costs* of making a wrong decision. 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 of faulty brakes. These considerations have led some researchers to consider how to embed default reasoning within probability theory or utility theory.

12.6.2 Truth maintenance systems

We have seen 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**.¹⁰ 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 $\text{TELL}(KB, \neg P)$. To avoid creating a contradiction, we must first execute $\text{RETRACT}(KB, P)$. This sounds easy enough.

BELIEF REVISION

¹⁰ 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.

TRUTH
MAINTENANCE
SYSTEM

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.

One 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 $\text{RETRACT}(\text{KB}, P_i)$ 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 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
JUSTIFICATION

A more efficient approach is the justification-based truth maintenance system, or **JTMS**. 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 $\text{TELL}(P)$ will cause Q to be added with the justification $\{P, P \Rightarrow Q\}$. In general, a sentence can have any number of justifications. Justifications make retraction efficient. Given the call $\text{RETRACT}(P)$, 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 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 $\text{Site}(\text{Swimming}, \text{Pitesti})$, $\text{Site}(\text{Athletics}, \text{Bucharest})$, and $\text{Site}(\text{Equestrian}, \text{Arad})$. 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 $\text{Site}(\text{Athletics}, \text{Sibiu})$ instead, the TMS avoids the need to start again from scratch. Instead, we simply retract $\text{Site}(\text{Athletics}, \text{Bucharest})$ and assert $\text{Site}(\text{Athletics}, \text{Sibiu})$ 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**, makes 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 in which all the assumptions in one of the assumption sets hold.

EXPLANATION 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 basis 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.

ASSUMPTION

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.

12.7 THE INTERNET SHOPPING WORLD

In this final section we put together all we have learned to encode knowledge for 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, and ranking which offers are best. In some cases the buyer's product description will be precise, as in *Canon Rebel XTi 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 in its full complexity—not a toy simulated environment. The agent's percepts are Web pages, but whereas a human

Example Online Store

Select from our fine line of products:

- Computers
- Cameras
- Books
- Videos
- Music

```
<h1>Example Online Store</h1>
<i>Select</i> from our fine line of products:
<ul>
<li> <a href="http://example.com/compu">Computers</a>
<li> <a href="http://example.com/camer">Cameras</a>
<li> <a href="http://example.com/books">Books</a>
<li> <a href="http://example.com/video">Videos</a>
<li> <a href="http://example.com/music">Music</a>
</ul>
```

Figure 12.8 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://example.com/books` is called a **uniform resource locator (URL)**. The markup `Books` means to create a hypertext link to *url* with the **anchor text** *Books*.

Web user would see pages displayed as an array of pixels on a screen, the shopping agent will perceive a page as a character string consisting of ordinary words interspersed with formatting commands in the HTML markup language. Figure 12.8 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 12.8 is simple compared to real shopping sites, which may include CSS, 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 concentrate on a simple agent that ignores most of these complications.

The agent's first task is to collect product offers that are relevant to a query. If the query is "laptops," then a Web 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 words "buy" or "price" or "add to cart" within an HTML link or form on the

page. For example, if the page contains a string of the form “<a...add to cart...” then it is an offer. This could be represented in first-order logic, but it is more straightforward to encode it into program code. We show how to do more sophisticated information extraction in Section 22.4.

12.7.1 Following links

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.¹¹ The agent will have knowledge of a number of stores, for example:

$$\begin{aligned} Amazon &\in OnlineStores \wedge Homepage(Amazon, "amazon.com") . \\ Ebay &\in OnlineStores \wedge Homepage(Ebay, "ebay.com") . \\ ExampleStore &\in OnlineStores \wedge Homepage(ExampleStore, "example.com") . \end{aligned}$$

These stores classify their goods into product categories, and provide links to the major categories from their home page. Minor categories can be reached through 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 zero or more relevant category links from a store’s home page, and then from one more link to the product offer. We can define relevance:

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

Here the predicate $Link(from, to)$ means that there is a hyperlink from the *from* URL to the *to* URL. 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 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:

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

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 $Name("laptops", LaptopComputers)$. Some more examples of the *Name* predicate appear in Figure 12.9(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., “notebooks” and “laptops.”

¹¹ 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 22.3.

$Books \subset Products$	$Name("books", Books)$
$MusicRecordings \subset Products$	$Name("music", MusicRecordings)$
$MusicCDs \subset MusicRecordings$	$Name("CDs", MusicCDs)$
$Electronics \subset Products$	$Name("electronics", Electronics)$
$DigitalCameras \subset Electronics$	$Name("digital cameras", DigitalCameras)$
$StereoEquipment \subset Electronics$	$Name("stereos", StereoEquipment)$
$Computers \subset Electronics$	$Name("computers", Computers)$
$DesktopComputers \subset Computers$	$Name("desktops", DesktopComputers)$
$LaptopComputers \subset Computers$	$Name("laptops", LaptopComputers)$
\dots	$Name("notebooks", LaptopComputers)$
(a)	(b)

Figure 12.9 (a) Taxonomy of product categories. (b) Names for those categories.

- 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:

$$\begin{aligned}
 &RelevantCategoryName(query, text) \Leftrightarrow \\
 &\exists c_1, c_2 \quad Name(query, c_1) \wedge Name(text, c_2) \wedge (c_1 \subseteq c_2 \vee c_2 \subseteq c_1) .
 \end{aligned} \tag{12.1}$$

Otherwise, the anchor text is irrelevant because it names a category outside this line, such as “clothes” 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 12.9(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

$$Name("CDs", CertificatesOfDeposit)$$

to the knowledge base in Figure 12.9(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 the very broad range of descriptions that a user can type and 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 airline seat.” 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 12.10 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 *Contents(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 *Contents* 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

12.7.2 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 22.4. 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 example.com site with the text

WRAPPER

IBM ThinkBook 970. Our price: \$399.00

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

$$\begin{aligned} \exists c, offer \quad & c \in LaptopComputers \wedge offer \in ProductOffers \wedge \\ & Manufacturer(c, IBM) \wedge Model(c, ThinkBook970) \wedge \\ & ScreenSize(c, Inches(14)) \wedge ScreenType(c, ColorLCD) \wedge \\ & MemorySize(c, Gigabytes(2)) \wedge CPUSpeed(c, GHz(1.2)) \wedge \\ & OfferedProduct(offer, c) \wedge Store(offer, GenStore) \wedge \\ & URL(offer, "example.com/computers/34356.html") \wedge \\ & Price(offer, \$(399)) \wedge Date(offer, Today) . \end{aligned}$$

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 the method of payment and on the buyer’s qualifications for certain discounts. The final task is to compare the offers that have been extracted. For example, consider these three offers:

A : 1.4 GHz CPU, 2GB RAM, 250 GB disk, \$299 .

B : 1.2 GHz CPU, 4GB RAM, 350 GB disk, \$500 .

C : 1.2 GHz CPU, 2GB RAM, 250 GB disk, \$399 .

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 such an agent, and that once we have some knowledge in this form, the rest follows naturally.

12.8 SUMMARY

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 and a feeling for the interesting philosophical issues that arise. 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.
- Building a large, general-purpose ontology is a significant challenge that has yet to be fully realized, although current frameworks seem to be quite robust.
- We presented an **upper ontology** based on categories and the event calculus. We covered categories, subcategories, parts, structured objects, measurements, substances, events, time and space, change, and beliefs.

- Natural kinds cannot be defined completely in logic, but properties of natural kinds can be represented.
- Actions, events, and time can be represented either in situation calculus or in more expressive representations such as event calculus. Such representations enable an agent to construct plans by logical inference.
- 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.
- **Truth maintenance systems** handle knowledge updates and revisions efficiently.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Briggs (1985) claims 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: Aristotle's *Metaphysics* (literally, what comes after the book on physics) is a near-synonym for *Ontology*. 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. 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 Linné (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, as documented by the *Handbook on Ontologies* (Staab, 2004). The OPENCYC project (Lenat and Guha, 1990; Matuszek *et al.*, 2006) has released a 150,000-concept ontology, with an upper ontology similar to the one in Figure 12.1 as well as specific concepts like “OLED Display” and “iPhone,” which is a type of “cellular phone,” which in turn is a type of “consumer electronics,” “phone,” “wireless communication device,” and other concepts. The DBPEDIA project extracts structured data from Wikipedia; specifically from Infoboxes: the boxes of attribute/value pairs that accompany many Wikipedia articles (Wu and Weld, 2008; Bizer *et al.*, 2007). As of mid-2009, DBPEDIA contains 2.6 million concepts, with about 100 facts per concept. The IEEE working group P1600.1 created the Suggested Upper Merged Ontology (SUMO) (Niles and Pease, 2001; Pease and Niles, 2002), which contains about 1000 terms in the upper ontology and links to over 20,000 domain-specific terms. Stoffel *et al.* (1997) describe algorithms for efficiently managing a very large ontology. A survey of techniques for extracting knowledge from Web pages is given by Etzioni *et al.* (2008).

On the Web, representation languages are emerging. RDF (Brickley and Guha, 2004) allows for assertions to be made in the form of relational triples, and provides some means for evolving the meaning of names over time. OWL (Smith *et al.*, 2004) is a description logic that supports inferences over these triples. So far, usage seems to be inversely proportional to representational complexity: the traditional HTML and CSS formats account for over 99% of Web content, followed by the simplest representation schemes, such as microformats (Khare, 2006) and RDFa (Adida and Birbeck, 2008), which use HTML and XHTML markup to add attributes to literal text. Usage of sophisticated RDF and OWL ontologies is not yet widespread, and the full vision of the Semantic Web (Berners-Lee *et al.*, 2001) has not yet been realized. 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, 2005). An inspirational discussion of the general project of commonsense knowledge representation appears in Hayes’s (1978, 1985b) “Naive Physics Manifesto.”

Successful deep ontologies within a specific field include the Gene Ontology project (Consortium, 2008) and CML, the Chemical Markup Language (Murray-Rust *et al.*, 2003).

Doubts about the feasibility of a single ontology for *all* knowledge are expressed by Doctorow (2001), Gruber (2004), Halevy *et al.* (2009), and Smith (2004), who states, “the initial project of building one single ontology ... has ... largely been abandoned.”

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, 1997) and overviews (Shanahan, 1999; Mueller, 2006). van Lambalgen and Hamm (2005) show how the logic of events maps onto the language we use to talk about events. An alternative to the event and situation calculi is the fluent calculus (Thielscher, 1999). James Allen introduced time intervals for the same reason (Allen, 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; van Beek and Manchak (1996) analyze algorithms for temporal reasoning. 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 and the **chronicles** in McDermott’s (1985) theory of plans were also important influences on the field and this chapter.

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(Butter₃, Butter)* rather than *Butter₃ ∈ 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 support of Quine (1960). Harry Bunt (1985) has provided an extensive analysis of its use in knowledge representation. Casati and Varzi (1999) cover parts, wholes, and the spatial locations.

Mental objects have been the subject of intensive study in philosophy and AI. There are three main approaches. The one taken in this chapter, based on modal logic and possible worlds, is the classical approach from philosophy (Hintikka, 1962; Kripke, 1963; Hughes and Cresswell, 1996). The book *Reasoning about Knowledge* (Fagin *et al.*, 1995) provides a thorough introduction. The second approach is a first-order theory in which mental objects are fluents. Davis (2005) and Davis and Morgenstern (2005) describe this approach. It relies on the possible-worlds formalism, and builds on work by Robert Moore (1980, 1985). The third approach is a **syntactic theory**, in which mental objects are represented by character

strings. A string is just a complex term denoting a list of symbols, so *CanFly(Clark)* can be represented by the list of symbols $[C, a, n, F, l, y, (, C, l, a, r, k,)]$. 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. Ernie Davis (1990) provides 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. 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" Woods's (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). 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), several researchers developed systems that incorporate 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, and Baader *et al.* (2007) present a comprehensive handbook of description logic. 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. Delgrande and Schaub (2003) discuss the merits of the variants, given 25 years of hindsight. Answer set programming can be seen as an extension of negation as failure or as a refinement of circum-

scription; 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-benefit inquiries was perhaps the first commercially successful 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. Forbus and de Kleer (1993) explain in depth how TMSs can be used in AI applications. Nayak and Williams (1997) show how an efficient incremental TMS called an ITMS makes it feasible to plan the operations of a NASA spacecraft in real time.

This chapter could 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. Davis (2008) gives an update to the ontology of liquids that describes the pouring of liquids into containers.

De Kleer and Brown (1985), Ken Forbus (1985), and Benjamin Kuipers (1985) independently and almost simultaneously developed systems that can reason about a physical system based on qualitative abstractions of the underlying equations. Qualitative physics soon developed to the point where it became possible to analyze an impressive variety of complex physical systems (Yip, 1991). Qualitative techniques have been used to construct novel designs for clocks, windshield wipers, and six-legged walkers (Subramanian and Wang, 1994). The collection *Readings in Qualitative Reasoning about Physical Systems* (Weld and

de Kleer, 1990) an encyclopedia article by Kuipers (2001), and a handbook article by Davis (2007) introduce 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 systems; see also (Davis, 2006). 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.

Minker (2001) collects papers by leading researchers in knowledge representation, summarizing 40 years of work in the field. 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, van Harmelen *et al.* (2007) contributes a handbook, and a special issue of *AI Journal* covers recent progress (Davis and Morgenstern, 2004). The biennial conference on *Theoretical Aspects of Reasoning About Knowledge* (TARK) covers applications of the theory of knowledge in AI, economics, and distributed systems.

EXERCISES

12.1 Define an ontology in first-order logic for tic-tac-toe. The ontology should contain situations, actions, squares, players, marks (X, O, or blank), and the notion of winning, losing, or drawing a game. Also define the notion of a forced win (or draw): a position from which a player can force a win (or draw) with the right sequence of actions. Write axioms for the domain. (Note: The axioms that enumerate the different squares and that characterize the winning positions are rather long. You need not write these out in full, but indicate clearly what they look like.)

12.2 Figure 12.1 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.

12.3 Develop a representational system for reasoning about windows in a window-based computer interface. In particular, your representation should be able to describe:

- The state of a window: minimized, displayed, or nonexistent.
- Which window (if any) is the active window.
- The position of every window at a given time.
- The order (front to back) of overlapping windows.
- The actions of creating, destroying, resizing, and moving windows; changing the state of a window; and bringing a window to the front. Treat these actions as atomic; that is, do not deal with the issue of relating them to mouse actions. Give axioms describing the effects of actions on fluents. You may use either event or situation calculus.

Assume an ontology containing *situations*, *actions*, *integers* (for x and y coordinates) and *windows*. Define a language over this ontology; that is, a list of constants, function symbols, and predicates with an English description of each. If you need to add more categories to the ontology (e.g., pixels), you may do so, but be sure to specify these in your write-up. You may (and should) use symbols defined in the text, but be sure to list these explicitly.

12.4 State the following in the language you developed for the previous exercise:

- a. In situation S_0 , window W_1 is behind W_2 but sticks out on the left and right. Do *not* state exact coordinates for these; describe the *general* situation.
- b. If a window is displayed, then its top edge is higher than its bottom edge.
- c. After you create a window w , it is displayed.
- d. A window can be minimized if it is displayed.

12.5 (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 scenario:

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?

To answer the questions below, your knowledge base must include background knowledge. You'll have to deal with what kind of things are at a supermarket, what is involved with purchasing the things one selects, what the purchases will 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. 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. 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 some money or a credit card to the supermarket? [Yes]
- i. Does John have less money after going to the supermarket? [Yes]

12.6 Make the necessary additions or changes to your knowledge base from the previous exercise so that the questions that follow can be answered. Include in your report a discussion of your changes, explaining why they were needed, whether they were minor or major, and what kinds of questions would necessitate further changes.

- 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]

12.7 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.

12.8 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.

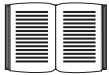
12.9 An alternative scheme for representing measures involves applying the units function to an abstract length object. In such a scheme, one would write $Inches(Length(L_1)) = 1.5$. How does this scheme compare with the one in the chapter? Issues include conversion axioms, names for abstract quantities (such as “50 dollars”), and comparisons of abstract measures in different units (50 inches is more than 50 centimeters).

12.10 Add sentences to extend the definition of the predicate $Name(s, c)$ so that a string such as “laptop computer” matches 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 sentences for handling plurals, conjunctions, etc.

12.11 Write event calculus axioms to describe the actions in the wumpus world.

12.12 State the interval-algebra relation that holds between every pair of the following real-world events:

- LK*: The life of President Kennedy.
- IK*: The infancy of President Kennedy.
- PK*: The presidency of President Kennedy.
- LJ*: The life of President Johnson.
- PJ*: The presidency of President Johnson.
- LO*: The life of President Obama.

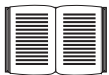


12.13 Investigate ways to extend the event calculus to handle *simultaneous* events. Is it possible to avoid a combinatorial explosion of axioms?

12.14 Construct a representation for exchange rates between currencies that allows for daily fluctuations.

12.15 Define the predicate *Fixed*, where $Fixed(Location(x))$ means that the location of object x is fixed over time.

12.16 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.



12.17 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.

12.18 (Adapted from Fagin *et al.* (1995).) Consider a game played with a deck of just 8 cards, 4 aces and 4 kings. The three players, Alice, Bob, and Carlos, are dealt two cards each. Without looking at them, they place the cards on their foreheads so that the other players can see them. Then the players take turns either announcing that they know what cards are on their own forehead, thereby winning the game, or saying "I don't know." Everyone knows the players are truthful and are perfect at reasoning about beliefs.

- a. Game 1. Alice and Bob have both said "I don't know." Carlos sees that Alice has two aces (A-A) and Bob has two kings (K-K). What should Carlos say? (*Hint*: consider all three possible cases for Carlos: A-A, K-K, A-K.)
- b. Describe each step of Game 1 using the notation of modal logic.
- c. Game 2. Carlos, Alice, and Bob all said "I don't know" on their first turn. Alice holds K-K and Bob holds A-K. What should Carlos say on his second turn?
- d. Game 3. Alice, Carlos, and Bob all say "I don't know" on their first turn, as does Alice on her second turn. Alice and Bob both hold A-K. What should Carlos say?
- e. Prove that there will always be a winner to this game.

12.19 The assumption of *logical omniscience*, discussed on page 453, is of course not true of any actual reasoners. Rather, it is an *idealization* of the reasoning process that may be more or less acceptable depending on the applications. Discuss the reasonableness of the assumption for each of the following applications of reasoning about knowledge:

- a. Partial knowledge adversary games, such as card games. Here one player wants to reason about what his opponent knows about the state of the game.
- b. Chess with a clock. Here the player may wish to reason about the limits of his opponent's or his own ability to find the best move in the time available. For instance, if player A has much more time left than player B, then A will sometimes make a move that greatly complicates the situation, in the hopes of gaining an advantage because he has more time to work out the proper strategy.

- c. A shopping agent in an environment in which there are costs of gathering information.
- d. Reasoning about public key cryptography, which rests on the intractability of certain computational problems.

12.20 Translate the following description logic expression (from page 457) into first-order logic, and comment on the result:

*And(Man, AtLeast(3, Son), AtMost(2, Daughter),
 All(Son, And(Unemployed, Married, All(Spouse, Doctor))),
 All(Daughter, And(Professor, Fills(Department, Physics, Math)))) .*

12.21 Recall that inheritance information in semantic networks can be captured logically by suitable implication sentences. This exercise investigates the efficiency of using such sentences for inheritance.

- a. Consider the information in a used-car catalog such as Kelly's Blue Book—for example, that 1973 Dodge vans are (or perhaps were once) worth \$575. Suppose all this information (for 11,000 models) is encoded as logical sentences, as suggested in the chapter. Write down three such sentences, including that for 1973 Dodge vans. How would you use the sentences to find the value of a *particular* car, 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 sentences about prices.
- d. Describe a situation in which neither forward nor backward chaining on the sentences 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 year and model have the same price.)

12.22 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.

12.23 One part of the shopping process that was not covered in this chapter is checking for compatibility between items. For example, if a digital camera is ordered, what accessory batteries, memory cards, and cases are compatible with the camera? Write a knowledge base that can determine the compatibility of a set of items and suggest replacements or additional items if the shopper makes a choice that is not compatible. The knowledge base should work with at least one line of products and extend easily to other lines.

12.24 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. The buyer must use the following grammar for product descriptions:

$$\begin{aligned} \textit{Description} &\rightarrow \textit{Category} [\textit{Connector} \textit{ Modifier}]^* \\ \textit{Connector} &\rightarrow \text{“with”} \mid \text{“and”} \mid \text{“,”} \\ \textit{Modifier} &\rightarrow \textit{Attribute} \mid \textit{Attribute Op Value} \\ \textit{Op} &\rightarrow \text{“=”} \mid \text{“>”} \mid \text{“<”} \end{aligned}$$

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 \$500” must be re-expressed as “computer with CPU > 2.5 GHz and price < \$500.” Implement a shopping agent that accepts descriptions in this language.

12.25 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.

13 QUANTIFYING UNCERTAINTY

In which we see how an agent can tame uncertainty with degrees of belief.

13.1 ACTING UNDER UNCERTAINTY

UNCERTAINTY

Agents may need to handle **uncertainty**, whether due to partial observability, nondeterminism, or a combination of the two. An agent may never know for certain what state it's in or where it will end up after a sequence of actions.

We have seen problem-solving agents (Chapter 4) and logical agents (Chapters 7 and 11) designed to handle uncertainty by keeping track of a **belief state**—a representation of the set of all possible world states that it might be in—and generating a contingency plan that handles every possible eventuality that its sensors may report during execution. Despite its many virtues, however, this approach has significant drawbacks when taken literally as a recipe for creating agent programs:

- When interpreting partial sensor information, a logical agent must consider *every logically possible* explanation for the observations, no matter how unlikely. This leads to impossible large and complex belief-state representations.
- A correct contingent plan that handles every eventuality can grow arbitrarily large and must consider arbitrarily unlikely contingencies.
- Sometimes there is no plan that is guaranteed to achieve the goal—yet the agent must act. It must have some way to compare the merits of plans that are not guaranteed.

Suppose, for example, that an automated taxi¹automated has the goal of delivering a passenger to the airport on time. The agent forms 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 5 miles away, a logical taxi 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 the 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 no meteorite hits the car, and” None of these conditions can be

deduced for sure, so the plan's success cannot be inferred. This is the **qualification problem** (page 268), for which we so far have seen no real solution.

Nonetheless, in some sense A_{90} is in fact the right thing to do. What do we mean by 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 (where the expectation is relative to the agent's knowledge 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 agent's knowledge 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_{180} , 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.



13.1.1 Summarizing uncertainty

Let's consider an example of uncertain reasoning: diagnosing a dental patient's toothache. Diagnosis—whether for medicine, automobile repair, or whatever—almost always involves uncertainty. Let us try to write rules for dental diagnosis using propositional logic, so that we can see how the logical approach breaks down. Consider the following simple rule:

$$\text{Toothache} \Rightarrow \text{Cavity} .$$

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:

$$\text{Toothache} \Rightarrow \text{Cavity} \vee \text{GumProblem} \vee \text{Abscess} \dots$$

Unfortunately, in order to make the rule true, we have to add an almost unlimited list of possible problems. We could try turning the rule into a causal rule:

$$\text{Cavity} \Rightarrow \text{Toothache} .$$

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. Trying to use logic to cope with a domain like medical diagnosis thus fails for three main reasons:

- **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:** Medical science has no complete theory for the domain.
- **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

LAZINESS

THEORETICAL
IGNORANCE
PRACTICAL
IGNORANCE

DEGREE OF BELIEF
PROBABILITY
THEORY



can at best provide only a **degree of belief** in the relevant sentences. Our main tool for dealing with degrees of belief is **probability theory**. In the terminology of Section 8.1, the **ontological commitments** of logic and probability theory are the same—that the world is composed of facts that do or do not hold in any particular case—but the **epistemological commitments** are different: a logical agent believes each sentence to be true or false or has no opinion, whereas a probabilistic agent may have a numerical degree of belief between 0 (for sentences that are certainly false) and 1 (certainly true).

Probability provides a way of summarizing the uncertainty that comes from our laziness and ignorance, thereby solving the qualification problem. 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 who has a toothache has a cavity. That is, we expect that out of all the situations that are indistinguishable from the current situation as far as our 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 dental knowledge, or from a combination of evidence sources.

One confusing point is that at the time of our diagnosis, there is no uncertainty in the actual world: the patient either has a cavity or doesn't. So what does it mean to say the probability of a cavity is 0.8? Shouldn't it be either 0 or 1? The answer is that probability statements are made with respect to a knowledge state, not with respect to the real world. We say “The probability that the patient has a cavity, *given that she has a toothache*, is 0.8.” If we later learn that the patient has a history of gum disease, we can make a different statement: “The probability that the patient has a cavity, *given that she has a toothache and a history of gum disease*, is 0.4.” If we gather further conclusive evidence against a cavity, we can say “The probability that the patient has a cavity, *given all we now know*, is almost 0.” Note that these statements do not contradict each other; each is a separate assertion about a different knowledge state.

13.1.2 Uncertainty and rational decisions

Consider again the A_{90} plan for getting to the airport. Suppose it gives us a 97% chance of catching our flight. Does this mean it is a rational choice? Not necessarily: there might be other plans, such as A_{180} , with higher probabilities. 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—not to mention a possibly unpleasant diet of airport food.

To make such choices, an agent must first have **preferences** between the different possible **outcomes** of the various plans. An 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 use **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.

PREFERENCE
OUTCOME
UTILITY THEORY

The utility of a state is relative to an agent. For example, the utility of a state in which White has checkmated Black in a game of chess is obviously high for the agent playing White, but low for the agent playing Black. But we can't go strictly by the scores of 1, 1/2, and 0 that are dictated by the rules of tournament chess—some players (including the authors) might be thrilled 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 account for any set of preferences—quirky or typical, noble or perverse. Note that utilities can account for altruism, simply by including the welfare of others as one of the factors.

Preferences, as expressed by utilities, are combined with probabilities in the general theory of rational decisions called **decision theory**:

DECISION THEORY

$$\text{Decision theory} = \text{probability theory} + \text{utility theory} .$$

MAXIMUM EXPECTED
UTILITY

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). Note that “expected” might seem like a vague, hypothetical term, but as it is used here it has a precise meaning: it means the “average,” or “statistical mean” of the outcomes, weighted by the probability of the outcome. We saw this principle in action in Chapter 5 when we touched briefly on optimal decisions in backgammon; it is in fact a completely general principle.

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 agents described in Chapters 4 and 7 that maintain a belief state reflecting the history of percepts to date. The primary difference is that the decision-theoretic agent's belief state represents not just the *possibilities* for world states but also their *probabilities*. 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 over time and predicting the environment. Chapter 16 covers utility theory in more depth, and Chapter 17 develops algorithms for planning sequences of actions in uncertain environments.

13.2 BASIC PROBABILITY NOTATION

For our agent to represent and use probabilistic information, we need a formal language. The language of probability theory has traditionally been informal, written by human mathematicians to other human mathematicians. Appendix A includes a standard introduction to elementary probability theory; here, we take an approach more suited to the needs of AI and more consistent with the concepts of formal logic.

```
function DT-AGENT(percept) returns an action
  persistent: 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.

13.2.1 What probabilities are about

SAMPLE SPACE

Like logical assertions, probabilistic assertions are about possible worlds. Whereas logical assertions say which possible worlds are strictly ruled out (all those in which the assertion is false), probabilistic assertions talk about how probable the various worlds are. In probability theory, the set of all possible worlds is called the **sample space**. The possible worlds are *mutually exclusive* and *exhaustive*—two possible worlds cannot both be the case, and one possible world must be the case. For example, if we are about to roll two (distinguishable) dice, there are 36 possible worlds to consider: (1,1), (1,2), . . . , (6,6). The Greek letter Ω (uppercase omega) is used to refer to the sample space, and ω (lowercase omega) refers to elements of the space, that is, particular possible worlds.

PROBABILITY MODEL

A fully specified **probability model** associates a numerical probability $P(\omega)$ with each possible world.¹ The basic axioms of probability theory say that every possible world has a probability between 0 and 1 and that the total probability of the set of possible worlds is 1:

$$0 \leq P(\omega) \leq 1 \text{ for every } \omega \text{ and } \sum_{\omega \in \Omega} P(\omega) = 1 .$$

(13.1)

For example, if we assume that each die is fair and the rolls don't interfere with each other, then each of the possible worlds (1,1), (1,2), . . . , (6,6) has probability 1/36. On the other hand, if the dice conspire to produce the same number, then the worlds (1,1), (2,2), (3,3), etc., might have higher probabilities, leaving the others with lower probabilities.

EVENT

Probabilistic assertions and queries are not usually about particular possible worlds, but about sets of them. For example, we might be interested in the cases where the two dice add up to 11, the cases where doubles are rolled, and so on. In probability theory, these sets are called **events**—a term already used extensively in Chapter 12 for a different concept. In AI, the sets are always described by **propositions** in a formal language. (One such language is described in Section 13.2.2.) For each proposition, the corresponding set contains just those possible worlds in which the proposition holds. The probability associated with a proposition

¹ For now, we assume a discrete, countable set of worlds. The proper treatment of the continuous case brings in certain complications that are less relevant for most purposes in AI.

is defined to be the sum of the probabilities of the worlds in which it holds:

$$\text{For any proposition } \phi, P(\phi) = \sum_{\omega \in \phi} P(\omega). \quad (13.2)$$

For example, when rolling fair dice, we have $P(\text{Total} = 11) = P((5, 6)) + P((6, 5)) = 1/36 + 1/36 = 1/18$. Note that probability theory does not require complete knowledge of the probabilities of each possible world. For example, if we believe the dice conspire to produce the same number, we might *assert* that $P(\text{doubles}) = 1/4$ without knowing whether the dice prefer double 6 to double 2. Just as with logical assertions, this assertion *constrains* the underlying probability model without fully determining it.

UNCONDITIONAL
PROBABILITY
PRIOR PROBABILITY

EVIDENCE

CONDITIONAL
PROBABILITY
POSTERIOR
PROBABILITY

Probabilities such as $P(\text{Total} = 11)$ and $P(\text{doubles})$ are called **unconditional** or **prior probabilities** (and sometimes just “priors” for short); they refer to degrees of belief in propositions *in the absence of any other information*. Most of the time, however, we have *some* information, usually called **evidence**, that has already been revealed. For example, the first die may already be showing a 5 and we are waiting with bated breath for the other one to stop spinning. In that case, we are interested not in the unconditional probability of rolling doubles, but the **conditional** or **posterior** probability (or just “posterior” for short) of rolling doubles *given that the first die is a 5*. This probability is written $P(\text{doubles} \mid \text{Die}_1 = 5)$, where the “ \mid ” is pronounced “given.” Similarly, if I am going to the dentist for a regular checkup, the probability $P(\text{cavity}) = 0.2$ might be of interest; but if I go to the dentist because I have a toothache, it’s $P(\text{cavity} \mid \text{toothache}) = 0.6$ that matters. Note that the precedence of “ \mid ” is such that any expression of the form $P(\dots \mid \dots)$ always means $P((\dots) \mid (\dots))$.

It is important to understand that $P(\text{cavity}) = 0.2$ is still *valid* after *toothache* is observed; it just isn’t especially useful. When making decisions, an agent needs to condition on *all* the evidence it has observed. It is also important to understand the difference between conditioning and logical implication. The assertion that $P(\text{cavity} \mid \text{toothache}) = 0.6$ does not mean “Whenever *toothache* is true, conclude that *cavity* is true with probability 0.6” rather it means “Whenever *toothache* is true *and we have no further information*, conclude that *cavity* is true with probability 0.6.” The extra condition is important; for example, if we had the further information that the dentist found no cavities, we definitely would not want to conclude that *cavity* is true with probability 0.6; instead we need to use $P(\text{cavity} \mid \text{toothache} \wedge \neg \text{cavity}) = 0$.

Mathematically speaking, conditional probabilities are defined in terms of unconditional probabilities as follows: for any propositions a and b , we have

$$P(a \mid b) = \frac{P(a \wedge b)}{P(b)}, \quad (13.3)$$

which holds whenever $P(b) > 0$. For example,

$$P(\text{doubles} \mid \text{Die}_1 = 5) = \frac{P(\text{doubles} \wedge \text{Die}_1 = 5)}{P(\text{Die}_1 = 5)}.$$

The definition makes sense if you remember that observing b rules out all those possible worlds where b is false, leaving a set whose total probability is just $P(b)$. Within that set, the a -worlds satisfy $a \wedge b$ and constitute a fraction $P(a \wedge b)/P(b)$.

PRODUCT RULE

The definition of conditional probability, Equation (13.3), can be written in a different form called the **product rule**:

$$P(a \wedge b) = P(a | b)P(b) ,$$

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 .

13.2.2 The language of propositions in probability assertions

In this chapter and the next, propositions describing sets of possible worlds are written in a notation that combines elements of propositional logic and constraint satisfaction notation. In the terminology of Section 2.4.7, it is a **factored representation**, in which a possible world is represented by a set of variable/value pairs.

RANDOM VARIABLE

DOMAIN

Variables in probability theory are called **random variables** and their names begin with an uppercase letter. Thus, in the dice example, *Total* and *Die₁* are random variables. Every random variable has a **domain**—the set of possible values it can take on. The domain of *Total* for two dice is the set $\{2, \dots, 12\}$ and the domain of *Die₁* is $\{1, \dots, 6\}$. A Boolean random variable has the domain $\{true, false\}$ (notice that values are always lowercase); for example, the proposition that doubles are rolled can be written as *Doubles* = *true*. By convention, propositions of the form $A = true$ are abbreviated simply as a , while $A = false$ is abbreviated as $\neg a$. (The uses of *doubles*, *cavity*, and *toothache* in the preceding section are abbreviations of this kind.) As in CSPs, domains can be sets of arbitrary tokens; we might choose the domain of *Age* to be $\{juvenile, teen, adult\}$ and the domain of *Weather* might be $\{sunny, rain, cloudy, snow\}$. When no ambiguity is possible, it is common to use a value by itself to stand for the proposition that a particular variable has that value; thus, *sunny* can stand for *Weather* = *sunny*.

The preceding examples all have finite domains. Variables can have infinite domains, too—either discrete (like the integers) or continuous (like the reals). For any variable with an ordered domain, inequalities are also allowed, such as *NumberOfAtomsInUniverse* $\geq 10^{70}$.

Finally, we can combine these sorts of elementary propositions (including the abbreviated forms for Boolean variables) by using the connectives of propositional logic. For example, we can express “The probability that the patient has a cavity, given that she is a teenager with no toothache, is 0.1” as follows:

$$P(cavity | \neg toothache \wedge teen) = 0.1 .$$

Sometimes we will want to talk about the probabilities of *all* the possible values of a random variable. We could write:

$$\begin{aligned} P(Weather = sunny) &= 0.6 \\ P(Weather = rain) &= 0.1 \\ P(Weather = cloudy) &= 0.29 \\ P(Weather = snow) &= 0.01 , \end{aligned}$$

but as an abbreviation we will allow

$$\mathbf{P}(Weather) = \langle 0.6, 0.1, 0.29, 0.01 \rangle ,$$

PROBABILITY
DISTRIBUTION

where the bold \mathbf{P} indicates that the result is a vector of numbers, and where we assume a pre-defined ordering $\langle \text{sunny}, \text{rain}, \text{cloudy}, \text{snow} \rangle$ on the domain of *Weather*. We say that the \mathbf{P} statement defines a **probability distribution** for the random variable *Weather*. The \mathbf{P} notation is also used for conditional distributions: $\mathbf{P}(X | Y)$ gives the values of $P(X = x_i | Y = y_j)$ for each possible i, j pair.

For continuous variables, it is not possible to write out the entire distribution as a vector, because there are infinitely many values. Instead, we can define the probability that a random variable takes on some value x as a parameterized function of x . For example, the sentence

$$P(\text{NoonTemp} = x) = \text{Uniform}_{[18C, 26C]}(x)$$

PROBABILITY
DENSITY FUNCTION

expresses the belief that the temperature at noon is distributed uniformly between 18 and 26 degrees Celsius. We call this a **probability density function**.

Probability density functions (sometimes called **pdfs**) differ in meaning from discrete distributions. Saying that the probability density is uniform from 18C to 26C means that there is a 100% chance that the temperature will fall somewhere in that 8C-wide region and a 50% chance that it will fall in any 4C-wide region, and so on. We write the probability density for a continuous random variable X at value x as $P(X = x)$ or just $P(x)$; the intuitive definition of $P(x)$ is the probability that X falls within an arbitrarily small region beginning at x , divided by the width of the region:

$$P(x) = \lim_{dx \rightarrow 0} P(x \leq X \leq x + dx) / dx .$$

For *NoonTemp* we have

$$P(\text{NoonTemp} = x) = \text{Uniform}_{[18C, 26C]}(x) = \begin{cases} \frac{1}{8C} & \text{if } 18C \leq x \leq 26C \\ 0 & \text{otherwise} \end{cases} ,$$

where C stands for centigrade (not for a constant). In $P(\text{NoonTemp} = 20.18C) = \frac{1}{8C}$, note that $\frac{1}{8C}$ is not a probability, it is a probability density. The probability that *NoonTemp* is *exactly* 20.18C is zero, because 20.18C is a region of width 0. 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.

JOINT PROBABILITY
DISTRIBUTION

In addition to distributions on single variables, we need notation for distributions on multiple variables. Commas are used for this. For example, $\mathbf{P}(\text{Weather}, \text{Cavity})$ denotes the probabilities of all combinations of the values of *Weather* and *Cavity*. This is a 4×2 table of probabilities called the **joint probability distribution** of *Weather* and *Cavity*. We can also mix variables with and without values; $\mathbf{P}(\text{sunny}, \text{Cavity})$ would be a two-element vector giving the probabilities of a sunny day with a cavity and a sunny day with no cavity. The \mathbf{P} notation makes certain expressions much more concise than they might otherwise be. For example, the product rules for all possible values of *Weather* and *Cavity* can be written as a single equation:

$$\mathbf{P}(\text{Weather}, \text{Cavity}) = \mathbf{P}(\text{Weather} | \text{Cavity}) \mathbf{P}(\text{Cavity}) ,$$

instead of as these $4 \times 2 = 8$ equations (using abbreviations W and C):

$$\begin{aligned}
 P(W = \text{sunny} \wedge C = \text{true}) &= P(W = \text{sunny} | C = \text{true}) P(C = \text{true}) \\
 P(W = \text{rain} \wedge C = \text{true}) &= P(W = \text{rain} | C = \text{true}) P(C = \text{true}) \\
 P(W = \text{cloudy} \wedge C = \text{true}) &= P(W = \text{cloudy} | C = \text{true}) P(C = \text{true}) \\
 P(W = \text{snow} \wedge C = \text{true}) &= P(W = \text{snow} | C = \text{true}) P(C = \text{true}) \\
 P(W = \text{sunny} \wedge C = \text{false}) &= P(W = \text{sunny} | C = \text{false}) P(C = \text{false}) \\
 P(W = \text{rain} \wedge C = \text{false}) &= P(W = \text{rain} | C = \text{false}) P(C = \text{false}) \\
 P(W = \text{cloudy} \wedge C = \text{false}) &= P(W = \text{cloudy} | C = \text{false}) P(C = \text{false}) \\
 P(W = \text{snow} \wedge C = \text{false}) &= P(W = \text{snow} | C = \text{false}) P(C = \text{false}) .
 \end{aligned}$$

As a degenerate case, $\mathbf{P}(\text{sunny}, \text{cavity})$ has no variables and thus is a one-element vector that is the probability of a sunny day with a cavity, which could also be written as $P(\text{sunny}, \text{cavity})$ or $P(\text{sunny} \wedge \text{cavity})$. We will sometimes use \mathbf{P} notation to derive results about individual P values, and when we say “ $\mathbf{P}(\text{sunny}) = 0.6$ ” it is really an abbreviation for “ $\mathbf{P}(\text{sunny})$ is the one-element vector $\langle 0.6 \rangle$, which means that $P(\text{sunny}) = 0.6$.”

Now we have defined a syntax for propositions and probability assertions and we have given part of the semantics: Equation (13.2) defines the probability of a proposition as the sum of the probabilities of worlds in which it holds. To complete the semantics, we need to say what the worlds are and how to determine whether a proposition holds in a world. We borrow this part directly from the semantics of propositional logic, as follows. *A possible world is defined to be an assignment of values to all of the random variables under consideration.* It is easy to see that this definition satisfies the basic requirement that possible worlds be mutually exclusive and exhaustive (Exercise 13.5). For example, if the random variables are *Cavity*, *Toothache*, and *Weather*, then there are $2 \times 2 \times 4 = 16$ possible worlds. Furthermore, the truth of any given proposition, no matter how complex, can be determined easily in such worlds using the same recursive definition of truth as for formulas in propositional logic.

From the preceding definition of possible worlds, it follows that a probability model is completely determined by the joint distribution for all of the random variables—the so-called **full joint probability distribution**. For example, if the variables are *Cavity*, *Toothache*, and *Weather*, then the full joint distribution is given by $\mathbf{P}(\text{Cavity}, \text{Toothache}, \text{Weather})$. This joint distribution can be represented as a $2 \times 2 \times 4$ table with 16 entries. Because every proposition’s probability is a sum over possible worlds, a full joint distribution suffices, in principle, for calculating the probability of any proposition.

13.2.3 Probability axioms and their reasonableness

The basic axioms of probability (Equations (13.1) and (13.2)) imply certain relationships among the degrees of belief that can be accorded to logically related propositions. For example, we can derive the familiar relationship between the probability of a proposition and the probability of its negation:

$$\begin{aligned}
 P(\neg a) &= \sum_{\omega \in \neg a} P(\omega) && \text{by Equation (13.2)} \\
 &= \sum_{\omega \in \neg a} P(\omega) + \sum_{\omega \in a} P(\omega) - \sum_{\omega \in a} P(\omega) \\
 &= \sum_{\omega \in \Omega} P(\omega) - \sum_{\omega \in a} P(\omega) && \text{grouping the first two terms} \\
 &= 1 - P(a) && \text{by (13.1) and (13.2).}
 \end{aligned}$$



INCLUSION-
EXCLUSION
PRINCIPLE

We can also derive the well-known formula for the probability of a disjunction, sometimes called the **inclusion–exclusion principle**:

$$P(a \vee b) = P(a) + P(b) - P(a \wedge b) . \quad (13.4)$$

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 $P(a \wedge b)$. The proof is left as an exercise (Exercise 13.6).

KOLMOGOROV'S
AXIOMS

Equations (13.1) and (13.4) 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 and how to handle the difficulties caused by continuous variables.² While Equation (13.2) has a definitional flavor, Equation (13.4) reveals that the axioms really do constrain the degrees of belief an agent can have concerning logically related propositions. This is analogous to the fact that a logical agent cannot simultaneously believe A , B , and $\neg(A \wedge B)$, because there is no possible world in which all three are true. With probabilities, however, 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 (even though they violate Kolmogorov's axioms)?

$$\begin{array}{ll} P(a) = 0.4 & P(a \wedge b) = 0.0 \\ P(b) = 0.3 & P(a \vee b) = 0.8 . \end{array} \quad (13.5)$$

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.

One argument for the axioms of probability, first stated in 1931 by Bruno de Finetti (and translated into English in de Finetti (1993)), is as follows: 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 .³ 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 wager for or against a at stakes that are consistent with the stated degree of belief. That is, Agent 2 could choose to accept Agent 1's bet that a will occur, offering \$6 against Agent 1's \$4. Or Agent 2 could accept Agent 1's bet that $\neg a$ will occur, offering \$4 against Agent 1's \$6. Then we observe the outcome of a , and whoever is right collects the money. 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.* For example, suppose that Agent 1 has the set of degrees of belief from Equation (13.5). Figure 13.2 shows that if Agent

² The difficulties include the **Vitali set**, a well-defined subset of the interval $[0, 1]$ with no well-defined size.

³ 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.

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 . De Finetti's theorem implies that no rational agent can have beliefs that violate the axioms of probability.

Agent 1		Agent 2		Outcomes and payoffs to Agent 1			
Proposition	Belief	Bet	Stakes	a, b	$a, \neg b$	$\neg a, b$	$\neg a, \neg b$
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 .

One common objection to de Finetti's theorem is 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.

Other strong philosophical arguments have been put forward for the use of probabilities, most notably those of Cox (1946), Carnap (1950), and Jaynes (2003). They each construct a set of axioms for reasoning with degrees of beliefs: no contradictions, correspondence with ordinary logic (for example, if belief in A goes up, then belief in $\neg A$ must go down), and so on. The only controversial axiom is that degrees of belief must be numbers, or at least act like numbers in that they must be transitive (if belief in A is greater than belief in B , which is greater than belief in C , then belief in A must be greater than C) and comparable (the belief in A must be one of equal to, greater than, or less than belief in B). It can then be proved that probability is the only approach that satisfies these axioms.

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.

13.3 INFERENCE USING FULL JOINT DISTRIBUTIONS

PROBABILISTIC INFERENCE

In this section we describe a simple method for **probabilistic inference**—that is, the computation of posterior probabilities for query propositions given observed evidence. We use the full joint distribution as the “knowledge base” from which answers to all questions may be derived. Along the way we also introduce several useful techniques for manipulating equations involving probabilities.

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, the fact 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. The subjective **Bayesian** view allows any self-consistent ascription of prior probabilities to propositions, but then insists on proper Bayesian updating as evidence arrives.

In the end, even a strict frequentist position involves subjective analysis because of the **reference class** problem: in trying to determine the outcome probability of a *particular* experiment, the frequentist has to place it in a reference class of “similar” experiments with known outcome frequencies. I. J. Good (1983, p. 27) wrote, “every event in life is unique, and every real-life probability that we estimate in practice is that of an event that has never occurred before.” For example, given a particular patient, a frequentist who wants to estimate the probability of a cavity will consider a reference class of other patients who are similar in important ways—age, symptoms, diet—and see what proportion of them had a cavity. If the dentist considers everything that is known about the patient—weight to the nearest gram, hair color, mother’s maiden name—then the reference class becomes empty. This has been a vexing problem in the philosophy of science.

The **principle of indifference** attributed to Laplace (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.

	<i>toothache</i>		\neg <i>toothache</i>	
	<i>catch</i>	\neg <i>catch</i>	<i>catch</i>	\neg <i>catch</i>
<i>cavity</i>	0.108	0.012	0.072	0.008
\neg <i>cavity</i>	0.016	0.064	0.144	0.576

Figure 13.3 A full joint distribution for the *Toothache*, *Cavity*, *Catch* world.

We begin with a 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.

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: simply identify those possible worlds in which the proposition is true and add up their probabilities. For example, there are six possible worlds 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**⁴ of *cavity*:

$$P(cavity) = 0.108 + 0.012 + 0.072 + 0.008 = 0.2 .$$

This process is called **marginalization**, or **summing out**—because we sum up the probabilities for each possible value of the other variables, thereby taking them out of the equation. We can write the following general marginalization rule for any sets of variables **Y** and **Z**:

$$P(Y) = \sum_{z \in Z} P(Y, z) , \quad (13.6)$$

where $\sum_{z \in Z}$ means to sum over all the possible combinations of values of the set of variables **Z**. We sometimes abbreviate this as \sum_z , leaving **Z** implicit. We just used the rule as

$$P(Cavity) = \sum_{z \in \{Catch, Toothache\}} P(Cavity, z) . \quad (13.7)$$

A variant of this rule involves conditional probabilities instead of joint probabilities, using the product rule:

$$P(Y) = \sum_z P(Y | z) P(z) . \quad (13.8)$$

This rule is called **conditioning**. Marginalization and conditioning turn out to be useful rules for all kinds of derivations involving probability expressions.

In most cases, we are interested in computing *conditional* probabilities of some variables, given evidence about others. Conditional probabilities can be found by first using

⁴ So called because of a common practice among actuaries of writing the sums of observed frequencies in the margins of insurance tables.

MARGINAL
PROBABILITY

MARGINALIZATION

CONDITIONING

Equation (13.3) 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:

$$\begin{aligned} P(\text{cavity} \mid \text{toothache}) &= \frac{P(\text{cavity} \wedge \text{toothache})}{P(\text{toothache})} \\ &= \frac{0.108 + 0.012}{0.108 + 0.012 + 0.016 + 0.064} = 0.6 . \end{aligned}$$

Just to check, we can also compute the probability that there is no cavity, given a toothache:

$$\begin{aligned} P(\neg \text{cavity} \mid \text{toothache}) &= \frac{P(\neg \text{cavity} \wedge \text{toothache})}{P(\text{toothache})} \\ &= \frac{0.016 + 0.064}{0.108 + 0.012 + 0.016 + 0.064} = 0.4 . \end{aligned}$$

The two values sum to 1.0, as they should. Notice that in these two calculations the term $1/P(\text{toothache})$ remains constant, no matter which value of *Cavity* we calculate. In fact, it can be viewed as a **normalization** constant for the distribution $\mathbf{P}(\text{Cavity} \mid \text{toothache})$, ensuring that it adds up to 1. Throughout the chapters dealing with probability, we use α to denote such constants. With this notation, we can write the two preceding equations in one:

$$\begin{aligned} \mathbf{P}(\text{Cavity} \mid \text{toothache}) &= \alpha \mathbf{P}(\text{Cavity}, \text{toothache}) \\ &= \alpha [\mathbf{P}(\text{Cavity}, \text{toothache}, \text{catch}) + \mathbf{P}(\text{Cavity}, \text{toothache}, \neg \text{catch})] \\ &= \alpha [\langle 0.108, 0.016 \rangle + \langle 0.012, 0.064 \rangle] = \alpha \langle 0.12, 0.08 \rangle = \langle 0.6, 0.4 \rangle . \end{aligned}$$

In other words, we can calculate $\mathbf{P}(\text{Cavity} \mid \text{toothache})$ even if we don't know the value of $P(\text{toothache})$! We temporarily forget about the factor $1/P(\text{toothache})$ and add up the values for *cavity* and $\neg \text{cavity}$, getting 0.12 and 0.08. Those are the correct relative proportions, but they don't sum to 1, so we normalize them by dividing each one by $0.12 + 0.08$, getting the true probabilities of 0.6 and 0.4. Normalization turns out to be a useful shortcut in many probability calculations, both to make the computation easier and to allow us to proceed when some probability assessment (such as $P(\text{toothache})$) is not available.

From the example, we can extract a general inference procedure. We begin with the case in which the query involves a single variable, X (*Cavity* in the example). Let \mathbf{E} be the list of evidence variables (just *Toothache* in the example), let \mathbf{e} be the list of observed values for them, and let \mathbf{Y} be the remaining unobserved variables (just *Catch* in the example). The query is $\mathbf{P}(X \mid \mathbf{e})$ and can be evaluated as

$$\mathbf{P}(X \mid \mathbf{e}) = \alpha \mathbf{P}(X, \mathbf{e}) = \alpha \sum_{\mathbf{y}} \mathbf{P}(X, \mathbf{e}, \mathbf{y}) , \quad (13.9)$$

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.

Given the full joint distribution to work with, Equation (13.9) can answer 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 we could easily have $n > 100$, making $O(2^n)$ impractical. The full joint distribution in tabular form is just not a practical tool for building reasoning systems. Instead, it should be viewed as the theoretical foundation on which more effective approaches may be built, just as truth tables formed a theoretical foundation for more practical algorithms like DPLL. The remainder of this chapter introduces some of the basic ideas required in preparation for the development of realistic systems in Chapter 14.

13.4 INDEPENDENCE

Let us expand the full joint distribution in Figure 13.3 by adding a fourth variable, *Weather*. The full joint distribution then becomes $\mathbf{P}(\textit{Toothache}, \textit{Catch}, \textit{Cavity}, \textit{Weather})$, which has $2 \times 2 \times 2 \times 4 = 32$ entries. It contains four “editions” of the table shown in Figure 13.3, one for each kind of weather. What relationship do these editions have to each other and to the original three-variable table? For example, how are $P(\textit{toothache}, \textit{catch}, \textit{cavity}, \textit{cloudy})$ and $P(\textit{toothache}, \textit{catch}, \textit{cavity})$ related? We can use the product rule:

$$\begin{aligned} P(\textit{toothache}, \textit{catch}, \textit{cavity}, \textit{cloudy}) \\ = P(\textit{cloudy} \mid \textit{toothache}, \textit{catch}, \textit{cavity}) P(\textit{toothache}, \textit{catch}, \textit{cavity}) . \end{aligned}$$

Now, unless one is in the deity business, one should not imagine that one’s dental problems influence the weather. And for indoor dentistry, at least, it seems safe to say that the weather does not influence the dental variables. Therefore, the following assertion seems reasonable:

$$P(\textit{cloudy} \mid \textit{toothache}, \textit{catch}, \textit{cavity}) = P(\textit{cloudy}) . \quad (13.10)$$

From this, we can deduce

$$P(\textit{toothache}, \textit{catch}, \textit{cavity}, \textit{cloudy}) = P(\textit{cloudy}) P(\textit{toothache}, \textit{catch}, \textit{cavity}) .$$

A similar equation exists for *every entry* in $\mathbf{P}(\textit{Toothache}, \textit{Catch}, \textit{Cavity}, \textit{Weather})$. In fact, we can write the general equation

$$\mathbf{P}(\textit{Toothache}, \textit{Catch}, \textit{Cavity}, \textit{Weather}) = \mathbf{P}(\textit{Toothache}, \textit{Catch}, \textit{Cavity}) \mathbf{P}(\textit{Weather}) .$$

Thus, the 32-element table for four variables can be constructed from one 8-element table and one 4-element table. This decomposition is illustrated schematically in Figure 13.4(a).

INDEPENDENCE

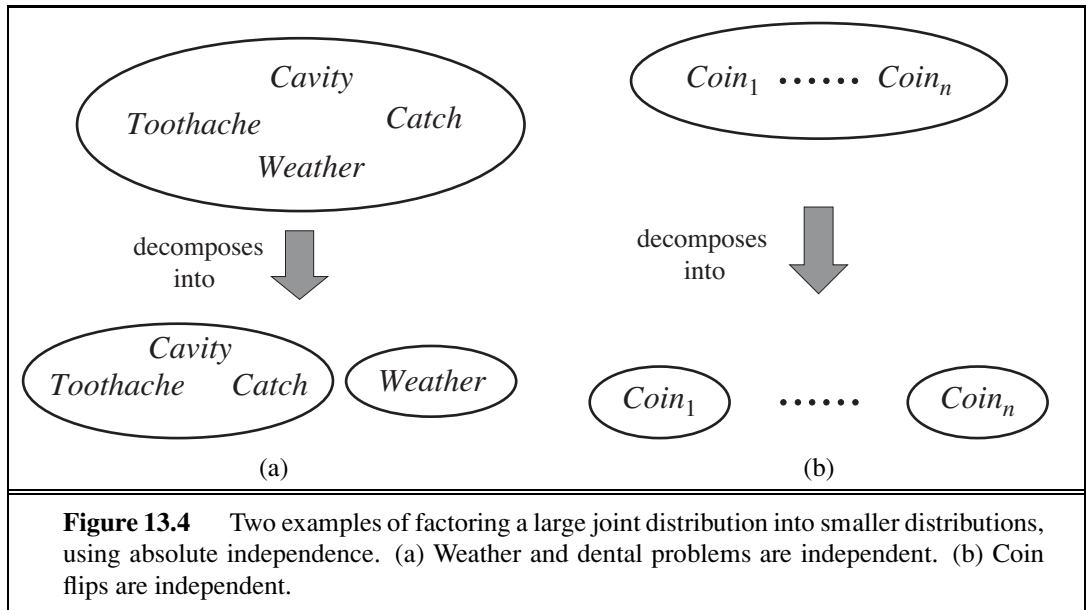
The property we used in Equation (13.10) is called **independence** (also **marginal independence** and **absolute independence**). In particular, the weather is independent of one’s dental problems. Independence between propositions a and b can be written as

$$P(a \mid b) = P(a) \quad \text{or} \quad P(b \mid a) = P(b) \quad \text{or} \quad P(a \wedge b) = P(a)P(b) . \quad (13.11)$$

All these forms are equivalent (Exercise 13.12). Independence between variables X and Y can be written as follows (again, these are all equivalent):

$$\mathbf{P}(X \mid Y) = \mathbf{P}(X) \quad \text{or} \quad \mathbf{P}(Y \mid X) = \mathbf{P}(Y) \quad \text{or} \quad \mathbf{P}(X, Y) = \mathbf{P}(X)\mathbf{P}(Y) .$$

Independence assertions are usually based on knowledge of the domain. As the toothache–weather example illustrates, 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 distribution can be *factored* into separate joint distributions on those subsets. For example, the full joint distribution on the outcome of n independent coin flips, $\mathbf{P}(C_1, \dots, C_n)$, has 2^n entries, but it 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 need more subtle methods than the straightforward concept of independence.

13.5 BAYES' RULE AND ITS USE

On page 486, we defined the **product rule**. It can actually be written in two forms:

$$P(a \wedge b) = P(a | b)P(b) \quad \text{and} \quad P(a \wedge b) = P(b | a)P(a).$$

Equating the two right-hand sides and dividing by $P(a)$, we get

$$P(b | a) = \frac{P(a | b)P(b)}{P(a)}. \quad (13.12)$$

BAYES' RULE

This equation is known as **Bayes' rule** (also Bayes' law or Bayes' theorem). This simple equation underlies most modern AI systems for probabilistic inference.

The more general case of Bayes' rule for multivalued variables can be written in the **P** notation as follows:

$$\mathbf{P}(Y | X) = \frac{\mathbf{P}(X | Y)\mathbf{P}(Y)}{\mathbf{P}(X)} ,$$

As before, 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**:

$$\mathbf{P}(Y | X, \mathbf{e}) = \frac{\mathbf{P}(X | Y, \mathbf{e})\mathbf{P}(Y | \mathbf{e})}{\mathbf{P}(X | \mathbf{e})} . \quad (13.13)$$

13.5.1 Applying Bayes' rule: The simple case

On the surface, Bayes' rule does not seem very useful. It allows us to compute the single term $P(b | a)$ in terms of three terms: $P(a | b)$, $P(b)$, and $P(a)$. That seems like two steps backwards, but 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. Often, we perceive as evidence the *effect* of some unknown *cause* and we would like to determine that cause. In that case, Bayes' rule becomes

$$P(\text{cause} | \text{effect}) = \frac{P(\text{effect} | \text{cause})P(\text{cause})}{P(\text{effect})} .$$

CAUSAL
DIAGNOSTIC

The conditional probability $P(\text{effect} | \text{cause})$ quantifies the relationship in the **causal** direction, whereas $P(\text{cause} | \text{effect})$ describes the **diagnostic** direction. In a task such as medical diagnosis, we often have conditional probabilities on causal relationships (that is, the doctor knows $P(\text{symptoms} | \text{disease})$) and want to derive a diagnosis, $P(\text{disease} | \text{symptoms})$. For example, a doctor knows that the disease meningitis causes the patient to have a stiff neck, say, 70% 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%. Letting s be the proposition that the patient has a stiff neck and m be the proposition that the patient has meningitis, we have

$$\begin{aligned} P(s | m) &= 0.7 \\ P(m) &= 1/50000 \\ P(s) &= 0.01 \\ P(m | s) &= \frac{P(s | m)P(m)}{P(s)} = \frac{0.7 \times 1/50000}{0.01} = 0.0014 . \end{aligned} \quad (13.14)$$

That is, we expect less than 1 in 700 patients with a stiff neck to have meningitis. Notice that even though a stiff neck is quite strongly indicated by meningitis (with probability 0.7), the probability of meningitis in the patient remains small. This is because the prior probability of stiff necks is much higher than that of meningitis.

Section 13.3 illustrated a process by which one can avoid assessing the prior 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, 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

$$\mathbf{P}(Y | X) = \alpha \mathbf{P}(X | Y) \mathbf{P}(Y) , \quad (13.15)$$

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 important, 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.



13.5.2 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 $P(\text{effect} | \text{cause})$. 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), we can read off the answer:

$$\mathbf{P}(\text{Cavity} | \text{toothache} \wedge \text{catch}) = \alpha \langle 0.108, 0.016 \rangle \approx \langle 0.871, 0.129 \rangle .$$

We know, however, that such an approach does not scale up to larger numbers of variables. We can try using Bayes' rule to reformulate the problem:

$$\begin{aligned} & \mathbf{P}(\text{Cavity} | \text{toothache} \wedge \text{catch}) \\ &= \alpha \mathbf{P}(\text{toothache} \wedge \text{catch} | \text{Cavity}) \mathbf{P}(\text{Cavity}) . \end{aligned} \quad (13.16)$$

For this reformulation to work, we need to know the conditional probabilities of the conjunction $\text{toothache} \wedge \text{catch}$ for each value of Cavity . That might be feasible for just two evidence variables, but again it does 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 13.4 provides a clue, but needs refining. It would be nice if *Toothache* and *Catch* were independent, but they are not: if the probe catches in the tooth, then it is likely that the tooth has a cavity and that the cavity 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.⁵ Mathematically, this property is written as

$$\mathbf{P}(\text{toothache} \wedge \text{catch} \mid \text{Cavity}) = \mathbf{P}(\text{toothache} \mid \text{Cavity})\mathbf{P}(\text{catch} \mid \text{Cavity}) . \quad (13.17)$$

This equation expresses the **conditional independence** of *toothache* and *catch* given *Cavity*. We can plug it into Equation (13.16) to obtain the probability of a cavity:

$$\begin{aligned} \mathbf{P}(\text{Cavity} \mid \text{toothache} \wedge \text{catch}) \\ = \alpha \mathbf{P}(\text{toothache} \mid \text{Cavity}) \mathbf{P}(\text{catch} \mid \text{Cavity}) \mathbf{P}(\text{Cavity}) . \end{aligned} \quad (13.18)$$

Now the information requirements are the same as for inference, using each piece of evidence separately: the prior probability $\mathbf{P}(\text{Cavity})$ for the query variable and the conditional probability of each effect, given its cause.

The general definition of **conditional independence** of two variables X and Y , given a third variable Z , is

$$\mathbf{P}(X, Y \mid Z) = \mathbf{P}(X \mid Z)\mathbf{P}(Y \mid Z) .$$

In the dentist domain, for example, it seems reasonable to assert conditional independence of the variables *Toothache* and *Catch*, given *Cavity*:

$$\mathbf{P}(\text{Toothache}, \text{Catch} \mid \text{Cavity}) = \mathbf{P}(\text{Toothache} \mid \text{Cavity})\mathbf{P}(\text{Catch} \mid \text{Cavity}) . \quad (13.19)$$

Notice that this assertion is somewhat stronger than Equation (13.17), which asserts independence only for specific values of *Toothache* and *Catch*. As with absolute independence in Equation (13.11), the equivalent forms

$$\mathbf{P}(X \mid Y, Z) = \mathbf{P}(X \mid Z) \quad \text{and} \quad \mathbf{P}(Y \mid X, Z) = \mathbf{P}(Y \mid Z)$$

can also be used (see Exercise 13.17). Section 13.4 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 (13.19), we can derive a decomposition as follows:

$$\begin{aligned} \mathbf{P}(\text{Toothache}, \text{Catch}, \text{Cavity}) \\ = \mathbf{P}(\text{Toothache}, \text{Catch} \mid \text{Cavity})\mathbf{P}(\text{Cavity}) \quad (\text{product rule}) \\ = \mathbf{P}(\text{Toothache} \mid \text{Cavity})\mathbf{P}(\text{Catch} \mid \text{Cavity})\mathbf{P}(\text{Cavity}) \quad (\text{using 13.19}). \end{aligned}$$

(The reader can easily check that this equation does in fact hold in Figure 13.3.) In this way, the original large table is decomposed into three smaller tables. The original table has seven

CONDITIONAL
INDEPENDENCE

⁵ We assume that the patient and dentist are distinct individuals.



SEPARATION

independent numbers ($2^3 = 8$ entries in the table, but they must sum to 1, so 7 are independent). The smaller tables contain five independent numbers (for a conditional probability distributions such as $\mathbf{P}(T|C)$ there are two rows of two numbers, and each row sums to 1, so that's two independent numbers; for a prior distribution like $\mathbf{P}(C)$ there is only one independent number). Going from seven to five might not seem like 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 $O(2^n)$. That means that *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 through conditional independence is one of the most important developments in the recent history of AI.

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

$$\mathbf{P}(\text{Cause}, \text{Effect}_1, \dots, \text{Effect}_n) = \mathbf{P}(\text{Cause}) \prod_i \mathbf{P}(\text{Effect}_i | \text{Cause}).$$

NAIVE BAYES

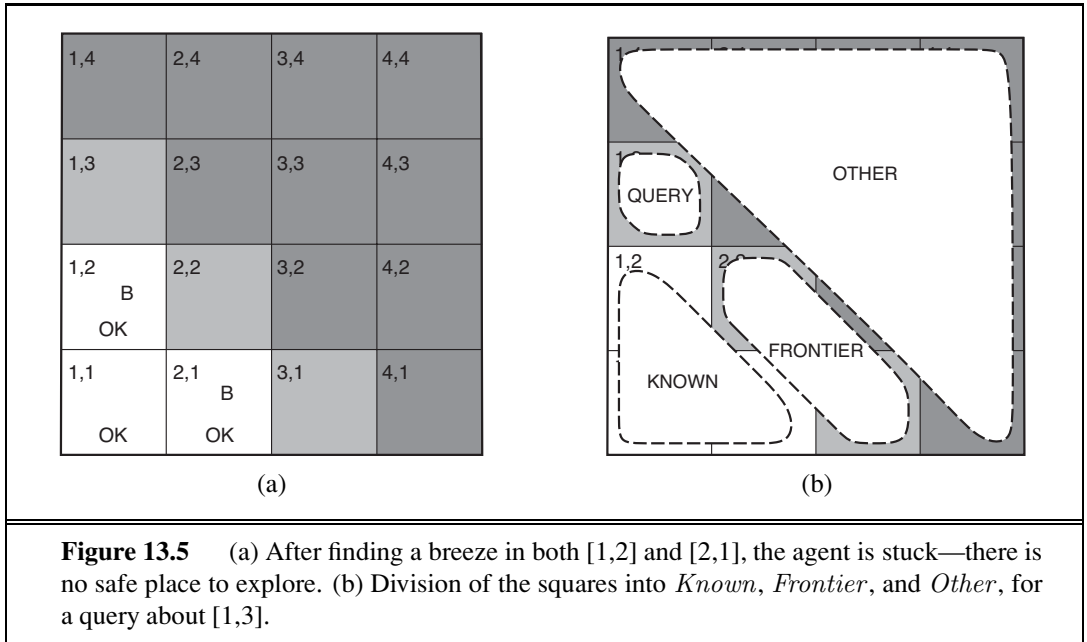
Such a probability distribution is called a **naive Bayes** model—“naive” because it is often used (as a simplifying assumption) in cases where the “effect” variables are *not* actually conditionally independent given the cause variable. (The naive Bayes model is sometimes called a **Bayesian classifier**, 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 conditional independence assumption is not true. Chapter 20 describes methods for learning naive Bayes distributions from observations.

13.6 THE WUMPUS WORLD REVISITED

We can combine 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 information about the world. For example, Figure 13.5 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 have to choose randomly. We will see that a probabilistic agent can do much better than the logical agent.

Our aim is to calculate the probability that each of the three squares contains a pit. (For this example we 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, $\mathbf{P}(P_{1,1}, \dots, P_{4,4}, B_{1,1}, B_{1,2}, B_{2,1})$. Applying the product rule, we have

$$\mathbf{P}(P_{1,1}, \dots, P_{4,4}, B_{1,1}, B_{1,2}, B_{2,1}) = \mathbf{P}(B_{1,1}, B_{1,2}, B_{2,1} \mid P_{1,1}, \dots, P_{4,4}) \mathbf{P}(P_{1,1}, \dots, P_{4,4}).$$

This decomposition makes it easy to see what the joint probability values should be. The first term is the conditional probability distribution of a breeze configuration, given a pit configuration; its values are 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,

$$\mathbf{P}(P_{1,1}, \dots, P_{4,4}) = \prod_{i,j=1,1}^{4,4} \mathbf{P}(P_{i,j}). \quad (13.20)$$

For a particular configuration with exactly n pits, $\mathbf{P}(P_{1,1}, \dots, P_{4,4}) = 0.2^n \times 0.8^{16-n}$.

In the situation in Figure 13.5(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 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 $\mathbf{P}(P_{1,3} \mid known, b)$: 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 of Equation (13.9), namely, summing over entries from the full joint distribution. Let $Unknown$ be the set of $P_{i,j}$ vari-

ables for squares other than the *Known* squares and the query square [1,3]. Then, by Equation (13.9), we have

$$\mathbf{P}(P_{1,3} \mid \textit{known}, b) = \alpha \sum_{\textit{unknown}} \mathbf{P}(P_{1,3}, \textit{unknown}, \textit{known}, b) .$$

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.

Surely, one might ask, aren't the other squares irrelevant? How could [4,4] affect whether [1,3] has a pit? Indeed, this intuition is correct. Let *Frontier* be the pit 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 pit variables for the other unknown squares; in this case, there are 10 other squares, as shown in Figure 13.5(b). The key insight is that the observed breezes are *conditionally independent* of the other variables, given the known, frontier, and query variables. 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 apply conditional independence:

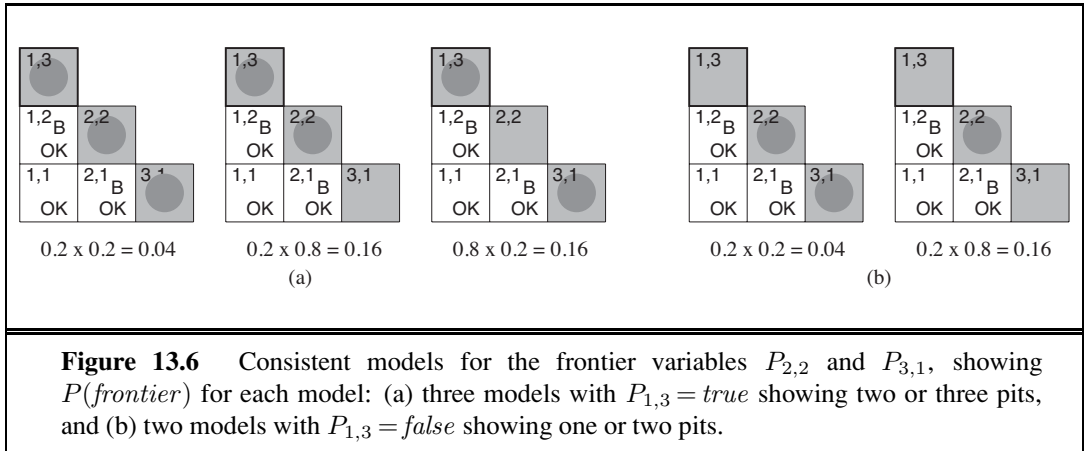
$$\begin{aligned} \mathbf{P}(P_{1,3} \mid \textit{known}, b) &= \alpha \sum_{\textit{unknown}} \mathbf{P}(P_{1,3}, \textit{known}, b, \textit{unknown}) \quad (\text{by Equation (13.9)}) \\ &= \alpha \sum_{\textit{unknown}} \mathbf{P}(b \mid P_{1,3}, \textit{known}, \textit{unknown}) \mathbf{P}(P_{1,3}, \textit{known}, \textit{unknown}) \\ &\quad (\text{by the product rule}) \\ &= \alpha \sum_{\textit{frontier}} \sum_{\textit{other}} \mathbf{P}(b \mid \textit{known}, P_{1,3}, \textit{frontier}, \textit{other}) \mathbf{P}(P_{1,3}, \textit{known}, \textit{frontier}, \textit{other}) \\ &= \alpha \sum_{\textit{frontier}} \sum_{\textit{other}} \mathbf{P}(b \mid \textit{known}, P_{1,3}, \textit{frontier}) \mathbf{P}(P_{1,3}, \textit{known}, \textit{frontier}, \textit{other}) , \end{aligned}$$

where the final step uses conditional independence: *b* is independent of *other* given *known*, *P*_{1,3}, and *frontier*. Now, the first term in this expression does not depend on the *Other* variables, so we can move the summation inward:

$$\begin{aligned} \mathbf{P}(P_{1,3} \mid \textit{known}, b) &= \alpha \sum_{\textit{frontier}} \mathbf{P}(b \mid \textit{known}, P_{1,3}, \textit{frontier}) \sum_{\textit{other}} \mathbf{P}(P_{1,3}, \textit{known}, \textit{frontier}, \textit{other}) . \end{aligned}$$

By independence, as in Equation (13.20), the prior term can be factored, and then the terms can be reordered:

$$\begin{aligned} \mathbf{P}(P_{1,3} \mid \textit{known}, b) &= \alpha \sum_{\textit{frontier}} \mathbf{P}(b \mid \textit{known}, P_{1,3}, \textit{frontier}) \sum_{\textit{other}} \mathbf{P}(P_{1,3}) P(\textit{known}) P(\textit{frontier}) P(\textit{other}) \\ &= \alpha P(\textit{known}) \mathbf{P}(P_{1,3}) \sum_{\textit{frontier}} \mathbf{P}(b \mid \textit{known}, P_{1,3}, \textit{frontier}) P(\textit{frontier}) \sum_{\textit{other}} P(\textit{other}) \\ &= \alpha' \mathbf{P}(P_{1,3}) \sum_{\textit{frontier}} \mathbf{P}(b \mid \textit{known}, P_{1,3}, \textit{frontier}) P(\textit{frontier}) , \end{aligned}$$



where the last step folds $P(\text{known})$ into the normalizing constant and uses the fact that $\sum_{\text{other}} P(\text{other})$ equals 1.

Now, there are just four terms in the summation over the frontier 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 $\mathbf{P}(b \mid \text{known}, P_{1,3}, \text{frontier})$ is 1 when the frontier 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 frontier variables that are consistent with the known facts. (Compare with the enumeration over models in Figure 7.5 on page 241.) The models and their associated prior probabilities— $P(\text{frontier})$ —are shown in Figure 13.6. We have

$$\mathbf{P}(P_{1,3} \mid \text{known}, b) = \alpha' \langle 0.2(0.04 + 0.16 + 0.16), 0.8(0.04 + 0.16) \rangle \approx \langle 0.31, 0.69 \rangle.$$

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]! Note that our logical agent from Chapter 7 did not know that [2,2] was worse than the other squares. Logic can tell us that it is unknown whether there is a pit in [2, 2], but we need probability to tell us how likely it is.

What this section has shown is that even seemingly complicated problems can be formulated precisely in probability theory and solved with 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 develop formal representations for such relationships as well as algorithms that operate on those representations to perform probabilistic inference efficiently.

13.7 SUMMARY

This chapter has suggested probability theory as a suitable foundation for uncertain reasoning and provided a gentle introduction to its use.

- Uncertainty arises because of both laziness and ignorance. It is inescapable in complex, nondeterministic, or partially observable environments.
- Probabilities express the agent's inability to reach a definite decision regarding the truth of a sentence. Probabilities summarize the agent's beliefs relative to the evidence.
- Decision theory combines the agent's beliefs and desires, defining the best action as the one that maximizes expected utility.
- Basic probability statements include **prior probabilities** and **conditional probabilities** over simple and complex propositions.
- The axioms of probability constrain the possible assignments of probabilities to propositions. An agent that violates the axioms must behave irrationally in some cases.
- 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, but when it is available it can be used to answer queries simply by adding up entries for the possible worlds corresponding to the query propositions.
- **Absolute independence** between subsets of random variables allows the full joint distribution to be factored into smaller joint distributions, greatly reducing its complexity. Absolute independence 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 runs 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, thereby improving on the decisions of a purely logical agent. Conditional independence makes these calculations tractable.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Probability theory was invented as a way of analyzing games of chance. In about 850 A.D. the Indian mathematician Mahaviracarya described how to arrange a set of bets that can't lose (what we now call a Dutch book). In Europe, the first significant systematic analyses were produced by Girolamo Cardano around 1565, although publication was posthumous (1663). By that time, probability had been established as a mathematical discipline due to a series of

results established in a famous correspondence between Blaise Pascal and Pierre de Fermat in 1654. As with probability itself, the results were initially motivated by gambling problems (see Exercise 13.9). The first published textbook on probability was *De Ratiociniis in Ludo Aleae* (Huygens, 1657). The “laziness and ignorance” view of uncertainty was described by John Arbuthnot in the preface of his translation of Huygens (Arbuthnot, 1692): “It is impossible for a Die, with such determin’d force and direction, not to fall on such determin’d side, only I don’t know the force and direction which makes it fall on such determin’d side, and therefore I call it Chance, which is nothing but the want of art...”

Laplace (1816) gave an exceptionally accurate and modern overview of probability; he was the first to use the example “take two urns, A and B, the first containing four white and two black balls, ...” The Rev. Thomas Bayes (1702–1761) introduced the rule for reasoning about conditional probabilities that was named after him (Bayes, 1763). Bayes only considered the case of uniform priors; it was Laplace who independently developed the general case. 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**. The principle is often attributed to Laplace, but he never isolated the principle explicitly. George Boole and John Venn both referred to it as the **principle of insufficient reason**; the modern name is due to 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), Richard Jeffrey (1983), and E. T. Jaynes (2003) 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 *e*.

CONFIRMATION

INDUCTIVE LOGIC

Carnap attempted to go further than Leibniz or Laplace by making this notion of degree of **confirmation** mathematically precise, as a logical relation between a and 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 by adversarial arguments that some fundamental difficulties would prevent a strict extension to languages capable of expressing arithmetic.

Cox's theorem (1946) shows that any system for uncertain reasoning that meets his set of assumptions is equivalent to probability theory. This gave renewed confidence to those who already favored probability, but others were not convinced, pointing to the assumptions (primarily that belief must be represented by a single number, and thus the belief in $\neg p$ must be a function of the belief in p). Halpern (1999) describes the assumptions and shows some gaps in Cox's original formulation. Horn (2003) shows how to patch up the difficulties. Jaynes (2003) has a similar argument that is easier to read.

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 straightforward reference-class method. Kyburg and Teng (2006) contrast probabilistic inference with nonmonotonic logic.

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 by 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). Lucas *et al.* (2004) gives an overview. 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 model 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 information retrieval, beginning with the work of Maron (1961). The probabilistic foundations of this technique, described further in Exercise 13.22, 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 Bertsekas and Tsitsiklis (2008) and Grinstead and Snell (1997). DeGroot and Schervish (2001) offer a combined introduction to probability and statistics from a Bayesian standpoint. 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 For each of the following statements, either prove it is true or give a counterexample.

- a. If $P(a | b, c) = P(b | a, c)$, then $P(a | c) = P(b | c)$
- b. If $P(a | b, c) = P(a)$, then $P(b | c) = P(b)$
- c. If $P(a | b) = P(a)$, then $P(a | b, c) = P(a | c)$

13.4 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?)

13.5 This question deals with the properties of possible worlds, defined on page 488 as assignments to all random variables. We will work with propositions that correspond to exactly one possible world because they pin down the assignments of all the variables. In probability theory, such propositions are called **atomic events**. For example, with Boolean variables X_1, X_2, X_3 , the proposition $x_1 \wedge \neg x_2 \wedge \neg x_3$ fixes the assignment of the variables; in the language of propositional logic, we would say it has exactly one model.

- a. Prove, for the case of n Boolean variables, that any two distinct atomic events are mutually exclusive; that is, their conjunction is equivalent to *false*.
- b. Prove that the disjunction of all possible atomic events is logically equivalent to *true*.
- c. Prove that any proposition is logically equivalent to the disjunction of the atomic events that entail its truth.

13.6 Prove Equation (13.4) from Equations (13.1) and (13.2).

13.7 Consider the set of all possible five-card poker hands dealt fairly from a standard deck of fifty-two cards.

- How many atomic events are there in the joint probability distribution (i.e., how many five-card hands are there)?
- What is the probability of each atomic event?
- What is the probability of being dealt a royal straight flush? Four of a kind?

13.8 Given the full joint distribution shown in Figure 13.3, calculate the following:

- $P(\textit{toothache})$.
- $P(\textit{Cavity})$.
- $P(\textit{Toothache} \mid \textit{cavity})$.
- $P(\textit{Cavity} \mid \textit{toothache} \vee \textit{catch})$.

13.9 In his letter of August 24, 1654, Pascal was trying to show how a pot of money should be allocated when a gambling game must end prematurely. Imagine a game where each turn consists of the roll of a die, player *E* gets a point when the die is even, and player *O* gets a point when the die is odd. The first player to get 7 points wins the pot. Suppose the game is interrupted with *E* leading 4–2. How should the money be fairly split in this case? What is the general formula? (Fermat and Pascal made several errors before solving the problem, but you should be able to get it right the first time.)

13.10 Deciding to put probability theory to good use, we encounter a slot machine with three independent wheels, each producing one of the four symbols BAR, BELL, LEMON, or CHERRY with equal probability. The slot machine has the following payout scheme for a bet of 1 coin (where “?” denotes that we don’t care what comes up for that wheel):

BAR/BAR/BAR pays 20 coins
 BELL/BELL/BELL pays 15 coins
 LEMON/LEMON/LEMON pays 5 coins
 CHERRY/CHERRY/CHERRY pays 3 coins
 CHERRY/CHERRY/? pays 2 coins
 CHERRY/?/? pays 1 coin

- Compute the expected “payback” percentage of the machine. In other words, for each coin played, what is the expected coin return?
- Compute the probability that playing the slot machine once will result in a win.
- Estimate the mean and median number of plays you can expect to make until you go broke, if you start with 10 coins. You can run a simulation to estimate this, rather than trying to compute an exact answer.

13.11 We wish to transmit an n -bit message to a receiving agent. The bits in the message are independently corrupted (flipped) during transmission with ϵ probability each. With an extra parity bit sent along with the original information, a message can be corrected by the receiver

if at most one bit in the entire message (including the parity bit) has been corrupted. Suppose we want to ensure that the correct message is received with probability at least $1 - \delta$. What is the maximum feasible value of n ? Calculate this value for the case $\epsilon = 0.001$, $\delta = 0.01$.

13.12 Show that the three forms of independence in Equation (13.11) are equivalent.

13.13 Consider two medical tests, A and B, for a virus. Test A is 95% effective at recognizing the virus when it is present, but has a 10% false positive rate (indicating that the virus is present, when it is not). Test B is 90% effective at recognizing the virus, but has a 5% false positive rate. The two tests use independent methods of identifying the virus. The virus is carried by 1% of all people. Say that a person is tested for the virus using only one of the tests, and that test comes back positive for carrying the virus. Which test returning positive is more indicative of someone really carrying the virus? Justify your answer mathematically.

13.14 Suppose you are given a coin that lands *heads* with probability x and *tails* with probability $1 - x$. Are the outcomes of successive flips of the coin independent of each other given that you know the value of x ? Are the outcomes of successive flips of the coin independent of each other if you do *not* know the value of x ? Justify your answer.

13.15 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?

13.16 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 (13.13).

13.17 Show that the statement of conditional independence

$$\mathbf{P}(X, Y | Z) = \mathbf{P}(X | Z)\mathbf{P}(Y | Z)$$

is equivalent to each of the statements

$$\mathbf{P}(X | Y, Z) = \mathbf{P}(X | Z) \quad \text{and} \quad \mathbf{P}(Y | X, Z) = \mathbf{P}(Y | Z) .$$

13.18 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 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?

13.19 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, Equation (13.14)). Now normalize these values so that they add to 1.

13.20 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?

13.21 (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.

- a. 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.)
- b. What if you know that 9 out of 10 Athenian taxis are green?

13.22 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 conditional independence assumption reasonable? Discuss.

13.23 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 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 $\mathbf{P}(P_{1,1}, \dots, P_{4,4})$ now? Redo the calculation for the probabilities of pits in $[1,3]$ and $[2,2]$.

13.24 Redo the probability calculation for pits in $[1,3]$ and $[2,2]$, assuming that each square contains a pit with probability 0.01, independent of the other squares. What can you say about the relative performance of a logical versus a probabilistic agent in this case?

13.25 Implement a hybrid probabilistic agent for the wumpus world, based on the hybrid agent in Figure 7.20 and the probabilistic inference procedure outlined in this chapter.



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 introduced the basic elements of probability theory and noted 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 possible worlds one by one is unnatural and tedious.

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. Bayesian networks can represent essentially *any* full joint probability distribution and in many cases can do so very concisely.

BAYESIAN NETWORK

¹ This is the most common name, but there are many synonyms, 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** is 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. Each node corresponds to a random variable, which 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 . The graph has no directed cycles (and hence is a directed acyclic graph, or DAG).
3. Each node X_i has a conditional probability distribution $\mathbf{P}(X_i \mid \text{Parents}(X_i))$ that quantifies the effect of the parents on the node.

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 is typically that X has a *direct influence* on Y , which suggests that causes should be parents of effects. 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 nearly always calls when he hears the alarm, but sometimes confuses the telephone ringing with

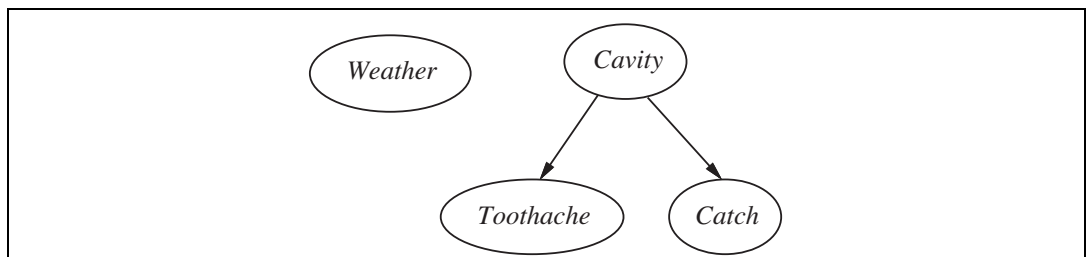
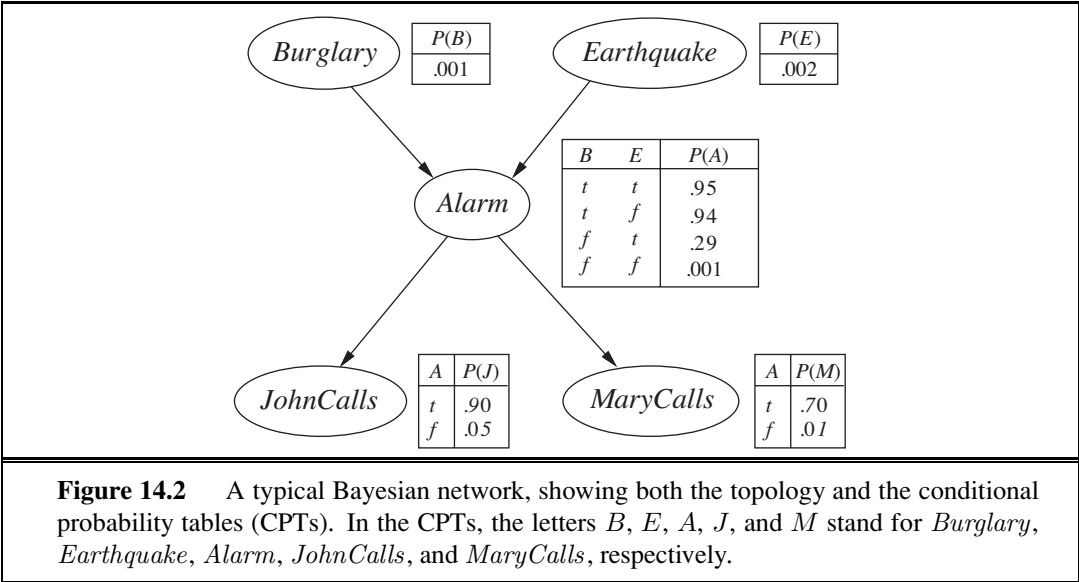


Figure 14.1 A simple Bayesian network in which *Weather* is independent of the other three variables and *Toothache* and *Catch* are conditionally independent, given *Cavity*.



the alarm and calls then, too. Mary, on the other hand, likes rather loud music and often misses the alarm altogether. Given the evidence of who has or has not called, we would like to estimate the probability of a burglary.

A Bayesian network for this domain appears in Figure 14.2. The network structure 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 burglaries directly, they do not notice minor earthquakes, and they do not confer before calling.

The conditional distributions in Figure 14.2 are 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 combination of values for the parent nodes—a miniature possible world, 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.

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*

CONDITIONAL
PROBABILITY TABLE

CONDITIONING CASE

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.

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.

14.2.1 Representing the full joint distribution

Viewed as a piece of “syntax,” a Bayesian network is a directed acyclic graph with some numeric parameters attached to each node. One way to define what the network means—its semantics—is to define the way in which it represents a specific joint distribution over all the variables. To do this, we first need to retract (temporarily) what we said earlier about the parameters associated with each node. We said that those parameters correspond to conditional probabilities $\mathbf{P}(X_i | \text{Parents}(X_i))$; this is a true statement, but until we assign semantics to the network as a whole, we should think of them just as numbers $\theta(X_i | \text{Parents}(X_i))$.

A generic entry in the joint distribution is the probability of a conjunction of particular assignments to each variable, such as $P(X_1 = x_1 \wedge \dots \wedge X_n = x_n)$. We use the notation $P(x_1, \dots, x_n)$ as an abbreviation for this. The value of this entry is given by the formula

$$P(x_1, \dots, x_n) = \prod_{i=1}^n \theta(x_i | \text{parents}(X_i)), \quad (14.1)$$

where $\text{parents}(X_i)$ denotes the values of $\text{Parents}(X_i)$ that appear in x_1, \dots, x_n . 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.

From this definition, it is easy to prove that the parameters $\theta(X_i | \text{Parents}(X_i))$ are exactly the conditional probabilities $\mathbf{P}(X_i | \text{Parents}(X_i))$ implied by the joint distribution (see Exercise 14.2). Hence, we can rewrite Equation (14.1) as

$$P(x_1, \dots, x_n) = \prod_{i=1}^n P(x_i | \text{parents}(X_i)). \quad (14.2)$$

In other words, the tables we have been calling conditional probability tables really *are* conditional probability tables according to the semantics defined in Equation (14.1).

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 multiply entries

from the joint distribution (using single-letter names for the variables):

$$\begin{aligned} P(j, m, a, \neg b, \neg e) &= P(j | a)P(m | a)P(a | \neg b \wedge \neg e)P(\neg b)P(\neg e) \\ &= 0.90 \times 0.70 \times 0.001 \times 0.999 \times 0.998 = 0.000628 . \end{aligned}$$

Section 13.3 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 (14.2) defines what a given Bayesian network means. The next step is to 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 (14.2) 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 entries in the joint distribution in terms of conditional probability, using the product rule (see page 486):

$$P(x_1, \dots, x_n) = P(x_n | x_{n-1}, \dots, x_1)P(x_{n-1}, \dots, x_1) .$$

Then we repeat the process, reducing each conjunctive probability to a conditional probability and a smaller conjunction. We end up with one big product:

$$\begin{aligned} P(x_1, \dots, x_n) &= P(x_n | x_{n-1}, \dots, x_1)P(x_{n-1} | x_{n-2}, \dots, x_1) \cdots P(x_2 | x_1)P(x_1) \\ &= \prod_{i=1}^n P(x_i | x_{i-1}, \dots, x_1) . \end{aligned}$$

CHAIN RULE

This identity is called the **chain rule**. It holds for any set of random variables. Comparing it with Equation (14.2), we see that the specification of the joint distribution is equivalent to the general assertion that, for every variable X_i in the network,

$$\mathbf{P}(X_i | X_{i-1}, \dots, X_1) = \mathbf{P}(X_i | Parents(X_i)) , \quad (14.3)$$

provided that $Parents(X_i) \subseteq \{X_{i-1}, \dots, X_1\}$. This last condition is satisfied by numbering the nodes in a way that is consistent with the partial order implicit in the graph structure.

What Equation (14.3) says is that the Bayesian network is a correct representation of the domain only if each node is conditionally independent of its other predecessors in the node ordering, given its parents. We can satisfy this condition with this methodology:

1. *Nodes*: First determine the set of variables that are required to model the domain. Now order them, $\{X_1, \dots, X_n\}$. Any order will work, but the resulting network will be more compact if the variables are ordered such that causes precede effects.
2. *Links*: For $i = 1$ to n do:
 - Choose, from X_1, \dots, X_{i-1} , a minimal set of parents for X_i , such that Equation (14.3) is satisfied.
 - For each parent insert a link from the parent to X_i .
 - CPTs: Write down the conditional probability table, $\mathbf{P}(X_i | Parents(X_i))$.



Intuitively, the parents of node X_i should contain all those nodes in X_1, \dots, 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:

$$\mathbf{P}(\text{MaryCalls} \mid \text{JohnCalls}, \text{Alarm}, \text{Earthquake}, \text{Burglary}) = \mathbf{P}(\text{MaryCalls} \mid \text{Alarm}) .$$

Thus, *Alarm* will be the only parent node for *MaryCalls*.

Because each node is connected only to earlier nodes, this construction method guarantees that the network is acyclic. Another important property of Bayesian networks is that they contain no redundant probability values. If there is no redundancy, then there is no chance for inconsistency: *it is impossible for the knowledge engineer or domain expert to create a Bayesian network that violates the axioms of probability.*

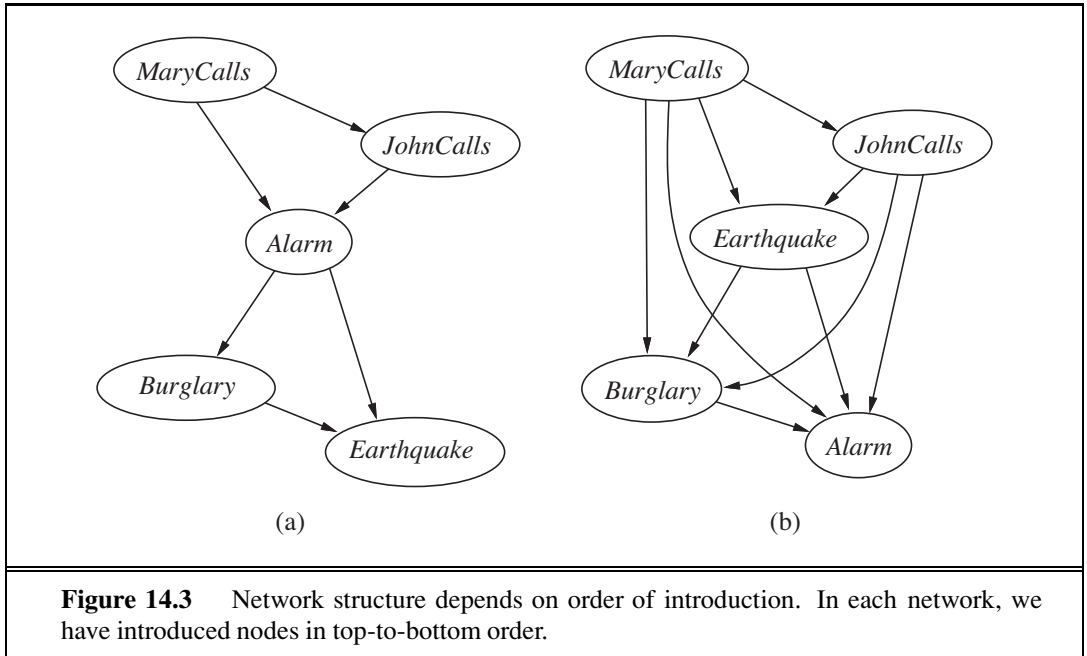


Compactness and node ordering

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 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 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 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.

LOCALLY
STRUCTURED
SPARSE



Even in a locally structured domain, we will get a compact Bayesian network only if we choose the node ordering well. What happens if we happen to choose the wrong order? Consider the burglary example again. Suppose we decide to add the nodes in the order *MaryCalls*, *JohnCalls*, *Alarm*, *Burglary*, *Earthquake*. We then 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 calls, 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:

$$\mathbf{P}(\text{Burglary} \mid \text{Alarm}, \text{JohnCalls}, \text{MaryCalls}) = \mathbf{P}(\text{Burglary} \mid \text{Alarm}) .$$

Hence we need just *Alarm* as parent.

- 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 as-



sessing 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 Section 13.5.1 (see also Exercise 8.13). 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 number 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.

14.2.2 Conditional independence relations in Bayesian networks

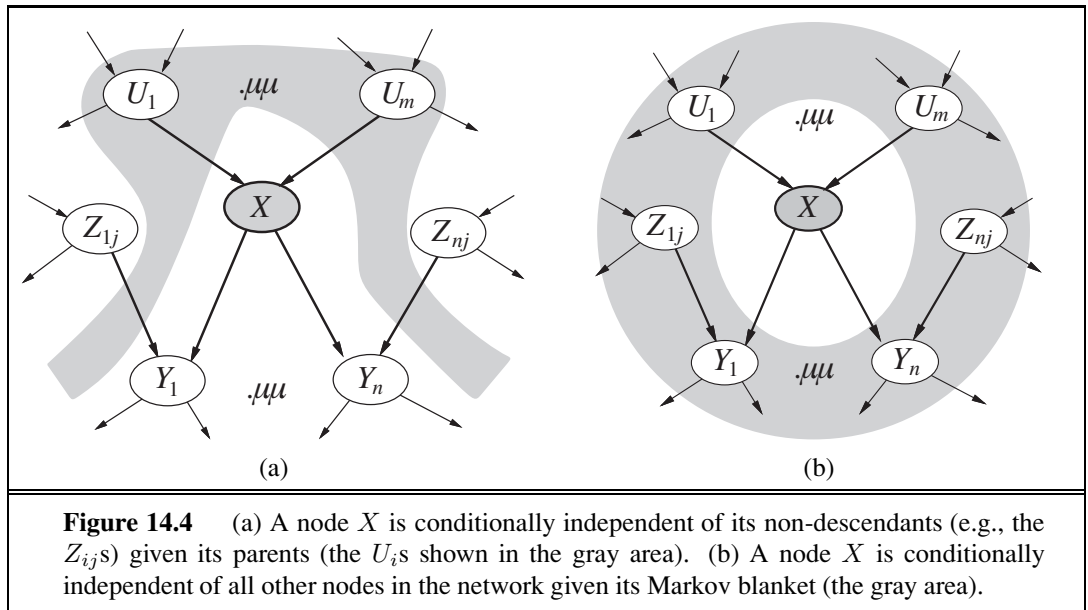
We have provided a “numerical” semantics for Bayesian networks in terms of the representation of the full joint distribution, as in Equation (14.2). Using this semantics to derive a method for constructing Bayesian networks, we were led to the consequence that a node is conditionally independent of its other 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 this we can derive the “numerical” semantics. The topological semantics² specifies that each variable is conditionally independent of its non-**descendants**, given its parents. For example, in Figure 14.2, *JohnCalls* is independent of *Burglary*, *Earthquake*, and *MaryCalls* given the value of *Alarm*. The definition is illustrated in Figure 14.4(a). From these conditional independence assertions and the interpretation of the network parameters $\theta(X_i | \text{Parents}(X_i))$ as specifications of conditional probabilities $\mathbf{P}(X_i | \text{Parents}(X_i))$, the full joint distribution given in Equation (14.2) can be reconstructed. In this sense, the “numerical” semantics and the “topological” semantics are equivalent.

Another important independence property is implied by the topological semantics: 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**. (Exercise 14.7 asks you to prove this.) For example, *Burglary* is independent of *JohnCalls* and *MaryCalls*, given *Alarm* and *Earthquake*. This property is illustrated in Figure 14.4(b).

² There is also a general topological criterion called **d-separation** for deciding whether a set of nodes **X** is conditionally 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 Pearl (1988) or Darwiche (2009). Shachter (1998) gives a more intuitive method of ascertaining d-separation.

DESCENDANT

MARKOV BLANKET



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.

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 a lake's inflows (rivers, runoff, precipitation) and outflows (rivers, evaporation, seepage) and the child is the change in the water level of the lake, then the value of the child is the sum of the inflow parents minus the sum of the outflow parents.

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

CANONICAL
DISTRIBUTION

DETERMINISTIC
NODES

NOISY-OR

LEAK NODE

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. (If some are missing, 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 q for each parent. Let us suppose these individual inhibition probabilities are as follows:

$$\begin{aligned} q_{\text{cold}} &= P(\neg \text{fever} \mid \text{cold}, \neg \text{flu}, \neg \text{malaria}) = 0.6, \\ q_{\text{flu}} &= P(\neg \text{fever} \mid \neg \text{cold}, \text{flu}, \neg \text{malaria}) = 0.2, \\ q_{\text{malaria}} &= P(\neg \text{fever} \mid \neg \text{cold}, \neg \text{flu}, \text{malaria}) = 0.1. \end{aligned}$$

Then, from this information and the noisy-OR assumptions, the entire CPT can be built. The general rule is that

$$P(x_i \mid \text{parents}(X_i)) = 1 - \prod_{\{j: X_j = \text{true}\}} q_j,$$

where the product is taken over the parents that are set to true for that row of the CPT. The following table illustrates this calculation:

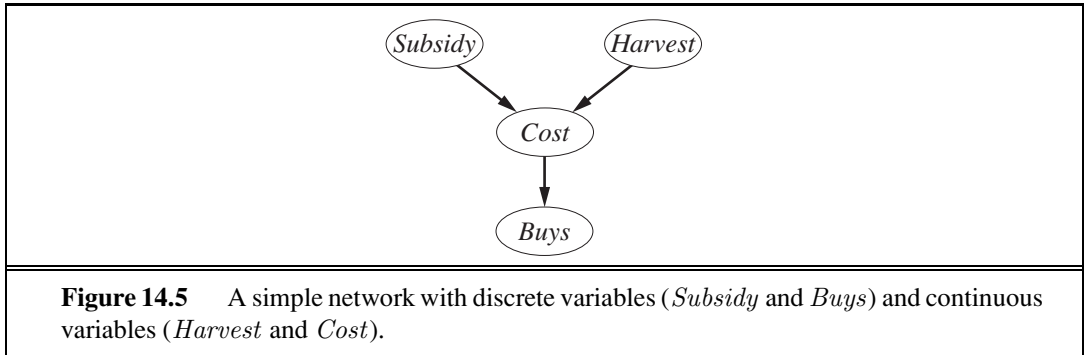
<i>Cold</i>	<i>Flu</i>	<i>Malaria</i>	$P(\text{Fever})$	$P(\neg \text{Fever})$
F	F	F	0.0	1.0
F	F	T	0.9	0.1
F	T	F	0.8	0.2
F	T	T	0.98	$0.02 = 0.2 \times 0.1$
T	F	F	0.4	0.6
T	F	T	0.94	$0.06 = 0.6 \times 0.1$
T	T	F	0.88	$0.12 = 0.6 \times 0.2$
T	T	T	0.988	$0.012 = 0.6 \times 0.2 \times 0.1$

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

DISCRETIZATION



possible values into a fixed set of intervals. For example, temperatures could be divided into ($<0^\circ\text{C}$), ($0^\circ\text{C}–100^\circ\text{C}$), and ($>100^\circ\text{C}$). Discretization is sometimes an adequate solution, but often results in a considerable loss of accuracy and very large CPTs. The most common 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. Yet another solution—sometimes called a **nonparametric** representation—is to define the conditional distribution implicitly with a collection of instances, each containing specific values of the parent and child variables. We explore this approach further in Chapter 18.

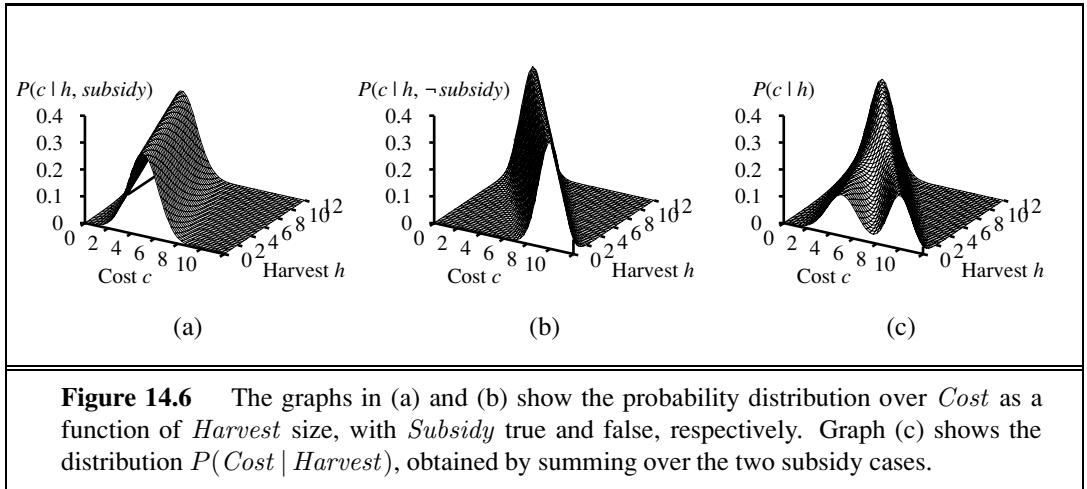
A network with both discrete and continuous variables is called a **hybrid Bayesian network**. To specify a hybrid network, we have to specify two new kinds of distributions: the conditional distribution for a continuous variable given discrete or continuous parents; and the conditional distribution 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.

For the *Cost* variable, we need to specify $\mathbf{P}(\text{Cost} \mid \text{Harvest}, \text{Subsidy})$. The discrete parent is handled by enumeration—that is, by specifying both $P(\text{Cost} \mid \text{Harvest}, \text{subsidy})$ and $P(\text{Cost} \mid \text{Harvest}, \neg \text{subsidy})$. 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 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 $\neg \text{subsidy}$, with different parameters:

$$P(c \mid h, \text{subsidy}) = N(a_t h + b_t, \sigma_t^2)(c) = \frac{1}{\sigma_t \sqrt{2\pi}} e^{-\frac{1}{2} \left(\frac{c - (a_t h + b_t)}{\sigma_t} \right)^2}$$

$$P(c \mid h, \neg \text{subsidy}) = N(a_f h + b_f, \sigma_f^2)(c) = \frac{1}{\sigma_f \sqrt{2\pi}} e^{-\frac{1}{2} \left(\frac{c - (a_f h + b_f)}{\sigma_f} \right)^2}.$$

For this example, then, the conditional distribution for *Cost* is specified by naming the linear Gaussian distribution and providing the parameters $a_t, b_t, \sigma_t, a_f, b_f$, and σ_f . Figures 14.6(a)



and (b) show these two relationships. Notice that in each case the slope is negative, because cost decreases as supply increases. (Of course, the assumption of linearity implies that the cost 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 distribution that is a multivariate Gaussian distribution (see Appendix A) over all the variables (Exercise 14.9). Furthermore, the posterior distribution given any evidence also has this property.³ When discrete variables are added as parents (not as children) of continuous variables, 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.

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:

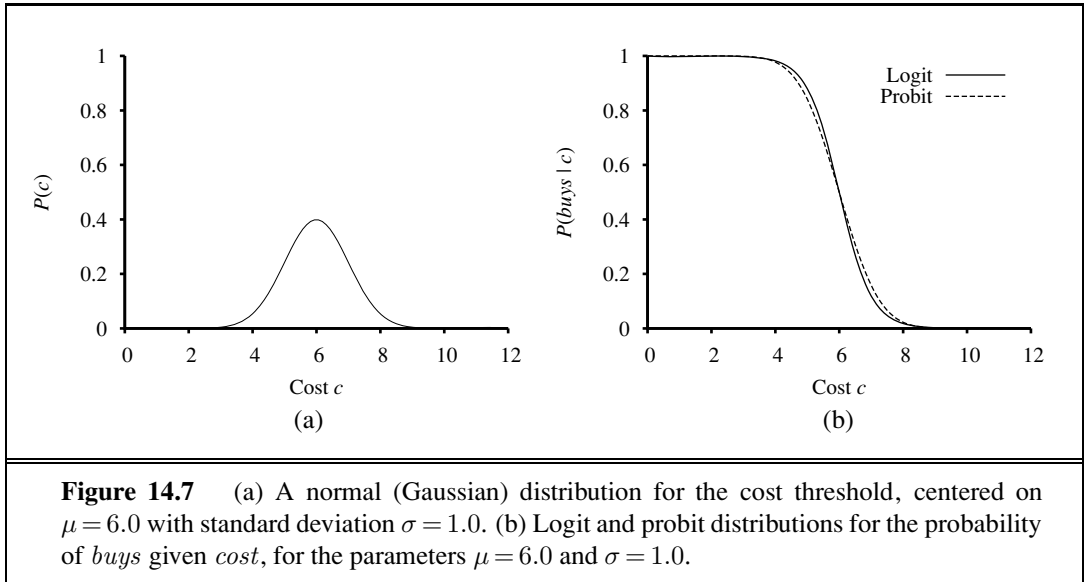
$$\Phi(x) = \int_{-\infty}^x N(0, 1)(x) dx .$$

Then the probability of *Buys* given *Cost* might be

$$P(\text{buys} | \text{Cost} = c) = \Phi((-c + \mu)/\sigma) ,$$

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. This **probit distribution**

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 see that inference for networks of discrete variables is NP-hard.



PROBIT
DISTRIBUTION

bution (pronounced “pro-bit” and short for “probability unit”) 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.

LOGIT DISTRIBUTION

LOGISTIC FUNCTION

An alternative to the probit model is the **logit distribution** (pronounced “low-jit”). It uses the **logistic function** $1/(1 + e^{-x})$ to produce a soft threshold:

$$P(buys | Cost = c) = \frac{1}{1 + \exp(-2\frac{c-\mu}{\sigma})}.$$

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.

14.4 EXACT INFERENCE IN BAYESIAN NETWORKS

EVENT

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**. To simplify the presentation, we will consider only one query variable at a time; the algorithms can easily be extended to queries with multiple variables. We will use the notation from Chapter 13: X denotes the query variable; \mathbf{E} denotes the set of evidence variables E_1, \dots, E_m , and \mathbf{e} is a particular observed event; \mathbf{Y} will denote the nonevidence, nonquery variables Y_1, \dots, Y_l (called the **hidden variables**). Thus, the complete set of variables is $\mathbf{X} = \{X\} \cup \mathbf{E} \cup \mathbf{Y}$. A typical query asks for the posterior probability distribution $\mathbf{P}(X | \mathbf{e})$.

HIDDEN VARIABLE

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:

$$\mathbf{P}(\text{Burglary} \mid JohnCalls = true, MaryCalls = true) = \langle 0.284, 0.716 \rangle .$$

In this section we 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.

14.4.1 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 $\mathbf{P}(X \mid \mathbf{e})$ can be answered using Equation (13.9), which we repeat here for convenience:

$$\mathbf{P}(X \mid \mathbf{e}) = \alpha \mathbf{P}(X, \mathbf{e}) = \alpha \sum_{\mathbf{y}} \mathbf{P}(X, \mathbf{e}, \mathbf{y}) .$$

Now, a Bayesian network gives a complete representation of the full joint distribution. More specifically, Equation (14.2) on page 513 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.*

Consider the query $\mathbf{P}(\text{Burglary} \mid JohnCalls = true, MaryCalls = true)$. The hidden variables for this query are *Earthquake* and *Alarm*. From Equation (13.9), using initial letters for the variables to shorten the expressions, we have⁴

$$\mathbf{P}(B \mid j, m) = \alpha \mathbf{P}(B, j, m) = \alpha \sum_e \sum_a \mathbf{P}(B, j, m, e, a) .$$

The semantics of Bayesian networks (Equation (14.2)) then gives us an expression in terms of CPT entries. For simplicity, we do this just for $Burglary = true$:

$$P(b \mid j, m) = \alpha \sum_e \sum_a P(b)P(e)P(a \mid b, e)P(j \mid a)P(m \mid a) .$$

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

$$P(b \mid j, m) = \alpha P(b) \sum_e P(e) \sum_a P(a \mid b, e)P(j \mid a)P(m \mid a) . \quad (14.4)$$

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

⁴ An expression such as $\sum_e P(a, e)$ means to sum $P(A = a, E = e)$ for all possible values of e . When E is Boolean, 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.



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 (14.4) 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. The algorithm is very similar in structure to the backtracking algorithm for solving CSPs (Figure 6.5) and the DPLL algorithm for satisfiability (Figure 7.17).

The space complexity of ENUMERATION-ASK is only linear in the number of variables: 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 $O(n 2^n)$ for the simple approach described earlier, but still rather grim.

Note that the tree in Figure 14.8 makes explicit the *repeated subexpressions* 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.

14.4.2 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 (14.4) 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.

Let us illustrate this process for the burglary network. We evaluate the expression

$$\mathbf{P}(B | j, m) = \alpha \underbrace{\mathbf{P}(B)}_{\mathbf{f}_1(B)} \sum_e \underbrace{P(e)}_{\mathbf{f}_2(E)} \sum_a \underbrace{\mathbf{P}(a | B, e)}_{\mathbf{f}_3(A, B, E)} \underbrace{P(j | a)}_{\mathbf{f}_4(A)} \underbrace{P(m | a)}_{\mathbf{f}_5(A)} .$$

Notice that we have annotated each part of the expression with the name of the corresponding **factor**; each factor is a matrix indexed by the values of its argument variables. For example, the factors $\mathbf{f}_4(A)$ and $\mathbf{f}_5(A)$ corresponding to $P(j | a)$ and $P(m | a)$ depend just on A because J and M are fixed by the query. They are therefore two-element vectors:

$$\mathbf{f}_4(A) = \begin{pmatrix} P(j | a) \\ P(j | \neg a) \end{pmatrix} = \begin{pmatrix} 0.90 \\ 0.05 \end{pmatrix} \quad \mathbf{f}_5(A) = \begin{pmatrix} P(m | a) \\ P(m | \neg a) \end{pmatrix} = \begin{pmatrix} 0.70 \\ 0.01 \end{pmatrix} .$$

$\mathbf{f}_3(A, B, E)$ will be a $2 \times 2 \times 2$ matrix, which is hard to show on the printed page. (The “first” element is given by $P(a | b, e) = 0.95$ and the “last” by $P(\neg a | \neg b, \neg e) = 0.999$.) In terms of factors, the query expression is written as

$$\mathbf{P}(B | j, m) = \alpha \mathbf{f}_1(B) \times \sum_e \mathbf{f}_2(E) \times \sum_a \mathbf{f}_3(A, B, E) \times \mathbf{f}_4(A) \times \mathbf{f}_5(A)$$

VARIABLE
ELIMINATION

FACTOR

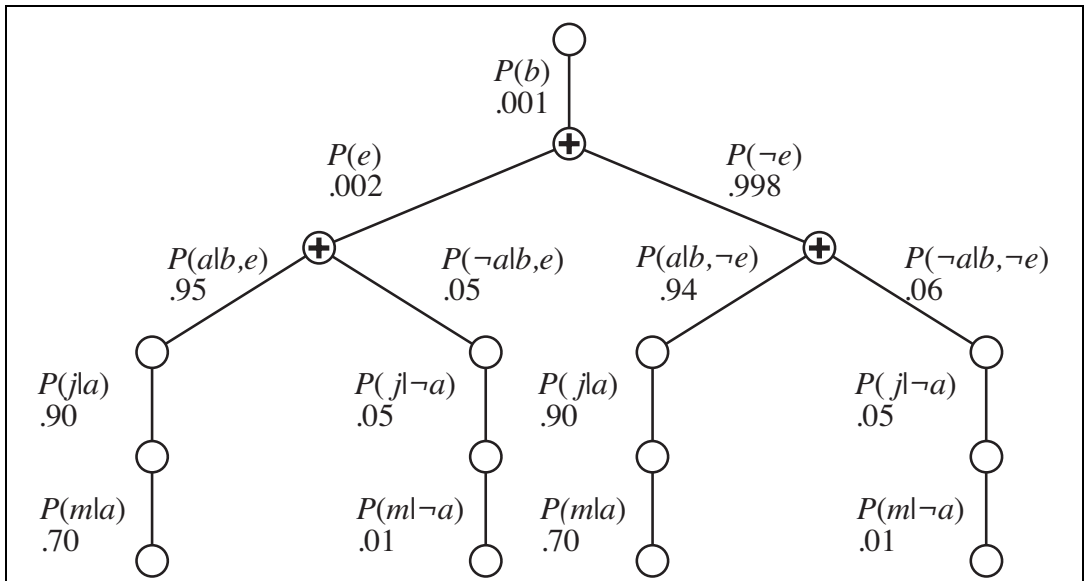


Figure 14.8 The structure of the expression shown in Equation (14.4). The evaluation proceeds top down, multiplying values along each path and summing at the “+” nodes. Notice the repetition of the paths for j and m .

```

function ENUMERATION-ASK( $X, \mathbf{e}, bn$ ) returns a distribution over  $X$ 
  inputs:  $X$ , the query variable
            $\mathbf{e}$ , observed values for variables  $\mathbf{E}$ 
            $bn$ , a Bayes net with variables  $\{X\} \cup \mathbf{E} \cup \mathbf{Y}$  /*  $\mathbf{Y}$  = hidden variables */

   $\mathbf{Q}(X) \leftarrow$  a distribution over  $X$ , initially empty
  for each value  $x_i$  of  $X$  do
     $\mathbf{Q}(x_i) \leftarrow$  ENUMERATE-ALL( $bn.VARS, \mathbf{e}_{x_i}$ )
    where  $\mathbf{e}_{x_i}$  is  $\mathbf{e}$  extended with  $X = x_i$ 
  return NORMALIZE( $\mathbf{Q}(X)$ )



---


function ENUMERATE-ALL( $vars, \mathbf{e}$ ) returns a real number
  if EMPTY?( $vars$ ) then return 1.0
   $Y \leftarrow$  FIRST( $vars$ )
  if  $Y$  has value  $y$  in  $\mathbf{e}$ 
    then return  $P(y \mid \text{parents}(Y)) \times$  ENUMERATE-ALL(REST( $vars$ ),  $\mathbf{e}$ )
    else return  $\sum_y P(y \mid \text{parents}(Y)) \times$  ENUMERATE-ALL(REST( $vars$ ),  $\mathbf{e}_y$ )
    where  $\mathbf{e}_y$  is  $\mathbf{e}$  extended with  $Y = y$ 

```

Figure 14.9 The enumeration algorithm for answering queries on Bayesian networks.

where the “ \times ” operator is not ordinary matrix multiplication but instead the **pointwise product** operation, to be described shortly.

The process of evaluation is a process of summing out variables (right to left) from pointwise products of factors to produce new factors, eventually yielding a factor that is the solution, i.e., the posterior distribution over the query variable. The steps are as follows:

- First, we sum out A from the product of \mathbf{f}_3 , \mathbf{f}_4 , and \mathbf{f}_5 . This gives us a new 2×2 factor $\mathbf{f}_6(B, E)$ whose indices range over just B and E :

$$\begin{aligned}\mathbf{f}_6(B, E) &= \sum_a \mathbf{f}_3(A, B, E) \times \mathbf{f}_4(A) \times \mathbf{f}_5(A) \\ &= (\mathbf{f}_3(a, B, E) \times \mathbf{f}_4(a) \times \mathbf{f}_5(a)) + (\mathbf{f}_3(\neg a, B, E) \times \mathbf{f}_4(\neg a) \times \mathbf{f}_5(\neg a)).\end{aligned}$$

Now we are left with the expression

$$\mathbf{P}(B \mid j, m) = \alpha \mathbf{f}_1(B) \times \sum_e \mathbf{f}_2(E) \times \mathbf{f}_6(B, E).$$

- Next, we sum out E from the product of \mathbf{f}_2 and \mathbf{f}_6 :

$$\begin{aligned}\mathbf{f}_7(B) &= \sum_e \mathbf{f}_2(E) \times \mathbf{f}_6(B, E) \\ &= \mathbf{f}_2(e) \times \mathbf{f}_6(B, e) + \mathbf{f}_2(\neg e) \times \mathbf{f}_6(B, \neg e).\end{aligned}$$

This leaves the expression

$$\mathbf{P}(B \mid j, m) = \alpha \mathbf{f}_1(B) \times \mathbf{f}_7(B)$$

which can be evaluated by taking the pointwise product and normalizing the result.

Examining this sequence, we see that two basic computational operations are required: pointwise product of a pair of factors, and summing out a variable from a product of factors. The next section describes each of these operations.

Operations on factors

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 and whose elements are given by the product of the corresponding elements in the two factors. Suppose the two factors have variables Y_1, \dots, Y_k in common. Then we have

$$\mathbf{f}(X_1 \dots X_j, Y_1 \dots Y_k, Z_1 \dots Z_l) = \mathbf{f}_1(X_1 \dots X_j, Y_1 \dots Y_k) \mathbf{f}_2(Y_1 \dots Y_k, Z_1 \dots Z_l).$$

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 $\mathbf{f}_2(B, C)$, the pointwise product $\mathbf{f}_1 \times \mathbf{f}_2 = \mathbf{f}_3(A, B, C)$ has $2^{1+1+1} = 8$ entries, as illustrated in Figure 14.10. Notice that the factor resulting from a pointwise product can contain more variables than any of the factors being multiplied and that the size of a factor is exponential in the number of variables. This is where both space and time complexity arise in the variable elimination algorithm.

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	$.3 \times .2 = .06$
T	F	.7	T	F	.8	T	T	F	$.3 \times .8 = .24$
F	T	.9	F	T	.6	T	F	T	$.7 \times .6 = .42$
F	F	.1	F	F	.4	T	F	F	$.7 \times .4 = .28$
						F	T	T	$.9 \times .2 = .18$
						F	T	F	$.9 \times .8 = .72$
						F	F	T	$.1 \times .6 = .06$
						F	F	F	$.1 \times .4 = .04$

Figure 14.10 Illustrating pointwise multiplication: $\mathbf{f}_1(A, B) \times \mathbf{f}_2(B, C) = \mathbf{f}_3(A, B, C)$.

Summing out a variable from a product of factors is done by adding up the submatrices formed by fixing the variable to each of its values in turn. For example, to sum out A from $\mathbf{f}_3(A, B, C)$, we write

$$\begin{aligned} \mathbf{f}(B, C) &= \sum_a \mathbf{f}_3(A, B, C) = \mathbf{f}_3(a, B, C) + \mathbf{f}_3(\neg a, B, C) \\ &= \begin{pmatrix} .06 & .24 \\ .42 & .28 \end{pmatrix} + \begin{pmatrix} .18 & .72 \\ .06 & .04 \end{pmatrix} = \begin{pmatrix} .24 & .96 \\ .48 & .32 \end{pmatrix}. \end{aligned}$$

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. For example, if we were to sum out E first in the burglary network, the relevant part of the expression would be

$$\sum_e \mathbf{f}_2(E) \times \mathbf{f}_3(A, B, E) \times \mathbf{f}_4(A) \times \mathbf{f}_5(A) = \mathbf{f}_4(A) \times \mathbf{f}_5(A) \times \sum_e \mathbf{f}_2(E) \times \mathbf{f}_3(A, B, E).$$

Now the pointwise product inside the summation is computed, and the variable is summed out of the resulting matrix.

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 functions for pointwise product and summing out, the variable elimination algorithm itself can be written quite simply, as shown in Figure 14.11.

Variable ordering and variable relevance

The algorithm in Figure 14.11 includes an unspecified ORDER function to choose an ordering for the variables. Every choice of ordering yields a valid algorithm, but different orderings cause different intermediate factors to be generated during the calculation. For example, in the calculation shown previously, we eliminated A before E ; if we do it the other way, the calculation becomes

$$\mathbf{P}(B \mid j, m) = \alpha \mathbf{f}_1(B) \times \sum_a \mathbf{f}_4(A) \times \mathbf{f}_5(A) \times \sum_e \mathbf{f}_2(E) \times \mathbf{f}_3(A, B, E),$$

during which a new factor $\mathbf{f}_6(A, B)$ will be generated.

In general, 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

```

function ELIMINATION-ASK( $X, \mathbf{e}, bn$ ) returns a distribution over  $X$ 
  inputs:  $X$ , the query variable
            $\mathbf{e}$ , observed values for variables  $\mathbf{E}$ 
            $bn$ , a Bayesian network specifying joint distribution  $\mathbf{P}(X_1, \dots, X_n)$ 

   $factors \leftarrow []$ 
  for each  $var$  in ORDER( $bn.VARS$ ) do
     $factors \leftarrow [MAKE-FACTOR(var, \mathbf{e}) | factors]$ 
    if  $var$  is a hidden variable then  $factors \leftarrow SUM-OUT(var, factors)$ 
  return NORMALIZE(POINTWISE-PRODUCT( $factors$ ))

```

Figure 14.11 The variable elimination algorithm for inference in Bayesian networks.

is determined by the order of elimination of variables and by the structure of the network. It turns out to be intractable to determine the optimal ordering, but several good heuristics are available. One fairly effective method is a greedy one: eliminate whichever variable minimizes the size of the next factor to be constructed.

Let us consider one more query: $\mathbf{P}(JohnCalls \mid Burglary = true)$. As usual, the first step is to write out the nested summation:

$$\mathbf{P}(J \mid b) = \alpha P(b) \sum_e P(e) \sum_a P(a \mid b, e) \mathbf{P}(J \mid a) \sum_m P(m \mid a).$$

Evaluating this expression from right to left, we notice something interesting: $\sum_m P(m \mid a)$ 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 $P(JohnCalls \mid Burglary = true)$ 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 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.

14.4.3 The complexity of exact inference

The complexity of exact inference in Bayesian networks depends strongly on 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.

For **multiply connected** networks, such as that of Figure 14.12(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*



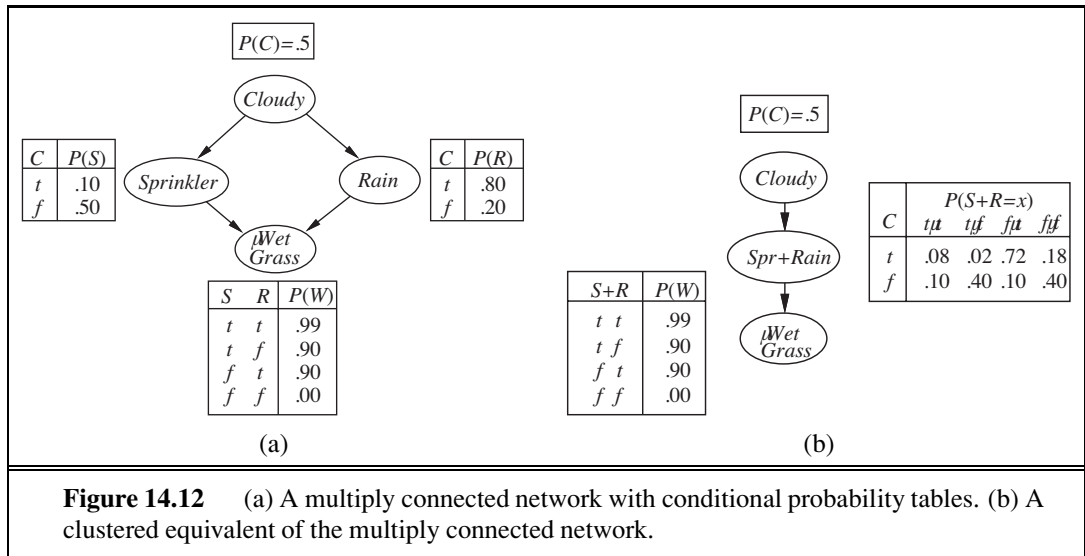
SINGLY CONNECTED

POLYTREE



MULTIPLY
CONNECTED





includes inference in propositional logic as a special case, inference in Bayesian networks is NP-hard. In fact, it can be shown (Exercise 14.16) 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 6, the difficulty of solving a discrete CSP is related to how “treelike” its constraint graph is. Measures such as **tree 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.

14.4.4 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.

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 14.12(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 14.12(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. Although this example doesn’t show it, the process of clustering often produces meganodes that share some variables.

Once the network is in polytree form, a special-purpose inference algorithm is required, because ordinary inference methods cannot handle meganodes that share variables with each other. Essentially, the algorithm is a form of constraint propagation (see Chapter 6) where the constraints ensure that neighboring meganodes 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 *linear* in the size of the clustered 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 necessarily be exponentially large.

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. Monte Carlo algorithms, of which simulated annealing (page 126) is an example, are used in many branches of science to estimate quantities that are difficult to calculate exactly. 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.

14.5.1 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 uniformly distributed in the range $[0, 1]$, it is a simple matter to sample any distribution on a single variable, whether discrete or continuous. (See Exercise 14.17.)

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 14.13. We can illustrate its operation on the network in Figure 14.12(a), assuming an ordering $[Cloudy, Sprinkler, Rain, WetGrass]$:

1. Sample from $\mathbf{P}(Cloudy) = \langle 0.5, 0.5 \rangle$, value is *true*.
2. Sample from $\mathbf{P}(Sprinkler \mid Cloudy = true) = \langle 0.1, 0.9 \rangle$, value is *false*.
3. Sample from $\mathbf{P}(Rain \mid Cloudy = true) = \langle 0.8, 0.2 \rangle$, value is *true*.
4. Sample from $\mathbf{P}(WetGrass \mid Sprinkler = false, Rain = true) = \langle 0.9, 0.1 \rangle$, value is *true*.

In this case, PRIOR-SAMPLE returns the event $[true, false, true, true]$.

```

function PRIOR-SAMPLE( $bn$ ) returns an event sampled from the prior specified by  $bn$ 
inputs:  $bn$ , a Bayesian network specifying joint distribution  $\mathbf{P}(X_1, \dots, X_n)$ 

 $\mathbf{x} \leftarrow$  an event with  $n$  elements
foreach variable  $X_i$  in  $X_1, \dots, X_n$  do
     $\mathbf{x}[i] \leftarrow$  a random sample from  $\mathbf{P}(X_i \mid \text{parents}(X_i))$ 
return  $\mathbf{x}$ 
    
```

Figure 14.13 A sampling algorithm that generates events from a Bayesian network. Each variable is sampled according to the conditional distribution given the values already sampled for the variable's parents.

It is easy to see that PRIOR-SAMPLE generates samples from the prior joint distribution specified by the network. First, let $S_{PS}(x_1, \dots, x_n)$ be the probability that a specific event is generated by the PRIOR-SAMPLE algorithm. *Just looking at the sampling process*, we have

$$S_{PS}(x_1 \dots x_n) = \prod_{i=1}^n P(x_i \mid \text{parents}(X_i))$$

because each sampling step depends only on the parent values. This expression should look 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 (14.2). That is, we have

$$S_{PS}(x_1 \dots x_n) = P(x_1 \dots x_n) .$$

This simple fact makes it 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 $N_{PS}(x_1, \dots, x_n)$ be the number of times the specific event x_1, \dots, x_n occurs in the set of samples. We expect this number, as a fraction of the total, to converge in the limit to its expected value according to the sampling probability:

$$\lim_{N \rightarrow \infty} \frac{N_{PS}(x_1, \dots, x_n)}{N} = S_{PS}(x_1, \dots, x_n) = P(x_1, \dots, x_n) . \quad (14.5)$$

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, \dots, x_m , where $m \leq n$, as follows:

$$P(x_1, \dots, x_m) \approx N_{PS}(x_1, \dots, x_m) / N . \quad (14.6)$$

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 14.14. First, it generates samples from the prior distribution specified 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 (14.6), 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 14.12(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 \mathbf{e} . It is a particular instance of the general statistical technique of **importance sampling**, tailored for inference in Bayesian networks. We begin by

IMPORTANCE
SAMPLING

```

function REJECTION-SAMPLING( $X, \mathbf{e}, bn, N$ ) returns an estimate of  $\mathbf{P}(X|\mathbf{e})$ 
  inputs:  $X$ , the query variable
            $\mathbf{e}$ , observed values for variables  $\mathbf{E}$ 
            $bn$ , a Bayesian network
            $N$ , the total number of samples to be generated
  local variables:  $\mathbf{N}$ , a vector of counts for each value of  $X$ , initially zero

  for  $j = 1$  to  $N$  do
     $\mathbf{x} \leftarrow \text{PRIOR-SAMPLE}(bn)$ 
    if  $\mathbf{x}$  is consistent with  $\mathbf{e}$  then
       $\mathbf{N}[x] \leftarrow \mathbf{N}[x] + 1$  where  $x$  is the value of  $X$  in  $\mathbf{x}$ 
  return NORMALIZE( $\mathbf{N}$ )

```

Figure 14.14 The rejection-sampling algorithm for answering queries given evidence in a Bayesian network.

describing how the algorithm works; then we show that it works correctly—that is, generates consistent probability estimates.

LIKELIHOOD-WEIGHTING (see Figure 14.15) fixes the values for the evidence variables \mathbf{E} and samples only the nonevidence variables. 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 14.12(a), with the query $\mathbf{P}(\text{Rain} \mid \text{Cloudy} = \text{true}, \text{WetGrass} = \text{true})$ and the ordering *Cloudy, Sprinkler, Rain, WetGrass*. (Any topological ordering will do.) The process goes as follows: First, the weight w is set to 1.0. Then an event is generated:

1. *Cloudy* is an evidence variable with value *true*. Therefore, we set

$$w \leftarrow w \times P(\text{Cloudy} = \text{true}) = 0.5 .$$

2. *Sprinkler* is not an evidence variable, so sample from $\mathbf{P}(\text{Sprinkler} \mid \text{Cloudy} = \text{true}) = \langle 0.1, 0.9 \rangle$; suppose this returns *false*.
3. Similarly, sample from $\mathbf{P}(\text{Rain} \mid \text{Cloudy} = \text{true}) = \langle 0.8, 0.2 \rangle$; suppose this returns *true*.
4. *WetGrass* is an evidence variable with value *true*. Therefore, we set

$$w \leftarrow w \times P(\text{WetGrass} = \text{true} \mid \text{Sprinkler} = \text{false}, \text{Rain} = \text{true}) = 0.45 .$$

Here WEIGHTED-SAMPLE returns the event $[\text{true}, \text{false}, \text{true}, \text{true}]$ with weight 0.45, and this is tallied under *Rain = true*.

To understand why likelihood weighting works, we start by examining the sampling probability S_{WS} for WEIGHTED-SAMPLE. Remember that the evidence variables \mathbf{E} are fixed

```

function LIKELIHOOD-WEIGHTING( $X, \mathbf{e}, bn, N$ ) returns an estimate of  $\mathbf{P}(X|\mathbf{e})$ 
  inputs:  $X$ , the query variable
            $\mathbf{e}$ , observed values for variables  $\mathbf{E}$ 
            $bn$ , a Bayesian network specifying joint distribution  $\mathbf{P}(X_1, \dots, X_n)$ 
            $N$ , the total number of samples to be generated
  local variables:  $\mathbf{W}$ , a vector of weighted counts for each value of  $X$ , initially zero

  for  $j = 1$  to  $N$  do
     $\mathbf{x}, w \leftarrow \text{WEIGHTED-SAMPLE}(bn, \mathbf{e})$ 
     $\mathbf{W}[x] \leftarrow \mathbf{W}[x] + w$  where  $x$  is the value of  $X$  in  $\mathbf{x}$ 
  return NORMALIZE( $\mathbf{W}$ )

```

```

function WEIGHTED-SAMPLE( $bn, \mathbf{e}$ ) returns an event and a weight
   $w \leftarrow 1$ ;  $\mathbf{x} \leftarrow$  an event with  $n$  elements initialized from  $\mathbf{e}$ 
  foreach variable  $X_i$  in  $X_1, \dots, X_n$  do
    if  $X_i$  is an evidence variable with value  $x_i$  in  $\mathbf{e}$ 
      then  $w \leftarrow w \times P(X_i = x_i \mid \text{parents}(X_i))$ 
      else  $\mathbf{x}[i] \leftarrow$  a random sample from  $\mathbf{P}(X_i \mid \text{parents}(X_i))$ 
  return  $\mathbf{x}, w$ 

```

Figure 14.15 The likelihood-weighting algorithm for inference in Bayesian networks. In WEIGHTED-SAMPLE, each nonevidence variable is sampled according to the conditional distribution given the values already sampled for the variable's parents, while a weight is accumulated based on the likelihood for each evidence variable.

with values \mathbf{e} . We call the nonevidence variables \mathbf{Z} (including the query variable X). The algorithm samples each variable in \mathbf{Z} given its parent values:

$$S_{WS}(\mathbf{z}, \mathbf{e}) = \prod_{i=1}^l P(z_i \mid \text{parents}(Z_i)). \quad (14.7)$$

Notice that $\text{Parents}(Z_i)$ can include both nonevidence 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. For example, when sampling *Sprinkler* the algorithm pays attention to the evidence *Cloudy = true* in its parent variable. 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.⁵ For example, when sampling *Sprinkler* and *Rain* the algorithm ignores the evidence in the child variable *WetGrass = true*; this means it will generate many samples with *Sprinkler = false* and *Rain = false* despite the fact that the evidence actually rules out this case.

⁵ Ideally, we would like to 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.

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_i s):

$$w(\mathbf{z}, \mathbf{e}) = \prod_{i=1}^m P(e_i | \text{parents}(E_i)) . \quad (14.8)$$

Multiplying Equations (14.7) and (14.8), we see that the *weighted* probability of a sample has the particularly convenient form

$$\begin{aligned} S_{WS}(\mathbf{z}, \mathbf{e}) w(\mathbf{z}, \mathbf{e}) &= \prod_{i=1}^l P(z_i | \text{parents}(Z_i)) \prod_{i=1}^m P(e_i | \text{parents}(E_i)) \\ &= P(\mathbf{z}, \mathbf{e}) \end{aligned} \quad (14.9)$$

because the two products cover all the variables in the network, allowing us to use Equation (14.2) 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:

$$\begin{aligned} \hat{P}(x | \mathbf{e}) &= \alpha \sum_{\mathbf{y}} N_{WS}(x, \mathbf{y}, \mathbf{e}) w(x, \mathbf{y}, \mathbf{e}) && \text{from LIKELIHOOD-WEIGHTING} \\ &\approx \alpha' \sum_{\mathbf{y}} S_{WS}(x, \mathbf{y}, \mathbf{e}) w(x, \mathbf{y}, \mathbf{e}) && \text{for large } N \\ &= \alpha' \sum_{\mathbf{y}} P(x, \mathbf{y}, \mathbf{e}) && \text{by Equation (14.9)} \\ &= \alpha' P(x, \mathbf{e}) = P(x | \mathbf{e}) . \end{aligned}$$

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 nonevidence variables will have no evidence in their parents and ancestors to guide the generation of samples. This means the samples will be simulations that bear little resemblance to the reality suggested by the evidence.

14.5.2 Inference by Markov chain simulation

Markov chain Monte Carlo (MCMC) algorithms work quite differently from rejection sampling and likelihood weighting. Instead of generating each sample from scratch, MCMC algorithms generate each sample by making a random change to the preceding sample. It is therefore helpful to think of an MCMC algorithm as being in a particular *current state* specifying a value for every variable and generating a *next state* by making random changes to the

GIBBS SAMPLING

current state. (If this reminds you of simulated annealing from Chapter 4 or WALKSAT from Chapter 7, that is because both are members of the MCMC family.) Here we describe a particular form of MCMC called **Gibbs sampling**, which is especially well suited for Bayesian networks. (Other forms, some of them significantly more powerful, are discussed in the notes at the end of the chapter.) We will first describe what the algorithm does, then we will explain why it works.

Gibbs sampling in Bayesian networks

The Gibbs sampling algorithm for Bayesian networks starts with an arbitrary state (with the evidence variables fixed at their observed values) and generates a next state by randomly sampling a value for one of the nonevidence variables X_i . The sampling for X_i is done *conditioned on the current values of the variables in the Markov blanket of X_i* . (Recall from page 517 that the Markov blanket of a variable consists of its parents, children, and children's parents.) The algorithm 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 $\mathbf{P}(\text{Rain} \mid \text{Sprinkler} = \text{true}, \text{WetGrass} = \text{true})$ applied to the network in Figure 14.12(a). The evidence variables *Sprinkler* and *WetGrass* are fixed to their observed values and the nonevidence variables *Cloudy* and *Rain* are initialized randomly—let us say to *true* and *false* respectively. Thus, the initial state is $[\text{true}, \text{true}, \text{false}, \text{true}]$. Now the nonevidence variables are sampled repeatedly in an arbitrary order. For example:

1. *Cloudy* is sampled, given the current values of its Markov blanket variables: in this case, we sample from $\mathbf{P}(\text{Cloudy} \mid \text{Sprinkler} = \text{true}, \text{Rain} = \text{false})$. (Shortly, we will show how to calculate this distribution.) Suppose the result is *Cloudy* = *false*. Then the new current state is $[\text{false}, \text{true}, \text{false}, \text{true}]$.
2. *Rain* is sampled, given the current values of its Markov blanket variables: in this case, we sample from $\mathbf{P}(\text{Rain} \mid \text{Cloudy} = \text{false}, \text{Sprinkler} = \text{true}, \text{WetGrass} = \text{true})$. Suppose this yields *Rain* = *true*. The new current state is $[\text{false}, \text{true}, \text{true}, \text{true}]$.

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 false, then the answer to the query is $\text{NORMALIZE}(\langle 20, 60 \rangle) = \langle 0.25, 0.75 \rangle$. The complete algorithm is shown in Figure 14.16.

Why Gibbs sampling works

We will now show that Gibbs sampling 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

```

function GIBBS-ASK( $X, \mathbf{e}, bn, N$ ) returns an estimate of  $\mathbf{P}(X|\mathbf{e})$ 
  local variables:  $\mathbf{N}$ , a vector of counts for each value of  $X$ , initially zero
                    $\mathbf{Z}$ , the nonevidence variables in  $bn$ 
                    $\mathbf{x}$ , the current state of the network, initially copied from  $\mathbf{e}$ 

  initialize  $\mathbf{x}$  with random values for the variables in  $\mathbf{Z}$ 
  for  $j = 1$  to  $N$  do
    for each  $Z_i$  in  $\mathbf{Z}$  do
      set the value of  $Z_i$  in  $\mathbf{x}$  by sampling from  $\mathbf{P}(Z_i|mb(Z_i))$ 
       $\mathbf{N}[x] \leftarrow \mathbf{N}[x] + 1$  where  $x$  is the value of  $X$  in  $\mathbf{x}$ 
  return NORMALIZE( $\mathbf{N}$ )

```

Figure 14.16 The Gibbs sampling algorithm for approximate inference in Bayesian networks; this version cycles through the variables, but choosing variables at random also works.

MARKOV CHAIN

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 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}' :

$$\pi_{t+1}(\mathbf{x}') = \sum_{\mathbf{x}} \pi_t(\mathbf{x}) q(\mathbf{x} \rightarrow \mathbf{x}').$$

STATIONARY
DISTRIBUTION

We 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

$$\pi(\mathbf{x}') = \sum_{\mathbf{x}} \pi(\mathbf{x}) q(\mathbf{x} \rightarrow \mathbf{x}') \quad \text{for all } \mathbf{x}'. \quad (14.10)$$

ERGODIC

Provided the transition probability distribution q is **ergodic**—that is, every state is reachable from every other and there are no strictly periodic cycles—there is exactly one distribution π satisfying this equation for any given q .

Equation (14.10) 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; that is,

$$\pi(\mathbf{x}) q(\mathbf{x} \rightarrow \mathbf{x}') = \pi(\mathbf{x}') q(\mathbf{x}' \rightarrow \mathbf{x}) \quad \text{for all } \mathbf{x}, \mathbf{x}'. \quad (14.11)$$

DETAILED BALANCE

When these equations hold, we say that $q(\mathbf{x} \rightarrow \mathbf{x}')$ is in **detailed balance** with $\pi(\mathbf{x})$.

We can show that detailed balance implies stationarity simply by summing over \mathbf{x} in Equation (14.11). 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.

The transition probability $q(\mathbf{x} \rightarrow \mathbf{x}')$ defined by the sampling step in GIBBS-ASK is actually a special case of the more general definition of Gibbs sampling, according to which each variable is sampled conditionally on the current values of *all* the other variables. We start by showing that this general definition of Gibbs sampling satisfies the detailed balance equation with a stationary distribution equal to $P(\mathbf{x} | \mathbf{e})$, (the true posterior distribution on the nonevidence variables). Then, we simply observe that, for Bayesian networks, sampling conditionally on all variables is equivalent to sampling conditionally on the variable's Markov blanket (see page 517).

To analyze the general Gibbs sampler, which samples each X_i in turn with a transition probability q_i that conditions on all the other variables, we define $\bar{\mathbf{x}}_i$ to be these other variables (except the evidence variables); their values in the current state are $\bar{\mathbf{x}}_i$. If we sample a new value x'_i for X_i conditionally on all the other variables, including the evidence, we have

$$q_i(\mathbf{x} \rightarrow \mathbf{x}') = q_i((x_i, \bar{\mathbf{x}}_i) \rightarrow (x'_i, \bar{\mathbf{x}}_i)) = P(x'_i | \bar{\mathbf{x}}_i, \mathbf{e}) .$$

Now we show that the transition probability for each step of the Gibbs sampler is in detailed balance with the true posterior:

$$\begin{aligned} \pi(\mathbf{x})q_i(\mathbf{x} \rightarrow \mathbf{x}') &= P(\mathbf{x} | \mathbf{e})P(x'_i | \bar{\mathbf{x}}_i, \mathbf{e}) = P(x_i, \bar{\mathbf{x}}_i | \mathbf{e})P(x'_i | \bar{\mathbf{x}}_i, \mathbf{e}) \\ &= P(x_i | \bar{\mathbf{x}}_i, \mathbf{e})P(\bar{\mathbf{x}}_i | \mathbf{e})P(x'_i | \bar{\mathbf{x}}_i, \mathbf{e}) \quad (\text{using the chain rule on the first term}) \\ &= P(x_i | \bar{\mathbf{x}}_i, \mathbf{e})P(x'_i, \bar{\mathbf{x}}_i | \mathbf{e}) \quad (\text{using the chain rule backward}) \\ &= \pi(\mathbf{x}')q_i(\mathbf{x}' \rightarrow \mathbf{x}) . \end{aligned}$$

We can think of the loop “**for each** Z_i **in** \mathbf{Z} **do**” in Figure 14.16 as defining one large transition probability q that is the sequential composition $q_1 \circ q_2 \circ \dots \circ q_n$ of the transition probabilities for the individual variables. It is easy to show (Exercise 14.19) that if each of q_i and q_j has π as its stationary distribution, then the sequential composition $q_i \circ q_j$ does too; hence the transition probability q for the whole loop has $P(\mathbf{x} | \mathbf{e})$ as its stationary distribution. Finally, unless the CPTs contain probabilities of 0 or 1—which can cause the state space to become disconnected—it is easy to see that q is ergodic. Hence, the samples generated by Gibbs sampling will eventually be drawn from the true posterior distribution.

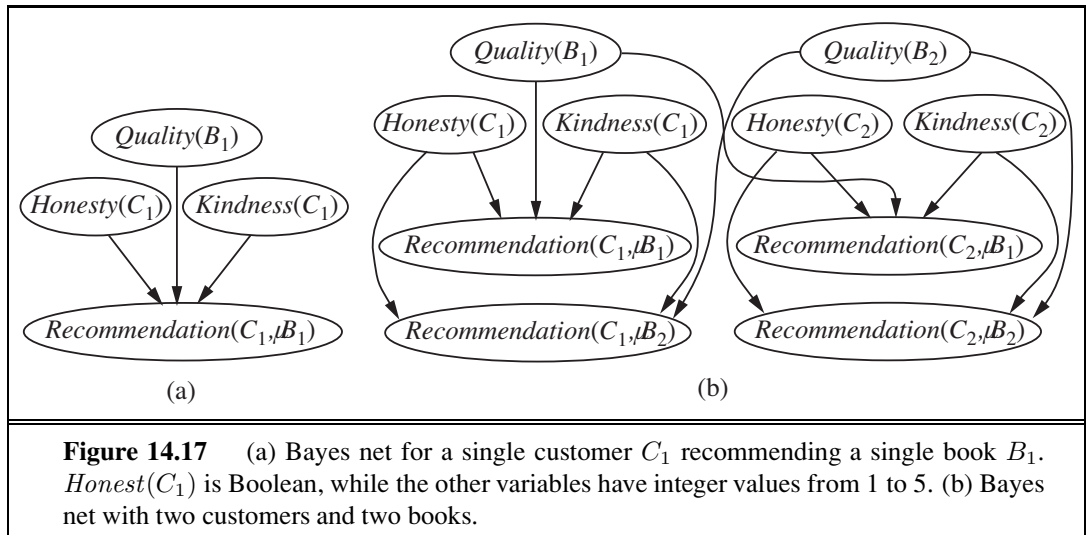
The final step is to show how to perform the general Gibbs sampling step—sampling X_i from $\mathbf{P}(X_i | \bar{\mathbf{x}}_i, \mathbf{e})$ —in a Bayesian network. Recall from page 517 that a variable is independent of all other variables given its Markov blanket; hence,

$$P(x'_i | \bar{\mathbf{x}}_i, \mathbf{e}) = P(x'_i | mb(X_i)) ,$$

where $mb(X_i)$ denotes the values of the variables in X_i 's Markov blanket, $MB(X_i)$. As shown in Exercise 14.7, 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:

$$P(x'_i | mb(X_i)) = \alpha P(x'_i | \text{parents}(X_i)) \times \prod_{Y_j \in \text{Children}(X_i)} P(y_j | \text{parents}(Y_j)) . \quad (14.12)$$

Hence, to flip each variable X_i conditioned on its Markov blanket, the number of multiplications required is equal to the number of X_i 's children.



14.6 RELATIONAL AND FIRST-ORDER PROBABILITY MODELS



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 random 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.*

For example, suppose that an online book retailer would like to provide overall evaluations of products based on recommendations received from its customers. The evaluation will take the form of a posterior distribution over the quality of the book, given the available evidence. The simplest solution to base the evaluation on the average recommendation, perhaps with a variance determined by the number of recommendations, but this fails to take into account the fact that some customers are kinder than others and some are less honest than others. Kind customers tend to give high recommendations even to fairly mediocre books, while dishonest customers give very high or very low recommendations for reasons other than quality—for example, they might work for a publisher.⁶

For a single customer C_1 , recommending a single book B_1 , the Bayes net might look like the one shown in Figure 14.17(a). (Just as in Section 9.1, expressions with parentheses such as $Honest(C_1)$ are just fancy symbols—in this case, fancy names for random variables.)

⁶ A game theorist would advise a dishonest customer to avoid detection by occasionally recommending a good book from a competitor. See Chapter 17.

With two customers and two books, the Bayes net looks like the one in Figure 14.17(b). For larger numbers of books and customers, it becomes completely impractical to specify the network by hand.

Fortunately, the network has a lot of repeated structure. Each $Recommendation(c, b)$ variable has as its parents the variables $Honest(c)$, $Kindness(c)$, and $Quality(b)$. Moreover, the CPTs for all the $Recommendation(c, b)$ variables are identical, as are those for all the $Honest(c)$ variables, and so on. The situation seems tailor-made for a first-order language. We would like to say something like

$$Recommendation(c, b) \sim RecCPT(Honest(c), Kindness(c), Quality(b))$$

with the intended meaning that a customer's recommendation for a book depends on the customer's honesty and kindness and the book's quality according to some fixed CPT. This section develops a language that lets us say exactly this, and a lot more besides.

14.6.1 Possible worlds

Recall from Chapter 13 that a probability model defines a set Ω of possible worlds with a probability $P(\omega)$ for each world ω . For Bayesian networks, the possible worlds are assignments of values to variables; for the Boolean case in particular, the possible worlds are identical to those of propositional logic. For a first-order probability model, then, it seems we need the possible worlds to be those of first-order logic—that is, a set of objects with relations among them and an interpretation that maps constant symbols to objects, predicate symbols to relations, and function symbols to functions on those objects. (See Section 8.2.) The model also needs to define a probability for each such possible world, just as a Bayesian network defines a probability for each assignment of values to variables.

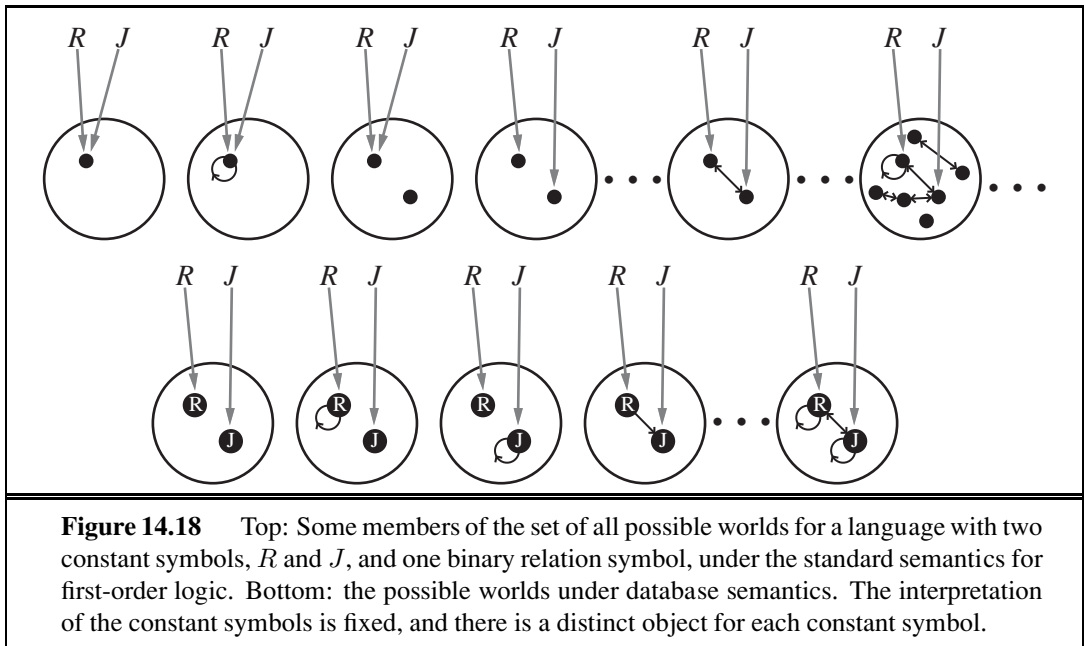
Let us suppose, for a moment, that we have figured out how to do this. Then, as usual (see page 485), we can obtain the probability of any first-order logical sentence ϕ as a sum over the possible worlds where it is true:

$$P(\phi) = \sum_{\omega: \phi \text{ is true in } \omega} P(\omega) . \quad (14.13)$$

Conditional probabilities $P(\phi | \mathbf{e})$ can be obtained similarly, so we can, in principle, ask any question we want of our model—e.g., “Which books are most likely to be recommended highly by dishonest customers?”—and get an answer. So far, so good.

There is, however, a problem: the set of first-order models is infinite. We saw this explicitly in Figure 8.4 on page 293, which we show again in Figure 14.18 (top). This means that (1) the summation in Equation (14.13) could be infeasible, and (2) specifying a complete, consistent distribution over an infinite set of worlds could be very difficult.

Section 14.6.2 explores one approach to dealing with this problem. The idea is to borrow not from the standard semantics of first-order logic but from the **database semantics** defined in Section 8.2.8 (page 299). The database semantics makes the **unique names assumption**—here, we adopt it for the constant symbols. It also assumes **domain closure**—there are no more objects than those that are named. We can then guarantee a finite set of possible worlds by making the set of objects in each world be exactly the set of constant



symbols that are used; as shown in Figure 14.18 (bottom), there is no uncertainty about the mapping from symbols to objects or about the objects that exist. We will call models defined in this way **relational probability models**, or RPMs.⁷ The most significant difference between the semantics of RPMs and the database semantics introduced in Section 8.2.8 is that RPMs do not make the closed-world assumption—obviously, assuming that every unknown fact is false doesn’t make sense in a probabilistic reasoning system!

When the underlying assumptions of database semantics fail to hold, RPMs won’t work well. For example, a book retailer might use an ISBN (International Standard Book Number) as a constant symbol to name each book, even though a given “logical” book (e.g., “Gone With the Wind”) may have several ISBNs. It would make sense to aggregate recommendations across multiple ISBNs, but the retailer may not know for sure which ISBNs are really the same book. (Note that we are not reifying the *individual copies* of the book, which might be necessary for used-book sales, car sales, and so on.) Worse still, each customer is identified by a login ID, but a dishonest customer may have thousands of IDs! In the computer security field, these multiple IDs are called **sibyls** and their use to confound a reputation system is called a **sibyl attack**. Thus, even a simple application in a relatively well-defined, online domain involves both **existence uncertainty** (what are the real books and customers underlying the observed data) and **identity uncertainty** (which symbol really refer to the same object). We need to bite the bullet and define probability models based on the standard semantics of first-order logic, for which the possible worlds vary in the objects they contain and in the mappings from symbols to objects. Section 14.6.3 shows how to do this.

⁷ The name *relational probability model* was given by Pfeffer (2000) to a slightly different representation, but the underlying ideas are the same.

14.6.2 Relational probability models

TYPE SIGNATURE

Like first-order logic, RPMs have constant, function, and predicate symbols. (It turns out to be easier to view predicates as functions that return *true* or *false*.) We will also assume a **type signature** for each function, that is, a specification of the type of each argument and the function's value. If the type of each object is known, many spurious possible worlds are eliminated by this mechanism. For the book-recommendation domain, the types are *Customer* and *Book*, and the type signatures for the functions and predicates are as follows:

$$Honest : Customer \rightarrow \{true, false\} \quad Kindness : Customer \rightarrow \{1, 2, 3, 4, 5\}$$

$$Quality : Book \rightarrow \{1, 2, 3, 4, 5\}$$

$$Recommendation : Customer \times Book \rightarrow \{1, 2, 3, 4, 5\}$$

The constant symbols will be whatever customer and book names appear in the retailer's data set. In the example given earlier (Figure 14.17(b)), these were C_1 , C_2 and B_1 , B_2 .

Given the constants and their types, together with the functions and their type signatures, the random variables of the RPM are obtained by instantiating each function with each possible combination of objects: $Honest(C_1)$, $Quality(B_2)$, $Recommendation(C_1, B_2)$, and so on. These are exactly the variables appearing in Figure 14.17(b). Because each type has only finitely many instances, the number of basic random variables is also finite.

To complete the RPM, we have to write the dependencies that govern these random variables. There is one dependency statement for each function, where each argument of the function is a logical variable (i.e., a variable that ranges over objects, as in first-order logic):

$$Honest(c) \sim \langle 0.99, 0.01 \rangle$$

$$Kindness(c) \sim \langle 0.1, 0.1, 0.2, 0.3, 0.3 \rangle$$

$$Quality(b) \sim \langle 0.05, 0.2, 0.4, 0.2, 0.15 \rangle$$

$$Recommendation(c, b) \sim RecCPT(Honest(c), Kindness(c), Quality(b))$$

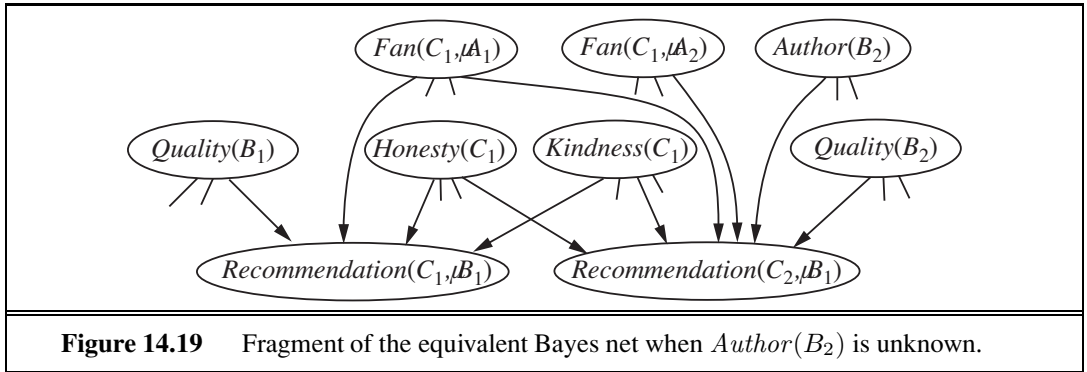
where *RecCPT* is a separately defined conditional distribution with $2 \times 5 \times 5 = 50$ rows, each with 5 entries. The semantics of the RPM can be obtained by instantiating these dependencies for all known constants, giving a Bayesian network (as in Figure 14.17(b)) that defines a joint distribution over the RPM's random variables.⁸

CONTEXT-SPECIFIC
INDEPENDENCE

We can refine the model by introducing a **context-specific independence** to reflect the fact that dishonest customers ignore quality when giving a recommendation; moreover, kindness plays no role in their decisions. A context-specific independence allows a variable to be independent of some of its parents given certain values of others; thus, $Recommendation(c, b)$ is independent of $Kindness(c)$ and $Quality(b)$ when $Honest(c) = false$:

$$Recommendation(c, b) \sim \begin{array}{ll} \text{if } Honest(c) & \text{then} \\ & HonestRecCPT(Kindness(c), Quality(b)) \\ \text{else} & \langle 0.4, 0.1, 0.0, 0.1, 0.4 \rangle \end{array} .$$

⁸ Some technical conditions must be observed to guarantee that the RPM defines a proper distribution. First, the dependencies must be *acyclic*, otherwise the resulting Bayesian network will have cycles and will not define a proper distribution. Second, the dependencies must be *well-founded*, that is, there can be no infinite ancestor chains, such as might arise from recursive dependencies. Under some circumstances (see Exercise 14.6), a fixed-point calculation yields a well-defined probability model for a recursive RPM.



This kind of dependency may look like an ordinary if–then–else statement on a programming language, but there is a key difference: the inference engine *doesn't necessarily know the value of the conditional test*!

We can elaborate this model in endless ways to make it more realistic. For example, suppose that an honest customer who is a fan of a book's author always gives the book a 5, regardless of quality:

$$\begin{aligned}
 Recommendation(c, b) \sim & \text{ if } Honest(c) \text{ then} \\
 & \text{ if } Fan(c, Author(b)) \text{ then Exactly}(5) \\
 & \text{ else } HonestRecCPT(Kindness(c), Quality(b)) \\
 & \text{ else } \langle 0.4, 0.1, 0.0, 0.1, 0.4 \rangle
 \end{aligned}$$

Again, the conditional test $Fan(c, Author(b))$ is unknown, but if a customer gives only 5s to a particular author's books and is not otherwise especially kind, then the posterior probability that the customer is a fan of that author will be high. Furthermore, the posterior distribution will tend to discount the customer's 5s in evaluating the quality of that author's books.

In the preceding example, we implicitly assumed that the value of $Author(b)$ is known for every b , but this may not be the case. How can the system reason about whether, say, C_1 is a fan of $Author(B_2)$ when $Author(B_2)$ is unknown? The answer is that the system may have to reason about *all possible authors*. Suppose (to keep things simple) that there are just two authors, A_1 and A_2 . Then $Author(B_2)$ is a random variable with two possible values, A_1 and A_2 , and it is a parent of $Recommendation(C_1, B_2)$. The variables $Fan(C_1, A_1)$ and $Fan(C_1, A_2)$ are parents too. The conditional distribution for $Recommendation(C_1, B_2)$ is then essentially a **multiplexer** in which the $Author(B_2)$ parent acts as a selector to choose which of $Fan(C_1, A_1)$ and $Fan(C_1, A_2)$ actually gets to influence the recommendation. A fragment of the equivalent Bayes net is shown in Figure 14.19. Uncertainty in the value of $Author(B_2)$, which affects the dependency structure of the network, is an instance of **relational uncertainty**.

In case you are wondering how the system can possibly work out who the author of B_2 is: consider the possibility that three other customers are fans of A_1 (and have no other favorite authors in common) and all three have given B_2 a 5, even though most other customers find it quite dismal. In that case, it is extremely likely that A_1 is the author of B_2 .

MULTIPLEXER

RELATIONAL
UNCERTAINTY

The emergence of sophisticated reasoning like this from an RPM model of just a few lines is an intriguing example of how probabilistic influences spread through the web of interconnections among objects in the model. As more dependencies and more objects are added, the picture conveyed by the posterior distribution often becomes clearer and clearer.

UNROLLING

The next question is how to do inference in RPMs. One approach is to collect the evidence and query and the constant symbols therein, construct the equivalent Bayes net, and apply any of the inference methods discussed in this chapter. This technique is called **unrolling**. The obvious drawback is that the resulting Bayes net may be very large. Furthermore, if there are many candidate objects for an unknown relation or function—for example, the unknown author of B_2 —then some variables in the network may have many parents.

Fortunately, much can be done to improve on generic inference algorithms. First, the presence of repeated substructure in the unrolled Bayes net means that many of the factors constructed during variable elimination (and similar kinds of tables constructed by clustering algorithms) will be identical; effective caching schemes have yielded speedups of three orders of magnitude for large networks. Second, inference methods developed to take advantage of context-specific independence in Bayes nets find many applications in RPMs. Third, MCMC inference algorithms have some interesting properties when applied to RPMs with relational uncertainty. MCMC works by sampling complete possible worlds, so in each state the relational structure is completely known. In the example given earlier, each MCMC state would specify the value of $Author(B_2)$, and so the other potential authors are no longer parents of the recommendation nodes for B_2 . For MCMC, then, relational uncertainty causes no increase in network complexity; instead, the MCMC process includes transitions that change the relational structure, and hence the dependency structure, of the unrolled network.

All of the methods just described assume that the RPM has to be partially or completely unrolled into a Bayesian network. This is exactly analogous to the method of **proposition-alization** for first-order logical inference. (See page 322.) Resolution theorem-provers and logic programming systems avoid propositionalizing by instantiating the logical variables only as needed to make the inference go through; that is, they *lift* the inference process above the level of ground propositional sentences and make each lifted step do the work of many ground steps. The same idea applied in probabilistic inference. For example, in the variable elimination algorithm, a lifted factor can represent an entire set of ground factors that assign probabilities to random variables in the RPM, where those random variables differ only in the constant symbols used to construct them. The details of this method are beyond the scope of this book, but references are given at the end of the chapter.

14.6.3 Open-universe probability models

We argued earlier that database semantics was appropriate for situations in which we know exactly the set of relevant objects that exist and can identify them unambiguously. (In particular, all observations about an object are correctly associated with the constant symbol that names it.) In many real-world settings, however, these assumptions are simply untenable. We gave the examples of multiple ISBNs and sibyl attacks in the book-recommendation domain (to which we will return in a moment), but the phenomenon is far more pervasive:

- A vision system doesn't know what exists, if anything, around the next corner, and may not know if the object it sees now is the same one it saw a few minutes ago.
- A text-understanding system does not know in advance the entities that will be featured in a text, and must reason about whether phrases such as “Mary,” “Dr. Smith,” “she,” “his cardiologist,” “his mother,” and so on refer to the same object.
- An intelligence analyst hunting for spies never knows how many spies there really are and can only guess whether various pseudonyms, phone numbers, and sightings belong to the same individual.

In fact, a major part of human cognition seems to require learning what objects exist and being able to connect observations—which almost never come with unique IDs attached—to hypothesized objects in the world.

OPEN UNIVERSE

For these reasons, we need to be able to write so-called **open-universe** probability models or OUPMs based on the standard semantics of first-order logic, as illustrated at the top of Figure 14.18. A language for OUPMs provides a way of writing such models easily while guaranteeing a unique, consistent probability distribution over the infinite space of possible worlds.

The basic idea is to understand how ordinary Bayesian networks and RPMs manage to define a unique probability model and to transfer that insight to the first-order setting. In essence, a Bayes net *generates* each possible world, event by event, in the topological order defined by the network structure, where each event is an assignment of a value to a variable. An RPM extends this to entire sets of events, defined by the possible instantiations of the logical variables in a given predicate or function. OUPMs go further by allowing generative steps that *add objects* to the possible world under construction, where the number and type of objects may depend on the objects that are already in that world. That is, the event being generated is not the assignment of a value to a variable, but the very *existence* of objects.

One way to do this in OUPMs is to add statements that define conditional distributions over the numbers of objects of various kinds. For example, in the book-recommendation domain, we might want to distinguish between *customers* (real people) and their *login IDs*. Suppose we expect somewhere between 100 and 10,000 distinct customers (whom we cannot observe directly). We can express this as a prior log-normal distribution⁹ as follows:

$$\# \text{ Customer} \sim \text{LogNormal}[6.9, 2.3^2]() .$$

We expect honest customers to have just one ID, whereas dishonest customers might have anywhere between 10 and 1000 IDs:

$$\# \text{ LoginID}(\text{Owner} = c) \sim \begin{array}{ll} \text{if } \text{Honest}(c) \text{ then } \text{Exactly}(1) \\ \text{else } \text{LogNormal}[6.9, 2.3^2]() . \end{array}$$

ORIGIN FUNCTION

This statement defines the number of login IDs for a given owner, who is a customer. The *Owner* function is called an **origin function** because it says where each generated object came from. In the formal semantics of BLOG (as distinct from first-order logic), the domain elements in each possible world are actually generation histories (e.g., “the fourth login ID of the seventh customer”) rather than simple tokens.

⁹ A distribution $\text{LogNormal}[\mu, \sigma^2](x)$ is equivalent to a distribution $N[\mu, \sigma^2](x)$ over $\log_e(x)$.

Subject to technical conditions of acyclicity and well-foundedness similar to those for RPMs, open-universe models of this kind define a unique distribution over possible worlds. Furthermore, there exist inference algorithms such that, for every such well-defined model and every first-order query, the answer returned approaches the true posterior arbitrarily closely in the limit. There are some tricky issues involved in designing these algorithms. For example, an MCMC algorithm cannot sample directly in the space of possible worlds when the size of those worlds is unbounded; instead, it samples finite, partial worlds, relying on the fact that only finitely many objects can be relevant to the query in distinct ways. Moreover, transitions must allow for merging two objects into one or splitting one into two. (Details are given in the references at the end of the chapter.) Despite these complications, the basic principle established in Equation (14.13) still holds: the probability of any sentence is well defined and can be calculated.

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 those mentioned above—computer vision, text understanding, and intelligence analysis—as well as many other kinds of sensor interpretation.

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 reason-

ing mechanisms can be built directly on top of probability theory, so 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 12.

- **Rule-based** approaches to uncertainty have also 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 for heads is again reasonable. Obviously, the two cases are different, yet the outcome 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.
- Probability makes the same ontological commitment as logic: that propositions 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 a proposition can be “sort of” true. Vagueness and uncertainty are in fact orthogonal issues.

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.

14.7.1 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.

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 14.12(a) (page 529). If we wanted to be able to do both causal and diagnostic reasoning, we would need the two rules

$$Rain \mapsto WetGrass \quad \text{and} \quad WetGrass \mapsto Rain .$$

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

$$Sprinkler \mapsto WetGrass \quad \text{and} \quad WetGrass \mapsto Rain .$$

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 can truth-functional systems be made useful in practice? 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. For these reasons, Bayesian networks have largely supplanted rule-based methods for uncertain reasoning.

14.7.2 Representing ignorance: Dempster–Shafer theory

DEMPSTER-SHAFFER
THEORY

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 you pick a coin from a magician’s pocket. 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.

MASS

The mathematical underpinnings of Dempster–Shafer theory have a similar flavor to those of probability theory; the main difference is that, instead of assigning probabilities to possible worlds, the theory assigns **masses** to *sets* of possible world, that is, to events. The masses still must add to 1 over all possible events. $Bel(A)$ is defined to be the sum of masses for all events that are subsets of (i.e., that entail) A , including A itself. With this definition, $Bel(A)$ and $Bel(\neg A)$ sum to *at most* 1, and the gap—the interval between $Bel(A)$ and $1 - Bel(\neg A)$ —is often interpreted as bounding the probability of A .

As with default reasoning, there is a problem in connecting beliefs to actions. Whenever there is a gap in the beliefs, then a decision problem can be defined such that a Dempster–Shafer system is unable to make a decision. In fact, the notion of utility in the Dempster–Shafer model is not yet well understood because the meanings of masses and beliefs themselves have yet to be understood. Pearl (1988) has argued that $Bel(A)$ should be interpreted not as a degree of belief in A but as the probability assigned to all the possible worlds (now interpreted as logical theories) in which A is *provable*. While there are cases in which this quantity might be of interest, it is not the same as the probability that A is true.

A Bayesian analysis of the coin-flipping example would suggest that no new formalism is necessary to handle such cases. The model would have two variables: the *Bias* of the coin (a number between 0 and 1, where 0 is a coin that always shows tails and 1 a coin that always shows heads) and the outcome of the next *Flip*. The prior probability distribution for *Bias*

would reflect our beliefs based on the source of the coin (the magician's pocket): some small probability that it is fair and some probability that it is heavily biased toward heads or tails. The conditional distribution $\mathbf{P}(\text{Flip} \mid \text{Bias})$ simply defines how the bias operates. If $\mathbf{P}(\text{Bias})$ is symmetric about 0.5, then our prior probability for the flip is

$$P(\text{Flip} = \text{heads}) = \int_0^1 P(\text{Bias} = x)P(\text{Flip} = \text{heads} \mid \text{Bias} = x) dx = 0.5 .$$

This is the same prediction as if we believe strongly that the coin is fair, but that does *not* mean that probability theory treats the two situations identically. The difference arises *after* the flips in computing the posterior distribution for *Bias*. If the coin came from a bank, then seeing it come up heads three times running would have almost no effect on our strong prior belief in its fairness; but if the coin comes from the magician's pocket, the same evidence will lead to a stronger posterior belief that the coin is biased toward heads. Thus, a Bayesian approach expresses our “ignorance” in terms of how our beliefs would change in the face of future information gathering.

14.7.3 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

$$\begin{aligned} T(A \wedge B) &= \min(T(A), T(B)) \\ T(A \vee B) &= \max(T(A), T(B)) \\ T(\neg A) &= 1 - T(A) . \end{aligned}$$

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 $T(Tall(Nate) \wedge Heavy(Nate)) = 0.4$, 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 $P(\text{Observer says Nate is tall} \mid \text{Height})$, perhaps using a **probit distribution** as described on page 522. 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

$$P(\text{Observer says Nate is tall and heavy} \mid \text{Height, Weight})$$

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.

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 an explicitly enumerated 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. **Open-universe probability models** handle **existence** and **identity uncertainty**, defining probability distributions over the infinite space of first-order possible worlds.
- 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). 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 poly-tree networks (Kim and Pearl, 1983) and explained the importance of causal rather than diagnostic probability models, in contrast to the certainty-factor systems then in vogue.

The first expert system using Bayesian networks was CONVINCER (Kim, 1983). Early applications in medicine included the MUNIN system for diagnosing neuromuscular disorders (Andersen *et al.*, 1989) and the PATHFINDER system for pathology (Heckerman, 1991). The CPCS system (Pradhan *et al.*, 1994) is a Bayesian network for internal medicine consisting

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.

of 448 nodes, 906 links and 8,254 conditional probability values. (The front cover shows a portion of the network.)

Applications in engineering include the Electric Power Research Institute's work on monitoring power generators (Morjaria *et al.*, 1995), NASA's work on displaying time-critical information at Mission Control in Houston (Horvitz and Barry, 1995), and the general field of **network tomography**, which aims to infer unobserved local properties of nodes and links in the Internet from observations of end-to-end message performance (Castro *et al.*, 2004). Perhaps 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). Another important application area is biology: Bayesian networks have been used for identifying human genes by reference to mouse genes (Zhang *et al.*, 2003), inferring cellular networks Friedman (2004), and many other tasks in bioinformatics. We could go on, but instead we'll refer you to Pourret *et al.* (2008), a 400-page guide to applications of Bayesian networks.

Ross Shachter (1986), working in the influence diagram community, developed the first complete algorithm for general Bayesian networks. His method was based on goal-directed reduction of the network using posterior-preserving transformations. 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 (Lauritzen and Spiegelhalter, 1988), is based on conversion to an undirected form of graphical model called a **Markov network**. This approach is implemented in the HUGIN system, an efficient and widely used tool for uncertain reasoning (Andersen *et al.*, 1989). Boutilier *et al.* (1996) show how to exploit context-specific independence in clustering algorithms.

The basic idea of variable elimination—that repeated computations within the overall sum-of-products expression can be avoided by caching—appeared in the symbolic probabilistic inference (SPI) algorithm (Shachter *et al.*, 1990). The elimination algorithm we describe is closest to that developed by Zhang and Poole (1994). 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. 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 likely 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 tree width of the network. Wexler and Meek (2009) describe a method of preventing exponential growth in the size of factors computed in variable elimination; their algorithm breaks down large factors into products of smaller factors and simultaneously computes an error bound for the resulting approximation.

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

MARKOV NETWORK

NONSERIAL DYNAMIC
PROGRAMMING

cHUGIN system (Olesen, 1993). Further analysis of linear Gaussian models, with connections to many other models used in statistics, appears in Roweis and Ghahramani (1999). The probit distribution is usually attributed to Gaddum (1933) and Bliss (1934), although it had been discovered several times in the 19th century. Bliss's work was expanded considerably by Finney (1947). The probit has been used widely for modeling discrete choice phenomena and can be extended to handle more than two choices (Daganzo, 1979). The logit model was introduced by Berkson (1944); initially much derided, it eventually became more popular than the probit model. Bishop (1995) gives a simple justification for its use.

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 6, 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**. 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).

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** λ that are adjusted to minimize a distance function D between the original and the reduced problem, often by solving the system of equations $\partial D / \partial \lambda = 0$. 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

VARIATIONAL
APPROXIMATION

VARIATIONAL
PARAMETER

MEAN FIELD

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 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. Since these early papers, variational methods have been applied to many specific families of models. The remarkable paper by Wainwright and Jordan (2008) provides a unifying theoretical analysis of the literature on variational methods.

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 BP) 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 BP 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) presented a promising empirical study of BP's performance, and Weiss and Freeman (2001) established strong convergence results for BP on linear Gaussian networks. Weiss (2000b) shows how an approximation called loopy belief propagation works, and when the approximation is correct. Yedidia *et al.* (2005) made further connections between loopy propagation and ideas from statistical physics.

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). The basic idea is that each sentence in the knowledge base expressed a *constraint* on the distribution over possible worlds; one sentence entails another if it expresses a stronger constraint. For example, the sentence $\forall x \ P(Hungry(x)) > 0.2$ rules out distributions in which any object is hungry with probability less than 0.2; thus, it entails the sentence $\forall x \ P(Hungry(x)) > 0.1$. It turns out that writing a *consistent* set of sentences in these languages is quite difficult and constructing a unique probability model nearly impossible unless one adopts the representation approach of Bayesian networks by writing suitable sentences about conditional probabilities.

Beginning in the early 1990s, researchers working on complex applications noticed the expressive limitations of Bayesian networks and developed various languages for writing "templates" with logical variables, from which large networks could be constructed automatically for each problem instance (Breese, 1992; Wellman *et al.*, 1992). The most important such language was BUGS (Bayesian inference Using Gibbs Sampling) (Gilks *et al.*, 1994), which combined Bayesian networks with the **indexed random variable** notation common in

BELIEF
PROPAGATION

TURBO DECODING

INDEXED RANDOM
VARIABLE

statistics. (In BUGS, an indexed random variable looks like $X[i]$, where i has a defined integer range.) These languages inherited the key property of Bayesian networks: every well-formed knowledge base defines a unique, consistent probability model. Languages with well-defined semantics based on unique names and domain closure drew on the representational capabilities of logic programming (Poole, 1993; Sato and Kameya, 1997; Kersting *et al.*, 2000) and semantic networks (Koller and Pfeffer, 1998; Pfeffer, 2000). Pfeffer (2007) went on to develop IBAL, which represents first-order probability models as probabilistic programs in a programming language extended with a randomization primitive. Another important thread was the combination of relational and first-order notations with (undirected) Markov networks (Taskar *et al.*, 2002; Domingos and Richardson, 2004), where the emphasis has been less on knowledge representation and more on learning from large data sets.

Initially, inference in these models was performed by generating an equivalent Bayesian network. Pfeffer *et al.* (1999) introduced a variable elimination algorithm that cached each computed factor for reuse by later computations involving the same relations but different objects, thereby realizing some of the computational gains of lifting. The first truly lifted inference algorithm was a lifted form of variable elimination described by Poole (2003) and subsequently improved by de Salvo Braz *et al.* (2007). Further advances, including cases where certain aggregate probabilities can be computed in closed form, are described by Milch *et al.* (2008) and Kisynski and Poole (2009). Pasula and Russell (2001) studied the application of MCMC to avoid building the complete equivalent Bayes net in cases of relational and identity uncertainty. Getoor and Taskar (2007) collect many important papers on first-order probability models and their use in machine learning.

RECORD LINKAGE

Probabilistic reasoning about identity uncertainty has two distinct origins. In statistics, the problem of **record linkage** arises when data records do not contain standard unique identifiers—for example, various citations of this book might name its first author “Stuart Russell” or “S. J. Russell” or even “Stewart Russle,” and other authors may use the some of the same names. Literally hundreds of companies exist solely to solve record linkage problems in financial, medical, census, and other data. Probabilistic analysis goes back to work by Dunn (1946); the Fellegi–Sunter model (1969), which is essentially naive Bayes applied to matching, still dominates current practice. The second origin for work on identity uncertainty is multitarget tracking (Sittler, 1964), which we cover in Chapter 15. For most of its history, work in symbolic AI assumed erroneously that sensors could supply sentences with unique identifiers for objects. The issue was studied in the context of language understanding by Charniak and Goldman (1992) and in the context of surveillance by (Huang and Russell, 1998) and Pasula *et al.* (1999). Pasula *et al.* (2003) developed a complex generative model for authors, papers, and citation strings, involving both relational and identity uncertainty, and demonstrated high accuracy for citation information extraction. The first formally defined language for open-universe probability models was BLOG (Milch *et al.*, 2005), which came with a complete (albeit slow) MCMC inference algorithm for all well-defined models. (The program code faintly visible on the front cover of this book is part of a BLOG model for detecting nuclear explosions from seismic signals as part of the UN Comprehensive Test Ban Treaty verification regime.) Laskey (2008) describes another open-universe modeling language called **multi-entity Bayesian networks**.

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. Pearl (1988) and Ruspini *et al.* (1992) analyze the relationship between the Dempster–Shafer theory and standard probability theory.

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) survey the connections between possibility theory and probability theory.

The resurgence of probability depended mainly on Pearl’s development 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. Eugene Charniak helped present the ideas to AI researchers with a popular article, “Bayesian networks without tears”¹¹ (1991), and book (1993). The book by Dean and Wellman (1991) also helped introduce Bayesian networks to AI researchers. One of the principal philosophical objections of the logicians 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. Goldszmidt and Pearl (1996) take a similar approach. 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. Often, programs speak louder than words, and the ready avail-

¹¹ The title of the original version of the article was “Pearl for swine.”

ability of high-quality software such as the Bayes Net toolkit (Murphy, 2001) accelerated the adoption of the technology.

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 (Lauritzen, 1996; Jensen, 2001; Korb and Nicholson, 2003; Jensen, 2007; Darwiche, 2009; Koller and Friedman, 2009) provide thorough treatments of the topics we have covered in this chapter. 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

14.1 We have a bag of three biased coins a , b , and c with probabilities of coming up heads of 20%, 60%, and 80%, respectively. One coin is drawn randomly from the bag (with equal likelihood of drawing each of the three coins), and then the coin is flipped three times to generate the outcomes X_1 , X_2 , and X_3 .

- a. Draw the Bayesian network corresponding to this setup and define the necessary CPTs.
- b. Calculate which coin was most likely to have been drawn from the bag if the observed flips come out heads twice and tails once.

14.2 Equation (14.1) on page 513 defines the joint distribution represented by a Bayesian network in terms of the parameters $\theta(X_i | \text{Parents}(X_i))$. This exercise asks you to derive the equivalence between the parameters and the conditional probabilities $\mathbf{P}(X_i | \text{Parents}(X_i))$ from this definition.

- a. Consider a simple network $X \rightarrow Y \rightarrow Z$ with three Boolean variables. Use Equations (13.3) and (13.6) (pages 485 and 492) to express the conditional probability $P(z | y)$ as the ratio of two sums, each over entries in the joint distribution $\mathbf{P}(X, Y, Z)$.
- b. Now use Equation (14.1) to write this expression in terms of the network parameters $\theta(X)$, $\theta(Y | X)$, and $\theta(Z | Y)$.
- c. Next, expand out the summations in your expression from part (b), writing out explicitly the terms for the true and false values of each summed variable. Assuming that all network parameters satisfy the constraint $\sum_{x_i} \theta(x_i | \text{parents}(X_i)) = 1$, show that the resulting expression reduces to $\theta(x | y)$.
- d. Generalize this derivation to show that $\theta(X_i | \text{Parents}(X_i)) = \mathbf{P}(X_i | \text{Parents}(X_i))$ for any Bayesian network.

ARC REVERSAL

14.3 The operation of **arc reversal** in a Bayesian network allows us to change the direction of an arc $X \rightarrow Y$ while preserving the joint probability distribution that the network represents (Shachter, 1986). Arc reversal may require introducing new arcs: all the parents of X also become parents of Y , and all parents of Y also become parents of X .

- Assume that X and Y start with m and n parents, respectively, and that all variables have k values. By calculating the change in size for the CPTs of X and Y , show that the total number of parameters in the network cannot decrease during arc reversal. (*Hint: the parents of X and Y need not be disjoint.*)
- Under what circumstances can the total number remain constant?
- Let the parents of X be $\mathbf{U} \cup \mathbf{V}$ and the parents of Y be $\mathbf{V} \cup \mathbf{W}$, where \mathbf{U} and \mathbf{W} are disjoint. The formulas for the new CPTs after arc reversal are as follows:

$$\mathbf{P}(Y | \mathbf{U}, \mathbf{V}, \mathbf{W}) = \sum_x \mathbf{P}(Y | \mathbf{V}, \mathbf{W}, x) \mathbf{P}(x | \mathbf{U}, \mathbf{V})$$

$$\mathbf{P}(X | \mathbf{U}, \mathbf{V}, \mathbf{W}, Y) = \mathbf{P}(Y | X, \mathbf{V}, \mathbf{W}) \mathbf{P}(X | \mathbf{U}, \mathbf{V}) / \mathbf{P}(Y | \mathbf{U}, \mathbf{V}, \mathbf{W}) .$$

Prove that the new network expresses the same joint distribution over all variables as the original network.

14.4 Consider the Bayesian network in Figure 14.2.

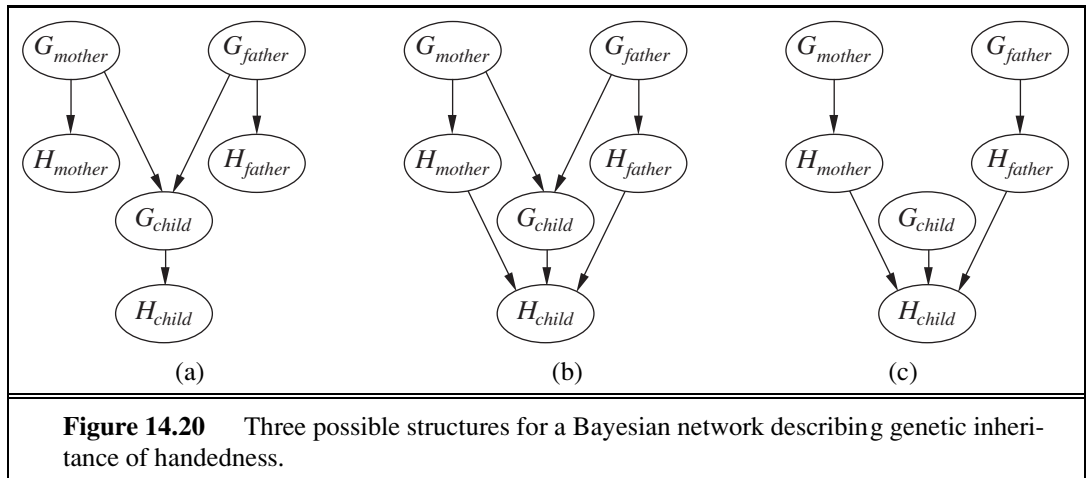
- If no evidence is observed, are *Burglary* and *Earthquake* independent? Prove this from the numerical semantics and from the topological semantics.
- If we observe $Alarm = true$, are *Burglary* and *Earthquake* independent? Justify your answer by calculating whether the probabilities involved satisfy the definition of conditional independence.

14.5 Suppose that in a Bayesian network containing an unobserved variable Y , all the variables in the Markov blanket $MB(Y)$ have been observed.

- Prove that removing the node Y from the network will not affect the posterior distribution for any other unobserved variable in the network.
- Discuss whether we can remove Y if we are planning to use (i) rejection sampling and (ii) likelihood weighting.

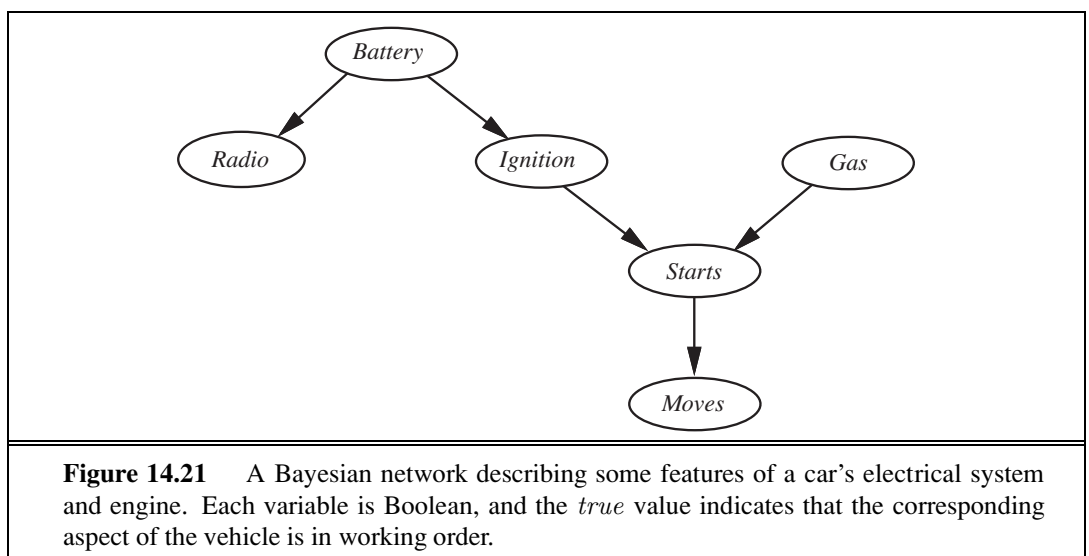
14.6 Let H_x be a random variable denoting the handedness of an individual x , with possible values l or r . A common hypothesis is that left- or right-handedness is inherited by a simple mechanism; that is, perhaps there is a gene G_x , also with values l or r , and perhaps actual handedness turns out mostly the same (with some probability s) as the gene an individual possesses. Furthermore, perhaps the gene itself is equally likely to be inherited from either of an individual's parents, with a small nonzero probability m of a random mutation flipping the handedness.

- Which of the three networks in Figure 14.20 claim that $\mathbf{P}(G_{father}, G_{mother}, G_{child}) = \mathbf{P}(G_{father})\mathbf{P}(G_{mother})\mathbf{P}(G_{child})$?
- Which of the three networks make independence claims that are consistent with the hypothesis about the inheritance of handedness?



- c. Which of the three networks is the best description of the hypothesis?
- d. Write down the CPT for the G_{child} node in network (a), in terms of s and m .
- e. Suppose that $P(G_{father} = l) = P(G_{mother} = l) = q$. In network (a), derive an expression for $P(G_{child} = l)$ in terms of m and q only, by conditioning on its parent nodes.
- f. Under conditions of genetic equilibrium, we expect the distribution of genes to be the same across generations. Use this to calculate the value of q , and, given what you know about handedness in humans, explain why the hypothesis described at the beginning of this question must be wrong.

14.7 The **Markov blanket** of a variable is defined on page 517. Prove that a variable is independent of all other variables in the network, given its Markov blanket and derive Equation (14.12) (page 538).



14.8 Consider the network for car diagnosis shown in Figure 14.21.

- 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.

14.9 Consider the family of linear Gaussian networks, as defined on page 520.

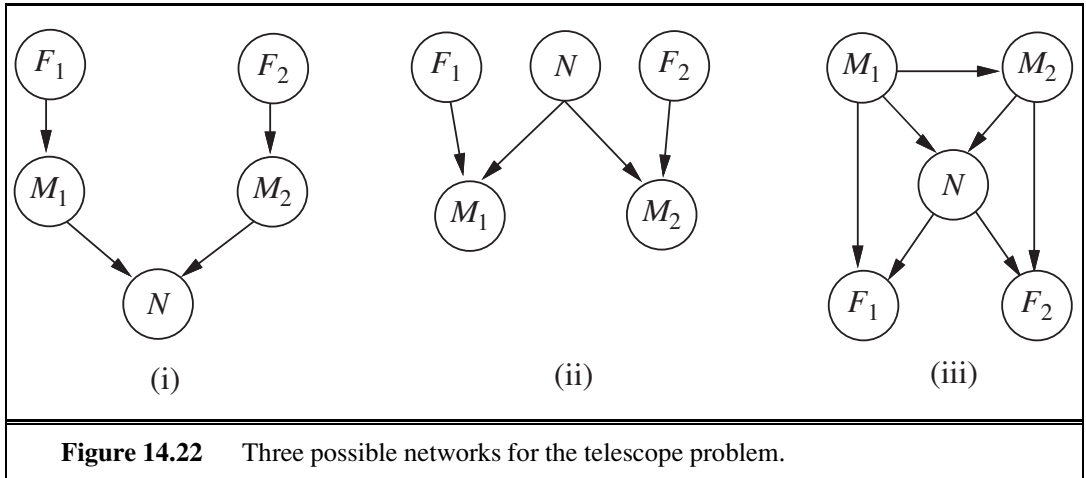
- a. In a two-variable network, let X_1 be the parent of X_2 , let X_1 have a Gaussian prior, and let $\mathbf{P}(X_2 | X_1)$ be a linear Gaussian distribution. Show that the joint distribution $P(X_1, X_2)$ is a multivariate Gaussian, and calculate its covariance matrix.
- b. Prove by induction that the joint distribution for a general linear Gaussian network on X_1, \dots, X_n is also a multivariate Gaussian.

14.10 The probit distribution defined on page 522 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.)

14.11 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? Why or why not?
- 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.



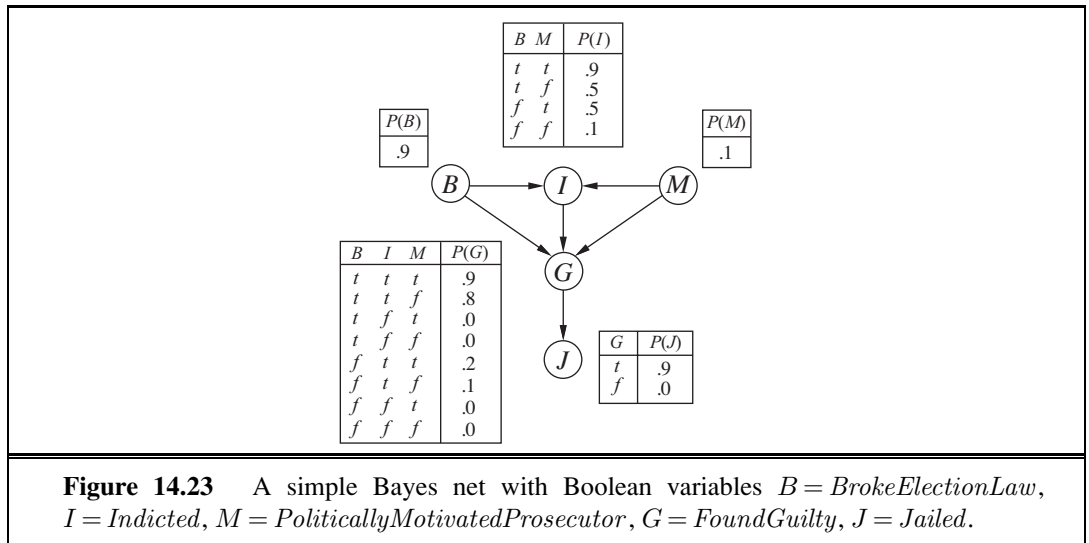
14.12 Two astronomers in different parts of the world make measurements M_1 and M_2 of the number of stars N in some small region of the sky, using their telescopes. Normally, there is a small possibility e of error by up to one star in each direction. Each telescope can also (with a much smaller probability 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 14.22.

- Which of these Bayesian networks are correct (but not necessarily efficient) representations of the preceding information?
- Which is the best network? Explain.
- Write out a conditional distribution for $\mathbf{P}(M_1 \mid N)$, 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 .
- Suppose $M_1 = 1$ and $M_2 = 3$. What are the *possible* numbers of stars if you assume no prior constraint on the values of N ?
- 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.

14.13 Consider the network shown in Figure 14.22(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 14.12. Using the enumeration algorithm (Figure 14.9 on page 525), calculate the probability distribution $\mathbf{P}(N \mid M_1 = 2, M_2 = 2)$.

14.14 Consider the Bayes net shown in Figure 14.23.

- Which of the following are asserted by the network *structure*?
 - $\mathbf{P}(B, I, M) = \mathbf{P}(B)\mathbf{P}(I)\mathbf{P}(M)$.
 - $\mathbf{P}(J \mid G) = \mathbf{P}(J \mid G, I)$.
 - $\mathbf{P}(M \mid G, B, I) = \mathbf{P}(M \mid G, B, I, J)$.



- Calculate the value of $P(b, i, \neg m, g, j)$.
- Calculate the probability that someone goes to jail given that they broke the law, have been indicted, and face a politically motivated prosecutor.
- A **context-specific independence** (see page 542) allows a variable to be independent of some of its parents given certain values of others. In addition to the usual conditional independences given by the graph structure, what context-specific independences exist in the Bayes net in Figure 14.23?
- Suppose we want to add the variable $P = \text{PresidentialPardon}$ to the network; draw the new network and briefly explain any links you add.

14.15 Consider the variable elimination algorithm in Figure 14.11 (page 528).

- Section 14.4 applies variable elimination to the query

$$\mathbf{P}(\text{Burglary} \mid \text{JohnCalls} = \text{true}, \text{MaryCalls} = \text{true}) .$$

Perform the calculations indicated and check that the answer is correct.

- Count the number of arithmetic operations performed, and compare it with the number performed by the enumeration algorithm.
- Suppose a network has the form of a *chain*: a sequence of Boolean variables X_1, \dots, X_n where $\text{Parents}(X_i) = \{X_{i-1}\}$ for $i = 2, \dots, n$. What is the complexity of computing $\mathbf{P}(X_1 \mid X_n = \text{true})$ using enumeration? Using variable elimination?
- 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.

14.16 Investigate the complexity of exact inference in general Bayesian networks:

- 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.

14.17 Consider the problem of generating a random sample from a specified distribution on a single variable. Assume you have a random number generator 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 $i \in \{1, \dots, k\}$. The **cumulative distribution** of X gives the probability that $X \in \{x_1, \dots, x_j\}$ for each possible j . (See also Appendix A.) Explain 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 run time per sample that is *constant* (i.e., independent of k).
- c. Now consider a continuous-valued variable with a parameterized 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?

14.18 Consider the query $\mathbf{P}(\text{Rain} \mid \text{Sprinkler} = \text{true}, \text{WetGrass} = \text{true})$ in Figure 14.12(a) (page 529) and how Gibbs sampling can answer it.

- a. How many states does the Markov chain have?
- b. Calculate the **transition matrix** \mathbf{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?

14.19 This exercise explores the stationary distribution for Gibbs sampling methods.

- a. The convex composition $[\alpha, q_1; 1 - \alpha, q_2]$ of q_1 and q_2 is a transition probability distribution that first chooses one of q_1 and q_2 with probabilities α and $1 - \alpha$, respectively, and then applies whichever is chosen. Prove that if q_1 and q_2 are in detailed balance with π , then their convex composition is also in detailed balance with π . (*Note*: this result justifies a variant of GIBBS-ASK in which variables are chosen at random rather than sampled in a fixed sequence.)
- b. Prove that if each of q_1 and q_2 has π as its stationary distribution, then the sequential composition $q = q_1 \circ q_2$ also has π as its stationary distribution.

14.20 The **Metropolis–Hastings** algorithm is a member of the MCMC family; as such, it is designed to generate samples \mathbf{x} (eventually) according to target probabilities $\pi(\mathbf{x})$. (Typically

CUMULATIVE
DISTRIBUTION

METROPOLIS-
HASTINGS

PROPOSAL
DISTRIBUTIONACCEPTANCE
PROBABILITY

we are interested in sampling from $\pi(\mathbf{x}) = P(\mathbf{x} | \mathbf{e})$.) Like simulated annealing, Metropolis–Hastings operates in two stages. First, it samples a new state \mathbf{x}' from a **proposal distribution** $q(\mathbf{x}' | \mathbf{x})$, given the current state \mathbf{x} . Then, it probabilistically accepts or rejects \mathbf{x}' according to the **acceptance probability**

$$\alpha(\mathbf{x}' | \mathbf{x}) = \min \left(1, \frac{\pi(\mathbf{x}')q(\mathbf{x} | \mathbf{x}')}{\pi(\mathbf{x})q(\mathbf{x}' | \mathbf{x})} \right).$$

If the proposal is rejected, the state remains at \mathbf{x} .

- a. Consider an ordinary Gibbs sampling step for a specific variable X_i . Show that this step, considered as a proposal, is guaranteed to be accepted by Metropolis–Hastings. (Hence, Gibbs sampling is a special case of Metropolis–Hastings.)
- b. Show that the two-step process above, viewed as a transition probability distribution, is in detailed balance with π .



14.21 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 for the three matches.
- 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 partially observable environments must be able to keep track of the current state, to the extent that their sensors allow. In Section 4.4 we showed a methodology for doing that: an agent maintains a **belief state** that represents which states of the world are currently possible. From the belief state and a **transition model**, the agent can predict how the world might evolve in the next time step. From the percepts observed and a **sensor model**, the agent can update the belief state. This is a pervasive idea: in Chapter 4 belief states were represented by explicitly enumerated sets of states, whereas in Chapters 7 and 11 they were represented by logical formulas. Those approaches defined belief states in terms of which world states were *possible*, but could say nothing about which states were *likely* or *unlikely*. In this chapter, we use probability theory to quantify the degree of belief in elements of the belief state.

As we show in Section 15.1, time itself is handled in the same way as in Chapter 7: a changing world is modeled using a variable for each aspect of the world state *at each point in time*. The transition and sensor models may be uncertain: the transition model describes the probability distribution of the variables at time t , given the state of the world at past times, while the sensor model describes the probability of each percept at time t , given the current state of the world. 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 examines the problems faced when keeping track of more than one thing.

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, we can make 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 recent food intake and insulin doses, 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, such as tracking the location of a robot, tracking the economic activity of a nation, and making sense of a spoken or written sequence of words. How can dynamic situations like these be modeled?

15.1.1 States and observations

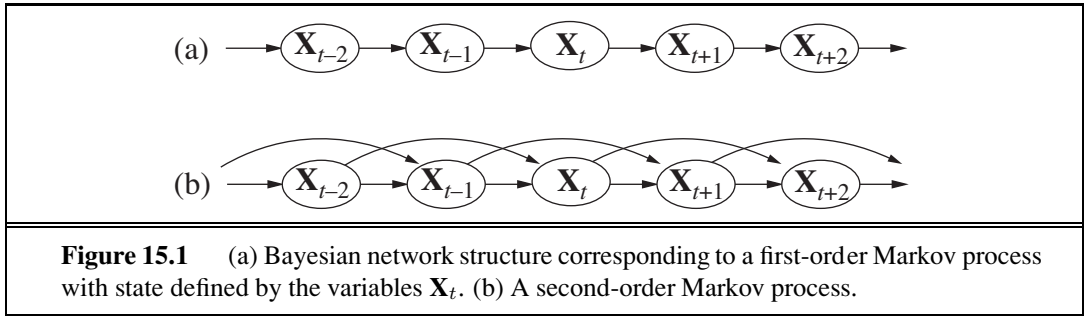
TIME SLICE

We view the world as a series of snapshots, or **time slices**, each of which contains a set of random variables, some observable and some not.¹ For simplicity, we will assume that the same subset of variables is observable in each time slice (although this is not strictly necessary in anything that follows). We will use \mathbf{X}_t to denote the set of state variables at time t , which are assumed to be unobservable, 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 .

Consider the following example: You are the security guard stationed at a 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 $Umbrella_t$ or U_t for short (whether the umbrella appears), and the set \mathbf{X}_t contains a single state variable $Rain_t$ or R_t for short (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 $StomachContents_t$. (Notice that $BloodSugar_t$ and $MeasuredBloodSugar_t$ are not the same variable; this is how we deal with noisy measurements of actual quantities.)

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 assume the interval between slices is fixed, so we can label times 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, \dots and evidence variables U_1, U_2, \dots . 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 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 .

¹ Uncertainty over *continuous* time can be modeled by **stochastic differential equations** (SDEs). The models studied in this chapter can be viewed as discrete-time approximations to SDEs.



15.1.2 Transition and sensor models

With the set of state and evidence variables for a given problem decided on, the next step is to specify how the world evolves (the transition model) and how the evidence variables get their values (the sensor model).

The transition model specifies the probability distribution over the latest state variables, given the previous values, that is, $\mathbf{P}(\mathbf{X}_t | \mathbf{X}_{0:t-1})$. Now we face a problem: the set $\mathbf{X}_{0:t-1}$ is unbounded in size as t increases. We solve the problem by making a **Markov assumption**—that the current state depends on only a *finite fixed number* of previous states. Processes satisfying this assumption were first studied in depth by the Russian statistician Andrei Markov (1856–1922) 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 provides enough information to make the future conditionally independent of the past, and we have

$$\mathbf{P}(\mathbf{X}_t | \mathbf{X}_{0:t-1}) = \mathbf{P}(\mathbf{X}_t | \mathbf{X}_{t-1}). \quad (15.1)$$

Hence, in a first-order Markov process, the transition model is the conditional distribution $\mathbf{P}(\mathbf{X}_t | \mathbf{X}_{t-1})$. 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.

Even with the Markov assumption there is still a problem: there are infinitely many possible values of t . Do we need to specify a different distribution for each time step? We avoid this problem 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 of rain, $\mathbf{P}(R_t | R_{t-1})$, is the same for all t , and we only have to specify one conditional probability table.

Now for the sensor model. The evidence variables \mathbf{E}_t *could* depend on previous variables as well as the current state variables, but any state that's worth its salt should suffice to generate the current sensor values. Thus, we make a **sensor Markov assumption** as follows:

$$\mathbf{P}(\mathbf{E}_t | \mathbf{X}_{0:t}, \mathbf{E}_{0:t-1}) = \mathbf{P}(\mathbf{E}_t | \mathbf{X}_t). \quad (15.2)$$

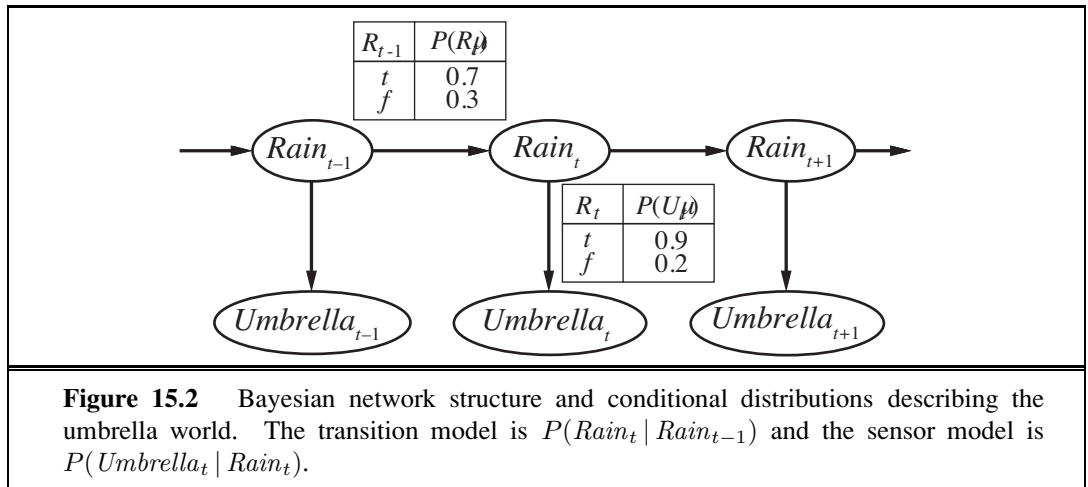
Thus, $\mathbf{P}(\mathbf{E}_t | \mathbf{X}_t)$ is our sensor model (sometimes called the **observation model**). Figure 15.2 shows both the transition model and the sensor model for the umbrella example. Notice the

MARKOV
ASSUMPTION

MARKOV PROCESS
FIRST-ORDER
MARKOV PROCESS

STATIONARY
PROCESS

SENSOR MARKOV
ASSUMPTION



direction of the dependence between state and sensors: the arrows go from the actual state of the world to sensor values because the state of the world *causes* the sensors to take on particular values: 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 specifying the transition and sensor models, we need to say how everything gets started—the prior probability distribution at time 0, $\mathbf{P}(\mathbf{X}_0)$. With that, we have a specification of the complete joint distribution over all the variables, using Equation (14.2). For any t ,

$$\mathbf{P}(\mathbf{X}_{0:t}, \mathbf{E}_{1:t}) = \mathbf{P}(\mathbf{X}_0) \prod_{i=1}^t \mathbf{P}(\mathbf{X}_i | \mathbf{X}_{i-1}) \mathbf{P}(\mathbf{E}_i | \mathbf{X}_i). \quad (15.3)$$

The three terms on the right-hand side are the initial state model $\mathbf{P}(\mathbf{X}_0)$, the transition model $\mathbf{P}(\mathbf{X}_i | \mathbf{X}_{i-1})$, and the sensor model $\mathbf{P}(\mathbf{E}_i | \mathbf{X}_i)$.

The structure in Figure 15.2 is a first-order Markov process—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, 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 ways to improve the accuracy of the approximation:

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, California, 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 $Pressure_t$ (perhaps at a range of locations) 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, but more often we find that error accumulates over time. In that case, accuracy can 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

BELIEF STATE

STATE ESTIMATION

- **Filtering:** This is the task of computing the **belief state**—the posterior distribution over the most recent state—given all evidence to date. Filtering² is also called **state estimation**. In our example, we wish to compute $\mathbf{P}(\mathbf{X}_t | \mathbf{e}_{1:t})$. 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 does to keep track of the current state so that rational decisions can be made. 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 to date. Prediction is useful for evaluating possible courses of action based on their expected outcomes.

² The term “filtering” refers to the roots of this problem in early work on signal processing, where the problem is to filter out the noise in a signal by estimating its underlying properties.

SMOOTHING

- **Smoothing:** 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. Smoothing provides a better estimate of the state than was available at the time, because it incorporates more evidence.³
- **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 inference tasks, we also have

- **Learning:** The transition and sensor models, if not yet known, can be learned 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.)

Note that learning requires smoothing, rather than filtering, because smoothing 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: unless you are an eyewitness, smoothing is *always* required to infer what happened at the murder scene from the observable variables.

The remainder of this section describes generic algorithms for the four inference tasks, independent of the particular kind of model employed. Improvements specific to each model are described in subsequent sections.

15.2.1 Filtering and prediction

As we pointed out in Section 7.7.3, a useful filtering algorithm needs to maintain a current state estimate and update it, rather than going back over the entire history of percepts for each update. (Otherwise, the cost of each update increases as time goes by.) In other words, given the result of filtering up to time t , the agent needs to compute the result for $t + 1$ from the new evidence \mathbf{e}_{t+1} ,

$$\mathbf{P}(\mathbf{X}_{t+1} | \mathbf{e}_{1:t+1}) = f(\mathbf{e}_{t+1}, \mathbf{P}(\mathbf{X}_t | \mathbf{e}_{1:t})) ,$$

for some function f . This process is called **recursive estimation**. We can view the calculation

³ In particular, when tracking a moving object with inaccurate position observations, smoothing gives a smoother estimated trajectory than filtering—hence the name.

as 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 when the formula is rearranged:

$$\begin{aligned}\mathbf{P}(\mathbf{X}_{t+1} | \mathbf{e}_{1:t+1}) &= \mathbf{P}(\mathbf{X}_{t+1} | \mathbf{e}_{1:t}, \mathbf{e}_{t+1}) \quad (\text{dividing up the evidence}) \\ &= \alpha \mathbf{P}(\mathbf{e}_{t+1} | \mathbf{X}_{t+1}, \mathbf{e}_{1:t}) \mathbf{P}(\mathbf{X}_{t+1} | \mathbf{e}_{1:t}) \quad (\text{using Bayes' rule}) \\ &= \alpha \mathbf{P}(\mathbf{e}_{t+1} | \mathbf{X}_{t+1}) \mathbf{P}(\mathbf{X}_{t+1} | \mathbf{e}_{1:t}) \quad (\text{by the sensor Markov assumption}).\end{aligned}\quad (15.4)$$

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 :

$$\begin{aligned}\mathbf{P}(\mathbf{X}_{t+1} | \mathbf{e}_{1:t+1}) &= \alpha \mathbf{P}(\mathbf{e}_{t+1} | \mathbf{X}_{t+1}) \sum_{\mathbf{x}_t} \mathbf{P}(\mathbf{X}_{t+1} | \mathbf{x}_t, \mathbf{e}_{1:t}) P(\mathbf{x}_t | \mathbf{e}_{1:t}) \\ &= \alpha \mathbf{P}(\mathbf{e}_{t+1} | \mathbf{X}_{t+1}) \sum_{\mathbf{x}_t} \mathbf{P}(\mathbf{X}_{t+1} | \mathbf{x}_t) P(\mathbf{x}_t | \mathbf{e}_{1:t}) \quad (\text{Markov assumption}).\end{aligned}\quad (15.5)$$

Within the summation, the first factor comes from the transition model and the second comes from 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 given by

$$\mathbf{f}_{1:t+1} = \alpha \text{FORWARD}(\mathbf{f}_{1:t}, \mathbf{e}_{t+1}),$$

where FORWARD implements the update described in Equation (15.5) and the process begins with $\mathbf{f}_{1:0} = \mathbf{P}(\mathbf{X}_0)$. 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 (Figure 15.2.) That is, we will compute $\mathbf{P}(R_2 | u_{1:2})$ as follows:

- On day 0, we have no observations, only the security guard’s prior beliefs; let’s assume that consists of $\mathbf{P}(R_0) = \langle 0.5, 0.5 \rangle$.
- On day 1, the umbrella appears, so $U_1 = \text{true}$. The prediction from $t = 0$ to $t = 1$ is

$$\begin{aligned}\mathbf{P}(R_1) &= \sum_{r_0} \mathbf{P}(R_1 | r_0) P(r_0) \\ &= \langle 0.7, 0.3 \rangle \times 0.5 + \langle 0.3, 0.7 \rangle \times 0.5 = \langle 0.5, 0.5 \rangle.\end{aligned}$$

Then the update step simply multiplies by the probability of the evidence for $t = 1$ and normalizes, as shown in Equation (15.4):

$$\begin{aligned}\mathbf{P}(R_1 | u_1) &= \alpha \mathbf{P}(u_1 | R_1) \mathbf{P}(R_1) = \alpha \langle 0.9, 0.2 \rangle \langle 0.5, 0.5 \rangle \\ &= \alpha \langle 0.45, 0.1 \rangle \approx \langle 0.818, 0.182 \rangle.\end{aligned}$$



- On day 2, the umbrella appears, so $U_2 = \text{true}$. 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 $t + k$:

$$\mathbf{P}(\mathbf{X}_{t+k+1} | \mathbf{e}_{1:t}) = \sum_{\mathbf{x}_{t+k}} \mathbf{P}(\mathbf{X}_{t+k+1} | \mathbf{x}_{t+k}) P(\mathbf{x}_{t+k} | \mathbf{e}_{1:t}). \quad (15.6)$$

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 537.) 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, unless the stationary distribution itself is strongly peaked in a small area of the state space. The more uncertainty there is in the transition model, the shorter will be the mixing time and the more the future is obscured.

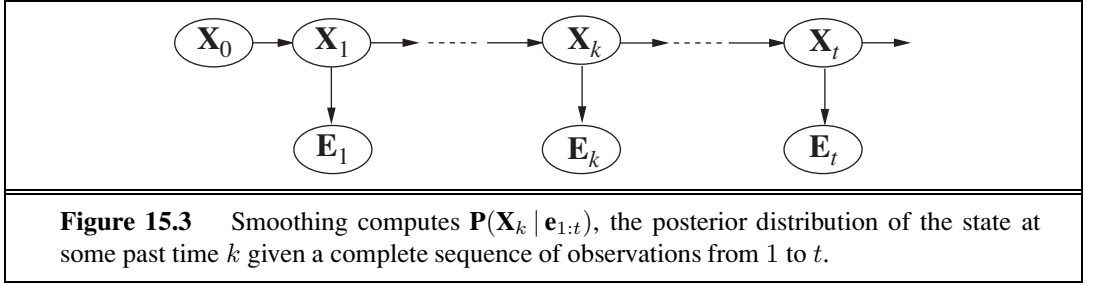
In addition to filtering and prediction, we can use a forward recursion to compute the **likelihood** of the evidence sequence, $P(\mathbf{e}_{1:t})$. This is a useful quantity if we want to compare different temporal models that might have produced the same evidence sequence (e.g., two different models for the persistence of rain). For this recursion, we use a likelihood message $\ell_{1:t}(\mathbf{X}_t) = \mathbf{P}(\mathbf{X}_t, \mathbf{e}_{1:t})$. It is a simple exercise to show that the message calculation is identical to that for filtering:

$$\ell_{1:t+1} = \text{FORWARD}(\ell_{1:t}, \mathbf{e}_{t+1}).$$

Having computed $\ell_{1:t}$, we obtain the actual likelihood by summing out \mathbf{X}_t :

$$L_{1:t} = P(\mathbf{e}_{1:t}) = \sum_{\mathbf{x}_t} \ell_{1:t}(\mathbf{x}_t). \quad (15.7)$$

Notice that the likelihood message represents the probabilities of longer and longer evidence sequences as time goes by and so becomes numerically smaller and smaller, leading to under-flow problems with floating-point arithmetic. This is an important problem in practice, but we shall not go into solutions here.



15.2.2 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 $0 \leq k < t$. (See Figure 15.3.) In anticipation of another recursive message-passing approach, we can split the computation into two parts—the evidence up to k and the evidence from $k + 1$ to t ,

$$\begin{aligned}
 \mathbf{P}(\mathbf{X}_k | \mathbf{e}_{1:t}) &= \mathbf{P}(\mathbf{X}_k | \mathbf{e}_{1:k}, \mathbf{e}_{k+1:t}) \\
 &= \alpha \mathbf{P}(\mathbf{X}_k | \mathbf{e}_{1:k}) \mathbf{P}(\mathbf{e}_{k+1:t} | \mathbf{X}_k, \mathbf{e}_{1:k}) \quad (\text{using Bayes' rule}) \\
 &= \alpha \mathbf{P}(\mathbf{X}_k | \mathbf{e}_{1:k}) \mathbf{P}(\mathbf{e}_{k+1:t} | \mathbf{X}_k) \quad (\text{using conditional independence}) \\
 &= \alpha \mathbf{f}_{1:k} \times \mathbf{b}_{k+1:t} .
 \end{aligned} \tag{15.8}$$

where “ \times ” represents pointwise multiplication of vectors. Here 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 (15.5). It turns out that the backward message $\mathbf{b}_{k+1:t}$ can be computed by a recursive process that runs *backward* from t :

$$\begin{aligned}
 \mathbf{P}(\mathbf{e}_{k+1:t} | \mathbf{X}_k) &= \sum_{\mathbf{x}_{k+1}} \mathbf{P}(\mathbf{e}_{k+1:t} | \mathbf{X}_k, \mathbf{x}_{k+1}) \mathbf{P}(\mathbf{x}_{k+1} | \mathbf{X}_k) \quad (\text{conditioning on } \mathbf{X}_{k+1}) \\
 &= \sum_{\mathbf{x}_{k+1}} P(\mathbf{e}_{k+1:t} | \mathbf{x}_{k+1}) \mathbf{P}(\mathbf{x}_{k+1} | \mathbf{X}_k) \quad (\text{by conditional independence}) \\
 &= \sum_{\mathbf{x}_{k+1}} P(\mathbf{e}_{k+1}, \mathbf{e}_{k+2:t} | \mathbf{x}_{k+1}) \mathbf{P}(\mathbf{x}_{k+1} | \mathbf{X}_k) \\
 &= \sum_{\mathbf{x}_{k+1}} P(\mathbf{e}_{k+1} | \mathbf{x}_{k+1}) P(\mathbf{e}_{k+2:t} | \mathbf{x}_{k+1}) \mathbf{P}(\mathbf{x}_{k+1} | \mathbf{X}_k) ,
 \end{aligned} \tag{15.9}$$

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

$$\mathbf{b}_{k+1:t} = \text{BACKWARD}(\mathbf{b}_{k+2:t}, \mathbf{e}_{k+1}) ,$$

where BACKWARD implements the update described in Equation (15.9). 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 (15.8) can both be computed by recursions through time, one running forward from 1 to k and using the filtering equation (15.5)

and the other running backward from t to $k + 1$ and using Equation (15.9). Note that the backward phase is initialized with $\mathbf{b}_{t+1:t} = \mathbf{P}(\mathbf{e}_{t+1:t} | \mathbf{X}_t) = \mathbf{P}(\mathbf{e}_{t+1:t} | \mathbf{X}_t) \mathbf{1}$, where $\mathbf{1}$ is a vector of 1s. (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 time $k = 1$, given the umbrella observations on days 1 and 2. From Equation (15.8), this is given by

$$\mathbf{P}(R_1 | u_1, u_2) = \alpha \mathbf{P}(R_1 | u_1) \mathbf{P}(u_2 | R_1). \quad (15.10)$$

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 (15.9):

$$\begin{aligned} \mathbf{P}(u_2 | R_1) &= \sum_{r_2} P(u_2 | r_2) P(r_2 | R_1) \\ &= (0.9 \times 1 \times \langle 0.7, 0.3 \rangle) + (0.2 \times 1 \times \langle 0.3, 0.7 \rangle) = \langle 0.69, 0.41 \rangle. \end{aligned}$$

Plugging this into Equation (15.10), 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 for rain on day 1 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 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.

The alert reader will have spotted that the Bayesian network structure shown in Figure 15.3 is a *polytree* as defined on page 528. 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).

The forward-backward algorithm forms the computational backbone for many applications that deal with sequences of noisy observations. As described so far, it has two practical drawbacks. The first is that its space complexity can be too high when 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 concomi-

tant 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.

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, independent of the lag d .

FIXED-LAG
SMOOTHING

15.2.3 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?

We could try this linear-time procedure: use smoothing to find the posterior distribution for the weather at each time step; then construct the sequence, using at each step the weather that is most likely according to the posterior. Such an approach should set off alarm bells in the reader’s head, because the posterior distributions computed by smoothing are distri-

```

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,  $\mathbf{P}(\mathbf{X}_0)$ 
  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]  $\leftarrow$  prior
  for  $i = 1$  to  $t$  do
    fv[ $i$ ]  $\leftarrow$  FORWARD(fv[ $i - 1$ ], ev[ $i$ ])
  for  $i = t$  downto 1 do
    sv[ $i$ ]  $\leftarrow$  NORMALIZE(fv[ $i$ ]  $\times$  b)
    b  $\leftarrow$  BACKWARD(b, ev[ $i$ ])
  return sv

```

Figure 15.4 The forward–backward algorithm for smoothing: computing posterior probabilities of a sequence of states given a sequence of observations. The FORWARD and BACKWARD operators are defined by Equations (15.5) and (15.9), respectively.

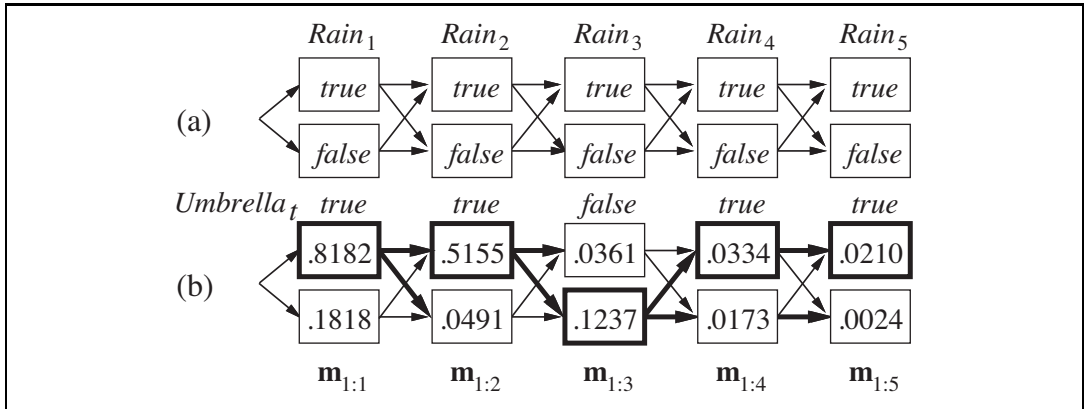


Figure 15.5 (a) Possible state sequences for $Rain_t$ can be viewed as paths through a graph of the possible states at each time step. (States are shown as rectangles to avoid confusion with nodes in a Bayes net.) (b) Operation of the Viterbi algorithm for the umbrella observation sequence $[true, true, false, true, true]$. For each t , we have shown the values of the message $m_{1:t}$, 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 $m_{1:5}$ gives the most likely sequence.

butions 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 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 x_{t+1} and most likely paths to each state x_t* . We can write this relationship as an equation connecting the probabilities of the paths:

$$\begin{aligned} & \max_{x_1 \dots x_t} P(x_1, \dots, x_t, x_{t+1} \mid e_{1:t+1}) \\ &= \alpha P(e_{t+1} \mid x_{t+1}) \max_{x_t} \left(P(x_{t+1} \mid x_t) \max_{x_1 \dots x_{t-1}} P(x_1, \dots, x_{t-1}, x_t \mid e_{1:t}) \right). \quad (15.11) \end{aligned}$$

Equation (15.11) is *identical* to the filtering equation (15.5) except that

1. The forward message $\mathbf{f}_{1:t} = \mathbf{P}(\mathbf{X}_t | \mathbf{e}_{1:t})$ is replaced by the message

$$\mathbf{m}_{1:t} = \max_{\mathbf{x}_1 \dots \mathbf{x}_{t-1}} \mathbf{P}(\mathbf{x}_1, \dots, \mathbf{x}_{t-1}, \mathbf{X}_t | \mathbf{e}_{1:t}),$$

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 (15.5) is replaced by the maximization over \mathbf{x}_t in Equation (15.11).

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 (15.11). 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 states outlined in bold). In order to identify the actual sequence, as opposed to just computing its probability, the algorithm will also need to record, for each state, the best state that leads to it; these are indicated by the bold arrows in Figure 15.5(b). The optimal sequence is identified by following these bold arrows backwards 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 time complexity is linear in t , the length of the sequence. Unlike filtering, which uses constant space, 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: $Rain_t$. What happens if you have a model with two or more state variables? You can still fit it into the HMM framework by combining the 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 simple and elegant matrix implementation of all the basic algorithms.⁴

⁴ The reader unfamiliar with basic operations on vectors and matrices might wish to consult Appendix A before proceeding with this section.

15.3.1 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 $1, \dots, S$, 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 at time t (call it e_t), we need only specify, for each state, how likely it is that the state causes e_t to appear: we need $P(e_t | X_t = i)$ for each state i . For mathematical convenience we place these values into an $S \times S$ diagonal matrix, \mathbf{O}_t whose i th diagonal entry is $P(e_t | X_t = i)$ and whose other entries are 0. For example, on day 1 in the umbrella world of Figure 15.5, $U_1 = \text{true}$, and on day 3, $U_3 = \text{false}$, so, from Figure 15.2, we have

$$\mathbf{O}_1 = \begin{pmatrix} 0.9 & 0 \\ 0 & 0.2 \end{pmatrix}; \quad \mathbf{O}_3 = \begin{pmatrix} 0.1 & 0 \\ 0 & 0.8 \end{pmatrix} .$$

Now, if we use column vectors to represent the forward and backward messages, all the computations become simple matrix–vector operations. The forward equation (15.5) becomes

$$\mathbf{f}_{1:t+1} = \alpha \mathbf{O}_{t+1} \mathbf{T}^\top \mathbf{f}_{1:t} \quad (15.12)$$

and the backward equation (15.9) becomes

$$\mathbf{b}_{k+1:t} = \mathbf{T} \mathbf{O}_{k+1} \mathbf{b}_{k+2:t} . \quad (15.13)$$

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 (15.8). The forward–backward algorithm achieves this by storing the \mathbf{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 \mathbf{f} and \mathbf{b} in the same direction. For example, the “forward” message \mathbf{f} can be propagated backward if we manipulate Equation (15.12) 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

```

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
  persistent:  $t$ , the current time, initially 1
                 $\mathbf{f}$ , the forward message  $\mathbf{P}(X_t|e_{1:t})$ , initially  $hmm.PRIOR$ 
                 $\mathbf{B}$ , 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
     $\mathbf{f} \leftarrow \text{FORWARD}(\mathbf{f}, e_t)$ 
    remove  $e_{t-d-1}$  from the beginning of  $e_{t-d:t}$ 
     $\mathbf{O}_{t-d} \leftarrow$  diagonal matrix containing  $\mathbf{P}(e_{t-d}|X_{t-d})$ 
     $\mathbf{B} \leftarrow \mathbf{O}_{t-d}^{-1} \mathbf{T}^{-1} \mathbf{B} \mathbf{O}_t$ 
  else  $\mathbf{B} \leftarrow \mathbf{B} \mathbf{O}_t$ 
   $t \leftarrow t + 1$ 
  if  $t > d$  then return  $\text{NORMALIZE}(\mathbf{f} \times \mathbf{B} \mathbf{1})$  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. Notice that the final output $\text{NORMALIZE}(\mathbf{f} \times \mathbf{B} \mathbf{1})$ is just $\alpha \mathbf{f} \times \mathbf{b}$, by Equation (15.14).

\mathbf{b} and \mathbf{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 are two significant restrictions on this algorithm: it requires that the transition matrix be invertible and that the sensor model have no zeroes—that is, that every observation be 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 (15.8), we need to compute

$$\alpha \mathbf{f}_{1:t-d} \times \mathbf{b}_{t-d+1:t}$$

for slice $t-d$. Then, when a new observation arrives, we need to compute

$$\alpha \mathbf{f}_{1:t-d+1} \times \mathbf{b}_{t-d+2:t+1}$$

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 (15.5).

Computing the backward message incrementally is trickier, 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 (15.13) d times to get

$$\mathbf{b}_{t-d+1:t} = \left(\prod_{i=t-d+1}^t \mathbf{TO}_i \right) \mathbf{b}_{t+1:t} = \mathbf{B}_{t-d+1:t} \mathbf{1}, \quad (15.14)$$

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:

$$\mathbf{b}_{t-d+2:t+1} = \left(\prod_{i=t-d+2}^{t+1} \mathbf{TO}_i \right) \mathbf{b}_{t+2:t+1} = \mathbf{B}_{t-d+2:t+1} \mathbf{1}. \quad (15.15)$$

Examining the product expressions in Equations (15.14) and (15.15), 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:

$$\mathbf{B}_{t-d+2:t+1} = \mathbf{O}_{t-d+1}^{-1} \mathbf{T}^{-1} \mathbf{B}_{t-d+1:t} \mathbf{TO}_{t+1}. \quad (15.16)$$

This equation provides an incremental update for the \mathbf{B} matrix, which in turn (through Equation (15.15)) 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.

15.3.2 Hidden Markov model example: Localization

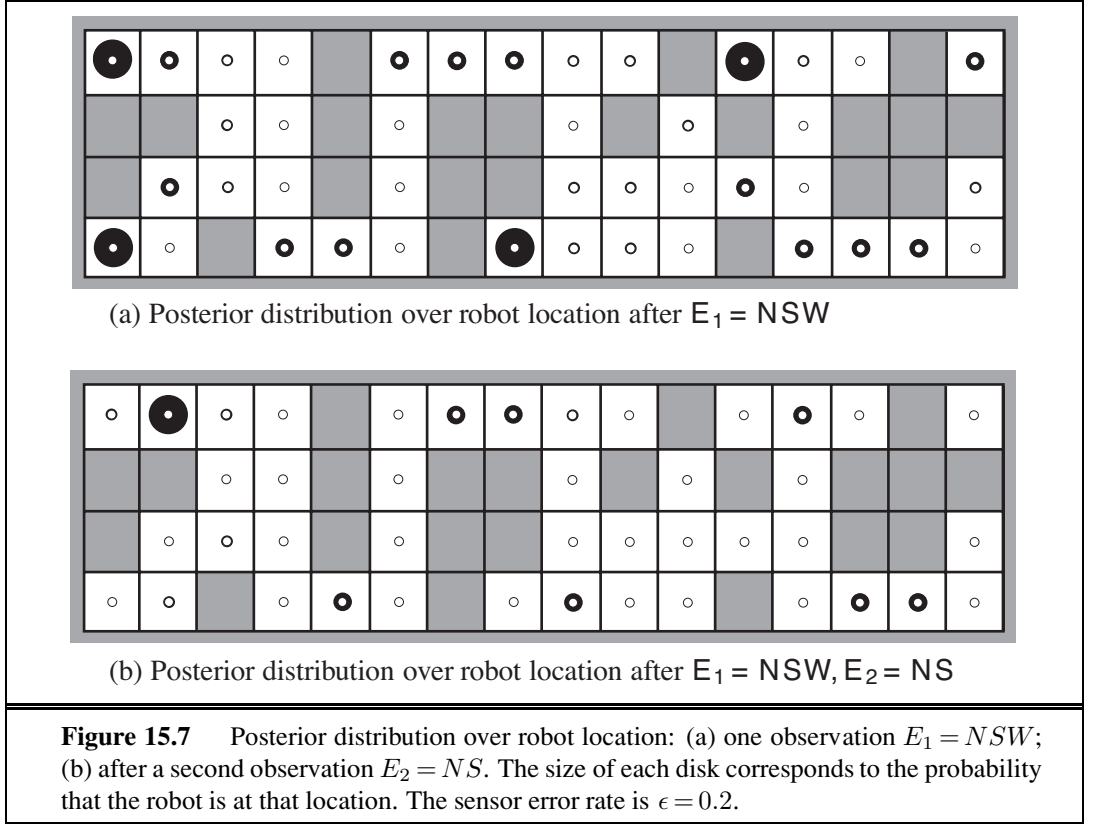
On page 145, we introduced a simple form of the **localization** problem for the vacuum world. In that version, the robot had a single nondeterministic *Move* action and its sensors reported perfectly whether or not obstacles lay immediately to the north, south, east, and west; the robot’s belief state was the set of possible locations it could be in.

Here we make the problem slightly more realistic by including a simple probability model for the robot’s motion and by allowing for noise in the sensors. The state variable X_t represents the location of the robot on the discrete grid; the domain of this variable is the set of empty squares $\{s_1, \dots, s_n\}$. Let $\text{NEIGHBORS}(s)$ be the set of empty squares that are adjacent to s and let $N(s)$ be the size of that set. Then the transition model for *Move* action says that the robot is equally likely to end up at any neighboring square:

$$P(X_{t+1} = j \mid X_t = i) = \mathbf{T}_{ij} = (1/N(i) \text{ if } j \in \text{NEIGHBORS}(i) \text{ else } 0).$$

We don’t know where the robot starts, so we will assume a uniform distribution over all the squares; that is, $P(X_0 = i) = 1/n$. For the particular environment we consider (Figure 15.7), $n = 42$ and the transition matrix \mathbf{T} has $42 \times 42 = 1764$ entries.

The sensor variable E_t has 16 possible values, each a four-bit sequence giving the presence or absence of an obstacle in a particular compass direction. We will use the notation



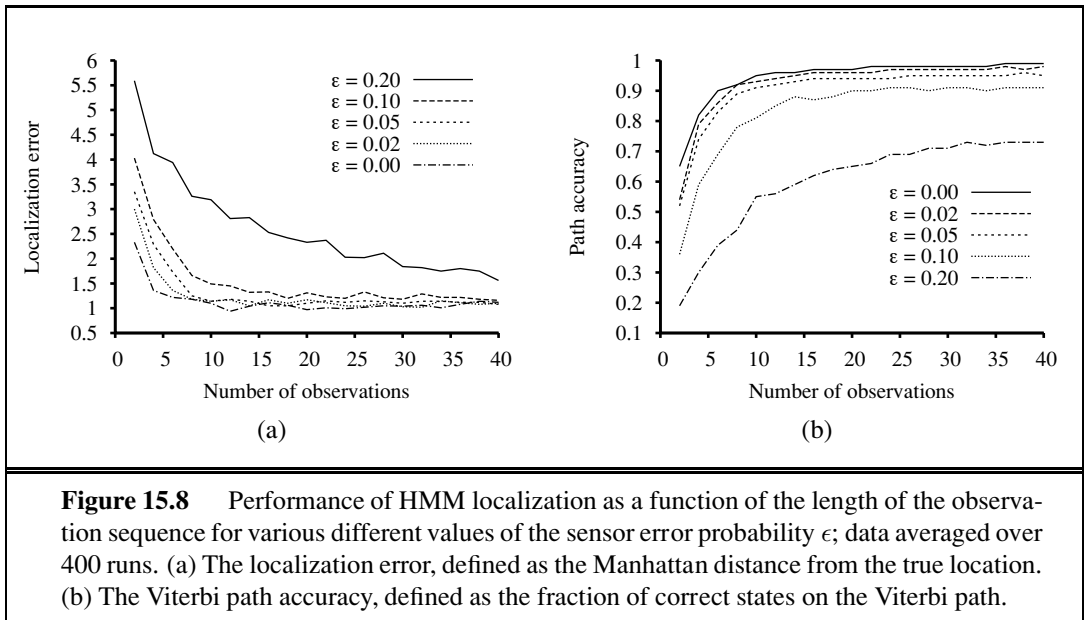
NS , for example, to mean that the north and south sensors report an obstacle and the east and west do not. Suppose that each sensor's error rate is ϵ and that errors occur independently for the four sensor directions. In that case, the probability of getting all four bits right is $(1 - \epsilon)^4$ and the probability of getting them all wrong is ϵ^4 . Furthermore, if d_{it} is the discrepancy—the number of bits that are different—between the true values for square i and the actual reading e_t , then the probability that a robot in square i would receive a sensor reading e_t is

$$P(E_t = e_t \mid X_t = i) = \mathbf{O}_{t_{ii}} = (1 - \epsilon)^{4-d_{it}} \epsilon^{d_{it}}.$$

For example, the probability that a square with obstacles to the north and south would produce a sensor reading NSE is $(1 - \epsilon)^3 \epsilon^1$.

Given the matrices \mathbf{T} and \mathbf{O}_t , the robot can use Equation (15.12) to compute the posterior distribution over locations—that is, to work out where it is. Figure 15.7 shows the distributions $\mathbf{P}(X_1 \mid E_1 = \text{NSW})$ and $\mathbf{P}(X_2 \mid E_1 = \text{NSW}, E_2 = \text{NS})$. This is the same maze we saw before in Figure 4.18 (page 146), but there we used logical filtering to find the locations that were *possible*, assuming perfect sensing. Those same locations are still the most *likely* with noisy sensing, but now *every* location has some nonzero probability.

In addition to filtering to estimate its current location, the robot can use smoothing (Equation (15.13)) to work out where it was at any given past time—for example, where it began at time 0—and it can use the Viterbi algorithm to work out the most likely path it has



taken to get where it is now. Figure 15.8 shows the localization error and Viterbi path accuracy for various values of the per-bit sensor error rate ϵ . Even when ϵ is 20%—which means that the overall sensor reading is wrong 59% of the time—the robot is usually able to work out its location within two squares after 25 observations. This is because of the algorithm’s ability to integrate evidence over time and to take into account the probabilistic constraints imposed on the location sequence by the transition model. When ϵ is 10%, the performance after a half-dozen observations is hard to distinguish from the performance with perfect sensing. Exercise 15.7 asks you to explore how robust the HMM localization algorithm is to errors in the prior distribution $\mathbf{P}(X_0)$ and in the transition model itself. Broadly speaking, high levels of localization and path accuracy are maintained even in the face of substantial errors in the models used.

The state variable for the example we have considered in this section is a physical location in the world. Other problems can, of course, include other aspects of the world. Exercise 15.8 asks you to consider a version of the vacuum robot that has the policy of going straight for as long as it can; only when it encounters an obstacle does it change to a new (randomly selected) heading. To model this robot, each state in the model consists of a (*location, heading*) pair. For the environment in Figure 15.7, which has 42 empty squares, this leads to 168 states and a transition matrix with $168^2 = 28,224$ entries—still a manageable number. If we add the possibility of dirt in the squares, the number of states is multiplied by 2^{42} and the transition matrix ends up with more than 10^{29} entries—no longer a manageable number; Section 15.5 shows how to use dynamic Bayesian networks to model domains with many state variables. If we allow the robot to move continuously rather than in a discrete grid, the number of states becomes infinite; the next section shows how to handle this case.

15.4 KALMAN FILTERS

KALMAN FILTERING

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 doing filtering: estimating state variables (here, position and velocity) from noisy observations over time. If the variables were discrete, we could model the system with a hidden Markov model. This section examines methods for handling continuous variables, using an algorithm called **Kalman filtering**, after one of its inventors, Rudolf E. Kalman.

The bird's flight might be specified by six continuous variables at each time point; three for position (X_t, Y_t, Z_t) and three for velocity $(\dot{X}_t, \dot{Y}_t, \dot{Z}_t)$. We will 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 time interval between observations be Δ , and assume constant velocity during the interval; then the position update is given by $X_{t+\Delta} = X_t + \dot{X}_t \Delta$. Adding Gaussian noise (to account for wind variation, etc.), we obtain a linear Gaussian transition model:

$$P(X_{t+\Delta} = x_{t+\Delta} \mid X_t = x_t, \dot{X}_t = \dot{x}_t) = N(x_t + \dot{x}_t \Delta, \sigma^2)(x_{t+\Delta}) .$$

The Bayesian network structure for a system with position vector \mathbf{X}_t and velocity $\dot{\mathbf{X}}_t$ is shown in Figure 15.9. 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

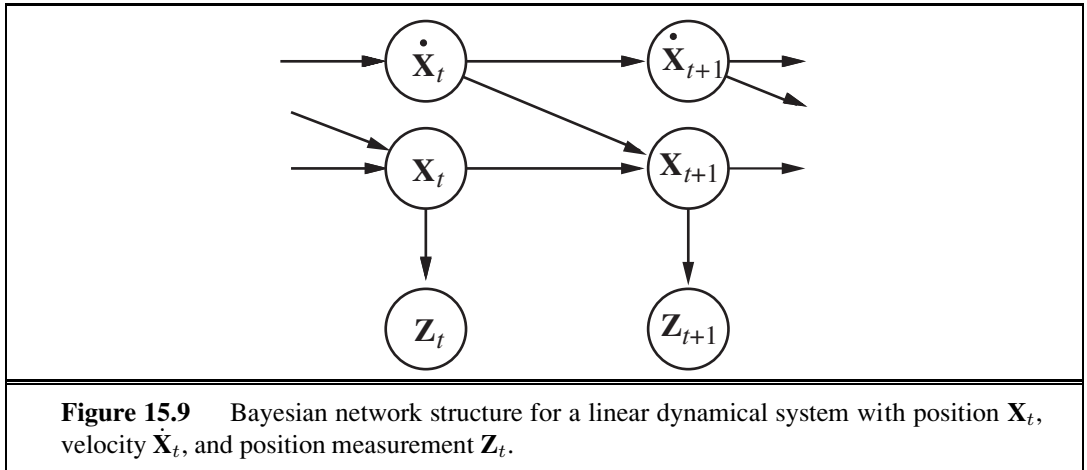
15.4.1 Updating Gaussian distributions

In Chapter 14 on page 521, 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 (15.5):

1. If the current distribution $\mathbf{P}(\mathbf{X}_t \mid \mathbf{e}_{1:t})$ is Gaussian and the transition model $\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{x}_t)$ is linear Gaussian, then the one-step predicted distribution given by

$$\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{e}_{1:t}) = \int_{\mathbf{x}_t} \mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{x}_t) P(\mathbf{x}_t \mid \mathbf{e}_{1:t}) d\mathbf{x}_t \quad (15.17)$$

is also a Gaussian distribution.



2. If the prediction $\mathbf{P}(\mathbf{X}_{t+1} | \mathbf{e}_{1:t})$ is Gaussian and the 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

$$\mathbf{P}(\mathbf{X}_{t+1} | \mathbf{e}_{1:t+1}) = \alpha \mathbf{P}(\mathbf{e}_{t+1} | \mathbf{X}_{t+1}) \mathbf{P}(\mathbf{X}_{t+1} | \mathbf{e}_{1:t}) \quad (15.18)$$

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 15.10 shows what happens for a simple example.

15.4.2 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; and later 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 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 use the same symbol α for all normalizing constants in this section.) The transition model 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 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 $\mathbf{P}(X_0)$, the one-step predicted distribution comes from Equation (15.17):

$$\begin{aligned} P(x_1) &= \int_{-\infty}^{\infty} P(x_1 | x_0) P(x_0) dx_0 = \alpha \int_{-\infty}^{\infty} e^{-\frac{1}{2} \left(\frac{(x_1 - x_0)^2}{\sigma_x^2} \right)} e^{-\frac{1}{2} \left(\frac{(x_0 - \mu_0)^2}{\sigma_0^2} \right)} dx_0 \\ &= \alpha \int_{-\infty}^{\infty} e^{-\frac{1}{2} \left(\frac{\sigma_0^2 (x_1 - x_0)^2 + \sigma_x^2 (x_0 - \mu_0)^2}{\sigma_0^2 \sigma_x^2} \right)} dx_0. \end{aligned}$$

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

$$P(x_1) = \alpha e^{-\frac{1}{2} \left(c - \frac{b^2}{4a} \right)} \int_{-\infty}^{\infty} e^{-\frac{1}{2} \left(a(x_0 - \frac{-b}{2a})^2 \right)} dx_0.$$

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. Then, we notice that the residual term is a quadratic in x_1 ; in fact, after simplification, we obtain

$$P(x_1) = \alpha e^{-\frac{1}{2} \left(\frac{(x_1 - \mu_0)^2}{\sigma_0^2 + \sigma_x^2} \right)}.$$

That is, the one-step predicted distribution is a Gaussian with the same mean μ_0 and a variance equal to the sum of the original variance σ_0^2 and the transition variance σ_x^2 .

To complete the update step, we need to condition on the observation at the first time step, namely, z_1 . From Equation (15.18), this is given by

$$\begin{aligned} P(x_1 | z_1) &= \alpha P(z_1 | x_1) P(x_1) \\ &= \alpha e^{-\frac{1}{2} \left(\frac{(z_1 - x_1)^2}{\sigma_z^2} \right)} e^{-\frac{1}{2} \left(\frac{(x_1 - \mu_0)^2}{\sigma_0^2 + \sigma_x^2} \right)}. \end{aligned}$$

Once again, we combine the exponents and complete the square (Exercise 15.11), obtaining

$$P(x_1 | z_1) = \alpha e^{-\frac{1}{2} \left(\frac{(x_1 - \frac{(\sigma_0^2 + \sigma_x^2)z_1 + \sigma_z^2 \mu_0}{\sigma_0^2 + \sigma_x^2 + \sigma_z^2})^2}{(\frac{\sigma_0^2 + \sigma_x^2}{\sigma_0^2 + \sigma_x^2 + \sigma_z^2}) \sigma_z^2 / (\sigma_0^2 + \sigma_x^2 + \sigma_z^2)} \right)}. \quad (15.19)$$

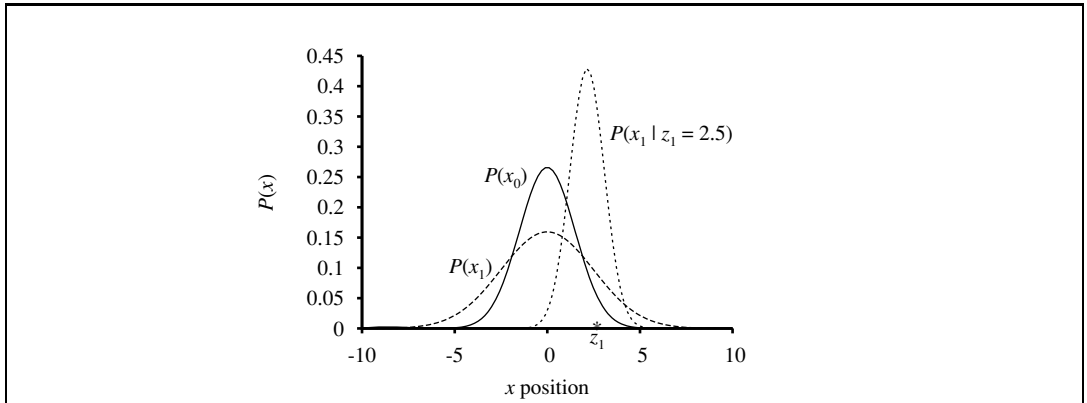


Figure 15.10 Stages in the Kalman filter update cycle for a random walk with a prior given by $\mu_0 = 0.0$ and $\sigma_0 = 1.0$, transition noise given by $\sigma_x = 2.0$, sensor noise given by $\sigma_z = 1.0$, and a first observation $z_1 = 2.5$ (marked on the x -axis). Notice how the prediction $P(x_1)$ is flattened out, relative to $P(x_0)$, by the transition noise. Notice also that the mean of the posterior distribution $P(x_1 | z_1)$ is slightly to the left of the observation z_1 because the mean is a weighted average of the prediction and the observation.

Thus, after one update cycle, we have a new Gaussian distribution for the state variable.

From the Gaussian formula in Equation (15.19), we see that the new mean and standard deviation can be calculated from the old mean and standard deviation as follows:

$$\mu_{t+1} = \frac{(\sigma_t^2 + \sigma_x^2)z_{t+1} + \sigma_z^2\mu_t}{\sigma_t^2 + \sigma_x^2 + \sigma_z^2} \quad \text{and} \quad \sigma_{t+1}^2 = \frac{(\sigma_t^2 + \sigma_x^2)\sigma_z^2}{\sigma_t^2 + \sigma_x^2 + \sigma_z^2}. \quad (15.20)$$

Figure 15.10 shows one update cycle for particular values of the transition and sensor models.

Equation (15.20) plays exactly the same role as the general filtering equation (15.5) or the HMM filtering equation (15.12). Because of the special nature of Gaussian distributions, however, the equations have some interesting additional properties. First, we can interpret the calculation for the new mean μ_{t+1} as simply a *weighted mean* of the new observation z_{t+1} and the old mean μ_t . If the observation is unreliable, then σ_z^2 is large and we pay more attention to the old mean; if the old mean is unreliable (σ_t^2 is large) or the process is highly unpredictable (σ_x^2 is large), then we pay more attention to the observation. Second, notice that the update for the variance σ_{t+1}^2 is *independent of the observation*. We can therefore compute in advance what the sequence of variance values will be. Third, the sequence of variance values converges quickly to a fixed value that depends only on σ_x^2 and σ_z^2 , thereby substantially simplifying the subsequent calculations. (See Exercise 15.12.)

15.4.3 The general case

The preceding derivation illustrates the key property of Gaussian distributions that allows Kalman filtering to work: the fact that the exponent is a quadratic form. This is true not just for the univariate case; the full multivariate Gaussian distribution has the form

$$N(\boldsymbol{\mu}, \boldsymbol{\Sigma})(\mathbf{x}) = \alpha e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})}.$$

Multiplying out the terms in the exponent makes it clear that the exponent is also a quadratic function of the values x_i in \mathbf{x} . As in the univariate case, the filtering update preserves the Gaussian nature of the state distribution.

Let us first define the general temporal model used with Kalman filtering. Both the transition model and the sensor model allow for a *linear* transformation with additive Gaussian noise. Thus, we have

$$\begin{aligned} P(\mathbf{x}_{t+1} | \mathbf{x}_t) &= N(\mathbf{F}\mathbf{x}_t, \Sigma_x)(\mathbf{x}_{t+1}) \\ P(\mathbf{z}_t | \mathbf{x}_t) &= N(\mathbf{H}\mathbf{x}_t, \Sigma_z)(\mathbf{z}_t), \end{aligned} \quad (15.21)$$

where \mathbf{F} and Σ_x are matrices describing the linear transition model and transition noise covariance, and \mathbf{H} and Σ_z are the corresponding matrices for the sensor model. Now the update equations for the mean and covariance, in their full, hairy horribleness, are

$$\begin{aligned} \mu_{t+1} &= \mathbf{F}\mu_t + \mathbf{K}_{t+1}(\mathbf{z}_{t+1} - \mathbf{H}\mathbf{F}\mu_t) \\ \Sigma_{t+1} &= (\mathbf{I} - \mathbf{K}_{t+1}\mathbf{H})(\mathbf{F}\Sigma_t\mathbf{F}^\top + \Sigma_x), \end{aligned} \quad (15.22)$$

where $\mathbf{K}_{t+1} = (\mathbf{F}\Sigma_t\mathbf{F}^\top + \Sigma_x)\mathbf{H}^\top(\mathbf{H}(\mathbf{F}\Sigma_t\mathbf{F}^\top + \Sigma_x)\mathbf{H}^\top + \Sigma_z)^{-1}$ is called the **Kalman gain matrix**. Believe it or not, these equations make some intuitive sense. For example, consider the update for the mean state estimate μ . The term $\mathbf{F}\mu_t$ is the *predicted* state at $t + 1$, so $\mathbf{H}\mathbf{F}\mu_t$ is the *predicted* observation. Therefore, the term $\mathbf{z}_{t+1} - \mathbf{H}\mathbf{F}\mu_t$ represents the error in the predicted observation. This is multiplied by \mathbf{K}_{t+1} to correct the predicted state; hence, \mathbf{K}_{t+1} is a measure of *how seriously to take the new observation* relative to the prediction. As in Equation (15.20), we also have the property that the variance update is independent of the observations. The sequence of values for Σ_t and \mathbf{K}_t can therefore be computed offline, and the actual calculations required during online tracking are quite modest.

To illustrate these equations at work, we have applied them to the problem of tracking an object moving on the X - Y plane. The state variables are $\mathbf{X} = (X, Y, \dot{X}, \dot{Y})^\top$, so \mathbf{F} , Σ_x , \mathbf{H} , and Σ_z are 4×4 matrices. Figure 15.11(a) shows the true trajectory, a series of noisy observations, and the trajectory estimated by Kalman filtering, along with the covariances indicated by the one-standard-deviation contours. The filtering process does a good job of tracking the actual motion, and, as expected, the variance quickly reaches a fixed point.

We can also derive equations for *smoothing* as well as filtering with linear Gaussian models. The smoothing results are shown in Figure 15.11(b). Notice how the variance in the position estimate is sharply reduced, except at the ends of the trajectory (why?), and that the estimated trajectory is much smoother.

15.4.4 Applicability of Kalman filtering

The Kalman filter and its elaborations are used in a vast array of applications. The “classical” application is in radar tracking of aircraft and missiles. Related applications include acoustic tracking of submarines and ground vehicles and visual tracking of vehicles and people. In a slightly more esoteric vein, Kalman filters are used to reconstruct particle trajectories from bubble-chamber photographs and ocean currents from satellite surface measurements. The range of application is much larger than just the tracking of motion: any system characterized by continuous state variables and noisy measurements will do. Such systems include pulp mills, chemical plants, nuclear reactors, plant ecosystems, and national economies.

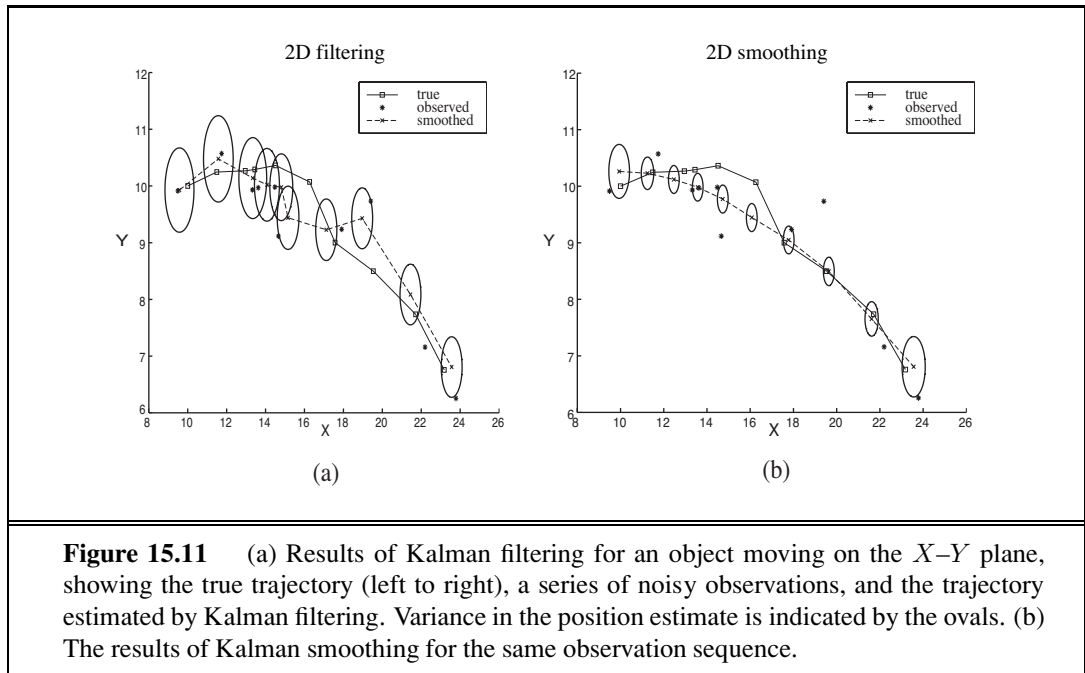


Figure 15.11 (a) Results of Kalman filtering for an object moving on the X - Y plane, showing the true trajectory (left to right), a series of noisy observations, and the trajectory estimated by Kalman filtering. Variance in the position estimate is indicated by the ovals. (b) The results of Kalman smoothing for the same observation sequence.

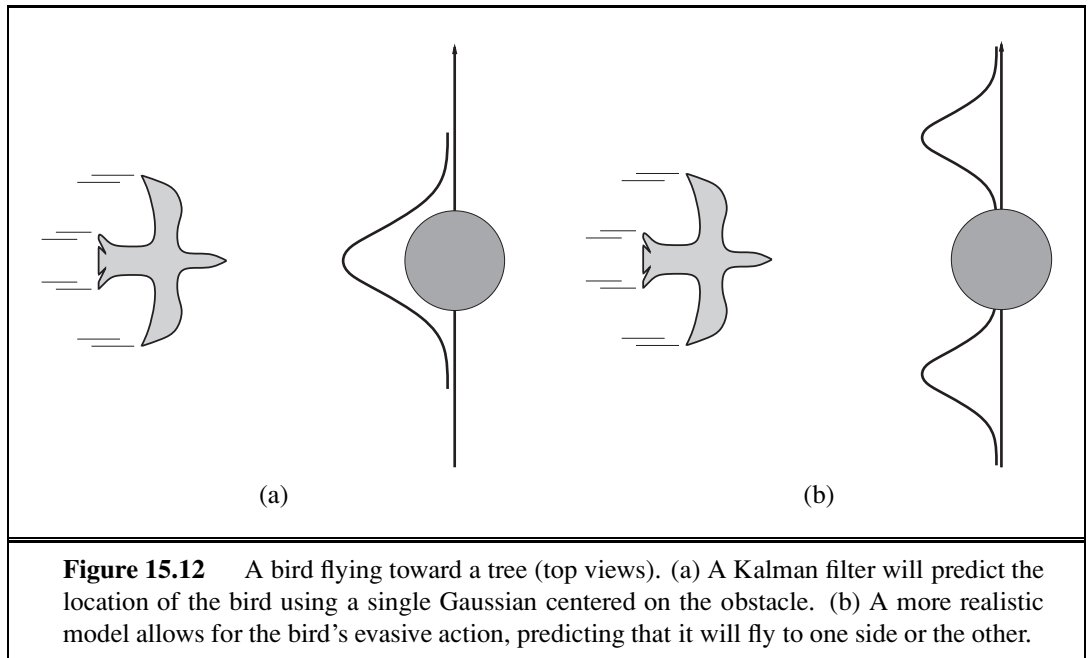
The fact that Kalman filtering can be applied to a system does not mean that the results will be valid or useful. The assumptions made—a linear Gaussian transition and sensor models—are very strong. The **extended Kalman filter (EKF)** attempts to overcome nonlinearities in the system being modeled. A system is **nonlinear** if the transition model cannot be described as a matrix multiplication of the state vector, as in Equation (15.21). The EKF works by modeling the system as *locally* linear in \mathbf{x}_t in the region of $\mathbf{x}_t = \boldsymbol{\mu}_t$, the mean of the current state distribution. This works well for smooth, well-behaved systems and allows the tracker to maintain and update a Gaussian state distribution that is a reasonable approximation to the true posterior. A detailed example is given in Chapter 25.

What does it mean for a system to be “unsmooth” or “poorly behaved”? Technically, it means that there is significant nonlinearity in system response within the region that is “close” (according to the covariance $\boldsymbol{\Sigma}_t$) to the current mean $\boldsymbol{\mu}_t$. To understand this idea in nontechnical terms, consider the example of trying to track a bird as it flies through the jungle. The bird appears to be heading at high speed straight for a tree trunk. The Kalman filter, whether regular or extended, can make only a Gaussian prediction of the location of the bird, and the mean of this Gaussian will be centered on the trunk, as shown in Figure 15.12(a). A reasonable model of the bird, on the other hand, would predict evasive action to one side or the other, as shown in Figure 15.12(b). Such a model is highly nonlinear, because the bird’s decision varies sharply depending on its precise location relative to the trunk.

To handle examples like these, we clearly need a more expressive language for representing the behavior of the system being modeled. Within the control theory community, for which problems such as evasive maneuvering by aircraft raise the same kinds of difficulties, the standard solution is the **switching Kalman filter**. In this approach, multiple Kalman fil-

EXTENDED KALMAN
FILTER (EKF)
NONLINEAR

SWITCHING KALMAN
FILTER



ters run in parallel, each using a different model of the system—for example, one for straight flight, one for sharp left turns, and one for sharp right turns. A weighted sum of predictions is used, where the weight depends on how well each filter fits the current data. We will see in the next section that this is simply a special case of the general dynamic Bayesian network model, obtained by adding a discrete “maneuver” state variable to the network shown in Figure 15.9. Switching Kalman filters are discussed further in Exercise 15.10.

15.5 DYNAMIC BAYESIAN NETWORKS

DYNAMIC BAYESIAN NETWORK

A **dynamic Bayesian network**, or **DBN**, is a Bayesian network that represents a temporal probability model of the kind described in Section 15.1. We have already seen examples of DBNs: the umbrella network in Figure 15.2 and the Kalman filter network in Figure 15.9. In general, each slice of a DBN can have any number of state variables \mathbf{X}_t and evidence variables \mathbf{E}_t . For simplicity, we assume that the variables and their links are exactly replicated from slice to slice and that the DBN represents a first-order Markov process, so that each variable can have parents only in its own slice or the immediately preceding slice.

It should be clear that every hidden Markov model can be represented as a DBN with a single state variable and a single evidence variable. It is also the case that every discrete-variable DBN can be represented as an HMM; as explained in Section 15.3, we can combine all the state variables in the DBN into a single state variable whose values are all possible tuples of values of the individual state variables. Now, if every HMM is a DBN and every DBN can be translated into an HMM, what's the difference? The difference is that, *by de-*



composing the state of a complex system into its constituent variables, the can take advantage of sparseness in the temporal probability model. Suppose, for example, that a DBN has 20 Boolean state variables, each of which has three parents in the preceding slice. Then the DBN transition model has $20 \times 2^3 = 160$ probabilities, whereas the corresponding HMM has 2^{20} states and therefore 2^{40} , or roughly a trillion, probabilities in the transition matrix. This is bad for at least three reasons: first, the HMM itself requires much more space; second, the huge transition matrix makes HMM inference much more expensive; and third, the problem of learning such a huge number of parameters makes the pure HMM model unsuitable for large problems. The relationship between DBNs and HMMs is roughly analogous to the relationship between ordinary Bayesian networks and full tabulated joint distributions.

We have already explained that every Kalman filter model can be represented in a DBN with continuous variables and linear Gaussian conditional distributions (Figure 15.9). It should be clear from the discussion at the end of the preceding section that *not* every DBN can be represented by a Kalman filter model. In a Kalman filter, the current state distribution is always a single multivariate Gaussian distribution—that is, a single “bump” in a particular location. DBNs, on the other hand, can model arbitrary distributions. For many real-world applications, this flexibility is essential. Consider, for example, the current location of my keys. They might be in my pocket, on the bedside table, on the kitchen counter, dangling from the front door, or locked in the car. A single Gaussian bump that included all these places would have to allocate significant probability to the keys being in mid-air in the front hall. Aspects of the real world such as purposive agents, obstacles, and pockets introduce “nonlinearities” that require combinations of discrete and continuous variables in order to get reasonable models.

15.5.1 Constructing DBNs

To construct a DBN, one must specify three kinds of information: the prior distribution over the state variables, $\mathbf{P}(\mathbf{X}_0)$; the transition model $\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{X}_t)$; and the sensor model $\mathbf{P}(\mathbf{E}_t \mid \mathbf{X}_t)$. To specify the transition and sensor models, one must also specify the topology of the connections between successive slices and between the state and evidence variables. Because the transition and sensor models are assumed to be stationary—the same for all t —it is most convenient simply to specify them for the first slice. For example, the complete DBN specification for the umbrella world is given by the three-node network shown in Figure 15.13(a). From this specification, the complete DBN with an unbounded number of time slices can be constructed as needed by copying the first slice.

Let us now consider a more interesting example: monitoring a battery-powered robot moving in the X - Y plane, as introduced at the end of Section 15.1. First, we need state variables, which will include both $\mathbf{X}_t = (X_t, Y_t)$ for position and $\dot{\mathbf{X}}_t = (\dot{X}_t, \dot{Y}_t)$ for velocity. We assume some method of measuring position—perhaps a fixed camera or onboard GPS (Global Positioning System)—yielding measurements \mathbf{Z}_t . The position at the next time step depends on the current position and velocity, as in the standard Kalman filter model. The velocity at the next step depends on the current velocity and the state of the battery. We add $Battery_t$ to represent the actual battery charge level, which has as parents the previous

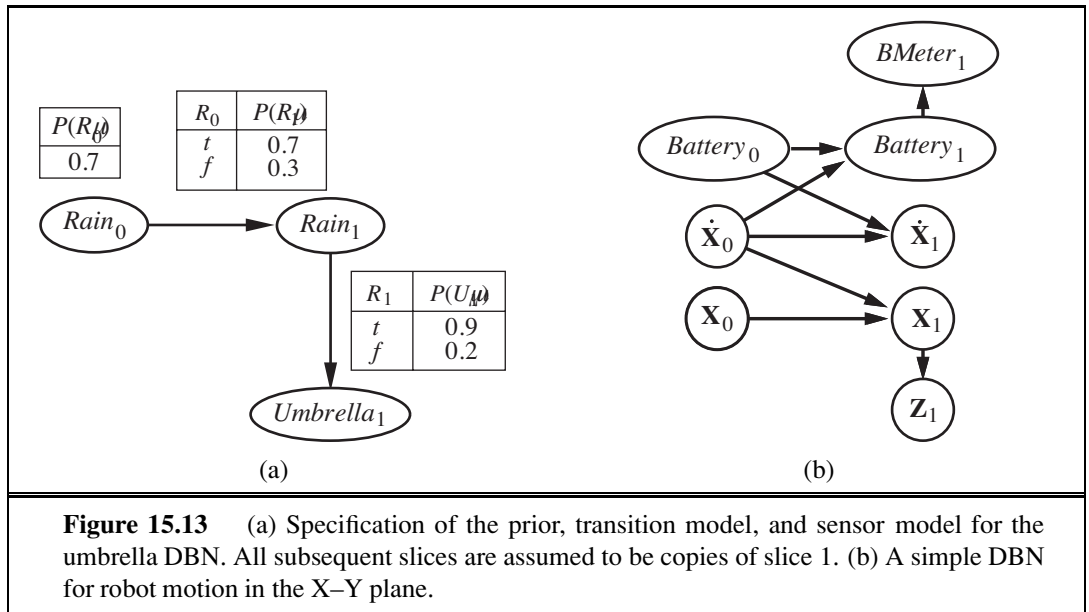


Figure 15.13 (a) Specification of the prior, transition model, and sensor model for the umbrella DBN. All subsequent slices are assumed to be copies of slice 1. (b) A simple DBN for robot motion in the X-Y plane.

battery level and the velocity, and we add $BMeter_t$, which measures the battery charge level. This gives us the basic model shown in Figure 15.13(b).

It is worth looking in more depth at the nature of the sensor model for $BMeter_t$. Let us suppose, for simplicity, that both $Battery_t$ and $BMeter_t$ can take on discrete values 0 through 5. If the meter is always accurate, then the CPT $\mathbf{P}(BMeter_t | Battery_t)$ should have probabilities of 1.0 “along the diagonal” and probabilities of 0.0 elsewhere. In reality, noise always creeps into measurements. For continuous measurements, a Gaussian distribution with a small variance might be used.⁵ For our discrete variables, we can approximate a Gaussian using a distribution in which the probability of error drops off in the appropriate way, so that the probability of a large error is very small. We use the term **Gaussian error model** to cover both the continuous and discrete versions.

Anyone with hands-on experience of robotics, computerized process control, or other forms of automatic sensing will readily testify to the fact that small amounts of measurement noise are often the least of one’s problems. Real sensors *fail*. When a sensor fails, it does not necessarily send a signal saying, “Oh, by the way, the data I’m about to send you is a load of nonsense.” Instead, it simply sends the nonsense. The simplest kind of failure is called a **transient failure**, where the sensor occasionally decides to send some nonsense. For example, the battery level sensor might have a habit of sending a zero when someone bumps the robot, even if the battery is fully charged.

Let’s see what happens when a transient failure occurs with a Gaussian error model that doesn’t accommodate such failures. Suppose, for example, that the robot is sitting quietly and observes 20 consecutive battery readings of 5. Then the battery meter has a temporary seizure

⁵ Strictly speaking, a Gaussian distribution is problematic because it assigns nonzero probability to large negative charge levels. The **beta distribution** is sometimes a better choice for a variable whose range is restricted.

and the next reading is $BMeter_{21} = 0$. What will the simple Gaussian error model lead us to believe about $Battery_{21}$? According to Bayes' rule, the answer depends on both the sensor model $\mathbf{P}(BMeter_{21} = 0 \mid Battery_{21})$ and the prediction $\mathbf{P}(Battery_{21} \mid BMeter_{1:20})$. If the probability of a large sensor error is significantly less likely than the probability of a transition to $Battery_{21} = 0$, even if the latter is very unlikely, then the posterior distribution will assign a high probability to the battery's being empty. A second reading of 0 at $t = 22$ will make this conclusion almost certain. If the transient failure then disappears and the reading returns to 5 from $t = 23$ onwards, the estimate for the battery level will quickly return to 5, as if by magic. This course of events is illustrated in the upper curve of Figure 15.14(a), which shows the expected value of $Battery_t$ over time, using a discrete Gaussian error model.

Despite the recovery, there is a time ($t = 22$) when the robot is convinced that its battery is empty; presumably, then, it should send out a mayday signal and shut down. Alas, its oversimplified sensor model has led it astray. How can this be fixed? Consider a familiar example from everyday human driving: on sharp curves or steep hills, one's "fuel tank empty" warning light sometimes turns on. Rather than looking for the emergency phone, one simply recalls that the fuel gauge sometimes gives a very large error when the fuel is sloshing around in the tank. The moral of the story is the following: *for the system to handle sensor failure properly, the sensor model must include the possibility of failure.*

The simplest kind of failure model for a sensor allows a certain probability that the sensor will return some completely incorrect value, regardless of the true state of the world. For example, if the battery meter fails by returning 0, we might say that

$$P(BMeter_t = 0 \mid Battery_t = 5) = 0.03 ,$$

which is presumably much larger than the probability assigned by the simple Gaussian error model. Let's call this the **transient failure model**. How does it help when we are faced with a reading of 0? Provided that the *predicted* probability of an empty battery, according to the readings so far, is much less than 0.03, then the best explanation of the observation $BMeter_{21} = 0$ is that the sensor has temporarily failed. Intuitively, we can think of the belief about the battery level as having a certain amount of "inertia" that helps to overcome temporary blips in the meter reading. The upper curve in Figure 15.14(b) shows that the transient failure model can handle transient failures without a catastrophic change in beliefs.

So much for temporary blips. What about a persistent sensor failure? Sadly, failures of this kind are all too common. If the sensor returns 20 readings of 5 followed by 20 readings of 0, then the transient sensor failure model described in the preceding paragraph will result in the robot gradually coming to believe that its battery is empty when in fact it may be that the meter has failed. The lower curve in Figure 15.14(b) shows the belief "trajectory" for this case. By $t = 25$ —five readings of 0—the robot is convinced that its battery is empty. Obviously, we would prefer the robot to believe that its battery meter is broken—if indeed this is the more likely event.

Unsurprisingly, to handle persistent failure, we need a **persistent failure model** that describes how the sensor behaves under normal conditions and after failure. To do this, we need to augment the state of the system with an additional variable, say, $BMBroken$, that describes the status of the battery meter. The persistence of failure must be modeled by an



TRANSIENT FAILURE
MODEL

PERSISTENT
FAILURE MODEL

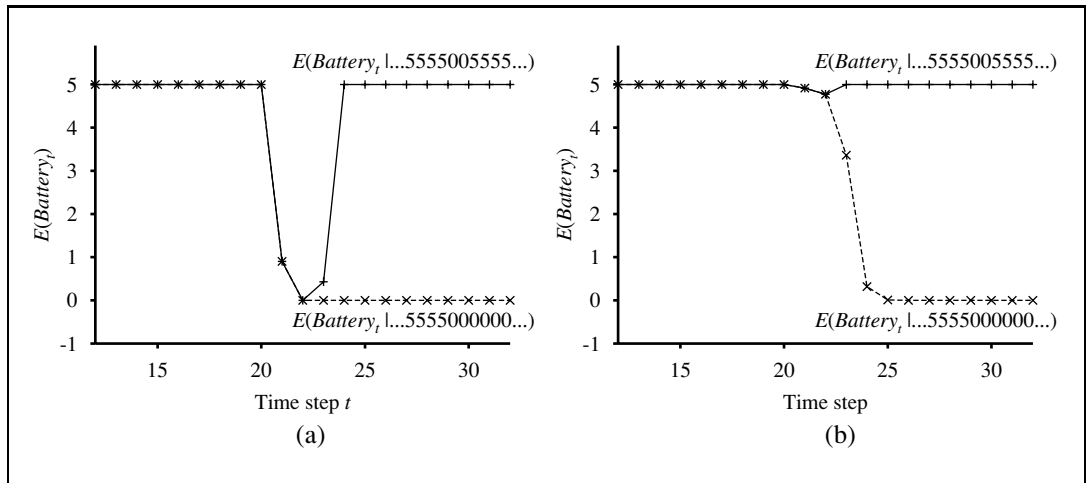


Figure 15.14 (a) Upper curve: trajectory of the expected value of $Battery_t$ for an observation sequence consisting of all 5s except for 0s at $t = 21$ and $t = 22$, using a simple Gaussian error model. Lower curve: trajectory when the observation remains at 0 from $t = 21$ onwards. (b) The same experiment run with the transient failure model. Notice that the transient failure is handled well, but the persistent failure results in excessive pessimism about the battery charge.

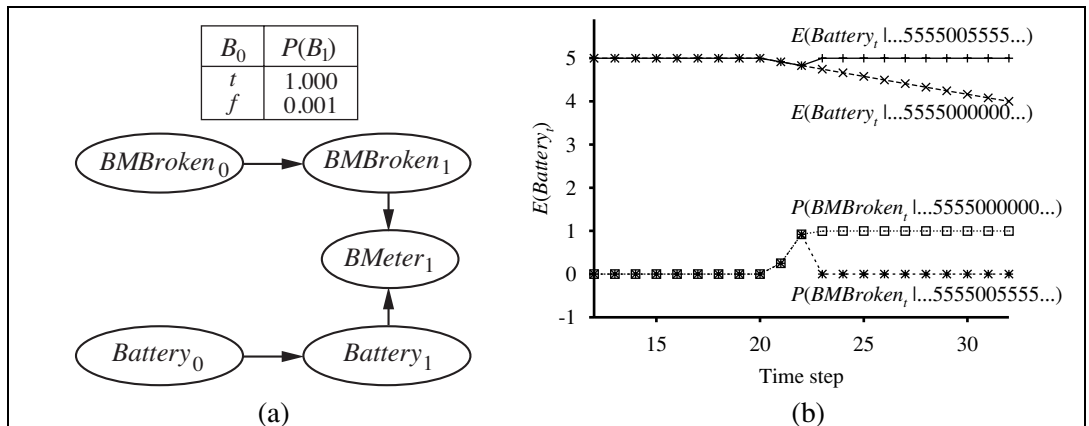
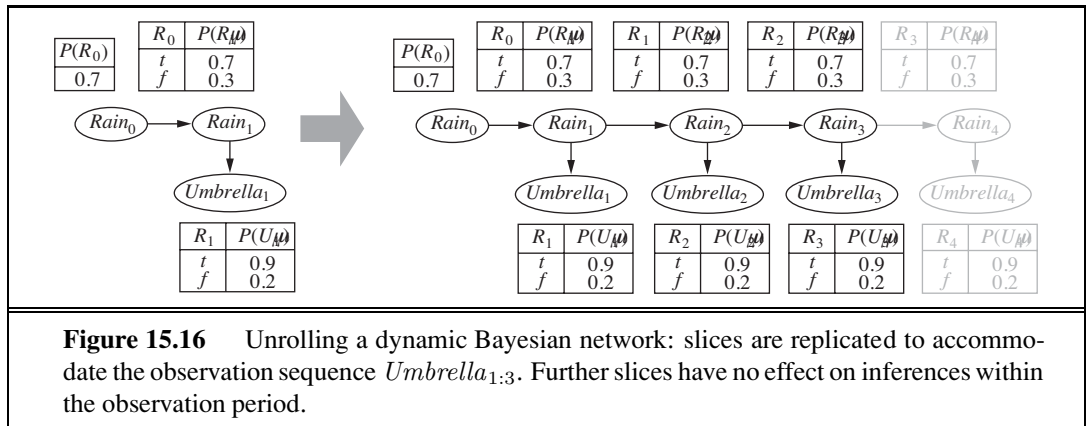


Figure 15.15 (a) A DBN fragment showing the sensor status variable required for modeling persistent failure of the battery sensor. (b) Upper curves: trajectories of the expected value of $Battery_t$ for the "transient failure" and "permanent failure" observation sequences. Lower curves: probability trajectories for $BMBroken$ given the two observation sequences.

PERSISTENCE ARC

arc linking $BMBroken_0$ to $BMBroken_1$. This **persistence arc** has a CPT that gives a small probability of failure in any given time step, say, 0.001, but specifies that the sensor stays broken once it breaks. When the sensor is OK, the sensor model for $BMeter$ is identical to the transient failure model; when the sensor is broken, it says $BMeter$ is always 0, regardless of the actual battery charge.



The persistent failure model for the battery sensor is shown in Figure 15.15(a). Its performance on the two data sequences (temporary blip and persistent failure) is shown in Figure 15.15(b). There are several things to notice about these curves. First, in the case of the temporary blip, the probability that the sensor is broken rises significantly after the second 0 reading, but immediately drops back to zero once a 5 is observed. Second, in the case of persistent failure, the probability that the sensor is broken rises quickly to almost 1 and stays there. Finally, once the sensor is known to be broken, the robot can only assume that its battery discharges at the “normal” rate, as shown by the gradually descending level of $E(Battery_t | \dots)$.

So far, we have merely scratched the surface of the problem of representing complex processes. The variety of transition models is huge, encompassing topics as disparate as modeling the human endocrine system and modeling multiple vehicles driving on a freeway. Sensor modeling is also a vast subfield in itself, but even subtle phenomena, such as sensor drift, sudden decalibration, and the effects of exogenous conditions (such as weather) on sensor readings, can be handled by explicit representation within dynamic Bayesian networks.

15.5.2 Exact inference in DBNs

Having sketched some ideas for representing complex processes as DBNs, we now turn to the question of inference. In a sense, this question has already been answered: dynamic Bayesian networks *are* Bayesian networks, and we already have algorithms for inference in Bayesian networks. Given a sequence of observations, one can construct the full Bayesian network representation of a DBN by replicating slices until the network is large enough to accommodate the observations, as in Figure 15.16. This technique, mentioned in Chapter 14 in the context of relational probability models, is called **unrolling**. (Technically, the DBN is equivalent to the semi-infinite network obtained by unrolling forever. Slices added beyond the last observation have no effect on inferences within the observation period and can be omitted.) Once the DBN is unrolled, one can use any of the inference algorithms—variable elimination, clustering methods, and so on—described in Chapter 14.

Unfortunately, a naive application of unrolling would not be particularly efficient. If we want to perform filtering or smoothing with a long sequence of observations $\mathbf{e}_{1:t}$, the

unrolled network would require $O(t)$ space and would thus grow without bound as more observations were added. Moreover, if we simply run the inference algorithm anew each time an observation is added, the inference time per update will also increase as $O(t)$.

Looking back to Section 15.2.1, we see that constant time and space per filtering update can be achieved if the computation can be done recursively. Essentially, the filtering update in Equation (15.5) works by *summing out* the state variables of the previous time step to get the distribution for the new time step. Summing out variables is exactly what the **variable elimination** (Figure 14.11) algorithm does, and it turns out that running variable elimination with the variables in temporal order exactly mimics the operation of the recursive filtering update in Equation (15.5). The modified algorithm keeps at most two slices in memory at any one time: starting with slice 0, we add slice 1, then sum out slice 0, then add slice 2, then sum out slice 1, and so on. In this way, we can achieve constant space and time per filtering update. (The same performance can be achieved by suitable modifications to the clustering algorithm.) Exercise 15.17 asks you to verify this fact for the umbrella network.

So much for the good news; now for the bad news: It turns out that the “constant” for the per-update time and space complexity is, in almost all cases, exponential in the number of state variables. What happens is that, as the variable elimination proceeds, the factors grow to include all the state variables (or, more precisely, all those state variables that have parents in the previous time slice). The maximum factor size is $O(d^{n+k})$ and the total update cost per step is $O(nd^{n+k})$, where d is the domain size of the variables and k is the maximum number of parents of any state variable.

Of course, this is much less than the cost of HMM updating, which is $O(d^{2n})$, but it is still infeasible for large numbers of variables. This grim fact is somewhat hard to accept. What it means is that *even though we can use DBNs to represent very complex temporal processes with many sparsely connected variables, we cannot reason efficiently and exactly about those processes*. The DBN model itself, which represents the prior joint distribution over all the variables, is factorable into its constituent CPTs, but the posterior joint distribution conditioned on an observation sequence—that is, the forward message—is generally *not* factorable. So far, no one has found a way around this problem, despite the fact that many important areas of science and engineering would benefit enormously from its solution. Thus, we must fall back on approximate methods.



15.5.3 Approximate inference in DBNs

Section 14.5 described two approximation algorithms: likelihood weighting (Figure 14.15) and Markov chain Monte Carlo (MCMC, Figure 14.16). Of the two, the former is most easily adapted to the DBN context. (An MCMC filtering algorithm is described briefly in the notes at the end of the chapter.) We will see, however, that several improvements are required over the standard likelihood weighting algorithm before a practical method emerges.

Recall that likelihood weighting works by sampling the nonevidence nodes of the network in topological order, weighting each sample by the likelihood it accords to the observed evidence variables. As with the exact algorithms, we could apply likelihood weighting directly to an unrolled DBN, but this would suffer from the same problems of increasing time



and space requirements per update as the observation sequence grows. The problem is that the standard algorithm runs each sample in turn, all the way through the network. Instead, we can simply run all N samples together through the DBN, one slice at a time. The modified algorithm fits the general pattern of filtering algorithms, with the set of N samples as the forward message. The first key innovation, then, is to *use the samples themselves as an approximate representation of the current state distribution*. This meets the requirement of a “constant” time per update, although the constant depends on the number of samples required to maintain an accurate approximation. There is also no need to unroll the DBN, because we need to have in memory only the current slice and the next slice.

In our discussion of likelihood weighting in Chapter 14, we pointed out that the algorithm’s accuracy suffers if the evidence variables are “downstream” from the variables being sampled, because in that case the samples are generated without any influence from the evidence. Looking at the typical structure of a DBN—say, the umbrella DBN in Figure 15.16—we see that indeed the early state variables will be sampled without the benefit of the later evidence. In fact, looking more carefully, we see that *none* of the state variables has *any* evidence variables among its ancestors! Hence, although the weight of each sample will depend on the evidence, the actual set of samples generated will be *completely independent* of the evidence. For example, even if the boss brings in the umbrella every day, the sampling process could still hallucinate endless days of sunshine. What this means in practice is that the fraction of samples that remain reasonably close to the actual series of events (and therefore have nonnegligible weights) drops exponentially with t , the length of the observation sequence. In other words, to maintain a given level of accuracy, we need to increase the number of samples exponentially with t . Given that a filtering algorithm that works in real time can use only a fixed number of samples, what happens in practice is that the error blows up after a very small number of update steps.



Clearly, we need a better solution. The second key innovation is to *focus the set of samples on the high-probability regions of the state space*. This can be done by throwing away samples that have very low weight, according to the observations, while replicating those that have high weight. In that way, the population of samples will stay reasonably close to reality. If we think of samples as a resource for modeling the posterior distribution, then it makes sense to use more samples in regions of the state space where the posterior is higher.

PARTICLE FILTERING

A family of algorithms called **particle filtering** is designed to do just that. Particle filtering works as follows: First, a population of N initial-state samples is created by sampling from the prior distribution $\mathbf{P}(\mathbf{X}_0)$. Then the update cycle is repeated for each time step:

1. Each sample is propagated forward by sampling the next state value \mathbf{x}_{t+1} given the current value \mathbf{x}_t for the sample, based on the transition model $\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{x}_t)$.
2. Each sample is weighted by the likelihood it assigns to the new evidence, $P(\mathbf{e}_{t+1} \mid \mathbf{x}_{t+1})$.
3. The population is *resampled* to generate a new population of N samples. Each new sample is selected from the current population; the probability that a particular sample is selected is proportional to its weight. The new samples are unweighted.

The algorithm is shown in detail in Figure 15.17, and its operation for the umbrella DBN is illustrated in Figure 15.18.

```

function PARTICLE-FILTERING( $\mathbf{e}, N, dbn$ ) returns a set of samples for the next time step
  inputs:  $\mathbf{e}$ , the new incoming evidence
            $N$ , the number of samples to be maintained
            $dbn$ , a DBN with prior  $\mathbf{P}(\mathbf{X}_0)$ , transition model  $\mathbf{P}(\mathbf{X}_1|\mathbf{X}_0)$ , sensor model  $\mathbf{P}(\mathbf{E}_1|\mathbf{X}_1)$ 
  persistent:  $S$ , a vector of samples of size  $N$ , initially generated from  $\mathbf{P}(\mathbf{X}_0)$ 
  local variables:  $W$ , a vector of weights of size  $N$ 

  for  $i = 1$  to  $N$  do
     $S[i] \leftarrow$  sample from  $\mathbf{P}(\mathbf{X}_1 | \mathbf{X}_0 = S[i])$  /* step 1 */
     $W[i] \leftarrow \mathbf{P}(\mathbf{e} | \mathbf{X}_1 = S[i])$  /* step 2 */
   $S \leftarrow$  WEIGHTED-SAMPLE-WITH-REPLACEMENT( $N, S, W$ ) /* step 3 */
  return  $S$ 

```

Figure 15.17 The particle filtering algorithm implemented as a recursive update operation with state (the set of samples). Each of the sampling operations involves sampling the relevant slice variables in topological order, much as in PRIOR-SAMPLE. The WEIGHTED-SAMPLE-WITH-REPLACEMENT operation can be implemented to run in $O(N)$ expected time. The step numbers refer to the description in the text.

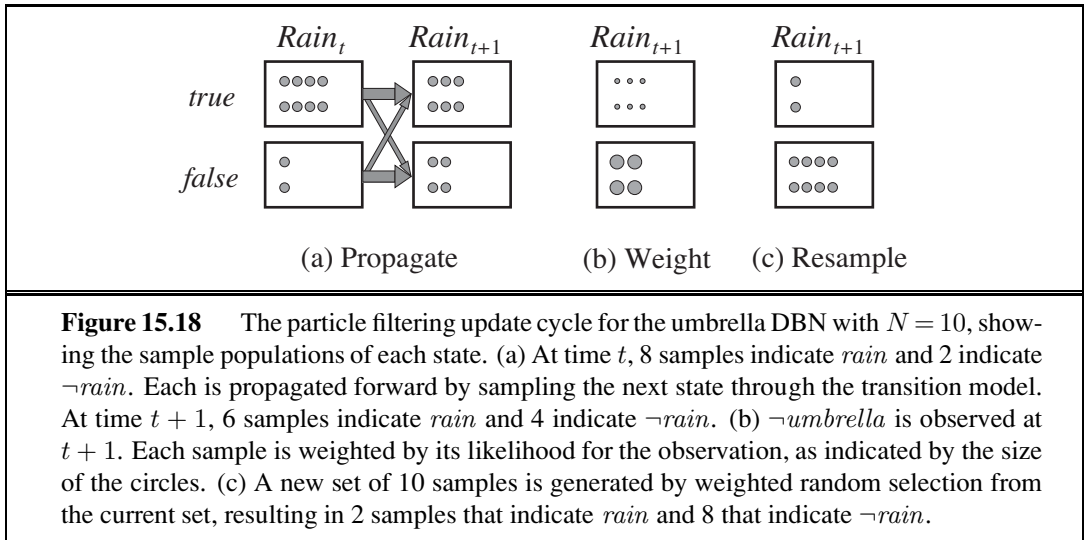


Figure 15.18 The particle filtering update cycle for the umbrella DBN with $N = 10$, showing the sample populations of each state. (a) At time t , 8 samples indicate *rain* and 2 indicate \neg *rain*. Each is propagated forward by sampling the next state through the transition model. At time $t + 1$, 6 samples indicate *rain* and 4 indicate \neg *rain*. (b) \neg *umbrella* is observed at $t + 1$. Each sample is weighted by its likelihood for the observation, as indicated by the size of the circles. (c) A new set of 10 samples is generated by weighted random selection from the current set, resulting in 2 samples that indicate *rain* and 8 that indicate \neg *rain*.

We can show that this algorithm is consistent—gives the correct probabilities as N tends to infinity—by considering what happens during one update cycle. We assume that the sample population starts with a correct representation of the forward message $\mathbf{f}_{1:t} = \mathbf{P}(\mathbf{X}_t | \mathbf{e}_{1:t})$ at time t . Writing $N(\mathbf{x}_t | \mathbf{e}_{1:t})$ for the number of samples occupying state \mathbf{x}_t after observations $\mathbf{e}_{1:t}$ have been processed, we therefore have

$$N(\mathbf{x}_t | \mathbf{e}_{1:t})/N = P(\mathbf{x}_t | \mathbf{e}_{1:t}) \quad (15.23)$$

for large N . Now we propagate each sample forward by sampling the state variables at $t + 1$, given the values for the sample at t . The number of samples reaching state \mathbf{x}_{t+1} from each

\mathbf{x}_t is the transition probability times the population of \mathbf{x}_t ; hence, the total number of samples reaching \mathbf{x}_{t+1} is

$$N(\mathbf{x}_{t+1} | \mathbf{e}_{1:t}) = \sum_{\mathbf{x}_t} P(\mathbf{x}_{t+1} | \mathbf{x}_t) N(\mathbf{x}_t | \mathbf{e}_{1:t}) .$$

Now we weight each sample by its likelihood for the evidence at $t + 1$. A sample in state \mathbf{x}_{t+1} receives weight $P(\mathbf{e}_{t+1} | \mathbf{x}_{t+1})$. The total weight of the samples in \mathbf{x}_{t+1} after seeing \mathbf{e}_{t+1} is therefore

$$W(\mathbf{x}_{t+1} | \mathbf{e}_{1:t+1}) = P(\mathbf{e}_{t+1} | \mathbf{x}_{t+1}) N(\mathbf{x}_{t+1} | \mathbf{e}_{1:t}) .$$

Now for the resampling step. Since each sample is replicated with probability proportional to its weight, the number of samples in state \mathbf{x}_{t+1} after resampling is proportional to the total weight in \mathbf{x}_{t+1} before resampling:

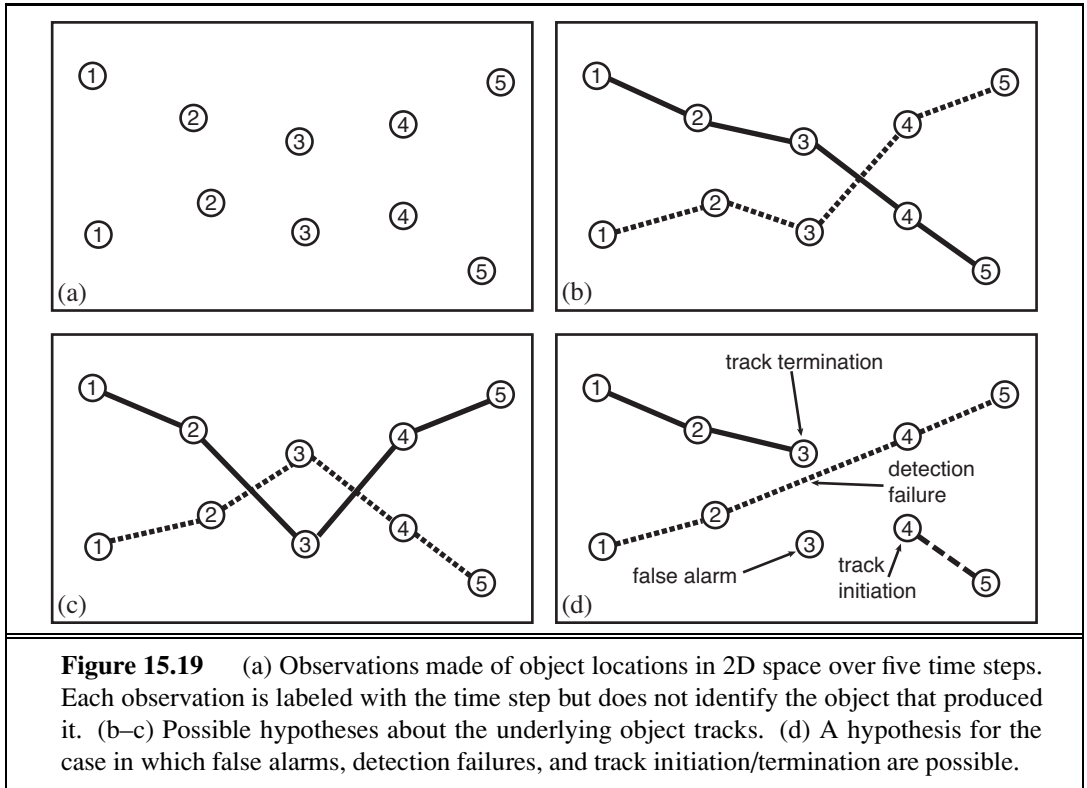
$$\begin{aligned} N(\mathbf{x}_{t+1} | \mathbf{e}_{1:t+1}) / N &= \alpha W(\mathbf{x}_{t+1} | \mathbf{e}_{1:t+1}) \\ &= \alpha P(\mathbf{e}_{t+1} | \mathbf{x}_{t+1}) N(\mathbf{x}_{t+1} | \mathbf{e}_{1:t}) \\ &= \alpha P(\mathbf{e}_{t+1} | \mathbf{x}_{t+1}) \sum_{\mathbf{x}_t} P(\mathbf{x}_{t+1} | \mathbf{x}_t) N(\mathbf{x}_t | \mathbf{e}_{1:t}) \\ &= \alpha N P(\mathbf{e}_{t+1} | \mathbf{x}_{t+1}) \sum_{\mathbf{x}_t} P(\mathbf{x}_{t+1} | \mathbf{x}_t) P(\mathbf{x}_t | \mathbf{e}_{1:t}) \quad (\text{by 15.23}) \\ &= \alpha' P(\mathbf{e}_{t+1} | \mathbf{x}_{t+1}) \sum_{\mathbf{x}_t} P(\mathbf{x}_{t+1} | \mathbf{x}_t) P(\mathbf{x}_t | \mathbf{e}_{1:t}) \\ &= P(\mathbf{x}_{t+1} | \mathbf{e}_{1:t+1}) \quad (\text{by 15.5}). \end{aligned}$$

Therefore the sample population after one update cycle correctly represents the forward message at time $t + 1$.

Particle filtering is *consistent*, therefore, but is it *efficient*? In practice, it seems that the answer is yes: particle filtering seems to maintain a good approximation to the true posterior using a constant number of samples. Under certain assumptions—in particular, that the probabilities in the transition and sensor models are strictly greater than 0 and less than 1—it is possible to prove that the approximation maintains bounded error with high probability. On the practical side, the range of applications has grown to include many fields of science and engineering; some references are given at the end of the chapter.

15.6 KEEPING TRACK OF MANY OBJECTS

The preceding sections have considered—without mentioning it—state estimation problems involving a single object. In this section, we see what happens when two or more objects generate the observations. What makes this case different from plain old state estimation is that there is now the possibility of *uncertainty* about which object generated which observation. This is the **identity uncertainty** problem of Section 14.6.3 (page 544), now viewed in a temporal context. In the control theory literature, this is the **data association** problem—that is, the problem of associating observation data with the objects that generated them.



The data association problem was studied originally in the context of radar tracking, where reflected pulses are detected at fixed time intervals by a rotating radar antenna. At each time step, multiple blips may appear on the screen, but there is no direct observation of which blips at time t belong to which blips at time $t - 1$. Figure 15.19(a) shows a simple example with two blips per time step for five steps. Let the two blip locations at time t be e_t^1 and e_t^2 . (The labeling of blips within a time step as “1” and “2” is completely arbitrary and carries no information.) Let us assume, for the time being, that exactly two aircraft, A and B , generated the blips; their true positions are X_t^A and X_t^B . Just to keep things simple, we’ll also assume that each aircraft moves independently according to a known transition model—e.g., a linear Gaussian model as used in the Kalman filter (Section 15.4).

Suppose we try to write down the overall probability model for this scenario, just as we did for general temporal processes in Equation (15.3) on page 569. As usual, the joint distribution factors into contributions for each time step as follows:

$$P(x_{0:t}^A, x_{0:t}^B, e_{1:t}^1, e_{1:t}^2) = P(x_0^A)P(x_0^B) \prod_{i=1}^t P(x_i^A | x_{i-1}^A)P(x_i^B | x_{i-1}^B) P(e_i^1, e_i^2 | x_i^A, x_i^B). \quad (15.24)$$

We would like to factor the observation term $P(e_i^1, e_i^2 | x_i^A, x_i^B)$ into a product of two terms, one for each object, but this would require knowing which observation was generated by which object. Instead, we have to sum over all possible ways of associating the observations

with the objects. Some of those ways are shown in Figure 15.19(b–c); in general, for n objects and T time steps, there are $(n!)^T$ ways of doing it—an awfully large number.

Mathematically speaking, the “way of associating the observations with the objects” is a collection of unobserved random variable that identify the source of each observation. We’ll write ω_t to denote the one-to-one mapping from objects to observations at time t , with $\omega_t(A)$ and $\omega_t(B)$ denoting the specific observations (1 or 2) that ω_t assigns to A and B . (For n objects, ω_t will have $n!$ possible values; here, $n! = 2$.) Because the labels “1” and “2” on the observations are assigned arbitrarily, the prior on ω_t is uniform and ω_t is independent of the states of the objects, x_t^A and x_t^B . So we can condition the observation term $P(e_i^1, e_i^2 | x_i^A, x_i^B)$ on ω_i and then simplify:

$$\begin{aligned} P(e_i^1, e_i^2 | x_i^A, x_i^B) &= \sum_{\omega_i} P(e_i^1, e_i^2 | x_i^A, x_i^B, \omega_i) P(\omega_i | x_i^A, x_i^B) \\ &= \sum_{\omega_i} P(e_i^{\omega_i(A)} | x_i^A) P(e_i^{\omega_i(B)} | x_i^B) P(\omega_i | x_i^A, x_i^B) \\ &= \frac{1}{2} \sum_{\omega_i} P(e_i^{\omega_i(A)} | x_i^A) P(e_i^{\omega_i(B)} | x_i^B). \end{aligned}$$

Plugging this into Equation (15.24), we get an expression that is only in terms of transition and sensor models for individual objects and observations.

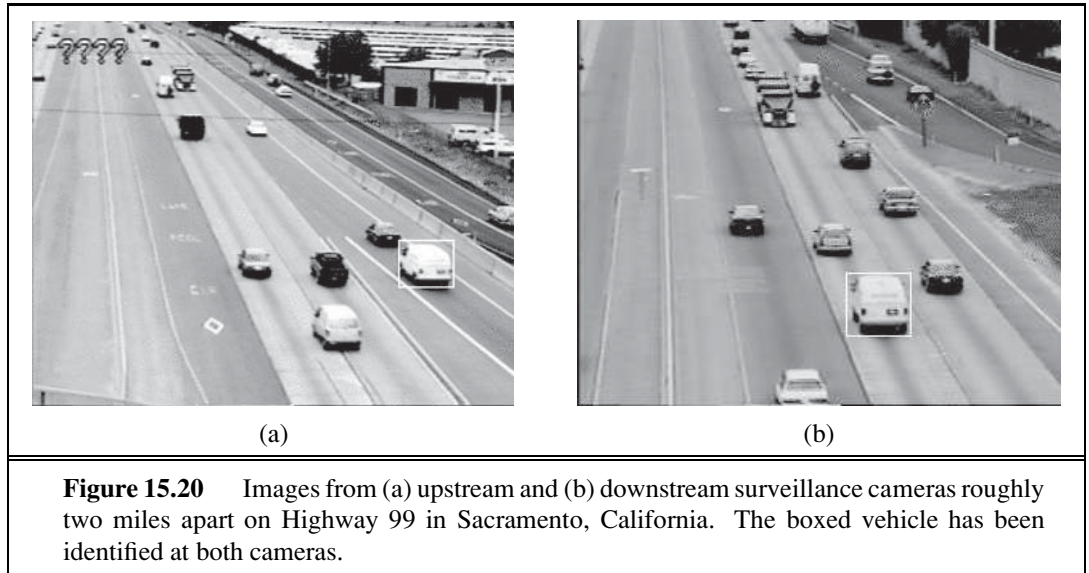
As for all probability models, inference means summing out the variables other than the query and the evidence. For filtering in HMMs and DBNs, we were able to sum out the state variables from 1 to $t - 1$ by a simple dynamic programming trick; for Kalman filters, we took advantage of special properties of Gaussians. For data association, we are less fortunate. There is no (known) efficient exact algorithm, for the same reason that there is none for the switching Kalman filter (page 589): the filtering distribution $P(x_t^A | e_{1:t}^1, e_{1:t}^2)$ for object A ends up as a mixture of exponentially many distributions, one for each way of picking a sequence of observations to assign to A .

As a result of the complexity of exact inference, many different approximate methods have been used. The simplest approach is to choose a single “best” assignment at each time step, given the predicted positions of the objects at the current time step. This assignment associates observations with objects and enables the track of each object to be updated and a prediction made for the next time step. For choosing the “best” assignment, it is common to use the so-called **nearest-neighbor filter**, which repeatedly chooses the closest pairing of predicted position and observation and adds that pairing to the assignment. The nearest-neighbor filter works well when the objects are well separated in state space and the prediction uncertainty and observation error are small—in other words, when there is no possibility of confusion. When there is more uncertainty as to the correct assignment, a better approach is to choose the assignment that maximizes the joint probability of the current observations given the predicted positions. This can be done very efficiently using the **Hungarian algorithm** (Kuhn, 1955), even though there are $n!$ assignments to choose from.

Any method that commits to a single best assignment at each time step fails miserably under more difficult conditions. In particular, if the algorithm commits to an incorrect assignment, the prediction at the next time step may be significantly wrong, leading to more

NEAREST-NEIGHBOR
FILTER

HUNGARIAN
ALGORITHM



incorrect assignments, and so on. Two modern approaches turn out to be much more effective. A **particle filtering** algorithm (see page 598) for data association works by maintaining a large collection of possible current assignments. An **MCMC** algorithm explores the space of assignment histories—for example, Figure 15.19(b–c) might be states in the MCMC state space—and can change its mind about previous assignment decisions. Current MCMC data association methods can handle many hundreds of objects in real time while giving a good approximation to the true posterior distributions.

The scenario described so far involved n known objects generating n observations at each time step. Real application of data association are typically much more complicated. Often, the reported observations include **false alarms** (also known as **clutter**), which are not caused by real objects. **Detection failures** can occur, meaning that no observation is reported for a real object. Finally, new objects arrive and old ones disappear. These phenomena, which create even more possible worlds to worry about, are illustrated in Figure 15.19(d).

Figure 15.20 shows two images from widely separated cameras on a California freeway. In this application, we are interested in two goals: estimating the time it takes, under current traffic conditions, to go from one place to another in the freeway system; and measuring *demand*, i.e., how many vehicles travel between any two points in the system at particular times of the day and on particular days of the week. Both goals require solving the data association problem over a wide area with many cameras and tens of thousands of vehicles per hour. With visual surveillance, false alarms are caused by moving shadows, articulated vehicles, reflections in puddles, etc.; detection failures are caused by occlusion, fog, darkness, and lack of visual contrast; and vehicles are constantly entering and leaving the freeway system. Furthermore, the appearance of any given vehicle can change dramatically between cameras depending on lighting conditions and vehicle pose in the image, and the transition model changes as traffic jams come and go. Despite these problems, modern data association algorithms have been successful in estimating traffic parameters in real-world settings.

FALSE ALARM

CLUTTER

DETECTION FAILURE

Data association is an essential foundation for keeping track of a complex world, because without it there is no way to combine multiple observations of any given object. When objects in the world interact with each other in complex activities, understanding the world requires combining data association with the relational and open-universe probability models of Section 14.6.3. This is currently an active area of research.

15.7 SUMMARY

This chapter has addressed the general problem of representing and reasoning about probabilistic temporal processes. The main points are as follows:

- The changing state of the world is handled by using a set of random variables to represent the state at each point in time.
- Representations can be designed to satisfy the **Markov property**, so that the future is independent of the past given the present. Combined with the assumption that the process is **stationary**—that is, the dynamics do not change over time—this greatly simplifies the representation.
- A temporal probability model can be thought of as containing a **transition model** describing the state evolution and a **sensor model** describing the observation process.
- The principal inference tasks in temporal models are **filtering**, **prediction**, **smoothing**, and computing the **most likely explanation**. Each of these can be achieved using simple, recursive algorithms whose run time is linear in the length of the sequence.
- Three families of temporal models were studied in more depth: **hidden Markov models**, **Kalman filters**, and **dynamic Bayesian networks** (which include the other two as special cases).
- Unless special assumptions are made, as in Kalman filters, exact inference with many state variables is intractable. In practice, the **particle filtering** algorithm seems to be an effective approximation algorithm.
- When trying to keep track of many objects, uncertainty arises as to which observations belong to which objects—the **data association** problem. The number of association hypotheses is typically intractably large, but MCMC and particle filtering algorithms for data association work well in practice.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Many of the basic ideas for estimating the state of dynamical systems came from the mathematician C. F. Gauss (1809), who formulated a deterministic least-squares algorithm for the problem of estimating orbits from astronomical observations. A. A. Markov (1913) developed what was later called the **Markov assumption** in his analysis of stochastic processes;

he estimated a first-order Markov chain on letters from the text of *Eugene Onegin*. The general theory of Markov chains and their mixing times is covered by Levin *et al.* (2008).

Significant classified work on filtering was done during World War II by Wiener (1942) for continuous-time processes and by Kolmogorov (1941) for discrete-time processes. Although this work led to important technological developments over the next 20 years, its use of a frequency-domain representation made many calculations quite cumbersome. Direct state-space modeling of the stochastic process turned out to be simpler, as shown by Peter Swerling (1959) and Rudolf Kalman (1960). The latter paper described what is now known as the Kalman filter for forward inference in linear systems with Gaussian noise; Kalman's results had, however, been obtained previously by the Danish statistician Thorvald Thiele (1880) and by the Russian mathematician Ruslan Stratonovich (1959), whom Kalman met in Moscow in 1960. After a visit to NASA Ames Research Center in 1960, Kalman saw the applicability of the method to the tracking of rocket trajectories, and the filter was later implemented for the Apollo missions. Important results on smoothing were derived by Rauch *et al.* (1965), and the impressively named Rauch–Tung–Striebel smoother is still a standard technique today. Many early results are gathered in Gelb (1974). Bar-Shalom and Fortmann (1988) give a more modern treatment with a Bayesian flavor, as well as many references to the vast literature on the subject. Chatfield (1989) and Box *et al.* (1994) cover the control theory approach to time series analysis.

The hidden Markov model and associated algorithms for inference and learning, including the forward–backward algorithm, were developed by Baum and Petrie (1966). The Viterbi algorithm first appeared in (Viterbi, 1967). Similar ideas also appeared independently in the Kalman filtering community (Rauch *et al.*, 1965). The forward–backward algorithm was one of the main precursors of the general formulation of the EM algorithm (Dempster *et al.*, 1977); see also Chapter 20. Constant-space smoothing appears in Binder *et al.* (1997b), as does the divide-and-conquer algorithm developed in Exercise 15.3. Constant-time fixed-lag smoothing for HMMs first appeared in Russell and Norvig (2003). HMMs have found many applications in language processing (Charniak, 1993), speech recognition (Rabiner and Juang, 1993), machine translation (Och and Ney, 2003), computational biology (Krogh *et al.*, 1994; Baldi *et al.*, 1994), financial economics Bhar and Hamori (2004) and other fields. There have been several extensions to the basic HMM model, for example the Hierarchical HMM (Fine *et al.*, 1998) and Layered HMM (Oliver *et al.*, 2004) introduce structure back into the model, replacing the single state variable of HMMs.

Dynamic Bayesian networks (DBNs) can be viewed as a sparse encoding of a Markov process and were first used in AI by Dean and Kanazawa (1989b), Nicholson and Brady (1992), and Kjaerulff (1992). The last work extends the HUGIN Bayes net system to accommodate dynamic Bayesian networks. The book by Dean and Wellman (1991) helped popularize DBNs and the probabilistic approach to planning and control within AI. Murphy (2002) provides a thorough analysis of DBNs.

Dynamic Bayesian networks have become popular for modeling a variety of complex motion processes in computer vision (Huang *et al.*, 1994; Intille and Bobick, 1999). Like HMMs, they have found applications in speech recognition (Zweig and Russell, 1998; Richardson *et al.*, 2000; Stephenson *et al.*, 2000; Nefian *et al.*, 2002; Livescu *et al.*, 2003), ge-

nomics (Murphy and Mian, 1999; Perrin *et al.*, 2003; Husmeier, 2003) and robot localization (Theocharous *et al.*, 2004). The link between HMMs and DBNs, and between the forward–backward algorithm and Bayesian network propagation, was made explicitly by Smyth *et al.* (1997). A further unification with Kalman filters (and other statistical models) appears in Roweis and Ghahramani (1999). Procedures exist for learning the parameters (Binder *et al.*, 1997a; Ghahramani, 1998) and structures (Friedman *et al.*, 1998) of DBNs.

The particle filtering algorithm described in Section 15.5 has a particularly interesting history. The first sampling algorithms for particle filtering (also called sequential Monte Carlo methods) were developed in the control theory community by Handschin and Mayne (1969), and the resampling idea that is the core of particle filtering appeared in a Russian control journal (Zaritskii *et al.*, 1975). It was later reinvented in statistics as **sequential importance-sampling resampling**, or **SIR** (Rubin, 1988; Liu and Chen, 1998), in control theory as particle filtering (Gordon *et al.*, 1993; Gordon, 1994), in AI as **survival of the fittest** (Kanazawa *et al.*, 1995), and in computer vision as **condensation** (Isard and Blake, 1996). The paper by Kanazawa *et al.* (1995) includes an improvement called **evidence reversal** whereby the state at time $t + 1$ is sampled conditional on both the state at time t and the evidence at time $t + 1$. This allows the evidence to influence sample generation directly and was proved by Doucet (1997) and Liu and Chen (1998) to reduce the approximation error. Particle filtering has been applied in many areas, including tracking complex motion patterns in video (Isard and Blake, 1996), predicting the stock market (de Freitas *et al.*, 2000), and diagnosing faults on planetary rovers (Verma *et al.*, 2004). A variant called the **Rao-Blackwellized particle filter** or **RBPF** (Doucet *et al.*, 2000; Murphy and Russell, 2001) applies particle filtering to a subset of state variables and, for each particle, performs exact inference on the remaining variables conditioned on the value sequence in the particle. In some cases RBPF works well with thousands of state variables. An application of RBPF to localization and mapping in robotics is described in Chapter 25. The book by Doucet *et al.* (2001) collects many important papers on **sequential Monte Carlo** (SMC) algorithms, of which particle filtering is the most important instance. Pierre Del Moral and colleagues have performed extensive theoretical analyses of SMC algorithms (Del Moral, 2004; Del Moral *et al.*, 2006).

MCMC methods (see Section 14.5.2) can be applied to the filtering problem; for example, Gibbs sampling can be applied directly to an unrolled DBN. To avoid the problem of increasing update times as the unrolled network grows, the **decayed MCMC** filter (Marthi *et al.*, 2002) prefers to sample more recent state variables, with a probability that decays as $1/k^2$ for a variable k steps into the past. Decayed MCMC is a provably nondivergent filter. Nondivergence theorems can also be obtained for certain types of **assumed-density filter**. An assumed-density filter assumes that the posterior distribution over states at time t belongs to a particular finitely parameterized family; if the projection and update steps take it outside this family, the distribution is projected back to give the best approximation within the family. For DBNs, the Boyen–Koller algorithm (Boyen *et al.*, 1999) and the **factored frontier** algorithm (Murphy and Weiss, 2001) assume that the posterior distribution can be approximated well by a product of small factors. Variational techniques (see Chapter 14) have also been developed for temporal models. Ghahramani and Jordan (1997) discuss an approximation algorithm for the **factorial HMM**, a DBN in which two or more independently evolving

EVIDENCE
REVERSALRAO-
BLACKWELLIZED
PARTICLE FILTERSEQUENTIAL MONTE
CARLO

DECAYED MCMC

ASSUMED-DENSITY
FILTERFACTORED
FRONTIER

FACTORIAL HMM

Markov chains are linked by a shared observation stream. Jordan *et al.* (1998) cover a number of other applications.

Data association for multitarget tracking was first described in a probabilistic setting by Sittler (1964). The first practical algorithm for large-scale problems was the “multiple hypothesis tracker” or MHT algorithm (Reid, 1979). Many important papers are collected by Bar-Shalom and Fortmann (1988) and Bar-Shalom (1992). The development of an MCMC algorithm for data association is due to Pasula *et al.* (1999), who applied it to traffic surveillance problems. Oh *et al.* (2009) provide a formal analysis and extensive experimental comparisons to other methods. Schulz *et al.* (2003) describe a data association method based on particle filtering. Ingemar Cox analyzed the complexity of data association (Cox, 1993; Cox and Hingorani, 1994) and brought the topic to the attention of the vision community. He also noted the applicability of the polynomial-time Hungarian algorithm to the problem of finding most-likely assignments, which had long been considered an intractable problem in the tracking community. The algorithm itself was published by Kuhn (1955), based on translations of papers published in 1931 by two Hungarian mathematicians, Dénes König and Jenő Egerváry. The basic theorem had been derived previously, however, in an unpublished Latin manuscript by the famous Prussian mathematician Carl Gustav Jacobi (1804–1851).

EXERCISES

15.1 Show that any second-order Markov process can be rewritten as a first-order Markov process with an augmented set of state variables. Can this always be done *parsimoniously*, i.e., without increasing the number of parameters needed to specify the transition model?

15.2 In this exercise, we examine what happens to the probabilities in the umbrella world in the limit of long time sequences.

- a. Suppose we observe an unending sequence of days on which the umbrella appears. Show that, as the days go by, the probability of rain on the current day increases monotonically toward a fixed point. Calculate this fixed point.
- b. Now consider *forecasting* further and further into the future, given just the first two umbrella observations. First, compute the probability $P(r_{2+k}|u_1, u_2)$ for $k = 1 \dots 20$ and plot the results. You should see that the probability converges towards a fixed point. Prove that the exact value of this fixed point is 0.5.

15.3 This exercise develops a space-efficient variant of the forward-backward algorithm described in Figure 15.4 (page 576). We wish to compute $\mathbf{P}(\mathbf{X}_k|\mathbf{e}_{1:t})$ for $k = 1, \dots, t$. This will be done with a divide-and-conquer approach.

- a. Suppose, for simplicity, that t is odd, and let the halfway point be $h = (t + 1)/2$. Show that $\mathbf{P}(\mathbf{X}_k|\mathbf{e}_{1:t})$ can be computed for $k = 1, \dots, h$ given just the initial forward message $\mathbf{f}_{1:0}$, the backward message $\mathbf{b}_{h+1:t}$, and the evidence $\mathbf{e}_{1:h}$.
- b. Show a similar result for the second half of the sequence.

- c. Given the results of (a) and (b), a recursive divide-and-conquer algorithm can be constructed by first running forward along the sequence and then backward from the end, storing just the required messages at the middle and the ends. Then the algorithm is called on each half. Write out the algorithm in detail.
- d. Compute the time and space complexity of the algorithm as a function of t , the length of the sequence. How does this change if we divide the input into more than two pieces?

15.4 On page 577, we outlined a flawed procedure for finding the most likely state sequence, given an observation sequence. The procedure involves finding the most likely state at each time step, using smoothing, and returning the sequence composed of these states. Show that, for some temporal probability models and observation sequences, this procedure returns an impossible state sequence (i.e., the posterior probability of the sequence is zero).

15.5 Equation (15.12) describes the filtering process for the matrix formulation of HMMs. Give a similar equation for the calculation of likelihoods, which was described generically in Equation (15.7).

15.6 Consider the vacuum worlds of Figure 4.18 (perfect sensing) and Figure 15.7 (noisy sensing). Suppose that the robot receives an observation sequence such that, with perfect sensing, there is exactly one possible location it could be in. Is this location necessarily the most probable location under noisy sensing for sufficiently small noise probability ϵ ? Prove your claim or find a counterexample.



15.7 In Section 15.3.2, the prior distribution over locations is uniform and the transition model assumes an equal probability of moving to any neighboring square. What if those assumptions are wrong? Suppose that the initial location is actually chosen uniformly from the northwest quadrant of the room and the *Move* action actually tends to move southeast. Keeping the HMM model fixed, explore the effect on localization and path accuracy as the southeasterly tendency increases, for different values of ϵ .

15.8 Consider a version of the vacuum robot (page 582) that has the policy of going straight for as long as it can; only when it encounters an obstacle does it change to a new (randomly selected) heading. To model this robot, each state in the model consists of a (*location, heading*) pair. Implement this model and see how well the Viterbi algorithm can track a robot with this model. The robot's policy is more constrained than the random-walk robot; does that mean that predictions of the most likely path are more accurate?

15.9 This exercise is concerned with filtering in an environment with no landmarks. Consider a vacuum robot in an empty room, represented by an $n \times m$ rectangular grid. The robot's location is hidden; the only evidence available to the observer is a noisy location sensor that gives an approximation to the robot's location. If the robot is at location (x, y) then with probability .1 the sensor gives the correct location, with probability .05 each it reports one of the 8 locations immediately surrounding (x, y) , with probability .025 each it reports one of the 16 locations that surround those 8, and with the remaining probability of .1 it reports "no reading." The robot's policy is to pick a direction and follow it with probability .8 on each step; the robot switches to a randomly selected new heading with probability .2 (or with

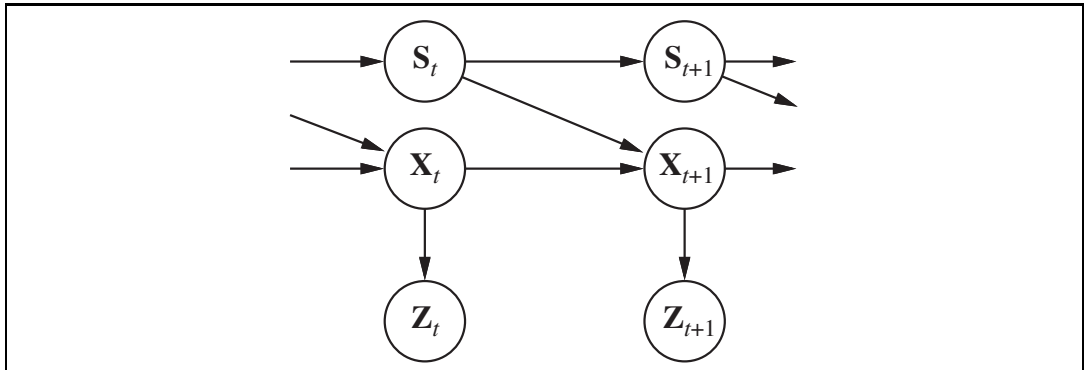


Figure 15.21 A Bayesian network representation of a switching Kalman filter. The switching variable S_t is a discrete state variable whose value determines the transition model for the continuous state variables \mathbf{X}_t . For any discrete state i , the transition model $\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{X}_t, S_t = i)$ is a linear Gaussian model, just as in a regular Kalman filter. The transition model for the discrete state, $\mathbf{P}(S_{t+1}|S_t)$, can be thought of as a matrix, as in a hidden Markov model.

probability 1 if it encounters a wall). Implement this as an HMM and do filtering to track the robot. How accurately can we track the robot's path?

15.10 Often, we wish to monitor a continuous-state system whose behavior switches unpredictably among a set of k distinct “modes.” For example, an aircraft trying to evade a missile can execute a series of distinct maneuvers that the missile may attempt to track. A Bayesian network representation of such a **switching Kalman filter** model is shown in Figure 15.21.

- Suppose that the discrete state S_t has k possible values and that the prior continuous state estimate $\mathbf{P}(\mathbf{X}_0)$ is a multivariate Gaussian distribution. Show that the prediction $\mathbf{P}(\mathbf{X}_1)$ is a **mixture of Gaussians**—that is, a weighted sum of Gaussians such that the weights sum to 1.
- Show that if the current continuous state estimate $\mathbf{P}(\mathbf{X}_t|\mathbf{e}_{1:t})$ is a mixture of m Gaussians, then in the general case the updated state estimate $\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t+1})$ will be a mixture of km Gaussians.
- What aspect of the temporal process do the weights in the Gaussian mixture represent?

The results in (a) and (b) show that the representation of the posterior grows without limit even for switching Kalman filters, which are among the simplest hybrid dynamic models.

15.11 Complete the missing step in the derivation of Equation (15.19) on page 586, the first update step for the one-dimensional Kalman filter.

15.12 Let us examine the behavior of the variance update in Equation (15.20) (page 587).

- Plot the value of σ_t^2 as a function of t , given various values for σ_x^2 and σ_z^2 .
- Show that the update has a fixed point σ^2 such that $\sigma_t^2 \rightarrow \sigma^2$ as $t \rightarrow \infty$, and calculate the value of σ^2 .
- Give a qualitative explanation for what happens as $\sigma_x^2 \rightarrow 0$ and as $\sigma_z^2 \rightarrow 0$.

15.13 A professor wants to know if students are getting enough sleep. Each day, the professor observes whether the students sleep in class, and whether they have red eyes. The professor has the following domain theory:

- The prior probability of getting enough sleep, with no observations, is 0.7.
- The probability of getting enough sleep on night t is 0.8 given that the student got enough sleep the previous night, and 0.3 if not.
- The probability of having red eyes is 0.2 if the student got enough sleep, and 0.7 if not.
- The probability of sleeping in class is 0.1 if the student got enough sleep, and 0.3 if not.

Formulate this information as a dynamic Bayesian network that the professor could use to filter or predict from a sequence of observations. Then reformulate it as a hidden Markov model that has only a single observation variable. Give the complete probability tables for the model.

15.14 For the DBN specified in Exercise 15.13 and for the evidence values

- \mathbf{e}_1 = not red eyes, not sleeping in class
- \mathbf{e}_2 = red eyes, not sleeping in class
- \mathbf{e}_3 = red eyes, sleeping in class

perform the following computations:

- a. State estimation: Compute $P(\text{EnoughSleep}_t | \mathbf{e}_{1:t})$ for each of $t = 1, 2, 3$.
- b. Smoothing: Compute $P(\text{EnoughSleep}_t | \mathbf{e}_{1:3})$ for each of $t = 1, 2, 3$.
- c. Compare the filtered and smoothed probabilities for $t = 1$ and $t = 2$.

15.15 Suppose that a particular student shows up with red eyes and sleeps in class every day. Given the model described in Exercise 15.13, explain why the probability that the student had enough sleep the previous night converges to a fixed point rather than continuing to go down as we gather more days of evidence. What is the fixed point? Answer this both numerically (by computation) and analytically.

15.16 This exercise analyzes in more detail the persistent-failure model for the battery sensor in Figure 15.15(a) (page 594).

- a. Figure 15.15(b) stops at $t = 32$. Describe qualitatively what should happen as $t \rightarrow \infty$ if the sensor continues to read 0.
- b. Suppose that the external temperature affects the battery sensor in such a way that transient failures become more likely as temperature increases. Show how to augment the DBN structure in Figure 15.15(a), and explain any required changes to the CPTs.
- c. Given the new network structure, can battery readings be used by the robot to infer the current temperature?

15.17 Consider applying the variable elimination algorithm to the umbrella DBN unrolled for three slices, where the query is $\mathbf{P}(R_3 | u_1, u_2, u_3)$. Show that the space complexity of the algorithm—the size of the largest factor—is the same, regardless of whether the rain variables are eliminated in forward or backward order.

16 MAKING SIMPLE DECISIONS

In which we see how an agent should make decisions so that it gets what it wants—on average, at least.

In this chapter, we fill in the details of how utility theory combines with probability theory to yield a decision-theoretic agent—an agent that can make rational decisions based on what it believes and what it wants. Such an agent can make decisions in contexts in which uncertainty and conflicting goals leave a logical agent with no way to decide: a goal-based agent has a binary distinction between good (goal) and bad (non-goal) states, while a decision-theoretic agent has a continuous measure of outcome quality.

Section 16.1 introduces the basic principle of decision theory: the maximization of expected utility. Section 16.2 shows that the behavior of any rational agent can be captured by supposing a utility function that is being maximized. Section 16.3 discusses the nature of utility functions in more detail, and in particular their relation to individual quantities such as money. Section 16.4 shows how to handle utility functions that depend on several quantities. In Section 16.5, we describe the implementation of decision-making systems. In particular, we introduce a formalism called a **decision network** (also known as an **influence diagram**) that extends Bayesian networks by incorporating actions and utilities. The remainder of the chapter discusses issues that arise in applications of decision theory to expert systems.

16.1 COMBINING BELIEFS AND DESIRES UNDER UNCERTAINTY

Decision theory, in its simplest form, deals with choosing among actions based on the desirability of their *immediate* outcomes; that is, the environment is assumed to be episodic in the sense defined on page 43. (This assumption is relaxed in Chapter 17.) In Chapter 3 we used the notation $\text{RESULT}(s_0, a)$ for the state that is the deterministic outcome of taking action a in state s_0 . In this chapter we deal with nondeterministic partially observable environments. Since the agent may not know the current state, we omit it and define $\text{RESULT}(a)$ as a *random variable* whose values are the possible outcome states. The probability of outcome s' , given evidence observations \mathbf{e} , is written

$$P(\text{RESULT}(a) = s' \mid a, \mathbf{e}),$$

where the a on the right-hand side of the conditioning bar stands for the event that action a is executed.¹

UTILITY FUNCTION

EXPECTED UTILITY

The agent's preferences are captured by a **utility function**, $U(s)$, which assigns a single number to express the desirability of a state. The **expected utility** of an action given the evidence, $EU(a|\mathbf{e})$, is just the average utility value of the outcomes, weighted by the probability that the outcome occurs:

$$EU(a|\mathbf{e}) = \sum_{s'} P(\text{RESULT}(a) = s' | a, \mathbf{e}) U(s'). \quad (16.1)$$

MAXIMUM EXPECTED UTILITY

The principle of **maximum expected utility** (MEU) says that a rational agent should choose the action that maximizes the agent's expected utility:

$$\text{action} = \operatorname{argmax}_a EU(a|\mathbf{e})$$

In a sense, the MEU principle could be seen as defining all of AI. All an intelligent agent has to do is calculate the various quantities, maximize utility over its actions, and away it goes. But this does not mean that the AI problem is *solved* by the definition!

The MEU principle *formalizes* the general notion that the agent should “do the right thing,” but goes only a small distance toward a full *operationalization* of that advice. Estimating the state of the world requires perception, learning, knowledge representation, and inference. Computing $P(\text{RESULT}(a) | a, \mathbf{e})$ requires a complete causal model of the world and, as we saw in Chapter 14, NP-hard inference in (very large) Bayesian networks. Computing the outcome utilities $U(s')$ often requires searching or planning, because an agent may not know how good a state is until it knows where it can get to from that state. So, decision theory is not a panacea that solves the AI problem—but it does provide a useful framework.

The MEU principle has a clear relation to the idea of performance measures introduced in Chapter 2. The basic idea is simple. Consider the environments that could lead to an agent having a given percept history, and consider the different agents that we could design. *If an agent acts so as to maximize a utility function that correctly reflects the performance measure, then the agent will achieve the highest possible performance score (averaged over all the possible environments).* This is the central justification for the MEU principle itself. While the claim may seem tautological, it does in fact embody a very important transition from a global, external criterion of rationality—the performance measure over environment histories—to a local, internal criterion involving the maximization of a utility function applied to the next state.



16.2 THE BASIS OF UTILITY THEORY

Intuitively, the principle of Maximum Expected Utility (MEU) seems like a reasonable way to make decisions, but it is by no means obvious that it is the *only* rational way. After all, why should maximizing the *average* utility be so special? What's wrong with an agent that

¹ Classical decision theory leaves the current state S_0 implicit, but we could make it explicit by writing $P(\text{RESULT}(a) = s' | a, \mathbf{e}) = \sum_s P(\text{RESULT}(s, a) = s' | a) P(S_0 = s | \mathbf{e})$.

maximizes the weighted sum of the cubes of the possible utilities, or tries to minimize the worst possible loss? Could an agent act rationally just by expressing preferences between states, without giving them numeric values? Finally, why should a utility function with the required properties exist at all? We shall see.

16.2.1 Constraints on rational preferences

These questions can be answered by writing down some constraints on the preferences that a rational agent should have and then showing that the MEU principle can be derived from the constraints. We use the following notation to describe an agent's preferences:

$A \succ B$ the agent prefers A over B .

$A \sim B$ the agent is indifferent between A and B .

$A \succsim B$ the agent prefers A over B or is indifferent between them.

Now the obvious question is, what sorts of things are A and B ? They could be states of the world, but more often than not there is uncertainty about what is really being offered. For example, an airline passenger who is offered “the pasta dish or the chicken” does not know what lurks beneath the tinfoil cover.² The pasta could be delicious or congealed, the chicken juicy or overcooked beyond recognition. We can think of the set of outcomes for each action as a **lottery**—think of each action as a ticket. A lottery L with possible outcomes S_1, \dots, S_n that occur with probabilities p_1, \dots, p_n is written

$$L = [p_1, S_1; p_2, S_2; \dots p_n, S_n].$$

In general, each outcome S_i of a lottery can be either an atomic state or another lottery. The primary issue for utility theory is to understand how preferences between complex lotteries are related to preferences between the underlying states in those lotteries. To address this issue we list six constraints that we require any reasonable preference relation to obey:

LOTTERY

ORDERABILITY

- **Orderability:** Given any two lotteries, a rational agent must either prefer one to the other or else rate the two as equally preferable. That is, the agent cannot avoid deciding. As we said on page 490, refusing to bet is like refusing to allow time to pass.

Exactly one of $(A \succ B)$, $(B \succ A)$, or $(A \sim B)$ holds.

TRANSITIVITY

- **Transitivity:** Given any three lotteries, if an agent prefers A to B and prefers B to C , then the agent must prefer A to C .

$$(A \succ B) \wedge (B \succ C) \Rightarrow (A \succ C).$$

CONTINUITY

- **Continuity:** If some lottery B is between A and C in preference, then there is some probability p for which the rational agent will be indifferent between getting B for sure and the lottery that yields A with probability p and C with probability $1 - p$.

$$A \succ B \succ C \Rightarrow \exists p [p, A; 1 - p, C] \sim B.$$

SUBSTITUTABILITY

- **Substitutability:** If an agent is indifferent between two lotteries A and B , then the agent is indifferent between two more complex lotteries that are the same except that B

² We apologize to readers whose local airlines no longer offer food on long flights.

is substituted for A in one of them. This holds regardless of the probabilities and the other outcome(s) in the lotteries.

$$A \sim B \Rightarrow [p, A; 1 - p, C] \sim [p, B; 1 - p, C] .$$

This also holds if we substitute \succ for \sim in this axiom.

MONOTONICITY

- **Monotonicity:** Suppose two lotteries have the same two possible outcomes, A and B . If an agent prefers A to B , then the agent must prefer the lottery that has a higher probability for A (and vice versa).

$$A \succ B \Rightarrow (p > q \Leftrightarrow [p, A; 1 - p, B] \succ [q, A; 1 - q, B]) .$$

DECOMPOSABILITY

- **Decomposability:** Compound lotteries can be reduced to simpler ones using the laws of probability. This has been called the “no fun in gambling” rule because it says that two consecutive lotteries can be compressed into a single equivalent lottery, as shown in Figure 16.1(b).³

$$[p, A; 1 - p, [q, B; 1 - q, C]] \sim [p, A; (1 - p)q, B; (1 - p)(1 - q), C] .$$

These constraints are known as the axioms of utility theory. Each axiom can be motivated by showing that an agent that violates it will exhibit patently irrational behavior in some situations. For example, we can motivate transitivity by making an agent with nontransitive preferences give us all its money. Suppose that the agent has the nontransitive preferences $A \succ B \succ C \succ A$, where A , B , and C are goods that can be freely exchanged. If the agent currently has A , then we could offer to trade C for A plus one cent. The agent prefers C , and so would be willing to make this trade. We could then offer to trade B for C , extracting another cent, and finally trade A for B . This brings us back where we started from, except that the agent has given us three cents (Figure 16.1(a)). We can keep going around the cycle until the agent has no money at all. Clearly, the agent has acted irrationally in this case.

16.2.2 Preferences lead to utility

Notice that the axioms of utility theory are really axioms about preferences—they say nothing about a utility function. But in fact from the axioms of utility we can derive the following consequences (for the proof, see von Neumann and Morgenstern, 1944):

- **Existence of Utility Function:** If an agent’s preferences obey the axioms of utility, then there exists a function U such that $U(A) > U(B)$ if and only if A is preferred to B , and $U(A) = U(B)$ if and only if the agent is indifferent between A and B .

$$U(A) > U(B) \Leftrightarrow A \succ B$$

$$U(A) = U(B) \Leftrightarrow A \sim B$$

- **Expected Utility of a Lottery:** The utility of a lottery is the sum of the probability of each outcome times the utility of that outcome.

$$U([p_1, S_1; \dots; p_n, S_n]) = \sum_i p_i U(S_i) .$$

³ We can account for the enjoyment of gambling by encoding gambling events into the state description; for example, “Have \$10 and gambled” could be preferred to “Have \$10 and didn’t gamble.”

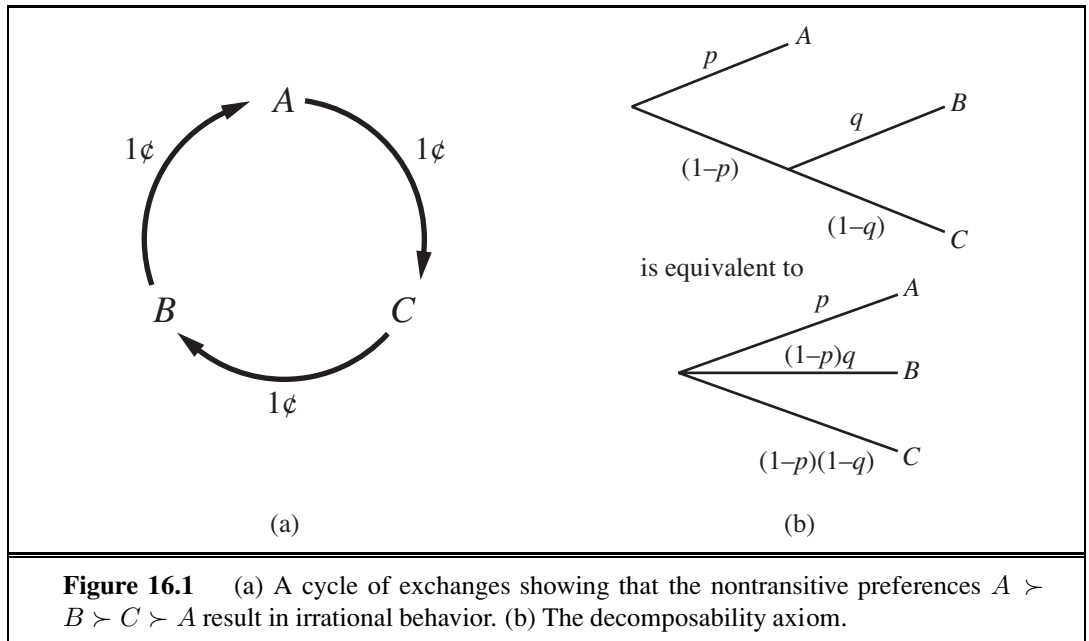


Figure 16.1 (a) A cycle of exchanges showing that the nontransitive preferences $A \succ B \succ C \succ A$ result in irrational behavior. (b) The decomposability axiom.

In other words, once the probabilities and utilities of the possible outcome states are specified, the utility of a compound lottery involving those states is completely determined. Because the outcome of a nondeterministic action is a lottery, it follows that an agent can act rationally—that is, consistently with its preferences—only by choosing an action that maximizes expected utility according to Equation (16.1).

The preceding theorems establish that a utility function *exists* for any rational agent, but they do not establish that it is *unique*. It is easy to see, in fact, that an agent's behavior would not change if its utility function $U(S)$ were transformed according to

$$U'(S) = aU(S) + b, \quad (16.2)$$

where a and b are constants and $a > 0$; an affine transformation.⁴ This fact was noted in Chapter 5 for two-player games of chance; here, we see that it is completely general.

As in game-playing, in a deterministic environment an agent just needs a preference ranking on states—the numbers don't matter. This is called a **value function** or **ordinal utility function**.

It is important to remember that the existence of a utility function that describes an agent's preference behavior does not necessarily mean that the agent is *explicitly* maximizing that utility function in its own deliberations. As we showed in Chapter 2, rational behavior can be generated in any number of ways. By observing a rational agent's preferences, however, an observer can construct the utility function that represents what the agent is actually trying to achieve (even if the agent doesn't know it).

⁴ In this sense, utilities resemble temperatures: a temperature in Fahrenheit is 1.8 times the Celsius temperature plus 32. You get the same results in either measurement system.

16.3 UTILITY FUNCTIONS

Utility is a function that maps from lotteries to real numbers. We know there are some axioms on utilities that all rational agents must obey. Is that all we can say about utility functions? Strictly speaking, that is it: an agent can have any preferences it likes. For example, an agent might prefer to have a prime number of dollars in its bank account; in which case, if it had \$16 it would give away \$3. This might be unusual, but we can't call it irrational. An agent might prefer a dented 1973 Ford Pinto to a shiny new Mercedes. Preferences can also interact: for example, the agent might prefer prime numbers of dollars only when it owns the Pinto, but when it owns the Mercedes, it might prefer more dollars to fewer. Fortunately, the preferences of real agents are usually more systematic, and thus easier to deal with.

16.3.1 Utility assessment and utility scales

If we want to build a decision-theoretic system that helps the agent make decisions or acts on his or her behalf, we must first work out what the agent's utility function is. This process, often called **preference elicitation**, involves presenting choices to the agent and using the observed preferences to pin down the underlying utility function.

PREFERENCE
ELICITATION

Equation (16.2) says that there is no absolute scale for utilities, but it is helpful, nonetheless, to establish *some* scale on which utilities can be recorded and compared for any particular problem. A scale can be established by fixing the utilities of any two particular outcomes, just as we fix a temperature scale by fixing the freezing point and boiling point of water. Typically, we fix the utility of a “best possible prize” at $U(S) = u_{\top}$ and a “worst possible catastrophe” at $U(S) = u_{\perp}$. **Normalized utilities** use a scale with $u_{\perp} = 0$ and $u_{\top} = 1$.

NORMALIZED
UTILITIES

Given a utility scale between u_{\top} and u_{\perp} , we can assess the utility of any particular prize S by asking the agent to choose between S and a **standard lottery** $[p, u_{\top}; (1-p), u_{\perp}]$. The probability p is adjusted until the agent is indifferent between S and the standard lottery. Assuming normalized utilities, the utility of S is given by p . Once this is done for each prize, the utilities for all lotteries involving those prizes are determined.

STANDARD LOTTERY



In medical, transportation, and environmental decision problems, among others, people's lives are at stake. In such cases, u_{\perp} is the value assigned to immediate death (or perhaps many deaths). *Although nobody feels comfortable with putting a value on human life, it is a fact that tradeoffs are made all the time.* Aircraft are given a complete overhaul at intervals determined by trips and miles flown, rather than after every trip. Cars are manufactured in a way that trades off costs against accident survival rates. Paradoxically, a refusal to “put a monetary value on life” means that life is often *undervalued*. Ross Shachter relates an experience with a government agency that commissioned a study on removing asbestos from schools. The decision analysts performing the study assumed a particular dollar value for the life of a school-age child, and argued that the rational choice under that assumption was to remove the asbestos. The agency, morally outraged at the idea of setting the value of a life, rejected the report out of hand. It then decided against asbestos removal—implicitly asserting a lower value for the life of a child than that assigned by the analysts.

MICROMORT

Some attempts have been made to find out the value that people place on their own lives. One common “currency” used in medical and safety analysis is the **micromort**, a one in a million chance of death. If you ask people how much they would pay to avoid a risk—for example, to avoid playing Russian roulette with a million-barreled revolver—they will respond with very large numbers, perhaps tens of thousands of dollars, but their actual behavior reflects a much lower monetary value for a micromort. For example, driving in a car for 230 miles incurs a risk of one micromort; over the life of your car—say, 92,000 miles—that’s 400 micromorts. People appear to be willing to pay about \$10,000 (at 2009 prices) more for a safer car that halves the risk of death, or about \$50 per micromort. A number of studies have confirmed a figure in this range across many individuals and risk types. Of course, this argument holds only for small risks. Most people won’t agree to kill themselves for \$50 million.

QALY

Another measure is the **QALY**, or quality-adjusted life year. Patients with a disability are willing to accept a shorter life expectancy to be restored to full health. For example, kidney patients on average are indifferent between living two years on a dialysis machine and one year at full health.

16.3.2 The utility of money

Utility theory has its roots in economics, and economics provides one obvious candidate for a utility measure: money (or more specifically, an agent’s total net assets). The almost universal exchangeability of money for all kinds of goods and services suggests that money plays a significant role in human utility functions.

MONOTONIC
PREFERENCE

It will usually be the case that an agent prefers more money to less, all other things being equal. We say that the agent exhibits a **monotonic preference** for more money. This does not mean that money behaves as a utility function, because it says nothing about preferences between *lotteries* involving money.

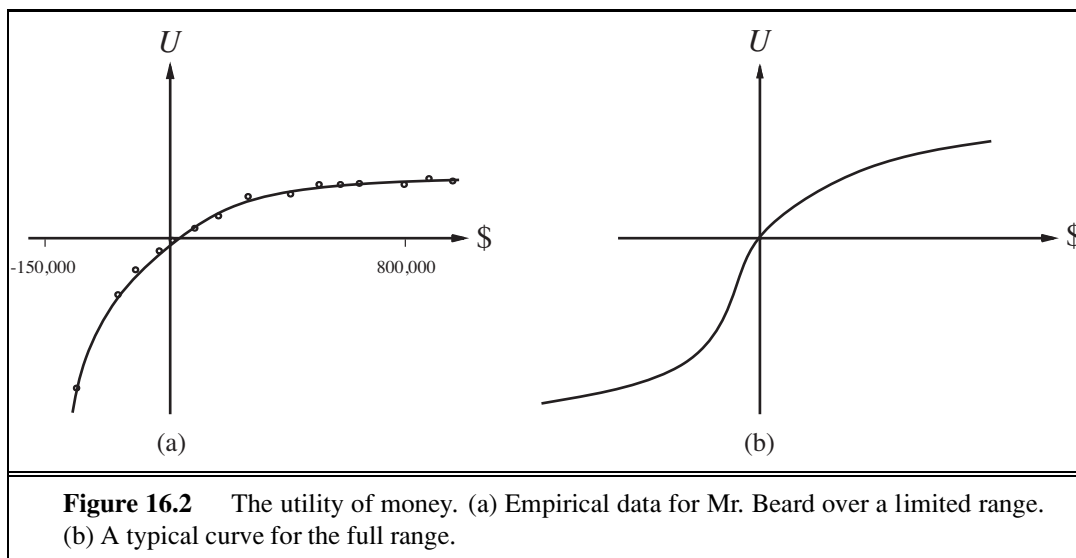
Suppose you have triumphed over the other competitors in a television game show. The host now offers you a choice: either you can take the \$1,000,000 prize or you can gamble it on the flip of a coin. If the coin comes up heads, you end up with nothing, but if it comes up tails, you get \$2,500,000. If you’re like most people, you would decline the gamble and pocket the million. Are you being irrational?

EXPECTED
MONETARY VALUE

Assuming the coin is fair, the **expected monetary value** (EMV) of the gamble is $\frac{1}{2}(\$0) + \frac{1}{2}(\$2,500,000) = \$1,250,000$, which is more than the original \$1,000,000. But that does not necessarily mean that accepting the gamble is a better decision. Suppose we use S_n to denote the state of possessing total wealth $\$n$, and that your current wealth is $\$k$. Then the expected utilities of the two actions of accepting and declining the gamble are

$$\begin{aligned} EU(\text{Accept}) &= \frac{1}{2}U(S_k) + \frac{1}{2}U(S_{k+2,500,000}) , \\ EU(\text{Decline}) &= U(S_{k+1,000,000}) . \end{aligned}$$

To determine what to do, we need to assign utilities to the outcome states. Utility is not directly proportional to monetary value, because the utility for your first million is very high (or so they say), whereas the utility for an additional million is smaller. Suppose you assign a utility of 5 to your current financial status (S_k), a 9 to the state $S_{k+2,500,000}$, and an 8 to the



state $S_{k+1,000,000}$. Then the rational action would be to decline, because the expected utility of accepting is only 7 (less than the 8 for declining). On the other hand, a billionaire would most likely have a utility function that is locally linear over the range of a few million more, and thus would accept the gamble.

In a pioneering study of actual utility functions, Grayson (1960) found that the utility of money was almost exactly proportional to the *logarithm* of the amount. (This idea was first suggested by Bernoulli (1738); see Exercise 16.3.) One particular utility curve, for a certain Mr. Beard, is shown in Figure 16.2(a). The data obtained for Mr. Beard's preferences are consistent with a utility function

$$U(S_{k+n}) = -263.31 + 22.09 \log(n + 150,000)$$

for the range between $n = -\$150,000$ and $n = \$800,000$.

We should not assume that this is the definitive utility function for monetary value, but it is likely that most people have a utility function that is concave for positive wealth. Going into debt is bad, but preferences between different levels of debt can display a reversal of the concavity associated with positive wealth. For example, someone already \$10,000,000 in debt might well accept a gamble on a fair coin with a gain of \$10,000,000 for heads and a loss of \$20,000,000 for tails.⁵ This yields the S-shaped curve shown in Figure 16.2(b).

If we restrict our attention to the positive part of the curves, where the slope is decreasing, then for any lottery L , the utility of being faced with that lottery is less than the utility of being handed the expected monetary value of the lottery as a sure thing:

$$U(L) < U(S_{EMV(L)}).$$

RISK-AVERSE

That is, agents with curves of this shape are **risk-averse**: they prefer a sure thing with a payoff that is less than the expected monetary value of a gamble. On the other hand, in the “desperate” region at large negative wealth in Figure 16.2(b), the behavior is **risk-seeking**.

RISK-SEEKING

⁵ Such behavior might be called desperate, but it is rational if one is already in a desperate situation.

CERTAINTY
EQUIVALENT

The value an agent will accept in lieu of a lottery is called the **certainty equivalent** of the lottery. Studies have shown that most people will accept about \$400 in lieu of a gamble that gives \$1000 half the time and \$0 the other half—that is, the certainty equivalent of the lottery is \$400, while the EMV is \$500. The difference between the EMV of a lottery and its certainty equivalent is called the **insurance premium**. Risk aversion is the basis for the insurance industry, because it means that insurance premiums are positive. People would rather pay a small insurance premium than gamble the price of their house against the chance of a fire. From the insurance company’s point of view, the price of the house is very small compared with the firm’s total reserves. This means that the insurer’s utility curve is approximately linear over such a small region, and the gamble costs the company almost nothing.

INSURANCE
PREMIUM

RISK-NEUTRAL

Notice that for *small* changes in wealth relative to the current wealth, almost any curve will be approximately linear. An agent that has a linear curve is said to be **risk-neutral**. For gambles with small sums, therefore, we expect risk neutrality. In a sense, this justifies the simplified procedure that proposed small gambles to assess probabilities and to justify the axioms of probability in Section 13.2.3.

16.3.3 Expected utility and post-decision disappointment

The rational way to choose the best action, a^* , is to maximize expected utility:

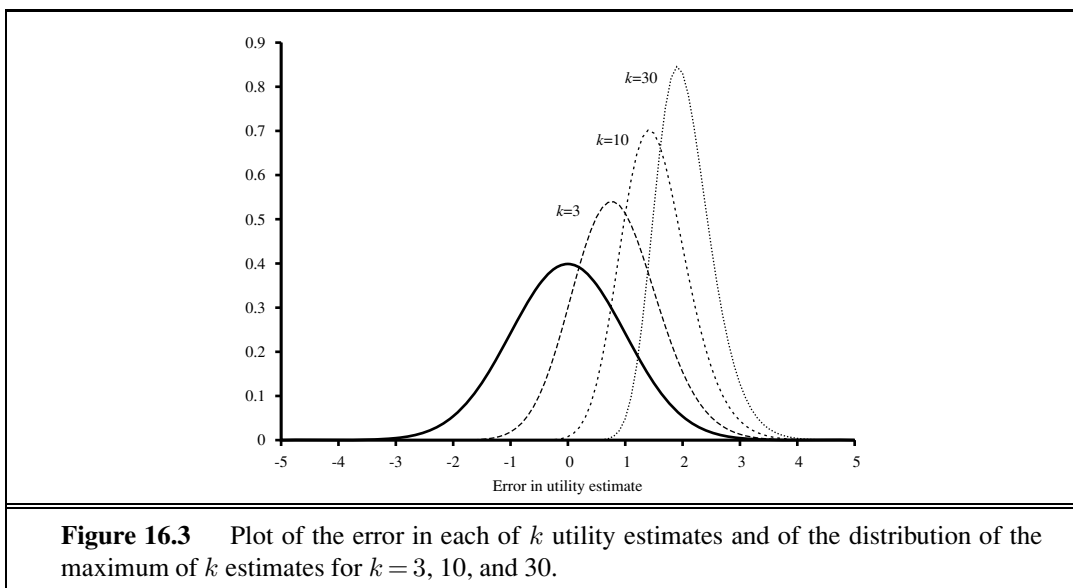
$$a^* = \operatorname{argmax}_a EU(a|\mathbf{e}) .$$

If we have calculated the expected utility correctly according to our probability model, and if the probability model correctly reflects the underlying stochastic processes that generate the outcomes, then, on average, we will get the utility we expect if the whole process is repeated many times.

In reality, however, our model usually oversimplifies the real situation, either because we don’t know enough (e.g., when making a complex investment decision) or because the computation of the true expected utility is too difficult (e.g., when estimating the utility of successor states of the root node in backgammon). In that case, we are really working with *estimates* $\widehat{EU}(a|\mathbf{e})$ of the true expected utility. We will assume, kindly perhaps, that the estimates are **unbiased**, that is, the expected value of the error, $E(\widehat{EU}(a|\mathbf{e}) - EU(a|\mathbf{e}))$, is zero. In that case, it still seems reasonable to choose the action with the highest estimated utility and to expect to receive that utility, on average, when the action is executed.

UNBIASED

Unfortunately, the real outcome will usually be significantly *worse* than we estimated, even though the estimate was unbiased! To see why, consider a decision problem in which there are k choices, each of which has true estimated utility of 0. Suppose that the error in each utility estimate has zero mean and standard deviation of 1, shown as the bold curve in Figure 16.3. Now, as we actually start to generate the estimates, some of the errors will be negative (pessimistic) and some will be positive (optimistic). Because we select the action with the *highest* utility estimate, we are obviously favoring the overly optimistic estimates, and that is the source of the bias. It is a straightforward matter to calculate the distribution of the maximum of the k estimates (see Exercise 16.11) and hence quantify the extent of our disappointment. The curve in Figure 16.3 for $k=3$ has a mean around 0.85, so the average disappointment will be about 85% of the standard deviation in the utility estimates.



With more choices, extremely optimistic estimates are more likely to arise: for $k = 30$, the disappointment will be around twice the standard deviation in the estimates.

OPTIMIZER'S CURSE

This tendency for the estimated expected utility of the best choice to be too high is called the **optimizer's curse** (Smith and Winkler, 2006). It afflicts even the most seasoned decision analysts and statisticians. Serious manifestations include believing that an exciting new drug that has cured 80% patients in a trial will cure 80% of patients (it's been chosen from $k =$ thousands of candidate drugs) or that a mutual fund advertised as having above-average returns will continue to have them (it's been chosen to appear in the advertisement out of $k =$ dozens of funds in the company's overall portfolio). It can even be the case that what appears to be the best choice may not be, if the variance in the utility estimate is high: a drug, selected from thousands tried, that has cured 9 of 10 patients is probably *worse* than one that has cured 800 of 1000.

The optimizer's curse crops up everywhere because of the ubiquity of utility-maximizing selection processes, so taking the utility estimates at face value is a bad idea. We can avoid the curse by using an explicit probability model $\mathbf{P}(\widehat{EU} \mid EU)$ of the error in the utility estimates. Given this model and a prior $\mathbf{P}(EU)$ on what we might reasonably expect the utilities to be, we treat the utility estimate, once obtained, as evidence and compute the posterior distribution for the true utility using Bayes' rule.

16.3.4 Human judgment and irrationality

NORMATIVE THEORY
DESCRIPTIVE
THEORY

Decision theory is a **normative theory**: it describes how a rational agent *should* act. A **descriptive theory**, on the other hand, describes how actual agents—for example, humans—really do act. The application of economic theory would be greatly enhanced if the two coincided, but there appears to be some experimental evidence to the contrary. The evidence suggests that humans are “predictably irrational” (Ariely, 2009).

The best-known problem is the Allais paradox (Allais, 1953). People are given a choice between lotteries A and B and then between C and D , which have the following prizes:

A : 80% chance of \$4000	C : 20% chance of \$4000
B : 100% chance of \$3000	D : 25% chance of \$3000

CERTAINTY EFFECT

REGRET

Most people consistently prefer B over A (taking the sure thing), and C over D (taking the higher EMV). The normative analysis disagrees! We can see this most easily if we use the freedom implied by Equation (16.2) to set $U(\$0) = 0$. In that case, then $B \succ A$ implies that $U(\$3000) > 0.8U(\$4000)$, whereas $C \succ D$ implies exactly the reverse. In other words, there is no utility function that is consistent with these choices. One explanation for the apparently irrational preferences is the **certainty effect** (Kahneman and Tversky, 1979): people are strongly attracted to gains that are certain. There are several reasons why this may be so. First, people may prefer to reduce their computational burden; by choosing certain outcomes, they don't have to compute with probabilities. But the effect persists even when the computations involved are very easy ones. Second, people may distrust the legitimacy of the stated probabilities. I trust that a coin flip is roughly 50/50 if I have control over the coin and the flip, but I may distrust the result if the flip is done by someone with a vested interest in the outcome.⁶ In the presence of distrust, it might be better to go for the sure thing.⁷ Third, people may be accounting for their emotional state as well as their financial state. People know they would experience **regret** if they gave up a certain reward (B) for an 80% chance at a higher reward and then lost. In other words, if A is chosen, there is a 20% chance of getting no money *and feeling like a complete idiot*, which is worse than just getting no money. So perhaps people who choose B over A and C over D are not being irrational; they are just saying that they are willing to give up \$200 of EMV to avoid a 20% chance of feeling like an idiot.

A related problem is the Ellsberg paradox. Here the prizes are fixed, but the probabilities are underconstrained. Your payoff will depend on the color of a ball chosen from an urn. You are told that the urn contains 1/3 red balls, and 2/3 either black or yellow balls, but you don't know how many black and how many yellow. Again, you are asked whether you prefer lottery A or B ; and then C or D :

A : \$100 for a red ball	C : \$100 for a red or yellow ball
B : \$100 for a black ball	D : \$100 for a black or yellow ball .

AMBIGUITY
AVERSION

It should be clear that if you think there are more red than black balls then you should prefer A over B and C over D ; if you think there are fewer red than black you should prefer the opposite. But it turns out that most people prefer A over B and also prefer D over C , even though there is no state of the world for which this is rational. It seems that people have **ambiguity aversion**: A gives you a 1/3 chance of winning, while B could be anywhere between 0 and 2/3. Similarly, D gives you a 2/3 chance, while C could be anywhere between 1/3 and 3/3. Most people elect the known probability rather than the unknown unknowns.

⁶ For example, the mathematician/magician Persi Diaconis can make a coin flip come out the way he wants every time (Landhuis, 2004).

⁷ Even the sure thing may not be certain. Despite cast-iron promises, we have not yet received that \$27,000,000 from the Nigerian bank account of a previously unknown deceased relative.

FRAMING EFFECT

Yet another problem is that the exact wording of a decision problem can have a big impact on the agent's choices; this is called the **framing effect**. Experiments show that people like a medical procedure that it is described as having a "90% survival rate" about twice as much as one described as having a "10% death rate," even though these two statements mean exactly the same thing. This discrepancy in judgment has been found in multiple experiments and is about the same whether the subjects were patients in a clinic, statistically sophisticated business school students, or experienced doctors.

ANCHORING EFFECT

People feel more comfortable making *relative* utility judgments rather than absolute ones. I may have little idea how much I might enjoy the various wines offered by a restaurant. The restaurant takes advantage of this by offering a \$200 bottle that it knows nobody will buy, but which serves to skew upward the customer's estimate of the value of all wines and make the \$55 bottle seem like a bargain. This is called the **anchoring effect**.

If human informants insist on contradictory preference judgments, there is nothing that automated agents can do to be consistent with them. Fortunately, preference judgments made by humans are often open to revision in the light of further consideration. Paradoxes like the Allais paradox are greatly reduced (but not eliminated) if the choices are explained better. In work at the Harvard Business School on assessing the utility of money, Keeney and Raiffa (1976, p. 210) found the following:

Subjects tend to be too risk-averse in the small and therefore . . . the fitted utility functions exhibit unacceptably large risk premiums for lotteries with a large spread. . . . Most of the subjects, however, can reconcile their inconsistencies and feel that they have learned an important lesson about how they want to behave. As a consequence, some subjects cancel their automobile collision insurance and take out more term insurance on their lives.

EVOLUTIONARY
PSYCHOLOGY

The evidence for human irrationality is also questioned by researchers in the field of **evolutionary psychology**, who point to the fact that our brain's decision-making mechanisms did not evolve to solve word problems with probabilities and prizes stated as decimal numbers. Let us grant, for the sake of argument, that the brain has built-in neural mechanism for computing with probabilities and utilities, or something functionally equivalent; if so, the required inputs would be obtained through accumulated experience of outcomes and rewards rather than through linguistic presentations of numerical values. It is far from obvious that we can directly access the brain's built-in neural mechanisms by presenting decision problems in linguistic/numerical form. The very fact that different wordings of the *same decision problem* elicit different choices suggests that the decision problem itself is not getting through. Spurred by this observation, psychologists have tried presenting problems in uncertain reasoning and decision making in "evolutionarily appropriate" forms; for example, instead of saying "90% survival rate," the experimenter might show 100 stick-figure animations of the operation, where the patient dies in 10 of them and survives in 90. (Boredom is a complicating factor in these experiments!) With decision problems posed in this way, people seem to be much closer to rational behavior than previously suspected.

16.4 MULTIATTRIBUTE UTILITY FUNCTIONS

MULTIATTRIBUTE UTILITY THEORY

Decision making in the field of public policy involves high stakes, in both money and lives. For example, in deciding what levels of harmful emissions to allow from a power plant, policy makers must weigh the prevention of death and disability against the benefit of the power and the economic burden of mitigating the emissions. Siting a new airport requires consideration of the disruption caused by construction; the cost of land; the distance from centers of population; the noise of flight operations; safety issues arising from local topography and weather conditions; and so on. Problems like these, in which outcomes are characterized by two or more attributes, are handled by **multiattribute utility theory**.

We will call the attributes $\mathbf{X} = X_1, \dots, X_n$; a complete vector of assignments will be $\mathbf{x} = \langle x_1, \dots, x_n \rangle$, where each x_i is either a numeric value or a discrete value with an assumed ordering on values. We will assume that higher values of an attribute correspond to higher utilities, all other things being equal. For example, if we choose *AbsenceOfNoise* as an attribute in the airport problem, then the greater its value, the better the solution.⁸ We begin by examining cases in which decisions can be made *without* combining the attribute values into a single utility value. Then we look at cases in which the utilities of attribute combinations can be specified very concisely.

16.4.1 Dominance

STRICT DOMINANCE

Suppose that airport site S_1 costs less, generates less noise pollution, and is safer than site S_2 . One would not hesitate to reject S_2 . We then say that there is **strict dominance** of S_1 over S_2 . In general, if an option is of lower value on all attributes than some other option, it need not be considered further. Strict dominance is often very useful in narrowing down the field of choices to the real contenders, although it seldom yields a unique choice. Figure 16.4(a) shows a schematic diagram for the two-attribute case.

That is fine for the deterministic case, in which the attribute values are known for sure. What about the general case, where the outcomes are uncertain? A direct analog of strict dominance can be constructed, where, despite the uncertainty, all possible concrete outcomes for S_1 strictly dominate all possible outcomes for S_2 . (See Figure 16.4(b).) Of course, this will probably occur even less often than in the deterministic case.

STOCHASTIC DOMINANCE

Fortunately, there is a more useful generalization called **stochastic dominance**, which occurs very frequently in real problems. Stochastic dominance is easiest to understand in the context of a single attribute. Suppose we believe that the cost of siting the airport at S_1 is uniformly distributed between \$2.8 billion and \$4.8 billion and that the cost at S_2 is uniformly distributed between \$3 billion and \$5.2 billion. Figure 16.5(a) shows these distributions, with cost plotted as a negative value. Then, given only the information that utility decreases with

⁸ In some cases, it may be necessary to subdivide the range of values so that utility varies monotonically within each range. For example, if the *RoomTemperature* attribute has a utility peak at 70°F, we would split it into two attributes measuring the difference from the ideal, one colder and one hotter. Utility would then be monotonically increasing in each attribute.

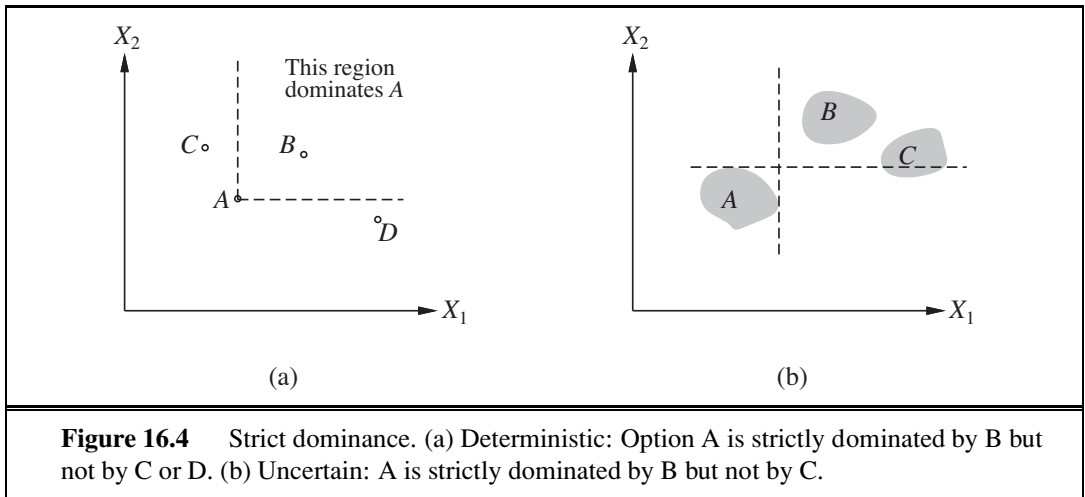


Figure 16.4 Strict dominance. (a) Deterministic: Option A is strictly dominated by B but not by C or D. (b) Uncertain: A is strictly dominated by B but not by C.

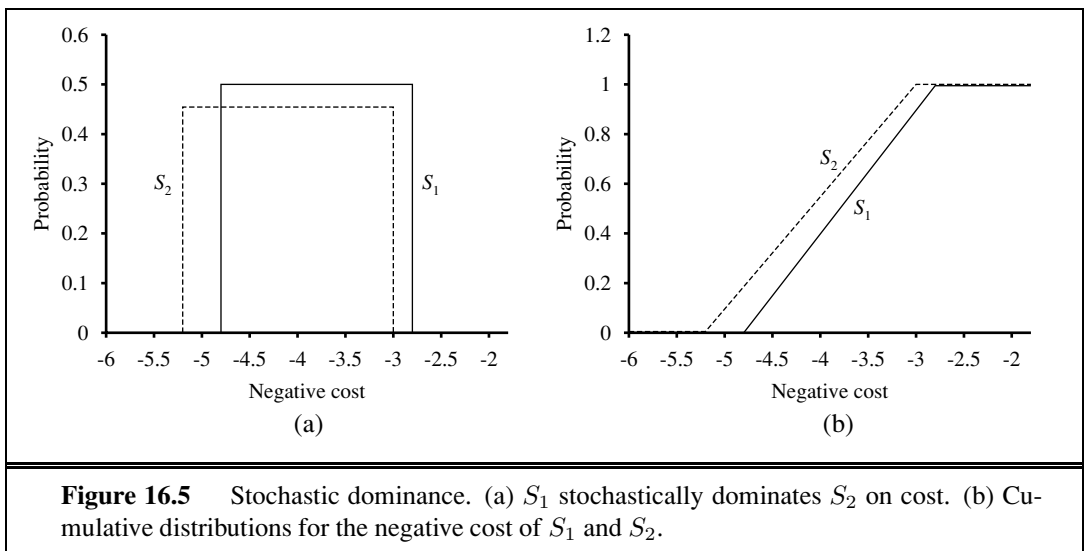


Figure 16.5 Stochastic dominance. (a) S_1 stochastically dominates S_2 on cost. (b) Cumulative distributions for the negative cost of S_1 and S_2 .

cost, we can say that S_1 stochastically dominates S_2 (i.e., S_2 can be discarded). It is important to note that this does *not* follow from comparing the expected costs. For example, if we knew the cost of S_1 to be *exactly* \$3.8 billion, then we would be *unable* to make a decision without additional information on the utility of money. (It might seem odd that *more* information on the cost of S_1 could make the agent *less* able to decide. The paradox is resolved by noting that in the absence of exact cost information, the decision is easier to make but is more likely to be wrong.)

The exact relationship between the attribute distributions needed to establish stochastic dominance is best seen by examining the **cumulative distributions**, shown in Figure 16.5(b). (See also Appendix A.) The cumulative distribution measures the probability that the cost is less than or equal to any given amount—that is, it integrates the original distribution. If the cumulative distribution for S_1 is always to the right of the cumulative distribution for S_2 ,

then, stochastically speaking, S_1 is cheaper than S_2 . Formally, if two actions A_1 and A_2 lead to probability distributions $p_1(x)$ and $p_2(x)$ on attribute X , then A_1 stochastically dominates A_2 on X if

$$\forall x \int_{-\infty}^x p_1(x') dx' \leq \int_{-\infty}^x p_2(x') dx' .$$



The relevance of this definition to the selection of optimal decisions comes from the following property: *if A_1 stochastically dominates A_2 , then for any monotonically nondecreasing utility function $U(x)$, the expected utility of A_1 is at least as high as the expected utility of A_2 .* Hence, if an action is stochastically dominated by another action on all attributes, then it can be discarded.

The stochastic dominance condition might seem rather technical and perhaps not so easy to evaluate without extensive probability calculations. In fact, it can be decided very easily in many cases. Suppose, for example, that the construction transportation cost depends on the distance to the supplier. The cost itself is uncertain, but the greater the distance, the greater the cost. If S_1 is closer than S_2 , then S_1 will dominate S_2 on cost. Although we will not present them here, there exist algorithms for propagating this kind of qualitative information among uncertain variables in **qualitative probabilistic networks**, enabling a system to make rational decisions based on stochastic dominance, without using any numeric values.

QUALITATIVE
PROBABILISTIC
NETWORKS

16.4.2 Preference structure and multiattribute utility

Suppose we have n attributes, each of which has d distinct possible values. To specify the complete utility function $U(x_1, \dots, x_n)$, we need d^n values in the worst case. Now, the worst case corresponds to a situation in which the agent's preferences have no regularity at all. Multiattribute utility theory is based on the supposition that the preferences of typical agents have much more structure than that. The basic approach is to identify regularities in the preference behavior we would expect to see and to use what are called **representation theorems** to show that an agent with a certain kind of preference structure has a utility function

$$U(x_1, \dots, x_n) = F[f_1(x_1), \dots, f_n(x_n)] ,$$

where F is, we hope, a simple function such as addition. Notice the similarity to the use of Bayesian networks to decompose the joint probability of several random variables.

REPRESENTATION
THEOREM

Preferences without uncertainty

Let us begin with the deterministic case. Remember that for deterministic environments the agent has a value function $V(x_1, \dots, x_n)$; the aim is to represent this function concisely. The basic regularity that arises in deterministic preference structures is called **preference independence**. Two attributes X_1 and X_2 are preferentially independent of a third attribute X_3 if the preference between outcomes $\langle x_1, x_2, x_3 \rangle$ and $\langle x'_1, x'_2, x_3 \rangle$ does not depend on the particular value x_3 for attribute X_3 .

PREFERENCE
INDEPENDENCE

Going back to the airport example, where we have (among other attributes) *Noise*, *Cost*, and *Deaths* to consider, one may propose that *Noise* and *Cost* are preferentially inde-

MUTUAL
PREFERENTIAL
INDEPENDENCE

pendent of *Deaths*. For example, if we prefer a state with 20,000 people residing in the flight path and a construction cost of \$4 billion over a state with 70,000 people residing in the flight path and a cost of \$3.7 billion when the safety level is 0.06 deaths per million passenger miles in both cases, then we would have the same preference when the safety level is 0.12 or 0.03; and the same independence would hold for preferences between any other pair of values for *Noise* and *Cost*. It is also apparent that *Cost* and *Deaths* are preferentially independent of *Noise* and that *Noise* and *Deaths* are preferentially independent of *Cost*. We say that the set of attributes $\{\textit{Noise}, \textit{Cost}, \textit{Deaths}\}$ exhibits **mutual preferential independence** (MPI). MPI says that, whereas each attribute may be important, it does not affect the way in which one trades off the other attributes against each other.

Mutual preferential independence is something of a mouthful, but thanks to a remarkable theorem due to the economist Gérard Debreu (1960), we can derive from it a very simple form for the agent's value function: *If attributes X_1, \dots, X_n are mutually preferentially independent, then the agent's preference behavior can be described as maximizing the function*

$$V(x_1, \dots, x_n) = \sum_i V_i(x_i),$$

where each V_i is a value function referring only to the attribute X_i . For example, it might well be the case that the airport decision can be made using a value function

$$V(\textit{noise}, \textit{cost}, \textit{deaths}) = -\textit{noise} \times 10^4 - \textit{cost} - \textit{deaths} \times 10^{12}.$$

ADDITIVE VALUE
FUNCTION

A value function of this type is called an **additive value function**. Additive functions are an extremely natural way to describe an agent's preferences and are valid in many real-world situations. For n attributes, assessing an additive value function requires assessing n separate one-dimensional value functions rather than one n -dimensional function; typically, this represents an exponential reduction in the number of preference experiments that are needed. Even when MPI does not strictly hold, as might be the case at extreme values of the attributes, an additive value function might still provide a good approximation to the agent's preferences. This is especially true when the violations of MPI occur in portions of the attribute ranges that are unlikely to occur in practice.

To understand MPI better, it helps to look at cases where it *doesn't* hold. Suppose you are at a medieval market, considering the purchase of some hunting dogs, some chickens, and some wicker cages for the chickens. The hunting dogs are very valuable, but if you don't have enough cages for the chickens, the dogs will eat the chickens; hence, the tradeoff between dogs and chickens depends strongly on the number of cages, and MPI is violated. The existence of these kinds of interactions among various attributes makes it much harder to assess the overall value function.

Preferences with uncertainty

When uncertainty is present in the domain, we also need to consider the structure of preferences between lotteries and to understand the resulting properties of utility functions, rather than just value functions. The mathematics of this problem can become quite complicated, so we present just one of the main results to give a flavor of what can be done. The reader is referred to Keeney and Raiffa (1976) for a thorough survey of the field.

UTILITY
INDEPENDENCE

The basic notion of **utility independence** extends preference independence to cover lotteries: a set of attributes \mathbf{X} is utility independent of a set of attributes \mathbf{Y} if preferences between lotteries on the attributes in \mathbf{X} are independent of the particular values of the attributes in \mathbf{Y} . A set of attributes is **mutually utility independent** (MUI) if each of its subsets is utility-independent of the remaining attributes. Again, it seems reasonable to propose that the airport attributes are MUI.

MUTUALLY UTILITY
INDEPENDENTMULTIPLICATIVE
UTILITY FUNCTION

MUI implies that the agent's behavior can be described using a **multiplicative utility function** (Keeney, 1974). The general form of a multiplicative utility function is best seen by looking at the case for three attributes. For conciseness, we use U_i to mean $U_i(x_i)$:

$$U = k_1U_1 + k_2U_2 + k_3U_3 + k_1k_2U_1U_2 + k_2k_3U_2U_3 + k_3k_1U_3U_1 + k_1k_2k_3U_1U_2U_3.$$

Although this does not look very simple, it contains just three single-attribute utility functions and three constants. In general, an n -attribute problem exhibiting MUI can be modeled using n single-attribute utilities and n constants. Each of the single-attribute utility functions can be developed independently of the other attributes, and this combination will be guaranteed to generate the correct overall preferences. Additional assumptions are required to obtain a purely additive utility function.

16.5 DECISION NETWORKS

INFLUENCE DIAGRAM
DECISION NETWORK

In this section, we look at a general mechanism for making rational decisions. The notation is often called an **influence diagram** (Howard and Matheson, 1984), but we will use the more descriptive term **decision network**. Decision networks combine Bayesian networks with additional node types for actions and utilities. We use airport siting as an example.

16.5.1 Representing a decision problem with a decision network

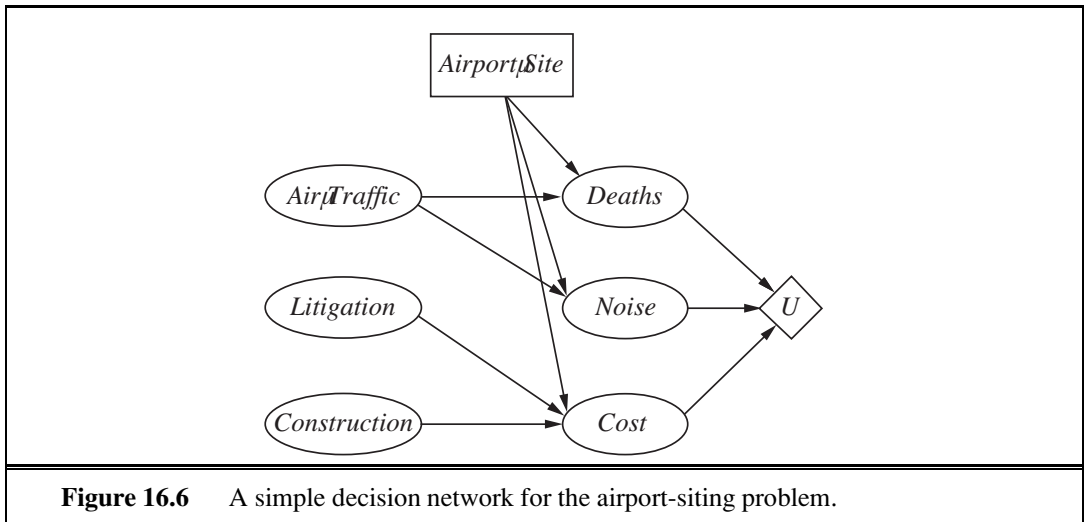
In its most general form, a decision network represents information about the agent's current state, its possible actions, the state that will result from the agent's action, and the utility of that state. It therefore provides a substrate for implementing utility-based agents of the type first introduced in Section 2.4.5. Figure 16.6 shows a decision network for the airport siting problem. It illustrates the three types of nodes used:

CHANCE NODES

- **Chance nodes** (ovals) represent random variables, just as they do in Bayesian networks. The agent could be uncertain about the construction cost, the level of air traffic and the potential for litigation, and the *Deaths*, *Noise*, and total *Cost* variables, each of which also depends on the site chosen. Each chance node has associated with it a conditional distribution that is indexed by the state of the parent nodes. In decision networks, the parent nodes can include decision nodes as well as chance nodes. Note that each of the current-state chance nodes could be part of a large Bayesian network for assessing construction costs, air traffic levels, or litigation potentials.

DECISION NODES

- **Decision nodes** (rectangles) represent points where the decision maker has a choice of



actions. In this case, the *AirportSite* action can take on a different value for each site under consideration. The choice influences the cost, safety, and noise that will result. In this chapter, we assume that we are dealing with a single decision node. Chapter 17 deals with cases in which more than one decision must be made.

UTILITY NODES

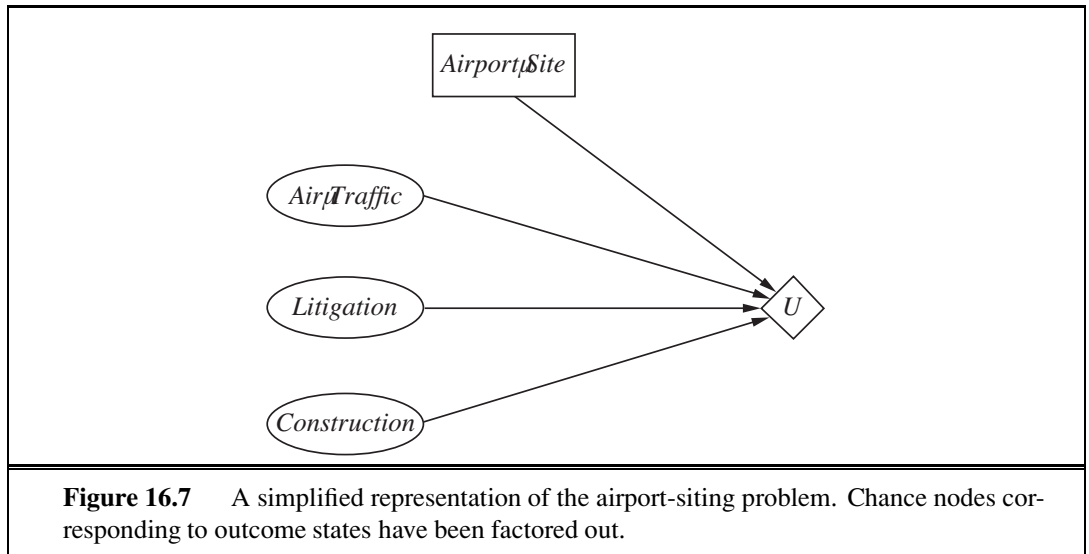
- **Utility nodes** (diamonds) represent the agent's utility function.⁹ The utility node has as parents all variables describing the outcome that directly affect utility. Associated with the utility node is a description of the agent's utility as a function of the parent attributes. The description could be just a tabulation of the function, or it might be a parameterized additive or linear function of the attribute values.

A simplified form is also used in many cases. The notation remains identical, but the chance nodes describing the outcome state are omitted. Instead, the utility node is connected directly to the current-state nodes and the decision node. In this case, rather than representing a utility function on outcome states, the utility node represents the *expected* utility associated with each action, as defined in Equation (16.1) on page 611; that is, the node is associated with an **action-utility function** (also known as a **Q-function** in reinforcement learning, as described in Chapter 21). Figure 16.7 shows the action-utility representation of the airport siting problem.

ACTION-UTILITY FUNCTION

Notice that, because the *Noise*, *Deaths*, and *Cost* chance nodes in Figure 16.6 refer to future states, they can never have their values set as evidence variables. Thus, the simplified version that omits these nodes can be used whenever the more general form can be used. Although the simplified form contains fewer nodes, the omission of an explicit description of the outcome of the siting decision means that it is less flexible with respect to changes in circumstances. For example, in Figure 16.6, a change in aircraft noise levels can be reflected by a change in the conditional probability table associated with the *Noise* node, whereas a change in the weight accorded to noise pollution in the utility function can be reflected by

⁹ These nodes are also called **value nodes** in the literature.



a change in the utility table. In the action-utility diagram, Figure 16.7, on the other hand, all such changes have to be reflected by changes to the action-utility table. Essentially, the action-utility formulation is a *compiled* version of the original formulation.

16.5.2 Evaluating decision networks

Actions are selected by evaluating the decision network for each possible setting of the decision node. Once the decision node is set, it behaves exactly like a chance node that has been set as an evidence variable. The algorithm for evaluating decision networks is the following:

1. Set the evidence variables for the current state.
2. For each possible value of the decision node:
 - (a) Set the decision node to that value.
 - (b) Calculate the posterior probabilities for the parent nodes of the utility node, using a standard probabilistic inference algorithm.
 - (c) Calculate the resulting utility for the action.
3. Return the action with the highest utility.

This is a straightforward extension of the Bayesian network algorithm and can be incorporated directly into the agent design given in Figure 13.1 on page 484. We will see in Chapter 17 that the possibility of executing several actions in sequence makes the problem much more interesting.

16.6 THE VALUE OF INFORMATION

In the preceding analysis, we have assumed that all relevant information, or at least all available information, is provided to the agent before it makes its decision. In practice, this is



INFORMATION VALUE THEORY

hardly ever the case. *One of the most important parts of decision making is knowing what questions to ask.* For example, a doctor cannot expect to be provided with the results of *all possible* diagnostic tests and questions at the time a patient first enters the consulting room.¹⁰ Tests are often expensive and sometimes hazardous (both directly and because of associated delays). Their importance depends on two factors: whether the test results would lead to a significantly better treatment plan, and how likely the various test results are.

This section describes **information value theory**, which enables an agent to choose what information to acquire. We assume that, prior to selecting a “real” action represented by the decision node, the agent can acquire the value of any of the potentially observable chance variables in the model. Thus, information value theory involves a simplified form of sequential decision making—simplified because the observation actions affect only the agent’s **belief state**, not the external physical state. The value of any particular observation must derive from the potential to affect the agent’s eventual physical action; and this potential can be estimated directly from the decision model itself.

16.6.1 A simple example

Suppose an oil company is hoping to buy one of n indistinguishable blocks of ocean-drilling rights. Let us assume further that exactly one of the blocks contains oil worth C dollars, while the others are worthless. The asking price of each block is C/n dollars. If the company is risk-neutral, then it will be indifferent between buying a block and not buying one.

Now suppose that a seismologist offers the company the results of a survey of block number 3, which indicates definitively whether the block contains oil. How much should the company be willing to pay for the information? The way to answer this question is to examine what the company would do if it had the information:

- With probability $1/n$, the survey will indicate oil in block 3. In this case, the company will buy block 3 for C/n dollars and make a profit of $C - C/n = (n-1)C/n$ dollars.
- With probability $(n-1)/n$, the survey will show that the block contains no oil, in which case the company will buy a different block. Now the probability of finding oil in one of the other blocks changes from $1/n$ to $1/(n-1)$, so the company makes an expected profit of $C/(n-1) - C/n = C/n(n-1)$ dollars.

Now we can calculate the expected profit, given the survey information:

$$\frac{1}{n} \times \frac{(n-1)C}{n} + \frac{n-1}{n} \times \frac{C}{n(n-1)} = C/n.$$

Therefore, the company should be willing to pay the seismologist up to C/n dollars for the information: the information is worth as much as the block itself.

The value of information derives from the fact that *with* the information, one’s course of action can be changed to suit the *actual* situation. One can discriminate according to the situation, whereas without the information, one has to do what’s best on average over the possible situations. In general, the value of a given piece of information is defined to be the difference in expected value between best actions before and after information is obtained.

¹⁰ In the United States, the only question that is always asked beforehand is whether the patient has insurance.

16.6.2 A general formula for perfect information

It is simple to derive a general mathematical formula for the value of information. We assume that exact evidence can be obtained about the value of some random variable E_j (that is, we learn $E_j = e_j$), so the phrase **value of perfect information** (VPI) is used.¹¹

Let the agent's initial evidence be \mathbf{e} . Then the value of the current best action α is defined by

$$EU(\alpha|\mathbf{e}) = \max_a \sum_{s'} P(\text{RESULT}(a) = s' | a, \mathbf{e}) U(s') ,$$

and the value of the new best action (after the new evidence $E_j = e_j$ is obtained) will be

$$EU(\alpha_{e_j}|\mathbf{e}, e_j) = \max_a \sum_{s'} P(\text{RESULT}(a) = s' | a, \mathbf{e}, e_j) U(s') .$$

But E_j is a random variable whose value is *currently* unknown, so to determine the value of discovering E_j , given current information \mathbf{e} we must average over all possible values e_{jk} that we might discover for E_j , using our *current* beliefs about its value:

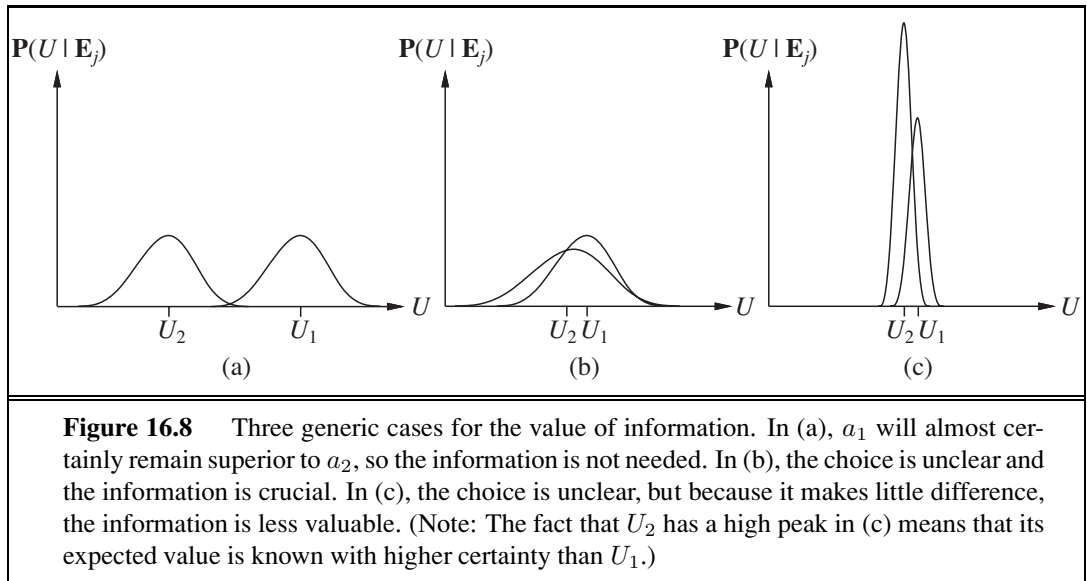
$$VPI_{\mathbf{e}}(E_j) = \left(\sum_k P(E_j = e_{jk}|\mathbf{e}) EU(\alpha_{e_{jk}}|\mathbf{e}, E_j = e_{jk}) \right) - EU(\alpha|\mathbf{e}) .$$

To get some intuition for this formula, consider the simple case where there are only two actions, a_1 and a_2 , from which to choose. Their current expected utilities are U_1 and U_2 . The information $E_j = e_{jk}$ will yield some new expected utilities U'_1 and U'_2 for the actions, but before we obtain E_j , we will have some probability distributions over the possible values of U'_1 and U'_2 (which we assume are independent).

Suppose that a_1 and a_2 represent two different routes through a mountain range in winter. a_1 is a nice, straight highway through a low pass, and a_2 is a winding dirt road over the top. Just given this information, a_1 is clearly preferable, because it is quite possible that a_2 is blocked by avalanches, whereas it is unlikely that anything blocks a_1 . U_1 is therefore clearly higher than U_2 . It is possible to obtain satellite reports E_j on the actual state of each road that would give new expectations, U'_1 and U'_2 , for the two crossings. The distributions for these expectations are shown in Figure 16.8(a). Obviously, in this case, it is not worth the expense of obtaining satellite reports, because it is unlikely that the information derived from them will change the plan. With no change, information has no value.

Now suppose that we are choosing between two different winding dirt roads of slightly different lengths and we are carrying a seriously injured passenger. Then, even when U_1 and U_2 are quite close, the distributions of U'_1 and U'_2 are very broad. There is a significant possibility that the second route will turn out to be clear while the first is blocked, and in this

¹¹ There is no loss of expressiveness in requiring perfect information. Suppose we wanted to model the case in which we become somewhat more certain about a variable. We can do that by introducing *another* variable about which we learn perfect information. For example, suppose we initially have broad uncertainty about the variable *Temperature*. Then we gain the perfect knowledge $\text{Thermometer} = 37$; this gives us imperfect information about the true *Temperature*, and the uncertainty due to measurement error is encoded in the sensor model $\mathbf{P}(\text{Thermometer} | \text{Temperature})$. See Exercise 16.17 for another example.



case the difference in utilities will be very high. The VPI formula indicates that it might be worthwhile getting the satellite reports. Such a situation is shown in Figure 16.8(b).

Finally, suppose that we are choosing between the two dirt roads in summertime, when blockage by avalanches is unlikely. In this case, satellite reports might show one route to be more scenic than the other because of flowering alpine meadows, or perhaps wetter because of errant streams. It is therefore quite likely that we would change our plan if we had the information. In this case, however, the difference in value between the two routes is still likely to be very small, so we will not bother to obtain the reports. This situation is shown in Figure 16.8(c).



In sum, *information has value to the extent that it is likely to cause a change of plan and to the extent that the new plan will be significantly better than the old plan.*

16.6.3 Properties of the value of information

One might ask whether it is possible for information to be deleterious: can it actually have negative expected value? Intuitively, one should expect this to be impossible. After all, one could in the worst case just ignore the information and pretend that one has never received it. This is confirmed by the following theorem, which applies to any decision-theoretic agent:



The expected value of information is nonnegative:

$$\forall \mathbf{e}, E_j \quad \text{VPI}_{\mathbf{e}}(E_j) \geq 0.$$

The theorem follows directly from the definition of VPI, and we leave the proof as an exercise (Exercise 16.18). It is, of course, a theorem about *expected* value, not *actual* value. Additional information can easily lead to a plan that *turns out to be* worse than the original plan if the information happens to be misleading. For example, a medical test that gives a false positive result may lead to unnecessary surgery; but that does not mean that the test shouldn't be done.

It is important to remember that VPI depends on the current state of information, which is why it is subscripted. It can change as more information is acquired. For any given piece of evidence E_j , the value of acquiring it can go down (e.g., if another variable strongly constrains the posterior for E_j) or up (e.g., if another variable provides a clue on which E_j builds, enabling a new and better plan to be devised). Thus, VPI is not additive. That is,

$$VPI_e(E_j, E_k) \neq VPI_e(E_j) + VPI_e(E_k) \quad (\text{in general}) .$$

VPI is, however, order independent. That is,

$$VPI_e(E_j, E_k) = VPI_e(E_j) + VPI_{e,e_j}(E_k) = VPI_e(E_k) + VPI_{e,e_k}(E_j) .$$

Order independence distinguishes sensing actions from ordinary actions and simplifies the problem of calculating the value of a sequence of sensing actions.

16.6.4 Implementation of an information-gathering agent

A sensible agent should ask questions in a reasonable order, should avoid asking questions that are irrelevant, should take into account the importance of each piece of information in relation to its cost, and should stop asking questions when that is appropriate. All of these capabilities can be achieved by using the value of information as a guide.

Figure 16.9 shows the overall design of an agent that can gather information intelligently before acting. For now, we assume that with each observable evidence variable E_j , there is an associated cost, $Cost(E_j)$, which reflects the cost of obtaining the evidence through tests, consultants, questions, or whatever. The agent requests what appears to be the most efficient observation in terms of utility gain per unit cost. We assume that the result of the action $Request(E_j)$ is that the next percept provides the value of E_j . If no observation is worth its cost, the agent selects a “real” action.

The agent algorithm we have described implements a form of information gathering that is called **myopic**. This is because it uses the VPI formula shortsightedly, calculating the value of information as if only a single evidence variable will be acquired. Myopic control is based on the same heuristic idea as greedy search and often works well in practice. (For example, it has been shown to outperform expert physicians in selecting diagnostic tests.)

MYOPIC

```
function INFORMATION-GATHERING-AGENT(percept) returns an action
  persistent: D, a decision network

  integrate percept into D
  j ← the value that maximizes  $VPI(E_j) / Cost(E_j)$ 
  if  $VPI(E_j) > Cost(E_j)$ 
    return REQUEST( $E_j$ )
  else return the best action from D
```

Figure 16.9 Design of a simple information-gathering agent. The agent works by repeatedly selecting the observation with the highest information value, until the cost of the next observation is greater than its expected benefit.

However, if there is no single evidence variable that will help a lot, a myopic agent might hastily take an action when it would have been better to request two or more variables first and then take action. A better approach in this situation would be to construct a *conditional plan* (as described in Section 11.3.2) that asks for variable values and takes different next steps depending on the answer.

One final consideration is the effect a series of questions will have on a human respondent. People may respond better to a series of questions if they “make sense,” so some expert systems are built to take this into account, asking questions in an order that maximizes the total utility of the system and human rather than an order that maximizes value of information.

16.7 DECISION-THEORETIC EXPERT SYSTEMS

DECISION ANALYSIS The field of **decision analysis**, which evolved in the 1950s and 1960s, studies the application of decision theory to actual decision problems. It is used to help make rational decisions in important domains where the stakes are high, such as business, government, law, military strategy, medical diagnosis and public health, engineering design, and resource management. The process involves a careful study of the possible actions and outcomes, as well as the preferences placed on each outcome. It is traditional in decision analysis to talk about two roles: the **decision maker** states preferences between outcomes, and the **decision analyst** enumerates the possible actions and outcomes and elicits preferences from the decision maker to determine the best course of action. Until the early 1980s, the main purpose of decision analysis was to help humans make decisions that actually reflect their own preferences. As more and more decision processes become automated, decision analysis is increasingly used to ensure that the automated processes are behaving as desired.

DECISION MAKER

DECISION ANALYST

Early expert system research concentrated on answering questions, rather than on making decisions. Those systems that did recommend actions rather than providing opinions on matters of fact generally did so using condition-action rules, rather than with explicit representations of outcomes and preferences. The emergence of Bayesian networks in the late 1980s made it possible to build large-scale systems that generated sound probabilistic inferences from evidence. The addition of decision networks means that expert systems can be developed that recommend optimal decisions, reflecting the preferences of the agent as well as the available evidence.

A system that incorporates utilities can avoid one of the most common pitfalls associated with the consultation process: confusing likelihood and importance. A common strategy in early medical expert systems, for example, was to rank possible diagnoses in order of likelihood and report the most likely. Unfortunately, this can be disastrous! For the majority of patients in general practice, the two most *likely* diagnoses are usually “There’s nothing wrong with you” and “You have a bad cold,” but if the third most likely diagnosis for a given patient is lung cancer, that’s a serious matter. Obviously, a testing or treatment plan should depend both on probabilities and utilities. Current medical expert systems can take into account the value of information to recommend tests, and then describe a differential diagnosis.

We now describe the knowledge engineering process for decision-theoretic expert systems. As an example we consider the problem of selecting a medical treatment for a kind of congenital heart disease in children (see Lucas, 1996).

About 0.8% of children are born with a heart anomaly, the most common being **aortic coarctation** (a constriction of the aorta). It can be treated with surgery, angioplasty (expanding the aorta with a balloon placed inside the artery), or medication. The problem is to decide what treatment to use and when to do it: the younger the infant, the greater the risks of certain treatments, but one mustn't wait too long. A decision-theoretic expert system for this problem can be created by a team consisting of at least one domain expert (a pediatric cardiologist) and one knowledge engineer. The process can be broken down into the following steps:

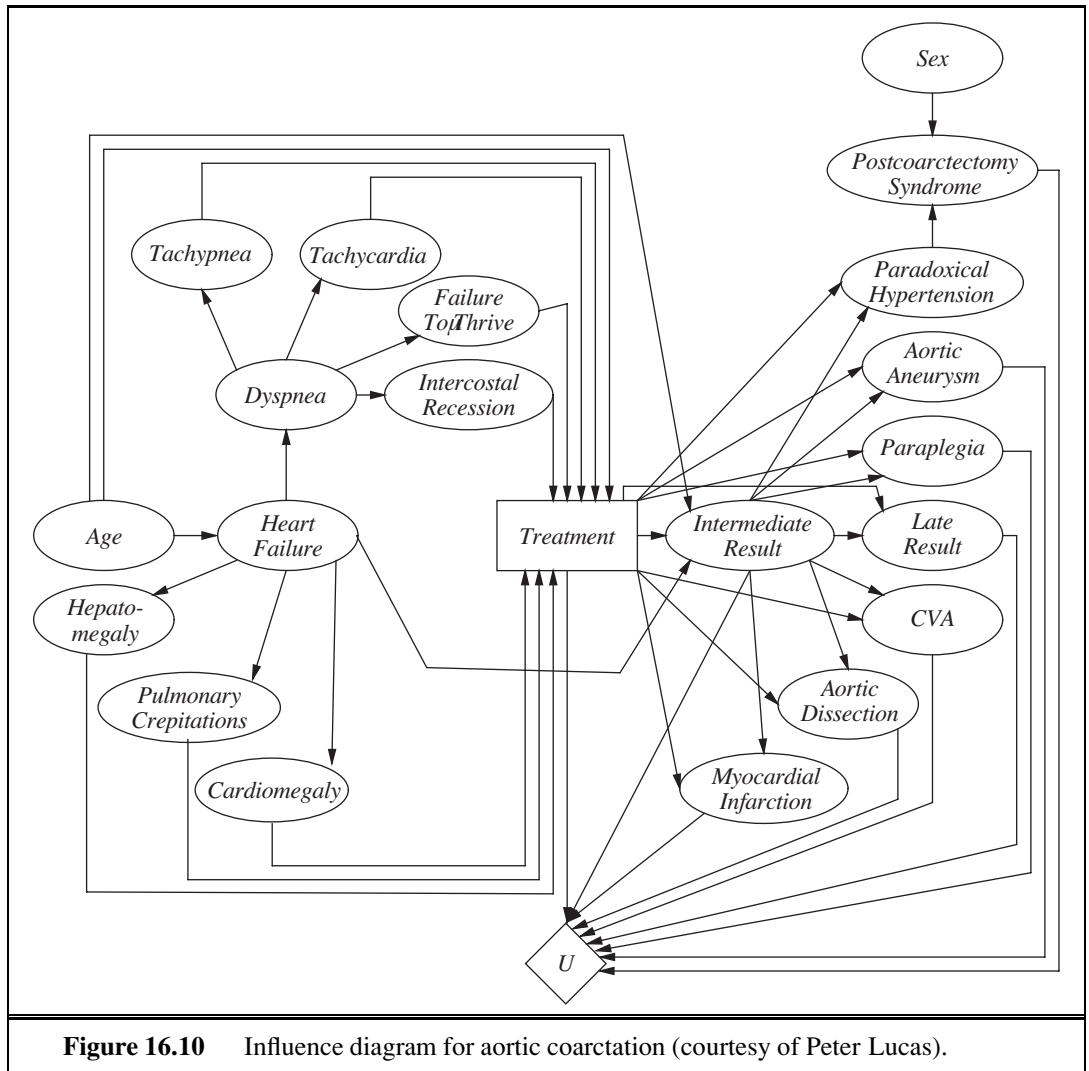
Create a causal model. Determine the possible symptoms, disorders, treatments, and outcomes. Then draw arcs between them, indicating what disorders cause what symptoms, and what treatments alleviate what disorders. Some of this will be well known to the domain expert, and some will come from the literature. Often the model will match well with the informal graphical descriptions given in medical textbooks.

Simplify to a qualitative decision model. Since we are using the model to make treatment decisions and not for other purposes (such as determining the joint probability of certain symptom/disorder combinations), we can often simplify by removing variables that are not involved in treatment decisions. Sometimes variables will have to be split or joined to match the expert's intuitions. For example, the original aortic coarctation model had a *Treatment* variable with values *surgery*, *angioplasty*, and *medication*, and a separate variable for *Timing* of the treatment. But the expert had a hard time thinking of these separately, so they were combined, with *Treatment* taking on values such as *surgery in 1 month*. This gives us the model of Figure 16.10.

Assign probabilities. Probabilities can come from patient databases, literature studies, or the expert's subjective assessments. Note that a diagnostic system will reason from symptoms and other observations to the disease or other cause of the problems. Thus, in the early years of building these systems, experts were asked for the probability of a cause given an effect. In general they found this difficult to do, and were better able to assess the probability of an effect given a cause. So modern systems usually assess causal knowledge and encode it directly in the Bayesian network structure of the model, leaving the diagnostic reasoning to the Bayesian network inference algorithms (Shachter and Heckerman, 1987).

Assign utilities. When there are a small number of possible outcomes, they can be enumerated and evaluated individually using the methods of Section 16.3.1. We would create a scale from best to worst outcome and give each a numeric value, for example 0 for death and 1 for complete recovery. We would then place the other outcomes on this scale. This can be done by the expert, but it is better if the patient (or in the case of infants, the patient's parents) can be involved, because different people have different preferences. If there are exponentially many outcomes, we need some way to combine them using multiattribute utility functions. For example, we may say that the costs of various complications are additive.

Verify and refine the model. To evaluate the system we need a set of correct (input, output) pairs; a so-called **gold standard** to compare against. For medical expert systems this usually means assembling the best available doctors, presenting them with a few cases,



and asking them for their diagnosis and recommended treatment plan. We then see how well the system matches their recommendations. If it does poorly, we try to isolate the parts that are going wrong and fix them. It can be useful to run the system “backward.” Instead of presenting the system with symptoms and asking for a diagnosis, we can present it with a diagnosis such as “heart failure,” examine the predicted probability of symptoms such as tachycardia, and compare with the medical literature.

SENSITIVITY ANALYSIS

Perform sensitivity analysis. This important step checks whether the best decision is sensitive to small changes in the assigned probabilities and utilities by systematically varying those parameters and running the evaluation again. If small changes lead to significantly different decisions, then it could be worthwhile to spend more resources to collect better data. If all variations lead to the same decision, then the agent will have more confidence that it is the right decision. Sensitivity analysis is particularly important, because one of the main

criticisms of probabilistic approaches to expert systems is that it is too difficult to assess the numerical probabilities required. Sensitivity analysis often reveals that many of the numbers need be specified only very approximately. For example, we might be uncertain about the conditional probability $P(\textit{tachycardia} \mid \textit{dyspnea})$, but if the optimal decision is reasonably robust to small variations in the probability, then our ignorance is less of a concern.

16.8 SUMMARY

This chapter shows how to combine utility theory with probability to enable an agent to select actions that will maximize its expected performance.

- **Probability theory** describes what an agent should believe on the basis of evidence, **utility theory** describes what an agent wants, and **decision theory** puts the two together to describe what an agent should do.
- We can use decision theory to build a system that makes decisions by considering all possible actions and choosing the one that leads to the best expected outcome. Such a system is known as a **rational agent**.
- Utility theory shows that an agent whose preferences between lotteries are consistent with a set of simple axioms can be described as possessing a utility function; furthermore, the agent selects actions as if maximizing its expected utility.
- **Multiattribute utility theory** deals with utilities that depend on several distinct attributes of states. **Stochastic dominance** is a particularly useful technique for making unambiguous decisions, even without precise utility values for attributes.
- **Decision networks** provide a simple formalism for expressing and solving decision problems. They are a natural extension of Bayesian networks, containing decision and utility nodes in addition to chance nodes.
- Sometimes, solving a problem involves finding more information before making a decision. The **value of information** is defined as the expected improvement in utility compared with making a decision without the information.
- **Expert systems** that incorporate utility information have additional capabilities compared with pure inference systems. In addition to being able to make decisions, they can use the value of information to decide which questions to ask, if any; they can recommend contingency plans; and they can calculate the sensitivity of their decisions to small changes in probability and utility assessments.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

The book *L'art de Penser*, also known as the *Port-Royal Logic* (Arnauld, 1662) states:

To judge what one must do to obtain a good or avoid an evil, it is necessary to consider not only the good and the evil in itself, but also the probability that it happens or does not happen; and to view geometrically the proportion that all these things have together.

Modern texts talk of *utility* rather than good and evil, but this statement correctly notes that one should multiply utility by probability (“view geometrically”) to give expected utility, and maximize that over all outcomes (“all these things”) to “judge what one must do.” It is remarkable how much this got right, 350 years ago, and only 8 years after Pascal and Fermat showed how to use probability correctly. The Port-Royal Logic also marked the first publication of Pascal’s wager.

Daniel Bernoulli (1738), investigating the St. Petersburg paradox (see Exercise 16.3), was the first to realize the importance of preference measurement for lotteries, writing “the *value* of an item must not be based on its *price*, but rather on the *utility* that it yields” (*italics his*). Utilitarian philosopher Jeremy Bentham (1823) proposed the **hedonic calculus** for weighing “pleasures” and “pains,” arguing that all decisions (not just monetary ones) could be reduced to utility comparisons.

The derivation of numerical utilities from preferences was first carried out by Ramsey (1931); the axioms for preference in the present text are closer in form to those rediscovered in *Theory of Games and Economic Behavior* (von Neumann and Morgenstern, 1944). A good presentation of these axioms, in the course of a discussion on risk preference, is given by Howard (1977). Ramsey had derived subjective probabilities (not just utilities) from an agent’s preferences; Savage (1954) and Jeffrey (1983) carry out more recent constructions of this kind. Von Winterfeldt and Edwards (1986) provide a modern perspective on decision analysis and its relationship to human preference structures. The micromort utility measure is discussed by Howard (1989). A 1994 survey by the *Economist* set the value of a life at between \$750,000 and \$2.6 million. However, Richard Thaler (1992) found irrational framing effects on the price one is willing to pay to avoid a risk of death versus the price one is willing to be paid to accept a risk. For a 1/1000 chance, a respondent wouldn’t pay more than \$200 to remove the risk, but wouldn’t accept \$50,000 to take on the risk. How much are people willing to pay for a QALY? When it comes down to a specific case of saving oneself or a family member, the number is approximately “whatever I’ve got.” But we can ask at a societal level: suppose there is a vaccine that would yield X QALYs but costs Y dollars; is it worth it? In this case people report a wide range of values from around \$10,000 to \$150,000 per QALY (Prades *et al.*, 2008). QALYs are much more widely used in medical and social policy decision making than are micromorts; see (Russell, 1990) for a typical example of an argument for a major change in public health policy on grounds of increased expected utility measured in QALYs.

The **optimizer’s curse** was brought to the attention of decision analysts in a forceful way by Smith and Winkler (2006), who pointed out that the financial benefits to the client projected by analysts for their proposed course of action almost never materialized. They trace this directly to the bias introduced by selecting an optimal action and show that a more complete Bayesian analysis eliminates the problem. The same underlying concept has been called **post-decision disappointment** by Harrison and March (1984) and was noted in the context of analyzing capital investment projects by Brown (1974). The optimizer’s curse is also closely related to the **winner’s curse** (Capen *et al.*, 1971; Thaler, 1992), which applies to competitive bidding in auctions: whoever wins the auction is very likely to have overestimated the value of the object in question. Capen *et al.* quote a petroleum engineer on the

POST-DECISION
DISAPPOINTMENT

WINNER’S CURSE

topic of bidding for oil-drilling rights: “If one wins a tract against two or three others he may feel fine about his good fortune. But how should he feel if he won against 50 others? Ill.” Finally, behind both curses is the general phenomenon of **regression to the mean**, whereby individuals selected on the basis of exceptional characteristics previously exhibited will, with high probability, become less exceptional in future.

The Allais paradox, due to Nobel Prize-winning economist Maurice Allais (1953) was tested experimentally (Tversky and Kahneman, 1982; Conlisk, 1989) to show that people are consistently inconsistent in their judgments. The Ellsberg paradox on ambiguity aversion was introduced in the Ph.D. thesis of Daniel Ellsberg (Ellsberg, 1962), who went on to become a military analyst at the RAND Corporation and to leak documents known as The Pentagon Papers, which contributed to the end of the Vietnam war and the resignation of President Nixon. Fox and Tversky (1995) describe a further study of ambiguity aversion. Mark Machina (2005) gives an overview of choice under uncertainty and how it can vary from expected utility theory.

There has been a recent outpouring of more-or-less popular books on human irrationality. The best known is *Predictably Irrational* (Ariely, 2009); others include *Sway* (Brafman and Brafman, 2009), *Nudge* (Thaler and Sunstein, 2009), *Kluge* (Marcus, 2009), *How We Decide* (Lehrer, 2009) and *On Being Certain* (Burton, 2009). They complement the classic (Kahneman *et al.*, 1982) and the article that started it all (Kahneman and Tversky, 1979). The field of evolutionary psychology (Buss, 2005), on the other hand, has run counter to this literature, arguing that humans are quite rational in evolutionarily appropriate contexts. Its adherents point out that irrationality is penalized by definition in an evolutionary context and show that in some cases it is an artifact of the experimental setup (Cummins and Allen, 1998). There has been a recent resurgence of interest in Bayesian models of cognition, overturning decades of pessimism (Oaksford and Chater, 1998; Elio, 2002; Chater and Oaksford, 2008).

Keeney and Raiffa (1976) give a thorough introduction to multiattribute utility theory. They describe early computer implementations of methods for eliciting the necessary parameters for a multiattribute utility function and include extensive accounts of real applications of the theory. In AI, the principal reference for MAUT is Wellman’s (1985) paper, which includes a system called URP (Utility Reasoning Package) that can use a collection of statements about preference independence and conditional independence to analyze the structure of decision problems. The use of stochastic dominance together with qualitative probability models was investigated extensively by Wellman (1988, 1990a). Wellman and Doyle (1992) provide a preliminary sketch of how a complex set of utility-independence relationships might be used to provide a structured model of a utility function, in much the same way that Bayesian networks provide a structured model of joint probability distributions. Bacchus and Grove (1995, 1996) and La Mura and Shoham (1999) give further results along these lines.

Decision theory has been a standard tool in economics, finance, and management science since the 1950s. Until the 1980s, decision trees were the main tool used for representing simple decision problems. Smith (1988) gives an overview of the methodology of decision analysis. Influence diagrams were introduced by Howard and Matheson (1984), based on earlier work at SRI (Miller *et al.*, 1976). Howard and Matheson’s method involved the

derivation of a decision tree from a decision network, but in general the tree is of exponential size. Shachter (1986) developed a method for making decisions based directly on a decision network, without the creation of an intermediate decision tree. This algorithm was also one of the first to provide complete inference for multiply connected Bayesian networks. Zhang *et al.* (1994) showed how to take advantage of conditional independence of information to reduce the size of trees in practice; they use the term *decision network* for networks that use this approach (although others use it as a synonym for influence diagram). Nilsson and Lauritzen (2000) link algorithms for decision networks to ongoing developments in clustering algorithms for Bayesian networks. Koller and Milch (2003) show how influence diagrams can be used to solve games that involve gathering information by opposing players, and Detwarasiti and Shachter (2005) show how influence diagrams can be used as an aid to decision making for a team that shares goals but is unable to share all information perfectly. The collection by Oliver and Smith (1990) has a number of useful articles on decision networks, as does the 1990 special issue of the journal *Networks*. Papers on decision networks and utility modeling also appear regularly in the journals *Management Science* and *Decision Analysis*.

The theory of information value was explored first in the context of statistical experiments, where a quasi-utility (entropy reduction) was used (Lindley, 1956). The Russian control theorist Ruslan Stratonovich (1965) developed the more general theory presented here, in which information has value by virtue of its ability to affect decisions. Stratonovich's work was not known in the West, where Ron Howard (1966) pioneered the same idea. His paper ends with the remark "If information value theory and associated decision theoretic structures do not in the future occupy a large part of the education of engineers, then the engineering profession will find that its traditional role of managing scientific and economic resources for the benefit of man has been forfeited to another profession." To date, the implied revolution in managerial methods has not occurred.

Recent work by Krause and Guestrin (2009) shows that computing the exact non-myopic value of information is intractable even in polytree networks. There are other cases—more restricted than general value of information—in which the myopic algorithm does provide a provably good approximation to the optimal sequence of observations (Krause *et al.*, 2008). In some cases—for example, looking for treasure buried in one of n places—ranking experiments in order of success probability divided by cost gives an optimal solution (Kadane and Simon, 1977).

Surprisingly few early AI researchers adopted decision-theoretic tools after the early applications in medical decision making described in Chapter 13. One of the few exceptions was Jerry Feldman, who applied decision theory to problems in vision (Feldman and Yakimovsky, 1974) and planning (Feldman and Sproull, 1977). After the resurgence of interest in probabilistic methods in AI in the 1980s, decision-theoretic expert systems gained widespread acceptance (Horvitz *et al.*, 1988; Cowell *et al.*, 2002). In fact, from 1991 onward, the cover design of the journal *Artificial Intelligence* has depicted a decision network, although some artistic license appears to have been taken with the direction of the arrows.

EXERCISES

16.1 (Adapted from David Heckerman.) This exercise concerns the **Almanac Game**, which is used by decision analysts to calibrate numeric estimation. For each of the questions that follow, give your best guess of the answer, that is, a number that you think is as likely to be too high as it is to be too low. Also give your guess at a 25th percentile estimate, that is, a number that you think has a 25% chance of being too high, and a 75% chance of being too low. Do the same for the 75th percentile. (Thus, you should give three estimates in all—low, median, and high—for each question.)

- a. Number of passengers who flew between New York and Los Angeles in 1989.
- b. Population of Warsaw in 1992.
- c. Year in which Coronado discovered the Mississippi River.
- d. Number of votes received by Jimmy Carter in the 1976 presidential election.
- e. Age of the oldest living tree, as of 2002.
- f. Height of the Hoover Dam in feet.
- g. Number of eggs produced in Oregon in 1985.
- h. Number of Buddhists in the world in 1992.
- i. Number of deaths due to AIDS in the United States in 1981.
- j. Number of U.S. patents granted in 1901.

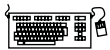
The correct answers appear after the last exercise of this chapter. From the point of view of decision analysis, the interesting thing is not how close your median guesses came to the real answers, but rather how often the real answer came within your 25% and 75% bounds. If it was about half the time, then your bounds are accurate. But if you're like most people, you will be more sure of yourself than you should be, and fewer than half the answers will fall within the bounds. With practice, you can calibrate yourself to give realistic bounds, and thus be more useful in supplying information for decision making. Try this second set of questions and see if there is any improvement:

- a. Year of birth of Zsa Zsa Gabor.
- b. Maximum distance from Mars to the sun in miles.
- c. Value in dollars of exports of wheat from the United States in 1992.
- d. Tons handled by the port of Honolulu in 1991.
- e. Annual salary in dollars of the governor of California in 1993.
- f. Population of San Diego in 1990.
- g. Year in which Roger Williams founded Providence, Rhode Island.
- h. Height of Mt. Kilimanjaro in feet.
- i. Length of the Brooklyn Bridge in feet.
- j. Number of deaths due to automobile accidents in the United States in 1992.

16.2 Chris considers four used cars before buying the one with maximum expected utility. Pat considers ten cars and does the same. All other things being equal, which one is more likely to have the better car? Which is more likely to be disappointed with their car's quality? By how much (in terms of standard deviations of expected quality)?

16.3 In 1713, Nicolas Bernoulli stated a puzzle, now called the St. Petersburg paradox, which works as follows. You have the opportunity to play a game in which a fair coin is tossed repeatedly until it comes up heads. If the first heads appears on the n th toss, you win 2^n dollars.

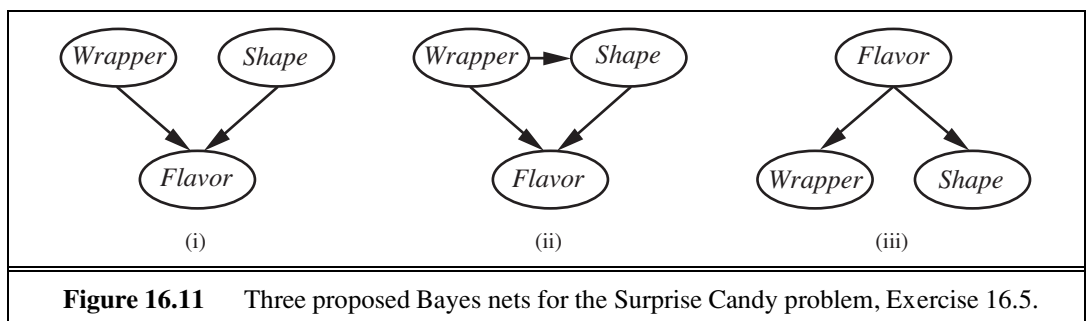
- Show that the expected monetary value of this game is infinite.
- How much would you, personally, pay to play the game?
- Nicolas's cousin Daniel Bernoulli resolved the apparent paradox in 1738 by suggesting that the utility of money is measured on a logarithmic scale (i.e., $U(S_n) = a \log_2 n + b$, where S_n is the state of having $\$n$). What is the expected utility of the game under this assumption?
- What is the maximum amount that it would be rational to pay to play the game, assuming that one's initial wealth is $\$k$?



16.4 Write a computer program to automate the process in Exercise 16.9. Try your program out on several people of different net worth and political outlook. Comment on the consistency of your results, both for an individual and across individuals.

16.5 The Surprise Candy Company makes candy in two flavors: 70% are strawberry flavor and 30% are anchovy flavor. Each new piece of candy starts out with a round shape; as it moves along the production line, a machine randomly selects a certain percentage to be trimmed into a square; then, each piece is wrapped in a wrapper whose color is chosen randomly to be red or brown. 80% of the strawberry candies are round and 80% have a red wrapper, while 90% of the anchovy candies are square and 90% have a brown wrapper. All candies are sold individually in sealed, identical, black boxes.

Now you, the customer, have just bought a Surprise candy at the store but have not yet opened the box. Consider the three Bayes nets in Figure 16.11.



- Which network(s) can correctly represent $\mathbf{P}(\text{Flavor}, \text{Wrapper}, \text{Shape})$?
- Which network is the best representation for this problem?

- c. Does network (i) assert that $\mathbf{P}(\text{Wrapper}|\text{Shape}) = \mathbf{P}(\text{Wrapper})$?
- d. What is the probability that your candy has a red wrapper?
- e. In the box is a round candy with a red wrapper. What is the probability that its flavor is strawberry?
- f. A unwrapped strawberry candy is worth s on the open market and an unwrapped anchovy candy is worth a . Write an expression for the value of an unopened candy box.
- g. A new law prohibits trading of unwrapped candies, but it is still legal to trade wrapped candies (out of the box). Is an unopened candy box now worth more than less than, or the same as before?

16.6 Prove that the judgments $B \succ A$ and $C \succ D$ in the Allais paradox (page 620) violate the axiom of substitutability.

16.7 Consider the Allais paradox described on page 620: an agent who prefers B over A (taking the sure thing), and C over D (taking the higher EMV) is not acting rationally, according to utility theory. Do you think this indicates a problem for the agent, a problem for the theory, or no problem at all? Explain.

16.8 Tickets to a lottery cost \$1. There are two possible prizes: a \$10 payoff with probability $1/50$, and a \$1,000,000 payoff with probability $1/2,000,000$. What is the expected monetary value of a lottery ticket? When (if ever) is it rational to buy a ticket? Be precise—show an equation involving utilities. You may assume current wealth of $\$k$ and that $U(S_k) = 0$. You may also assume that $U(S_{k+10}) = 10 \times U(S_{k+1})$, but you may not make any assumptions about $U(S_{k+1,000,000})$. Sociological studies show that people with lower income buy a disproportionate number of lottery tickets. Do you think this is because they are worse decision makers or because they have a different utility function? Consider the value of contemplating the possibility of winning the lottery versus the value of contemplating becoming an action hero while watching an adventure movie.

16.9 Assess your own utility for different incremental amounts of money by running a series of preference tests between some definite amount M_1 and a lottery $[p, M_2; (1-p), 0]$. Choose different values of M_1 and M_2 , and vary p until you are indifferent between the two choices. Plot the resulting utility function.

16.10 How much is a micromort worth to you? Devise a protocol to determine this. Ask questions based both on paying to avoid risk and being paid to accept risk.

16.11 Let continuous variables X_1, \dots, X_k be independently distributed according to the same probability density function $f(x)$. Prove that the density function for $\max\{X_1, \dots, X_k\}$ is given by $kf(x)(F(x))^{k-1}$, where F is the cumulative distribution for f .

16.12 Economists often make use of an exponential utility function for money: $U(x) = -e^{x/R}$, where R is a positive constant representing an individual's risk tolerance. Risk tolerance reflects how likely an individual is to accept a lottery with a particular expected monetary value (EMV) versus some certain payoff. As R (which is measured in the same units as x) becomes larger, the individual becomes less risk-averse.

- a. Assume Mary has an exponential utility function with $R = \$500$. Mary is given the choice between receiving \$500 with certainty (probability 1) or participating in a lottery which has a 60% probability of winning \$5000 and a 40% probability of winning nothing. Assuming Mary acts rationally, which option would she choose? Show how you derived your answer.
- b. Consider the choice between receiving \$100 with certainty (probability 1) or participating in a lottery which has a 50% probability of winning \$500 and a 50% probability of winning nothing. Approximate the value of R (to 3 significant digits) in an exponential utility function that would cause an individual to be indifferent to these two alternatives. (You might find it helpful to write a short program to help you solve this problem.)

16.13 Repeat Exercise 16.16, using the action-utility representation shown in Figure 16.7.

16.14 For either of the airport-siting diagrams from Exercises 16.16 and 16.13, to which conditional probability table entry is the utility most sensitive, given the available evidence?

16.15 Consider a student who has the choice to buy or not buy a textbook for a course. We'll model this as a decision problem with one Boolean decision node, B , indicating whether the agent chooses to buy the book, and two Boolean chance nodes, M , indicating whether the student has mastered the material in the book, and P , indicating whether the student passes the course. Of course, there is also a utility node, U . A certain student, Sam, has an additive utility function: 0 for not buying the book and -\$100 for buying it; and \$2000 for passing the course and 0 for not passing. Sam's conditional probability estimates are as follows:

$$\begin{aligned} P(p|b, m) &= 0.9 & P(m|b) &= 0.9 \\ P(p|b, \neg m) &= 0.5 & P(m|\neg b) &= 0.7 \\ P(p|\neg b, m) &= 0.8 \\ P(p|\neg b, \neg m) &= 0.3 \end{aligned}$$

You might think that P would be independent of B given M . But this course has an open-book final—so having the book helps.

- a. Draw the decision network for this problem.
- b. Compute the expected utility of buying the book and of not buying it.
- c. What should Sam do?



16.16 This exercise completes the analysis of the airport-siting problem in Figure 16.6.

- a. Provide reasonable variable domains, probabilities, and utilities for the network, assuming that there are three possible sites.
- b. Solve the decision problem.
- c. What happens if changes in technology mean that each aircraft generates half the noise?
- d. What if noise avoidance becomes three times more important?
- e. Calculate the VPI for *AirTraffic*, *Litigation*, and *Construction* in your model.

16.17 (Adapted from Pearl (1988).) A used-car buyer can decide to carry out various tests with various costs (e.g., kick the tires, take the car to a qualified mechanic) and then, depending on the outcome of the tests, decide which car to buy. We will assume that the buyer is deciding whether to buy car c_1 , that there is time to carry out at most one test, and that t_1 is the test of c_1 and costs \$50.

A car can be in good shape (quality q^+) or bad shape (quality q^-), and the tests might help indicate what shape the car is in. Car c_1 costs \$1,500, and its market value is \$2,000 if it is in good shape; if not, \$700 in repairs will be needed to make it in good shape. The buyer's estimate is that c_1 has a 70% chance of being in good shape.

- a. Draw the decision network that represents this problem.
- b. Calculate the expected net gain from buying c_1 , given no test.
- c. Tests can be described by the probability that the car will pass or fail the test given that the car is in good or bad shape. We have the following information:
 $P(\text{pass}(c_1, t_1) | q^+(c_1)) = 0.8$
 $P(\text{pass}(c_1, t_1) | q^-(c_1)) = 0.35$
 Use Bayes' theorem to calculate the probability that the car will pass (or fail) its test and hence the probability that it is in good (or bad) shape given each possible test outcome.
- d. Calculate the optimal decisions given either a pass or a fail, and their expected utilities.
- e. Calculate the value of information of the test, and derive an optimal conditional plan for the buyer.

16.18 Recall the definition of *value of information* in Section 16.6.

- a. Prove that the value of information is nonnegative and order independent.
- b. Explain why it is that some people would prefer not to get some information—for example, not wanting to know the sex of their baby when an ultrasound is done.
- c. A function f on sets is **submodular** if, for any element x and any sets A and B such that $A \subseteq B$, adding x to A gives a greater increase in f than adding x to B :

$$A \subseteq B \Rightarrow (f(A \cup \{x\}) - f(A)) \geq (f(B \cup \{x\}) - f(B)) .$$

Submodularity captures the intuitive notion of *diminishing returns*. Is the value of information, viewed as a function f on sets of possible observations, submodular? Prove this or find a counterexample.

The answers to Exercise 16.1 (where M stands for million): First set: 3M, 1.6M, 1541, 41M, 4768, 221, 649M, 295M, 132, 25,546. Second set: 1917, 155M, 4,500M, 11M, 120,000, 1.1M, 1636, 19,340, 1,595, 41,710.

17 MAKING COMPLEX DECISIONS

In which we examine methods for deciding what to do today, given that we may decide again tomorrow.

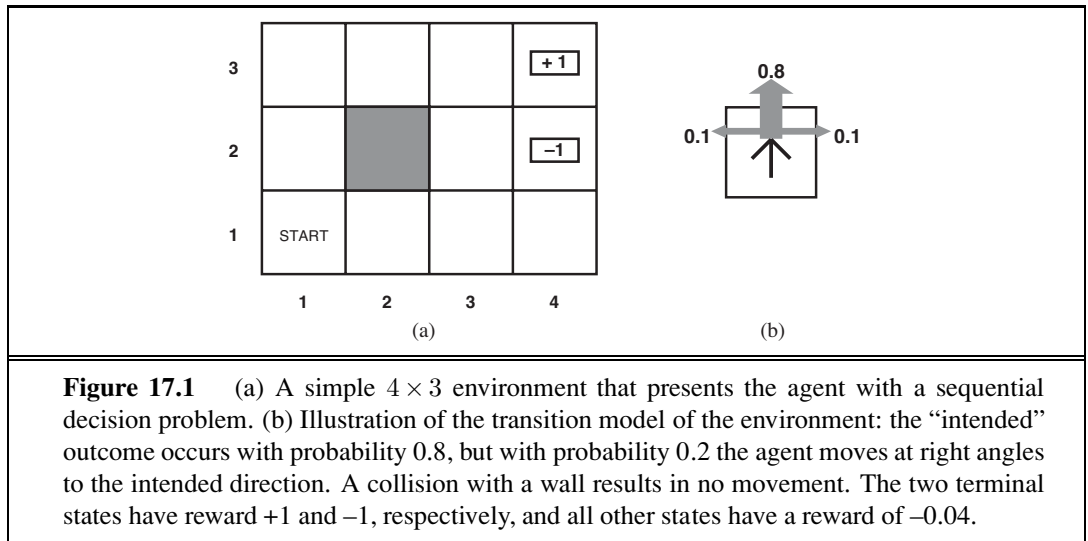
SEQUENTIAL
DECISION PROBLEM

In this chapter, we address the computational issues involved in making decisions in a stochastic environment. Whereas Chapter 16 was concerned with one-shot or episodic decision problems, in which the utility of each action's outcome was well known, we are concerned here with **sequential decision problems**, in which the agent's utility depends on a sequence of decisions. Sequential decision problems incorporate utilities, uncertainty, and sensing, and include search and planning problems as special cases. Section 17.1 explains how sequential decision problems are defined, and Sections 17.2 and 17.3 explain how they can be solved to produce optimal behavior that balances the risks and rewards of acting in an uncertain environment. Section 17.4 extends these ideas to the case of partially observable environments, and Section 17.4.3 develops a complete design for decision-theoretic agents in partially observable environments, combining dynamic Bayesian networks from Chapter 15 with decision networks from Chapter 16.

The second part of the chapter covers environments with multiple agents. In such environments, the notion of optimal behavior is complicated by the interactions among the agents. Section 17.5 introduces the main ideas of **game theory**, including the idea that rational agents might need to behave randomly. Section 17.6 looks at how multiagent systems can be designed so that multiple agents can achieve a common goal.

17.1 SEQUENTIAL DECISION PROBLEMS

Suppose that an agent is situated in the 4×3 environment shown in Figure 17.1(a). Beginning in the start state, it must choose an action at each time step. The interaction with the environment terminates when the agent reaches one of the goal states, marked +1 or -1. Just as for search problems, the actions available to the agent in each state are given by $\text{ACTIONS}(s)$, sometimes abbreviated to $A(s)$; in the 4×3 environment, the actions in every state are *Up*, *Down*, *Left*, and *Right*. We assume for now that the environment is **fully observable**, so that the agent always knows where it is.



If the environment were deterministic, a solution would be easy: *[Up, Up, Right, Right, Right]*. Unfortunately, the environment won’t always go along with this solution, because the actions are unreliable. The particular model of stochastic motion that we adopt is illustrated in Figure 17.1(b). Each action achieves the intended effect with probability 0.8, but the rest of the time, the action moves the agent at right angles to the intended direction. Furthermore, if the agent bumps into a wall, it stays in the same square. For example, from the start square (1,1), the action *Up* moves the agent to (1,2) with probability 0.8, but with probability 0.1, it moves right to (2,1), and with probability 0.1, it moves left, bumps into the wall, and stays in (1,1). In such an environment, the sequence *[Up, Up, Right, Right, Right]* goes up around the barrier and reaches the goal state at (4,3) with probability $0.8^5 = 0.32768$. There is also a small chance of accidentally reaching the goal by going the other way around with probability $0.1^4 \times 0.8$, for a grand total of 0.32776. (See also Exercise 17.1.)

As in Chapter 3, the **transition model** (or just “model,” whenever no confusion can arise) describes the outcome of each action in each state. Here, the outcome is stochastic, so we write $P(s' | s, a)$ to denote the probability of reaching state s' if action a is done in state s . We will assume that transitions are **Markovian** in the sense of Chapter 15, that is, the probability of reaching s' from s depends only on s and not on the history of earlier states. For now, you can think of $P(s' | s, a)$ as a big three-dimensional table containing probabilities. Later, in Section 17.4.3, we will see that the transition model can be represented as a **dynamic Bayesian network**, just as in Chapter 15.

To complete the definition of the task environment, we must specify the utility function for the agent. Because the decision problem is sequential, the utility function will depend on a sequence of states—an **environment history**—rather than on a single state. Later in this section, we investigate how such utility functions can be specified in general; for now, we simply stipulate that in each state s , the agent receives a **reward** $R(s)$, which may be positive or negative, but must be bounded. For our particular example, the reward is -0.04 in all states except the terminal states (which have rewards +1 and -1). The utility of an

environment history is just (for now) the *sum* of the rewards received. For example, if the agent reaches the +1 state after 10 steps, its total utility will be 0.6. The negative reward of -0.04 gives the agent an incentive to reach (4,3) quickly, so our environment is a stochastic generalization of the search problems of Chapter 3. Another way of saying this is that the agent does not enjoy living in this environment and so wants to leave as soon as possible.

MARKOV DECISION
PROCESS

To sum up: a sequential decision problem for a fully observable, stochastic environment with a Markovian transition model and additive rewards is called a **Markov decision process**, or **MDP**, and consists of a set of states (with an initial state s_0); a set $\text{ACTIONS}(s)$ of actions in each state; a transition model $P(s' | s, a)$; and a reward function $R(s)$.¹

POLICY

The next question is, what does a solution to the problem look like? We have seen that any fixed action sequence won't solve the problem, because the agent might end up in a state other than the goal. Therefore, a solution must specify what the agent should do for *any* state that the agent might reach. A solution of this kind is called a **policy**. It is traditional to denote a policy by π , and $\pi(s)$ is the action recommended by the policy π for state s . If the agent has a complete policy, then no matter what the outcome of any action, the agent will always know what to do next.

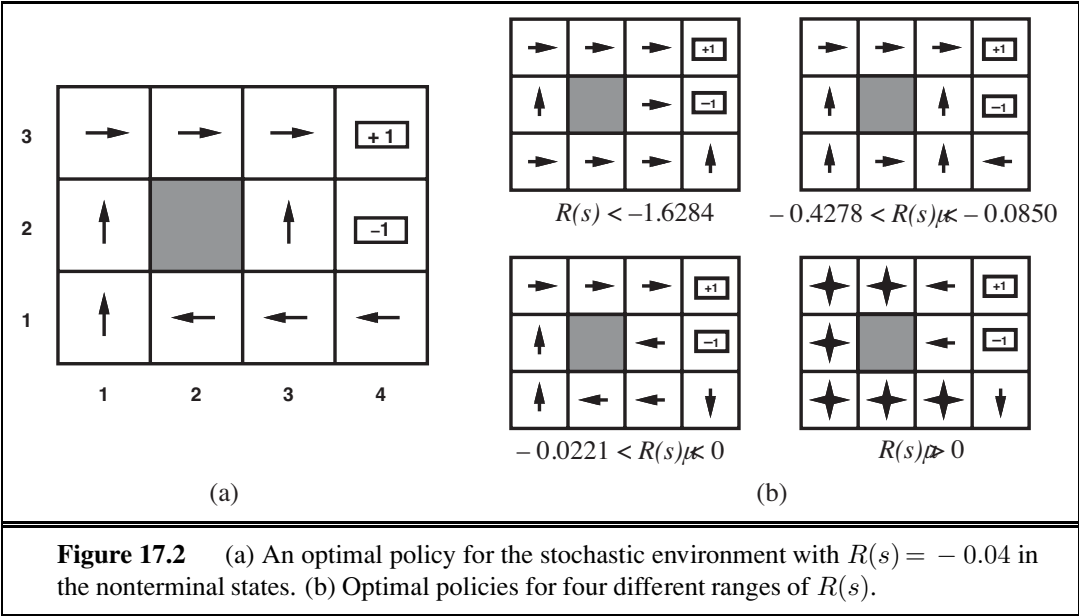
OPTIMAL POLICY

Each time a given policy is executed starting from the initial state, the stochastic nature of the environment may lead to a different environment history. The quality of a policy is therefore measured by the *expected* utility of the possible environment histories generated by that policy. An **optimal policy** is a policy that yields the highest expected utility. We use π^* to denote an optimal policy. Given π^* , the agent decides what to do by consulting its current percept, which tells it the current state s , and then executing the action $\pi^*(s)$. A policy represents the agent function explicitly and is therefore a description of a simple reflex agent, computed from the information used for a utility-based agent.

An optimal policy for the world of Figure 17.1 is shown in Figure 17.2(a). Notice that, because the cost of taking a step is fairly small compared with the penalty for ending up in (4,2) by accident, the optimal policy for the state (3,1) is conservative. The policy recommends taking the long way round, rather than taking the shortcut and thereby risking entering (4,2).

The balance of risk and reward changes depending on the value of $R(s)$ for the nonterminal states. Figure 17.2(b) shows optimal policies for four different ranges of $R(s)$. When $R(s) \leq -1.6284$, life is so painful that the agent heads straight for the nearest exit, even if the exit is worth -1 . When $-0.4278 \leq R(s) \leq -0.0850$, life is quite unpleasant; the agent takes the shortest route to the +1 state and is willing to risk falling into the -1 state by accident. In particular, the agent takes the shortcut from (3,1). When life is only slightly dreary ($-0.0221 < R(s) < 0$), the optimal policy takes *no risks at all*. In (4,1) and (3,2), the agent heads directly away from the -1 state so that it cannot fall in by accident, even though this means banging its head against the wall quite a few times. Finally, if $R(s) > 0$, then life is positively enjoyable and the agent avoids *both* exits. As long as the actions in (4,1), (3,2),

¹ Some definitions of MDPs allow the reward to depend on the action and outcome too, so the reward function is $R(s, a, s')$. This simplifies the description of some environments but does not change the problem in any fundamental way, as shown in Exercise 17.4.



and (3,3) are as shown, every policy is optimal, and the agent obtains infinite total reward because it never enters a terminal state. Surprisingly, it turns out that there are six other optimal policies for various ranges of $R(s)$; Exercise 17.5 asks you to find them.

The careful balancing of risk and reward is a characteristic of MDPs that does not arise in deterministic search problems; moreover, it is a characteristic of many real-world decision problems. For this reason, MDPs have been studied in several fields, including AI, operations research, economics, and control theory. Dozens of algorithms have been proposed for calculating optimal policies. In sections 17.2 and 17.3 we describe two of the most important algorithm families. First, however, we must complete our investigation of utilities and policies for sequential decision problems.

17.1.1 Utilities over time

In the MDP example in Figure 17.1, the performance of the agent was measured by a sum of rewards for the states visited. This choice of performance measure is not arbitrary, but it is not the only possibility for the utility function on environment histories, which we write as $U_h([s_0, s_1, \dots, s_n])$. Our analysis draws on **multiattribute utility theory** (Section 16.4) and is somewhat technical; the impatient reader may wish to skip to the next section.

The first question to answer is whether there is a **finite horizon** or an **infinite horizon** for decision making. A finite horizon means that there is a *fixed* time N after which nothing matters—the game is over, so to speak. Thus, $U_h([s_0, s_1, \dots, s_{N+k}]) = U_h([s_0, s_1, \dots, s_N])$ for all $k > 0$. For example, suppose an agent starts at (3,1) in the 4×3 world of Figure 17.1, and suppose that $N = 3$. Then, to have any chance of reaching the +1 state, the agent must head directly for it, and the optimal action is to go *Up*. On the other hand, if $N = 100$, then there is plenty of time to take the safe route by going *Left*. So, with a finite horizon,

FINITE HORIZON
INFINITE HORIZON



NONSTATIONARY
POLICY

STATIONARY POLICY

the optimal action in a given state could change over time. We say that the optimal policy for a finite horizon is **nonstationary**. With no fixed time limit, on the other hand, there is no reason to behave differently in the same state at different times. Hence, the optimal action depends only on the current state, and the optimal policy is **stationary**. Policies for the infinite-horizon case are therefore simpler than those for the finite-horizon case, and we deal mainly with the infinite-horizon case in this chapter. (We will see later that for partially observable environments, the infinite-horizon case is not so simple.) Note that “infinite horizon” does not necessarily mean that all state sequences are infinite; it just means that there is no fixed deadline. In particular, there can be finite state sequences in an infinite-horizon MDP containing a terminal state.

STATIONARY
PREFERENCE

The next question we must decide is how to calculate the utility of state sequences. In the terminology of multiattribute utility theory, each state s_i can be viewed as an **attribute** of the state sequence $[s_0, s_1, s_2, \dots]$. To obtain a simple expression in terms of the attributes, we will need to make some sort of preference-independence assumption. The most natural assumption is that the agent’s preferences between state sequences are **stationary**. Stationarity for preferences means the following: if two state sequences $[s_0, s_1, s_2, \dots]$ and $[s'_0, s'_1, s'_2, \dots]$ begin with the same state (i.e., $s_0 = s'_0$), then the two sequences should be preference-ordered the same way as the sequences $[s_1, s_2, \dots]$ and $[s'_1, s'_2, \dots]$. In English, this means that if you prefer one future to another starting tomorrow, then you should still prefer that future if it were to start today instead. Stationarity is a fairly innocuous-looking assumption with very strong consequences: it turns out that under stationarity there are just two coherent ways to assign utilities to sequences:

ADDITIVE REWARD

1. **Additive rewards:** The utility of a state sequence is

$$U_h([s_0, s_1, s_2, \dots]) = R(s_0) + R(s_1) + R(s_2) + \dots$$

The 4×3 world in Figure 17.1 uses additive rewards. Notice that additivity was used implicitly in our use of path cost functions in heuristic search algorithms (Chapter 3).

DISCOUNTED
REWARD

2. **Discounted rewards:** The utility of a state sequence is

$$U_h([s_0, s_1, s_2, \dots]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \dots,$$

DISCOUNT FACTOR

where the **discount factor** γ is a number between 0 and 1. The discount factor describes the preference of an agent for current rewards over future rewards. When γ is close to 0, rewards in the distant future are viewed as insignificant. When γ is 1, discounted rewards are exactly equivalent to additive rewards, so additive rewards are a special case of discounted rewards. Discounting appears to be a good model of both animal and human preferences over time. A discount factor of γ is equivalent to an interest rate of $(1/\gamma) - 1$.

For reasons that will shortly become clear, we assume discounted rewards in the remainder of the chapter, although sometimes we allow $\gamma = 1$.

Lurking beneath our choice of infinite horizons is a problem: if the environment does not contain a terminal state, or if the agent never reaches one, then all environment histories will be infinitely long, and utilities with additive, undiscounted rewards will generally be

infinite. While we can agree that $+\infty$ is better than $-\infty$, comparing two state sequences with $+\infty$ utility is more difficult. There are three solutions, two of which we have seen already:

1. With discounted rewards, the utility of an infinite sequence is *finite*. In fact, if $\gamma < 1$ and rewards are bounded by $\pm R_{\max}$, we have

$$U_h([s_0, s_1, s_2, \dots]) = \sum_{t=0}^{\infty} \gamma^t R(s_t) \leq \sum_{t=0}^{\infty} \gamma^t R_{\max} = R_{\max}/(1 - \gamma), \quad (17.1)$$

using the standard formula for the sum of an infinite geometric series.

2. If the environment contains terminal states *and if the agent is guaranteed to get to one eventually*, then we will never need to compare infinite sequences. A policy that is guaranteed to reach a terminal state is called a **proper policy**. With proper policies, we can use $\gamma = 1$ (i.e., additive rewards). The first three policies shown in Figure 17.2(b) are proper, but the fourth is improper. It gains infinite total reward by staying away from the terminal states when the reward for the nonterminal states is positive. The existence of improper policies can cause the standard algorithms for solving MDPs to fail with additive rewards, and so provides a good reason for using discounted rewards.

PROPER POLICY

AVERAGE REWARD

3. Infinite sequences can be compared in terms of the **average reward** obtained per time step. Suppose that square (1,1) in the 4×3 world has a reward of 0.1 while the other nonterminal states have a reward of 0.01. Then a policy that does its best to stay in (1,1) will have higher average reward than one that stays elsewhere. Average reward is a useful criterion for some problems, but the analysis of average-reward algorithms is beyond the scope of this book.

In sum, discounted rewards present the fewest difficulties in evaluating state sequences.

17.1.2 Optimal policies and the utilities of states

Having decided that the utility of a given state sequence is the sum of discounted rewards obtained during the sequence, we can compare policies by comparing the *expected* utilities obtained when executing them. We assume the agent is in some initial state s and define S_t (a random variable) to be the state the agent reaches at time t when executing a particular policy π . (Obviously, $S_0 = s$, the state the agent is in now.) The probability distribution over state sequences S_1, S_2, \dots , is determined by the initial state s , the policy π , and the transition model for the environment.

The expected utility obtained by executing π starting in s is given by

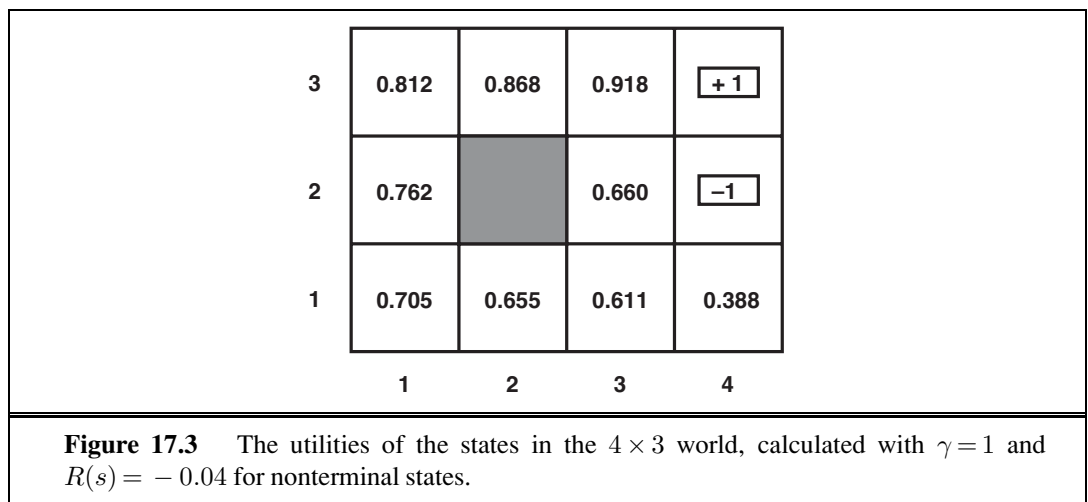
$$U^\pi(s) = E \left[\sum_{t=0}^{\infty} \gamma^t R(S_t) \right], \quad (17.2)$$

where the expectation is with respect to the probability distribution over state sequences determined by s and π . Now, out of all the policies the agent could choose to execute starting in s , one (or more) will have higher expected utilities than all the others. We'll use π_s^* to denote one of these policies:

$$\pi_s^* = \operatorname{argmax}_{\pi} U^\pi(s). \quad (17.3)$$

Remember that π_s^* is a policy, so it recommends an action for every state; its connection with s in particular is that it's an optimal policy when s is the starting state. A remarkable consequence of using discounted utilities with infinite horizons is that the optimal policy is *independent* of the starting state. (Of course, the *action sequence* won't be independent; remember that a policy is a function specifying an action for each state.) This fact seems intuitively obvious: if policy π_a^* is optimal starting in a and policy π_b^* is optimal starting in b , then, when they reach a third state c , there's no good reason for them to disagree with each other, or with π_c^* , about what to do next.² So we can simply write π^* for an optimal policy.

Given this definition, the true utility of a state is just $U^{\pi^*}(s)$ —that is, the expected sum of discounted rewards if the agent executes an optimal policy. We write this as $U(s)$, matching the notation used in Chapter 16 for the utility of an outcome. Notice that $U(s)$ and $R(s)$ are quite different quantities; $R(s)$ is the “short term” reward for being in s , whereas $U(s)$ is the “long term” total reward from s onward. Figure 17.3 shows the utilities for the 4×3 world. Notice that the utilities are higher for states closer to the +1 exit, because fewer steps are required to reach the exit.



The utility function $U(s)$ allows the agent to select actions by using the principle of maximum expected utility from Chapter 16—that is, choose the action that maximizes the expected utility of the subsequent state:

$$\pi^*(s) = \operatorname{argmax}_{a \in A(s)} \sum_{s'} P(s' | s, a) U(s'). \quad (17.4)$$

The next two sections describe algorithms for finding optimal policies.

² Although this seems obvious, it does not hold for finite-horizon policies or for other ways of combining rewards over time. The proof follows directly from the uniqueness of the utility function on states, as shown in Section 17.2.

17.2 VALUE ITERATION

VALUE ITERATION

In this section, we present an algorithm, called **value iteration**, for calculating an optimal policy. The basic idea is to calculate the utility of each state and then use the state utilities to select an optimal action in each state.

17.2.1 The Bellman equation for utilities



Section 17.1.2 defined the utility of being in a state as the expected sum of discounted rewards from that point onwards. From this, it follows that there is a direct relationship between the utility of a state and the utility of its neighbors: *the utility of a state is the immediate reward for that state plus the expected discounted utility of the next state, assuming that the agent chooses the optimal action.* That is, the utility of a state is given by

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' | s, a) U(s'). \quad (17.5)$$

BELLMAN EQUATION

This is called the **Bellman equation**, after Richard Bellman (1957). The utilities of the states—defined by Equation (17.2) as the expected utility of subsequent state sequences—are solutions of the set of Bellman equations. In fact, they are the *unique* solutions, as we show in Section 17.2.3.

Let us look at one of the Bellman equations for the 4×3 world. The equation for the state (1,1) is

$$U(1,1) = -0.04 + \gamma \max \begin{cases} 0.8U(1,2) + 0.1U(2,1) + 0.1U(1,1), & (Up) \\ 0.9U(1,1) + 0.1U(1,2), & (Left) \\ 0.9U(1,1) + 0.1U(2,1), & (Down) \\ 0.8U(2,1) + 0.1U(1,2) + 0.1U(1,1) \end{cases} \quad (Right)$$

When we plug in the numbers from Figure 17.3, we find that *Up* is the best action.

17.2.2 The value iteration algorithm

The Bellman equation is the basis of the value iteration algorithm for solving MDPs. If there are n possible states, then there are n Bellman equations, one for each state. The n equations contain n unknowns—the utilities of the states. So we would like to solve these simultaneous equations to find the utilities. There is one problem: the equations are *nonlinear*, because the “max” operator is not a linear operator. Whereas systems of linear equations can be solved quickly using linear algebra techniques, systems of nonlinear equations are more problematic. One thing to try is an *iterative* approach. We start with arbitrary initial values for the utilities, calculate the right-hand side of the equation, and plug it into the left-hand side—thereby updating the utility of each state from the utilities of its neighbors. We repeat this until we reach an equilibrium. Let $U_i(s)$ be the utility value for state s at the i th iteration. The iteration step, called a **Bellman update**, looks like this:

BELLMAN UPDATE

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' | s, a) U_i(s'), \quad (17.6)$$

```

function VALUE-ITERATION( $mdp, \epsilon$ ) returns a utility function
  inputs:  $mdp$ , an MDP with states  $S$ , actions  $A(s)$ , transition model  $P(s' | s, a)$ ,
           rewards  $R(s)$ , discount  $\gamma$ 
            $\epsilon$ , the maximum error allowed in the utility of any state
  local variables:  $U, U'$ , vectors of utilities for states in  $S$ , initially zero
                      $\delta$ , the maximum change in the utility of any state in an iteration

  repeat
     $U \leftarrow U'; \delta \leftarrow 0$ 
    for each state  $s$  in  $S$  do
       $U'[s] \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' | s, a) U[s']$ 
      if  $|U'[s] - U[s]| > \delta$  then  $\delta \leftarrow |U'[s] - U[s]|$ 
  until  $\delta < \epsilon(1 - \gamma)/\gamma$ 
  return  $U$ 

```

Figure 17.4 The value iteration algorithm for calculating utilities of states. The termination condition is from Equation (17.8).

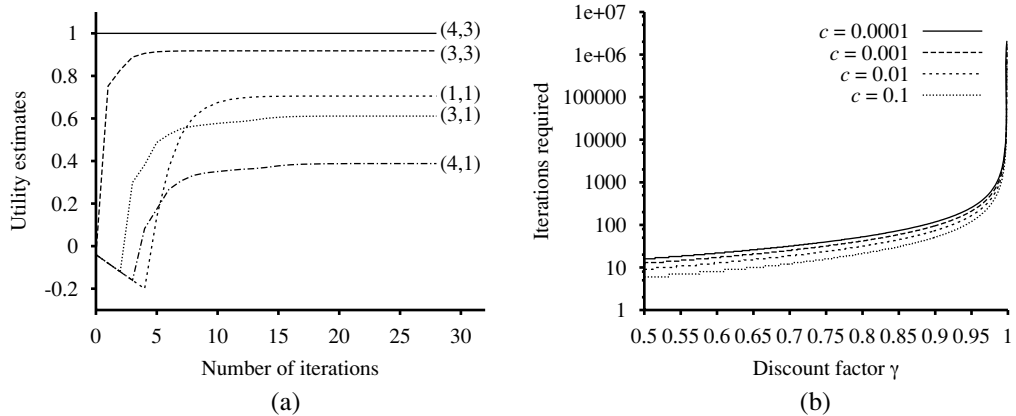


Figure 17.5 (a) Graph showing the evolution of the utilities of selected states using value iteration. (b) The number of value iterations k required to guarantee an error of at most $\epsilon = c \cdot R_{\max}$, for different values of c , as a function of the discount factor γ .

where the update is assumed to be applied simultaneously to all the states at each iteration. If we apply the Bellman update infinitely often, we are guaranteed to reach an equilibrium (see Section 17.2.3), in which case the final utility values must be solutions to the Bellman equations. In fact, they are also the *unique* solutions, and the corresponding policy (obtained using Equation (17.4)) is optimal. The algorithm, called VALUE-ITERATION, is shown in Figure 17.4.

We can apply value iteration to the 4×3 world in Figure 17.1(a). Starting with initial values of zero, the utilities evolve as shown in Figure 17.5(a). Notice how the states at differ-

ent distances from (4,3) accumulate negative reward until a path is found to (4,3), whereupon the utilities start to increase. We can think of the value iteration algorithm as *propagating information* through the state space by means of local updates.

17.2.3 Convergence of value iteration

We said that value iteration eventually converges to a unique set of solutions of the Bellman equations. In this section, we explain why this happens. We introduce some useful mathematical ideas along the way, and we obtain some methods for assessing the error in the utility function returned when the algorithm is terminated early; this is useful because it means that we don't have to run forever. This section is quite technical.

CONTRACTION

The basic concept used in showing that value iteration converges is the notion of a **contraction**. Roughly speaking, a contraction is a function of one argument that, when applied to two different inputs in turn, produces two output values that are “closer together,” by at least some constant factor, than the original inputs. For example, the function “divide by two” is a contraction, because, after we divide any two numbers by two, their difference is halved. Notice that the “divide by two” function has a fixed point, namely zero, that is unchanged by the application of the function. From this example, we can discern two important properties of contractions:

- A contraction has only one fixed point; if there were two fixed points they would not get closer together when the function was applied, so it would not be a contraction.
- When the function is applied to any argument, the value must get closer to the fixed point (because the fixed point does not move), so repeated application of a contraction always reaches the fixed point in the limit.

Now, suppose we view the Bellman update (Equation (17.6)) as an operator B that is applied simultaneously to update the utility of every state. Let U_i denote the vector of utilities for all the states at the i th iteration. Then the Bellman update equation can be written as

$$U_{i+1} \leftarrow B U_i .$$

MAX NORM

Next, we need a way to measure distances between utility vectors. We will use the **max norm**, which measures the “length” of a vector by the absolute value of its biggest component:

$$\|U\| = \max_s |U(s)| .$$

With this definition, the “distance” between two vectors, $\|U - U'\|$, is the maximum difference between any two corresponding elements. The main result of this section is the following: *Let U_i and U'_i be any two utility vectors. Then we have*

$$\|B U_i - B U'_i\| \leq \gamma \|U_i - U'_i\| . \quad (17.7)$$

That is, the Bellman update is a contraction by a factor of γ on the space of utility vectors. (Exercise 17.6 provides some guidance on proving this claim.) Hence, from the properties of contractions in general, it follows that value iteration always converges to a unique solution of the Bellman equations whenever $\gamma < 1$.



We can also use the contraction property to analyze the *rate* of convergence to a solution. In particular, we can replace U'_i in Equation (17.7) with the *true* utilities U , for which $BU = U$. Then we obtain the inequality

$$\|BU_i - U\| \leq \gamma \|U_i - U\|.$$

So, if we view $\|U_i - U\|$ as the *error* in the estimate U_i , we see that the error is reduced by a factor of at least γ on each iteration. This means that value iteration converges exponentially fast. We can calculate the number of iterations required to reach a specified error bound ϵ as follows: First, recall from Equation (17.1) that the utilities of all states are bounded by $\pm R_{\max}/(1 - \gamma)$. This means that the maximum initial error $\|U_0 - U\| \leq 2R_{\max}/(1 - \gamma)$. Suppose we run for N iterations to reach an error of at most ϵ . Then, because the error is reduced by at least γ each time, we require $\gamma^N \cdot 2R_{\max}/(1 - \gamma) \leq \epsilon$. Taking logs, we find

$$N = \lceil \log(2R_{\max}/\epsilon(1 - \gamma)) / \log(1/\gamma) \rceil$$

iterations suffice. Figure 17.5(b) shows how N varies with γ , for different values of the ratio ϵ/R_{\max} . The good news is that, because of the exponentially fast convergence, N does not depend much on the ratio ϵ/R_{\max} . The bad news is that N grows rapidly as γ becomes close to 1. We can get fast convergence if we make γ small, but this effectively gives the agent a short horizon and could miss the long-term effects of the agent's actions.

The error bound in the preceding paragraph gives some idea of the factors influencing the run time of the algorithm, but is sometimes overly conservative as a method of deciding when to stop the iteration. For the latter purpose, we can use a bound relating the error to the size of the Bellman update on any given iteration. From the contraction property (Equation (17.7)), it can be shown that if the update is small (i.e., no state's utility changes by much), then the error, compared with the true utility function, also is small. More precisely,

$$\text{if } \|U_{i+1} - U_i\| < \epsilon(1 - \gamma)/\gamma \text{ then } \|U_{i+1} - U\| < \epsilon. \quad (17.8)$$

This is the termination condition used in the VALUE-ITERATION algorithm of Figure 17.4.

So far, we have analyzed the error in the utility function returned by the value iteration algorithm. *What the agent really cares about, however, is how well it will do if it makes its decisions on the basis of this utility function.* Suppose that after i iterations of value iteration, the agent has an estimate U_i of the true utility U and obtains the MEU policy π_i based on one-step look-ahead using U_i (as in Equation (17.4)). Will the resulting behavior be nearly as good as the optimal behavior? This is a crucial question for any real agent, and it turns out that the answer is yes. $U^{\pi_i}(s)$ is the utility obtained if π_i is executed starting in s , and the **policy loss** $\|U^{\pi_i} - U\|$ is the most the agent can lose by executing π_i instead of the optimal policy π^* . The policy loss of π_i is connected to the error in U_i by the following inequality:

$$\text{if } \|U_i - U\| < \epsilon \text{ then } \|U^{\pi_i} - U\| < 2\epsilon\gamma/(1 - \gamma). \quad (17.9)$$

In practice, it often occurs that π_i becomes optimal long before U_i has converged. Figure 17.6 shows how the maximum error in U_i and the policy loss approach zero as the value iteration process proceeds for the 4×3 environment with $\gamma = 0.9$. The policy π_i is optimal when $i = 4$, even though the maximum error in U_i is still 0.46.

Now we have everything we need to use value iteration in practice. We know that it converges to the correct utilities, we can bound the error in the utility estimates if we



POLICY LOSS

stop after a finite number of iterations, and we can bound the policy loss that results from executing the corresponding MEU policy. As a final note, all of the results in this section depend on discounting with $\gamma < 1$. If $\gamma = 1$ and the environment contains terminal states, then a similar set of convergence results and error bounds can be derived whenever certain technical conditions are satisfied.

17.3 POLICY ITERATION

In the previous section, we observed that it is possible to get an optimal policy even when the utility function estimate is inaccurate. If one action is clearly better than all others, then the exact magnitude of the utilities on the states involved need not be precise. This insight suggests an alternative way to find optimal policies. The **policy iteration** algorithm alternates the following two steps, beginning from some initial policy π_0 :

POLICY ITERATION

POLICY EVALUATION

- **Policy evaluation:** given a policy π_i , calculate $U_i = U^{\pi_i}$, the utility of each state if π_i were to be executed.

POLICY IMPROVEMENT

- **Policy improvement:** Calculate a new MEU policy π_{i+1} , using one-step look-ahead based on U_i (as in Equation (17.4)).

The algorithm terminates when the policy improvement step yields no change in the utilities. At this point, we know that the utility function U_i is a fixed point of the Bellman update, so it is a solution to the Bellman equations, and π_i must be an optimal policy. Because there are only finitely many policies for a finite state space, and each iteration can be shown to yield a better policy, policy iteration must terminate. The algorithm is shown in Figure 17.7.

The policy improvement step is obviously straightforward, but how do we implement the POLICY-EVALUATION routine? It turns out that doing so is much simpler than solving the standard Bellman equations (which is what value iteration does), because the action in each state is fixed by the policy. At the i th iteration, the policy π_i specifies the action $\pi_i(s)$ in

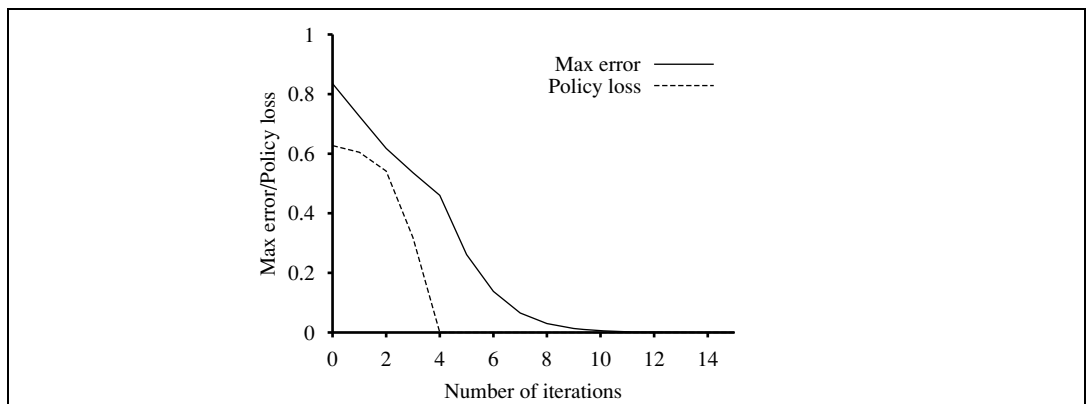


Figure 17.6 The maximum error $\|U_i - U\|$ of the utility estimates and the policy loss $\|U^{\pi_i} - U\|$, as a function of the number of iterations of value iteration.

state s . This means that we have a simplified version of the Bellman equation (17.5) relating the utility of s (under π_i) to the utilities of its neighbors:

$$U_i(s) = R(s) + \gamma \sum_{s'} P(s' | s, \pi_i(s)) U_i(s') . \quad (17.10)$$

For example, suppose π_i is the policy shown in Figure 17.2(a). Then we have $\pi_i(1, 1) = Up$, $\pi_i(1, 2) = Up$, and so on, and the simplified Bellman equations are

$$\begin{aligned} U_i(1, 1) &= -0.04 + 0.8U_i(1, 2) + 0.1U_i(1, 1) + 0.1U_i(2, 1) , \\ U_i(1, 2) &= -0.04 + 0.8U_i(1, 3) + 0.2U_i(1, 2) , \\ &\vdots \end{aligned}$$

The important point is that these equations are *linear*, because the “max” operator has been removed. For n states, we have n linear equations with n unknowns, which can be solved exactly in time $O(n^3)$ by standard linear algebra methods.

For small state spaces, policy evaluation using exact solution methods is often the most efficient approach. For large state spaces, $O(n^3)$ time might be prohibitive. Fortunately, it is not necessary to do *exact* policy evaluation. Instead, we can perform some number of simplified value iteration steps (simplified because the policy is fixed) to give a reasonably good approximation of the utilities. The simplified Bellman update for this process is

$$U_{i+1}(s) \leftarrow R(s) + \gamma \sum_{s'} P(s' | s, \pi_i(s)) U_i(s') ,$$

and this is repeated k times to produce the next utility estimate. The resulting algorithm is called **modified policy iteration**. It is often much more efficient than standard policy iteration or value iteration.

MODIFIED POLICY
ITERATION

```

function POLICY-ITERATION(mdp) returns a policy
  inputs: mdp, an MDP with states  $S$ , actions  $A(s)$ , transition model  $P(s' | s, a)$ 
  local variables:  $U$ , a vector of utilities for states in  $S$ , initially zero
                    $\pi$ , a policy vector indexed by state, initially random

  repeat
     $U \leftarrow \text{POLICY-EVALUATION}(\pi, U, \text{mdp})$ 
     $\text{unchanged?} \leftarrow \text{true}$ 
    for each state  $s$  in  $S$  do
      if  $\max_{a \in A(s)} \sum_{s'} P(s' | s, a) U[s'] > \sum_{s'} P(s' | s, \pi[s]) U[s']$  then do
         $\pi[s] \leftarrow \operatorname{argmax}_{a \in A(s)} \sum_{s'} P(s' | s, a) U[s']$ 
         $\text{unchanged?} \leftarrow \text{false}$ 
  until  $\text{unchanged?}$ 
  return  $\pi$ 

```

Figure 17.7 The policy iteration algorithm for calculating an optimal policy.

The algorithms we have described so far require updating the utility or policy for all states at once. It turns out that this is not strictly necessary. In fact, on each iteration, we can pick *any subset* of states and apply *either* kind of updating (policy improvement or simplified value iteration) to that subset. This very general algorithm is called **asynchronous policy iteration**. Given certain conditions on the initial policy and initial utility function, asynchronous policy iteration is guaranteed to converge to an optimal policy. The freedom to choose any states to work on means that we can design much more efficient heuristic algorithms—for example, algorithms that concentrate on updating the values of states that are likely to be reached by a good policy. This makes a lot of sense in real life: if one has no intention of throwing oneself off a cliff, one should not spend time worrying about the exact value of the resulting states.

17.4 PARTIALLY OBSERVABLE MDPs

The description of Markov decision processes in Section 17.1 assumed that the environment was **fully observable**. With this assumption, the agent always knows which state it is in. This, combined with the Markov assumption for the transition model, means that the optimal policy depends only on the current state. When the environment is only **partially observable**, the situation is, one might say, much less clear. The agent does not necessarily know which state it is in, so it cannot execute the action $\pi(s)$ recommended for that state. Furthermore, the utility of a state s and the optimal action in s depend not just on s , but also on *how much the agent knows* when it is in s . For these reasons, **partially observable MDPs** (or POMDPs—pronounced “pom-dee-pees”) are usually viewed as much more difficult than ordinary MDPs. We cannot avoid POMDPs, however, because the real world is one.

17.4.1 Definition of POMDPs

To get a handle on POMDPs, we must first define them properly. A POMDP has the same elements as an MDP—the transition model $P(s' | s, a)$, actions $A(s)$, and reward function $R(s)$ —but, like the partially observable search problems of Section 4.4, it also has a **sensor model** $P(e | s)$. Here, as in Chapter 15, the sensor model specifies the probability of perceiving evidence e in state s .³ For example, we can convert the 4×3 world of Figure 17.1 into a POMDP by adding a noisy or partial sensor instead of assuming that the agent knows its location exactly. Such a sensor might measure the *number of adjacent walls*, which happens to be 2 in all the nonterminal squares except for those in the third column, where the value is 1; a noisy version might give the wrong value with probability 0.1.

In Chapters 4 and 11, we studied nondeterministic and partially observable planning problems and identified the **belief state**—the set of actual states the agent might be in—as a key concept for describing and calculating solutions. In POMDPs, the belief state b becomes a *probability distribution* over all possible states, just as in Chapter 15. For example, the initial

³ As with the reward function for MDPs, the sensor model can also depend on the action and outcome state, but again this change is not fundamental.

belief state for the 4×3 POMDP could be the uniform distribution over the nine nonterminal states, i.e., $\langle \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, 0, 0 \rangle$. We write $b(s)$ for the probability assigned to the actual state s by belief state b . The agent can calculate its current belief state as the conditional probability distribution over the actual states given the sequence of percepts and actions so far. This is essentially the **filtering** task described in Chapter 15. The basic recursive filtering equation (15.5 on page 572) shows how to calculate the new belief state from the previous belief state and the new evidence. For POMDPs, we also have an action to consider, but the result is essentially the same. If $b(s)$ was the previous belief state, and the agent does action a and then perceives evidence e , then the new belief state is given by

$$b'(s') = \alpha P(e | s') \sum_s P(s' | s, a) b(s) ,$$

where α is a normalizing constant that makes the belief state sum to 1. By analogy with the update operator for filtering (page 572), we can write this as

$$b' = \text{FORWARD}(b, a, e) . \quad (17.11)$$

In the 4×3 POMDP, suppose the agent moves *Left* and its sensor reports 1 adjacent wall; then it's quite likely (although not guaranteed, because both the motion and the sensor are noisy) that the agent is now in (3,1). Exercise 17.13 asks you to calculate the exact probability values for the new belief state.



The fundamental insight required to understand POMDPs is this: *the optimal action depends only on the agent's current belief state*. That is, the optimal policy can be described by a mapping $\pi^*(b)$ from belief states to actions. It does *not* depend on the *actual* state the agent is in. This is a good thing, because the agent does not know its actual state; all it knows is the belief state. Hence, the decision cycle of a POMDP agent can be broken down into the following three steps:

1. Given the current belief state b , execute the action $a = \pi^*(b)$.
2. Receive percept e .
3. Set the current belief state to $\text{FORWARD}(b, a, e)$ and repeat.

Now we can think of POMDPs as requiring a search in belief-state space, just like the methods for sensorless and contingency problems in Chapter 4. The main difference is that the POMDP belief-state space is *continuous*, because a POMDP belief state is a probability distribution. For example, a belief state for the 4×3 world is a point in an 11-dimensional continuous space. An action changes the belief state, not just the physical state. Hence, the action is evaluated at least in part according to the information the agent acquires as a result. POMDPs therefore include the value of information (Section 16.6) as one component of the decision problem.

Let's look more carefully at the outcome of actions. In particular, let's calculate the probability that an agent in belief state b reaches belief state b' after executing action a . Now, if we knew the action *and the subsequent percept*, then Equation (17.11) would provide a *deterministic* update to the belief state: $b' = \text{FORWARD}(b, a, e)$. Of course, the subsequent percept is not yet known, so the agent might arrive in one of several possible belief states b' , depending on the percept that is received. The probability of perceiving e , given that a was

performed starting in belief state b , is given by summing over all the actual states s' that the agent might reach:

$$\begin{aligned} P(e|a, b) &= \sum_{s'} P(e|a, s', b) P(s'|a, b) \\ &= \sum_{s'} P(e | s') P(s'|a, b) \\ &= \sum_{s'} P(e | s') \sum_s P(s' | s, a) b(s) . \end{aligned}$$

Let us write the probability of reaching b' from b , given action a , as $P(b' | b, a)$. Then that gives us

$$\begin{aligned} P(b' | b, a) &= P(b'|a, b) = \sum_e P(b'|e, a, b) P(e|a, b) \\ &= \sum_e P(b'|e, a, b) \sum_{s'} P(e | s') \sum_s P(s' | s, a) b(s) , \end{aligned} \quad (17.12)$$

where $P(b'|e, a, b)$ is 1 if $b' = \text{FORWARD}(b, a, e)$ and 0 otherwise.

Equation (17.12) can be viewed as defining a transition model for the belief-state space. We can also define a reward function for belief states (i.e., the expected reward for the actual states the agent might be in):

$$\rho(b) = \sum_s b(s) R(s) .$$

Together, $P(b' | b, a)$ and $\rho(b)$ define an *observable* MDP on the space of belief states. Furthermore, it can be shown that an optimal policy for this MDP, $\pi^*(b)$, is also an optimal policy for the original POMDP. In other words, *solving a POMDP on a physical state space can be reduced to solving an MDP on the corresponding belief-state space*. This fact is perhaps less surprising if we remember that the belief state is always observable to the agent, by definition.

Notice that, although we have reduced POMDPs to MDPs, the MDP we obtain has a continuous (and usually high-dimensional) state space. None of the MDP algorithms described in Sections 17.2 and 17.3 applies directly to such MDPs. The next two subsections describe a value iteration algorithm designed specifically for POMDPs and an online decision-making algorithm, similar to those developed for games in Chapter 5.

17.4.2 Value iteration for POMDPs

Section 17.2 described a value iteration algorithm that computed one utility value for each state. With infinitely many belief states, we need to be more creative. Consider an optimal policy π^* and its application in a specific belief state b : the policy generates an action, then, for each subsequent percept, the belief state is updated and a new action is generated, and so on. For this specific b , therefore, the policy is exactly equivalent to a **conditional plan**, as defined in Chapter 4 for nondeterministic and partially observable problems. Instead of thinking about policies, let us think about conditional plans and how the expected utility of executing a fixed conditional plan varies with the initial belief state. We make two observations:



1. Let the utility of executing a *fixed* conditional plan p starting in physical state s be $\alpha_p(s)$. Then the expected utility of executing p in belief state b is just $\sum_s b(s)\alpha_p(s)$, or $b \cdot \alpha_p$ if we think of them both as vectors. Hence, the expected utility of a fixed conditional plan varies *linearly* with b ; that is, it corresponds to a hyperplane in belief space.
2. At any given belief state b , the optimal policy will choose to execute the conditional plan with highest expected utility; and the expected utility of b under the optimal policy is just the utility of that conditional plan:

$$U(b) = U^{\pi^*}(b) = \max_p b \cdot \alpha_p.$$

If the optimal policy π^* chooses to execute p starting at b , then it is reasonable to expect that it might choose to execute p in belief states that are very close to b ; in fact, if we bound the depth of the conditional plans, then there are only finitely many such plans and the continuous space of belief states will generally be divided into *regions*, each corresponding to a particular conditional plan that is optimal in that region.

From these two observations, we see that the utility function $U(b)$ on belief states, being the maximum of a collection of hyperplanes, will be *piecewise linear* and *convex*.

To illustrate this, we use a simple two-state world. The states are labeled 0 and 1, with $R(0) = 0$ and $R(1) = 1$. There are two actions: *Stay* stays put with probability 0.9 and *Go* switches to the other state with probability 0.9. For now we will assume the discount factor $\gamma = 1$. The sensor reports the correct state with probability 0.6. Obviously, the agent should *Stay* when it thinks it's in state 1 and *Go* when it thinks it's in state 0.

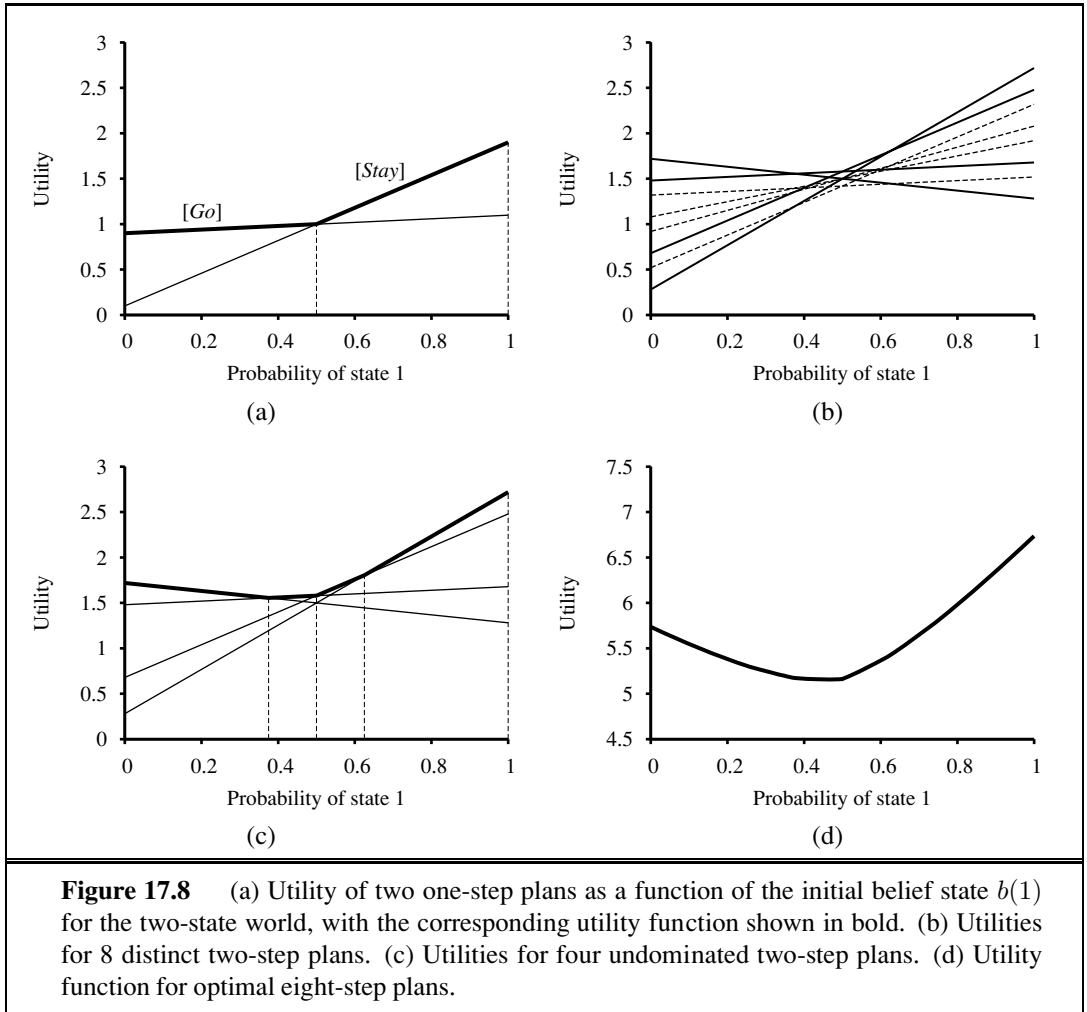
The advantage of a two-state world is that the belief space can be viewed as one-dimensional, because the two probabilities must sum to 1. In Figure 17.8(a), the x -axis represents the belief state, defined by $b(1)$, the probability of being in state 1. Now let us consider the one-step plans $[Stay]$ and $[Go]$, each of which receives the reward for the current state followed by the (discounted) reward for the state reached after the action:

$$\begin{aligned}\alpha_{[Stay]}(0) &= R(0) + \gamma(0.9R(0) + 0.1R(1)) = 0.1 \\ \alpha_{[Stay]}(1) &= R(1) + \gamma(0.9R(1) + 0.1R(0)) = 1.9 \\ \alpha_{[Go]}(0) &= R(0) + \gamma(0.9R(1) + 0.1R(0)) = 0.9 \\ \alpha_{[Go]}(1) &= R(1) + \gamma(0.9R(0) + 0.1R(1)) = 1.1\end{aligned}$$

The hyperplanes (lines, in this case) for $b \cdot \alpha_{[Stay]}$ and $b \cdot \alpha_{[Go]}$ are shown in Figure 17.8(a) and their maximum is shown in bold. The bold line therefore represents the utility function for the finite-horizon problem that allows just one action, and in each “piece” of the piecewise linear utility function the optimal action is the first action of the corresponding conditional plan. In this case, the optimal one-step policy is to *Stay* when $b(1) > 0.5$ and *Go* otherwise.

Once we have utilities $\alpha_p(s)$ for all the conditional plans p of depth 1 in each physical state s , we can compute the utilities for conditional plans of depth 2 by considering each possible first action, each possible subsequent percept, and then each way of choosing a depth-1 plan to execute for each percept:

```
[Stay; if Percept = 0 then Stay else Stay]
[Stay; if Percept = 0 then Stay else Go] ...
```



There are eight distinct depth-2 plans in all, and their utilities are shown in Figure 17.8(b). Notice that four of the plans, shown as dashed lines, are suboptimal across the entire belief space—we say these plans are **dominated**, and they need not be considered further. There are four undominated plans, each of which is optimal in a specific region, as shown in Figure 17.8(c). The regions partition the belief-state space.

We repeat the process for depth 3, and so on. In general, let p be a depth- d conditional plan whose initial action is a and whose depth- $d-1$ subplan for percept e is $p.e$; then

$$\alpha_p(s) = R(s) + \gamma \left(\sum_{s'} P(s' | s, a) \sum_e P(e | s') \alpha_{p.e}(s') \right). \quad (17.13)$$

This recursion naturally gives us a value iteration algorithm, which is sketched in Figure 17.9. The structure of the algorithm and its error analysis are similar to those of the basic value iteration algorithm in Figure 17.4 on page 653; the main difference is that instead of computing one utility number for each state, POMDP-VALUE-ITERATION maintains a collection of

```

function POMDP-VALUE-ITERATION(pomdp,  $\epsilon$ ) returns a utility function
  inputs: pomdp, a POMDP with states  $S$ , actions  $A(s)$ , transition model  $P(s' | s, a)$ ,
           sensor model  $P(e | s)$ , rewards  $R(s)$ , discount  $\gamma$ 
            $\epsilon$ , the maximum error allowed in the utility of any state
  local variables:  $U, U'$ , sets of plans  $p$  with associated utility vectors  $\alpha_p$ 

   $U' \leftarrow$  a set containing just the empty plan  $[\ ]$ , with  $\alpha_{[\ ]}(s) = R(s)$ 
  repeat
     $U \leftarrow U'$ 
     $U' \leftarrow$  the set of all plans consisting of an action and, for each possible next percept,
                a plan in  $U$  with utility vectors computed according to Equation (17.13)
     $U' \leftarrow \text{REMOVE-DOMINATED-PLANS}(U')$ 
  until MAX-DIFFERENCE( $U, U'$ )  $< \epsilon(1 - \gamma)/\gamma$ 
  return  $U$ 

```

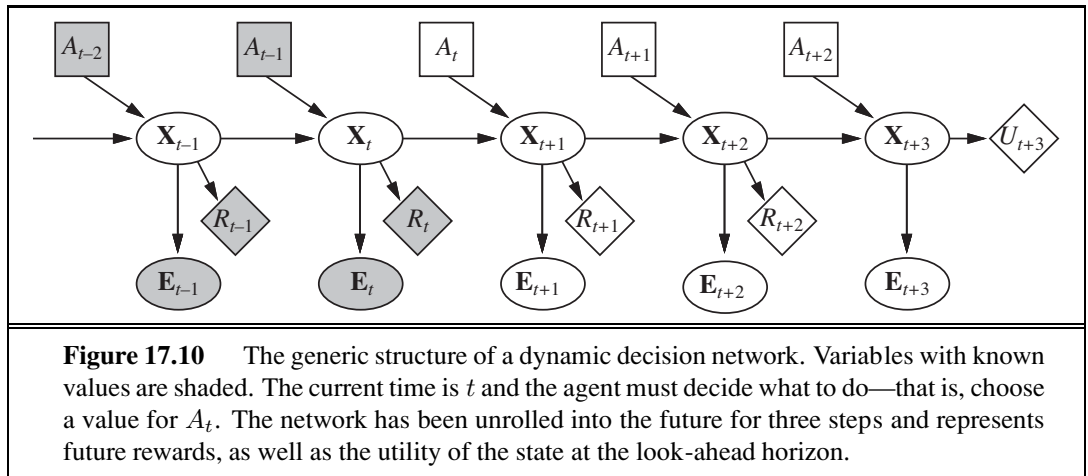
Figure 17.9 A high-level sketch of the value iteration algorithm for POMDPs. The REMOVE-DOMINATED-PLANS step and MAX-DIFFERENCE test are typically implemented as linear programs.

undominated plans with their utility hyperplanes. The algorithm's complexity depends primarily on how many plans get generated. Given $|A|$ actions and $|E|$ possible observations, it is easy to show that there are $|A|^{O(|E|^{d-1})}$ distinct depth- d plans. Even for the lowly two-state world with $d = 8$, the exact number is 2^{255} . The elimination of dominated plans is essential for reducing this doubly exponential growth: the number of undominated plans with $d = 8$ is just 144. The utility function for these 144 plans is shown in Figure 17.8(d).

Notice that even though state 0 has lower utility than state 1, the intermediate belief states have even lower utility because the agent lacks the information needed to choose a good action. This is why information has value in the sense defined in Section 16.6 and optimal policies in POMDPs often include information-gathering actions.

Given such a utility function, an executable policy can be extracted by looking at which hyperplane is optimal at any given belief state b and executing the first action of the corresponding plan. In Figure 17.8(d), the corresponding optimal policy is still the same as for depth-1 plans: *Stay* when $b(1) > 0.5$ and *Go* otherwise.

In practice, the value iteration algorithm in Figure 17.9 is hopelessly inefficient for larger problems—even the 4×3 POMDP is too hard. The main reason is that, given n conditional plans at level d , the algorithm constructs $|A| \cdot n^{|E|}$ conditional plans at level $d + 1$ before eliminating the dominated ones. Since the 1970s, when this algorithm was developed, there have been several advances including more efficient forms of value iteration and various kinds of policy iteration algorithms. Some of these are discussed in the notes at the end of the chapter. For general POMDPs, however, finding optimal policies is very difficult (PSPACE-hard, in fact—i.e., very hard indeed). Problems with a few dozen states are often infeasible. The next section describes a different, approximate method for solving POMDPs, one based on look-ahead search.



17.4.3 Online agents for POMDPs

In this section, we outline a simple approach to agent design for partially observable, stochastic environments. The basic elements of the design are already familiar:

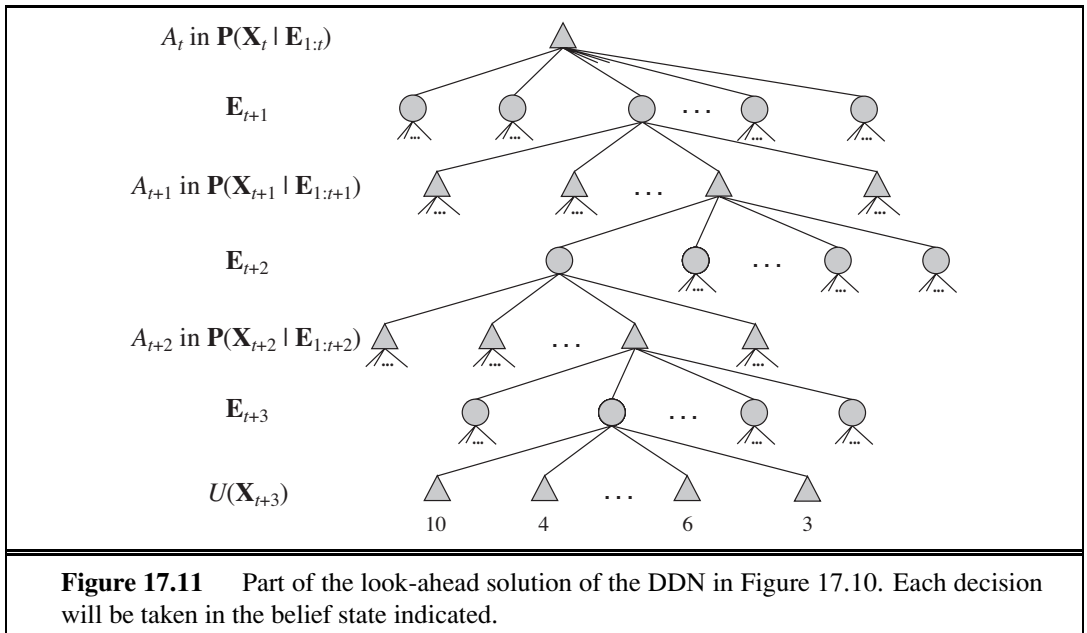
- The transition and sensor models are represented by a **dynamic Bayesian network** (DBN), as described in Chapter 15.
- The dynamic Bayesian network is extended with decision and utility nodes, as used in **decision networks** in Chapter 16. The resulting model is called a **dynamic decision network**, or DDN.
- A filtering algorithm is used to incorporate each new percept and action and to update the belief state representation.
- Decisions are made by projecting forward possible action sequences and choosing the best one.

DYNAMIC DECISION
NETWORK

DBNs are **factored representations** in the terminology of Chapter 2; they typically have an exponential complexity advantage over atomic representations and can model quite substantial real-world problems. The agent design is therefore a practical implementation of the **utility-based agent** sketched in Chapter 2.

In the DBN, the single state S_t becomes a set of state variables \mathbf{X}_t , and there may be multiple evidence variables \mathbf{E}_t . We will use A_t to refer to the action at time t , so the transition model becomes $\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{X}_t, A_t)$ and the sensor model becomes $\mathbf{P}(\mathbf{E}_t|\mathbf{X}_t)$. We will use R_t to refer to the reward received at time t and U_t to refer to the utility of the state at time t . (Both of these are random variables.) With this notation, a dynamic decision network looks like the one shown in Figure 17.10.

Dynamic decision networks can be used as inputs for any POMDP algorithm, including those for value and policy iteration methods. In this section, we focus on look-ahead methods that project action sequences forward from the current belief state in much the same way as do the game-playing algorithms of Chapter 5. The network in Figure 17.10 has been projected three steps into the future; the current and future decisions A and the future observations



\mathbf{E} and rewards R are all unknown. Notice that the network includes nodes for the *rewards* for \mathbf{X}_{t+1} and \mathbf{X}_{t+2} , but the *utility* for \mathbf{X}_{t+3} . This is because the agent must maximize the (discounted) sum of all future rewards, and $U(\mathbf{X}_{t+3})$ represents the reward for \mathbf{X}_{t+3} and all subsequent rewards. As in Chapter 5, we assume that U is available only in some approximate form: if exact utility values were available, look-ahead beyond depth 1 would be unnecessary.

Figure 17.11 shows part of the search tree corresponding to the three-step look-ahead DDN in Figure 17.10. Each of the triangular nodes is a belief state in which the agent makes a decision A_{t+i} for $i = 0, 1, 2, \dots$. The round (chance) nodes correspond to choices by the environment, namely, what evidence \mathbf{E}_{t+i} arrives. Notice that there are no chance nodes corresponding to the action outcomes; this is because the belief-state update for an action is deterministic regardless of the actual outcome.

The belief state at each triangular node can be computed by applying a filtering algorithm to the sequence of percepts and actions leading to it. In this way, the algorithm takes into account the fact that, for decision A_{t+i} , the agent *will* have available percepts $\mathbf{E}_{t+1}, \dots, \mathbf{E}_{t+i}$, even though at time t it does not know what those percepts will be. In this way, a decision-theoretic agent automatically takes into account the value of information and will execute information-gathering actions where appropriate.

A decision can be extracted from the search tree by backing up the utility values from the leaves, taking an average at the chance nodes and taking the maximum at the decision nodes. This is similar to the EXPECTIMINIMAX algorithm for game trees with chance nodes, except that (1) there can also be rewards at non-leaf states and (2) the decision nodes correspond to belief states rather than actual states. The time complexity of an exhaustive search to depth d is $O(|A|^d \cdot |\mathbf{E}|^d)$, where $|A|$ is the number of available actions and $|\mathbf{E}|$ is the number of possible percepts. (Notice that this is far less than the number of depth- d conditional

plans generated by value iteration.) For problems in which the discount factor γ is not too close to 1, a shallow search is often good enough to give near-optimal decisions. It is also possible to approximate the averaging step at the chance nodes, by sampling from the set of possible percepts instead of summing over all possible percepts. There are various other ways of finding good approximate solutions quickly, but we defer them to Chapter 21.

Decision-theoretic agents based on dynamic decision networks have a number of advantages compared with other, simpler agent designs presented in earlier chapters. In particular, they handle partially observable, uncertain environments and can easily revise their “plans” to handle unexpected evidence. With appropriate sensor models, they can handle sensor failure and can plan to gather information. They exhibit “graceful degradation” under time pressure and in complex environments, using various approximation techniques. So what is missing? One defect of our DDN-based algorithm is its reliance on forward search through state space, rather than using the hierarchical and other advanced planning techniques described in Chapter 11. There have been attempts to extend these techniques into the probabilistic domain, but so far they have proved to be inefficient. A second, related problem is the basically propositional nature of the DDN language. We would like to be able to extend some of the ideas for first-order probabilistic languages to the problem of decision making. Current research has shown that this extension is possible and has significant benefits, as discussed in the notes at the end of the chapter.

17.5 DECISIONS WITH MULTIPLE AGENTS: GAME THEORY

GAME THEORY

This chapter has concentrated on making decisions in uncertain environments. But what if the uncertainty is due to other agents and the decisions they make? And what if the decisions of those agents are in turn influenced by our decisions? We addressed this question once before, when we studied games in Chapter 5. There, however, we were primarily concerned with turn-taking games in fully observable environments, for which minimax search can be used to find optimal moves. In this section we study the aspects of **game theory** that analyze games with simultaneous moves and other sources of partial observability. (Game theorists use the terms **perfect information** and **imperfect information** rather than fully and partially observable.) Game theory can be used in at least two ways:

1. **Agent design:** Game theory can analyze the agent’s decisions and compute the expected utility for each decision (under the assumption that other agents are acting optimally according to game theory). For example, in the game **two-finger Morra**, two players, O and E , simultaneously display one or two fingers. Let the total number of fingers be f . If f is odd, O collects f dollars from E ; and if f is even, E collects f dollars from O . Game theory can determine the best strategy against a rational player and the expected return for each player.⁴

⁴ Morra is a recreational version of an **inspection game**. In such games, an inspector chooses a day to inspect a facility (such as a restaurant or a biological weapons plant), and the facility operator chooses a day to hide all the nasty stuff. The inspector wins if the days are different, and the facility operator wins if they are the same.

2. **Mechanism design:** When an environment is inhabited by many agents, it might be possible to define the rules of the environment (i.e., the game that the agents must play) so that the collective good of all agents is maximized when each agent adopts the game-theoretic solution that maximizes its own utility. For example, game theory can help design the protocols for a collection of Internet traffic routers so that each router has an incentive to act in such a way that global throughput is maximized. Mechanism design can also be used to construct intelligent **multiagent systems** that solve complex problems in a distributed fashion.

17.5.1 Single-move games

We start by considering a restricted set of games: ones where all players take action simultaneously and the result of the game is based on this single set of actions. (Actually, it is not crucial that the actions take place at exactly the same time; what matters is that no player has knowledge of the other players’ choices.) The restriction to a single move (and the very use of the word “game”) might make this seem trivial, but in fact, game theory is serious business. It is used in decision-making situations including the auctioning of oil drilling rights and wireless frequency spectrum rights, bankruptcy proceedings, product development and pricing decisions, and national defense—situations involving billions of dollars and hundreds of thousands of lives. A single-move game is defined by three components:

- PLAYER
- ACTION
- PAYOFF FUNCTION
- STRATEGIC FORM
- **Players** or agents who will be making decisions. Two-player games have received the most attention, although n -player games for $n > 2$ are also common. We give players capitalized names, like *Alice* and *Bob* or O and E .
 - **Actions** that the players can choose. We will give actions lowercase names, like *one* or *testify*. The players may or may not have the same set of actions available.
 - **A payoff function** that gives the utility to each player for each combination of actions by all the players. For single-move games the payoff function can be represented by a matrix, a representation known as the **strategic form** (also called **normal form**). The payoff matrix for two-finger Morra is as follows:

	O : one	O : two
E : one	$E = +2, O = -2$	$E = -3, O = +3$
E : two	$E = -3, O = +3$	$E = +4, O = -4$

For example, the lower-right corner shows that when player O chooses action *two* and E also chooses *two*, the payoff is +4 for E and −4 for O .

- STRATEGY
- PURE STRATEGY
- MIXED STRATEGY
- STRATEGY PROFILE
- OUTCOME
- Each player in a game must adopt and then execute a **strategy** (which is the name used in game theory for a *policy*). A **pure strategy** is a deterministic policy; for a single-move game, a pure strategy is just a single action. For many games an agent can do better with a **mixed strategy**, which is a randomized policy that selects actions according to a probability distribution. The mixed strategy that chooses action a with probability p and action b otherwise is written $[p:a; (1 - p):b]$. For example, a mixed strategy for two-finger Morra might be $[0.5:one; 0.5:two]$. A **strategy profile** is an assignment of a strategy to each player; given the strategy profile, the game’s **outcome** is a numeric value for each player.

SOLUTION

A **solution** to a game is a strategy profile in which each player adopts a rational strategy. We will see that the most important issue in game theory is to define what “rational” means when each agent chooses only part of the strategy profile that determines the outcome. It is important to realize that outcomes are actual results of playing a game, while solutions are theoretical constructs used to analyze a game. We will see that some games have a solution only in mixed strategies. But that does not mean that a player must literally be adopting a mixed strategy to be rational.

PRISONER'S
DILEMMA

Consider the following story: Two alleged burglars, Alice and Bob, are caught red-handed near the scene of a burglary and are interrogated separately. A prosecutor offers each a deal: if you testify against your partner as the leader of a burglary ring, you’ll go free for being the cooperative one, while your partner will serve 10 years in prison. However, if you both testify against each other, you’ll both get 5 years. Alice and Bob also know that if both refuse to testify they will serve only 1 year each for the lesser charge of possessing stolen property. Now Alice and Bob face the so-called **prisoner’s dilemma**: should they testify or refuse? Being rational agents, Alice and Bob each want to maximize their own expected utility. Let’s assume that Alice is callously unconcerned about her partner’s fate, so her utility decreases in proportion to the number of years she will spend in prison, regardless of what happens to Bob. Bob feels exactly the same way. To help reach a rational decision, they both construct the following payoff matrix:

	<i>Alice: testify</i>	<i>Alice: refuse</i>
<i>Bob: testify</i>	$A = -5, B = -5$	$A = -10, B = 0$
<i>Bob: refuse</i>	$A = 0, B = -10$	$A = -1, B = -1$

Alice analyzes the payoff matrix as follows: “Suppose Bob testifies. Then I get 5 years if I testify and 10 years if I don’t, so in that case testifying is better. On the other hand, if Bob refuses, then I get 0 years if I testify and 1 year if I refuse, so in that case as well testifying is better. So in either case, it’s better for me to testify, so that’s what I must do.”

DOMINANT
STRATEGY
STRONG
DOMINATION

Alice has discovered that *testify* is a **dominant strategy** for the game. We say that a strategy s for player p **strongly dominates** strategy s' if the outcome for s is better for p than the outcome for s' , for every choice of strategies by the other player(s). Strategy s **weakly dominates** s' if s is better than s' on at least one strategy profile and no worse on any other. A dominant strategy is a strategy that dominates all others. It is irrational to play a dominated strategy, and irrational not to play a dominant strategy if one exists. Being rational, Alice chooses the dominant strategy. We need just a bit more terminology: we say that an outcome is **Pareto optimal**⁵ if there is no other outcome that all players would prefer. An outcome is **Pareto dominated** by another outcome if all players would prefer the other outcome.

WEAK DOMINATION

PERETO OPTIMAL
PERETO DOMINATED

If Alice is clever as well as rational, she will continue to reason as follows: Bob’s dominant strategy is also to testify. Therefore, he will testify and we will both get five years. When each player has a dominant strategy, the combination of those strategies is called a **dominant strategy equilibrium**. In general, a strategy profile forms an **equilibrium** if no player can benefit by switching strategies, given that every other player sticks with the same

DOMINANT
STRATEGY
EQUILIBRIUM
EQUILIBRIUM

⁵ Pareto optimality is named after the economist Vilfredo Pareto (1848–1923).



NASH EQUILIBRIUM

strategy. An equilibrium is essentially a **local optimum** in the space of policies; it is the top of a peak that slopes downward along every dimension, where a dimension corresponds to a player's strategy choices.

The mathematician John Nash (1928–) proved that *every game has at least one equilibrium*. The general concept of equilibrium is now called **Nash equilibrium** in his honor. Clearly, a dominant strategy equilibrium is a Nash equilibrium (Exercise 17.16), but some games have Nash equilibria but no dominant strategies.

The *dilemma* in the prisoner's dilemma is that the equilibrium outcome is worse for both players than the outcome they would get if they both refused to testify. In other words, $(testify, testify)$ is Pareto dominated by the $(-1, -1)$ outcome of $(refuse, refuse)$. Is there any way for Alice and Bob to arrive at the $(-1, -1)$ outcome? It is certainly an *allowable* option for both of them to refuse to testify, but it is hard to see how rational agents can get there, given the definition of the game. Either player contemplating playing *refuse* will realize that he or she would do better by playing *testify*. That is the attractive power of an equilibrium point. Game theorists agree that being a Nash equilibrium is a necessary condition for being a solution—although they disagree whether it is a sufficient condition.

It is easy enough to get to the $(refuse, refuse)$ solution if we modify the game. For example, we could change to a **repeated game** in which the players know that they will meet again. Or the agents might have moral beliefs that encourage cooperation and fairness. That means they have a different utility function, necessitating a different payoff matrix, making it a different game. We will see later that agents with limited computational powers, rather than the ability to reason absolutely rationally, can reach non-equilibrium outcomes, as can an agent that knows that the other agent has limited rationality. In each case, we are considering a different game than the one described by the payoff matrix above.

Now let's look at a game that has no dominant strategy. Acme, a video game console manufacturer, has to decide whether its next game machine will use Blu-ray discs or DVDs. Meanwhile, the video game software producer Best needs to decide whether to produce its next game on Blu-ray or DVD. The profits for both will be positive if they agree and negative if they disagree, as shown in the following payoff matrix:

	<i>Acme:bluray</i>	<i>Acme:dvd</i>
<i>Best:bluray</i>	$A = +9, B = +9$	$A = -4, B = -1$
<i>Best:dvd</i>	$A = -3, B = -1$	$A = +5, B = +5$

There is no dominant strategy equilibrium for this game, but there are *two* Nash equilibria: $(bluray, bluray)$ and (dvd, dvd) . We know these are Nash equilibria because if either player unilaterally moves to a different strategy, that player will be worse off. Now the agents have a problem: *there are multiple acceptable solutions, but if each agent aims for a different solution, then both agents will suffer*. How can they agree on a solution? One answer is that both should choose the Pareto-optimal solution $(bluray, bluray)$; that is, we can restrict the definition of “solution” to the unique Pareto-optimal Nash equilibrium *provided that one exists*. Every game has at least one Pareto-optimal solution, but a game might have several, or they might not be equilibrium points. For example, if $(bluray, bluray)$ had payoff $(5, 5)$, then there would be two equal Pareto-optimal equilibrium points. To choose between



COORDINATION
GAME

them the agents can either guess or *communicate*, which can be done either by establishing a convention that orders the solutions before the game begins or by negotiating to reach a mutually beneficial solution during the game (which would mean including communicative actions as part of a sequential game). Communication thus arises in game theory for exactly the same reasons that it arose in multiagent planning in Section 11.4. Games in which players need to communicate like this are called **coordination games**.

A game can have more than one Nash equilibrium; how do we know that every game must have at least one? Some games have no *pure-strategy* Nash equilibria. Consider, for example, any pure-strategy profile for two-finger Morra (page 666). If the total number of fingers is even, then *O* will want to switch; on the other hand (so to speak), if the total is odd, then *E* will want to switch. Therefore, no pure strategy profile can be an equilibrium and we must look to mixed strategies instead.

ZERO-SUM GAME

But *which* mixed strategy? In 1928, von Neumann developed a method for finding the *optimal* mixed strategy for two-player, **zero-sum games**—games in which the sum of the payoffs is always zero.⁶ Clearly, Morra is such a game. For two-player, zero-sum games, we know that the payoffs are equal and opposite, so we need consider the payoffs of only one player, who will be the maximizer (just as in Chapter 5). For Morra, we pick the even player *E* to be the maximizer, so we can define the payoff matrix by the values $U_E(e, o)$ —the payoff to *E* if *E* does *e* and *O* does *o*. (For convenience we call player *E* “her” and *O* “him.”) Von Neumann’s method is called the **maximin** technique, and it works as follows:

MAXIMIN

- Suppose we change the rules as follows: first *E* picks her strategy and reveals it to *O*. Then *O* picks his strategy, with knowledge of *E*’s strategy. Finally, we evaluate the expected payoff of the game based on the chosen strategies. This gives us a turn-taking game to which we can apply the standard **minimax** algorithm from Chapter 5. Let’s suppose this gives an outcome $U_{E,O}$. Clearly, this game favors *O*, so the true utility U of the original game (from *E*’s point of view) is *at least* $U_{E,O}$. For example, if we just look at pure strategies, the minimax game tree has a root value of -3 (see Figure 17.12(a)), so we know that $U \geq -3$.
- Now suppose we change the rules to force *O* to reveal his strategy first, followed by *E*. Then the minimax value of this game is $U_{O,E}$, and because this game favors *E* we know that U is *at most* $U_{O,E}$. With pure strategies, the value is $+2$ (see Figure 17.12(b)), so we know $U \leq +2$.

Combining these two arguments, we see that the true utility U of the solution to the original game must satisfy

$$U_{E,O} \leq U \leq U_{O,E} \quad \text{or in this case,} \quad -3 \leq U \leq 2.$$



To pinpoint the value of U , we need to turn our analysis to mixed strategies. First, observe the following: *once the first player has revealed his or her strategy, the second player might as well choose a pure strategy*. The reason is simple: if the second player plays a mixed strategy, $[p: \text{one}; (1-p): \text{two}]$, its expected utility is a linear combination $(p \cdot u_{\text{one}} + (1-p) \cdot u_{\text{two}})$ of

⁶ or a constant—see page 162.

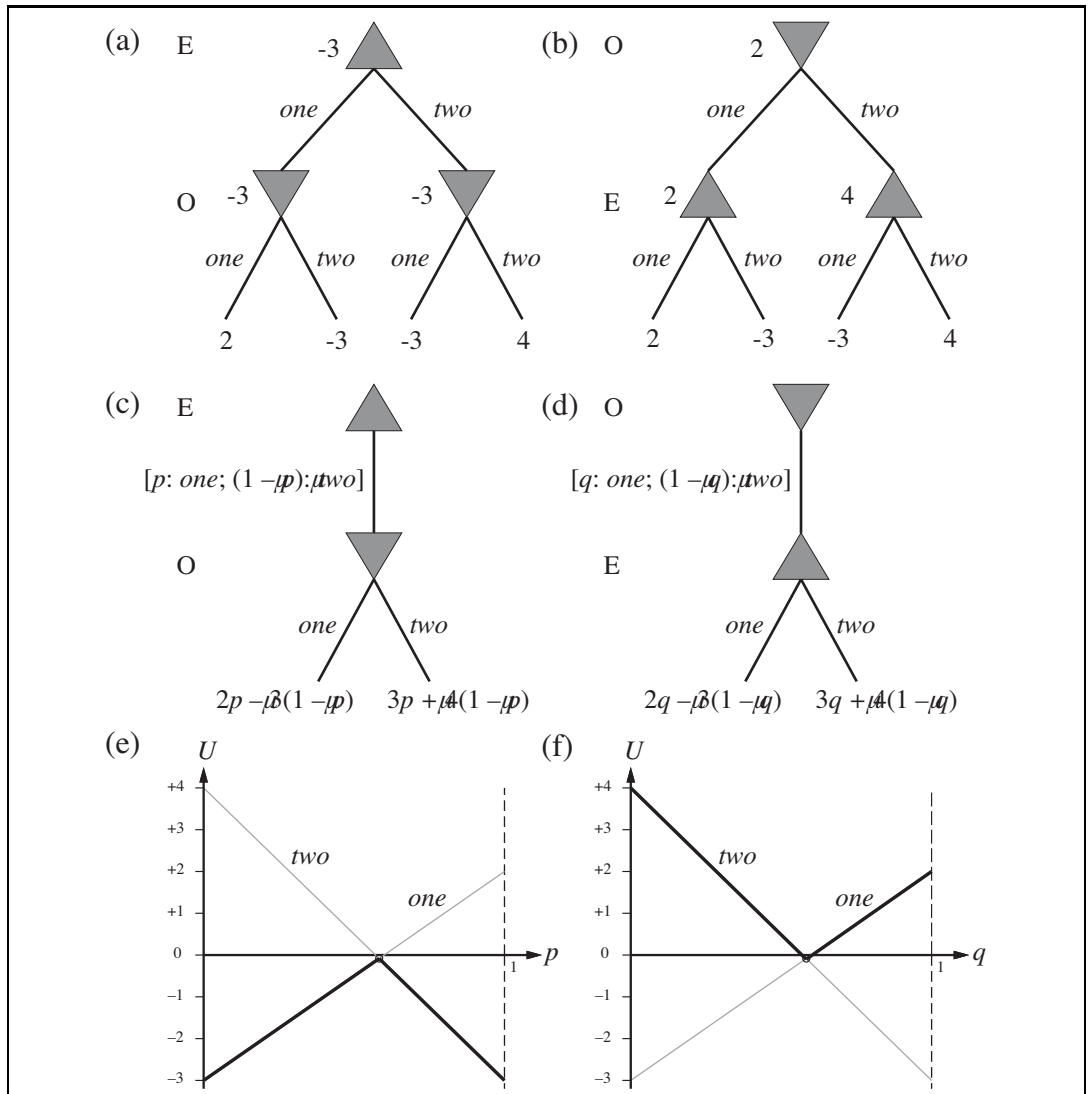


Figure 17.12 (a) and (b): Minimax game trees for two-finger Morra if the players take turns playing pure strategies. (c) and (d): Parameterized game trees where the first player plays a mixed strategy. The payoffs depend on the probability parameter (p or q) in the mixed strategy. (e) and (f): For any particular value of the probability parameter, the second player will choose the “better” of the two actions, so the value of the first player’s mixed strategy is given by the heavy lines. The first player will choose the probability parameter for the mixed strategy at the intersection point.

the utilities of the pure strategies, u_{one} and u_{two} . This linear combination can never be better than the better of u_{one} and u_{two} , so the second player can just choose the better one.

With this observation in mind, the minimax trees can be thought of as having infinitely many branches at the root, corresponding to the infinitely many mixed strategies the first

player can choose. Each of these leads to a node with two branches corresponding to the pure strategies for the second player. We can depict these infinite trees finitely by having one “parameterized” choice at the root:

- If E chooses first, the situation is as shown in Figure 17.12(c). E chooses the strategy $[p: \text{one}; (1-p): \text{two}]$ at the root, and then O chooses a pure strategy (and hence a move) given the value of p . If O chooses *one*, the expected payoff (to E) is $2p - 3(1-p) = 5p - 3$; if O chooses *two*, the expected payoff is $-3p + 4(1-p) = 4 - 7p$. We can draw these two payoffs as straight lines on a graph, where p ranges from 0 to 1 on the x -axis, as shown in Figure 17.12(e). O , the minimizer, will always choose the lower of the two lines, as shown by the heavy lines in the figure. Therefore, the best that E can do at the root is to choose p to be at the intersection point, which is where

$$5p - 3 = 4 - 7p \quad \Rightarrow \quad p = 7/12.$$

The utility for E at this point is $U_{E,O} = -1/12$.

- If O moves first, the situation is as shown in Figure 17.12(d). O chooses the strategy $[q: \text{one}; (1-q): \text{two}]$ at the root, and then E chooses a move given the value of q . The payoffs are $2q - 3(1-q) = 5q - 3$ and $-3q + 4(1-q) = 4 - 7q$.⁷ Again, Figure 17.12(f) shows that the best O can do at the root is to choose the intersection point:

$$5q - 3 = 4 - 7q \quad \Rightarrow \quad q = 7/12.$$

The utility for E at this point is $U_{O,E} = -1/12$.

Now we know that the true utility of the original game lies between $-1/12$ and $-1/12$, that is, it is exactly $-1/12$! (The moral is that it is better to be O than E if you are playing this game.) Furthermore, the true utility is attained by the mixed strategy $[7/12: \text{one}; 5/12: \text{two}]$, which should be played by both players. This strategy is called the **maximin equilibrium** of the game, and is a Nash equilibrium. Note that each component strategy in an equilibrium mixed strategy has the same expected utility. In this case, both *one* and *two* have the same expected utility, $-1/12$, as the mixed strategy itself.

Our result for two-finger Morra is an example of the general result by von Neumann: *every two-player zero-sum game has a maximin equilibrium when you allow mixed strategies*. Furthermore, every Nash equilibrium in a zero-sum game is a maximin for both players. A player who adopts the maximin strategy has two guarantees: First, no other strategy can do better against an opponent who plays well (although some other strategies might be better at exploiting an opponent who makes irrational mistakes). Second, the player continues to do just as well even if the strategy is revealed to the opponent.

The general algorithm for finding maximin equilibria in zero-sum games is somewhat more involved than Figures 17.12(e) and (f) might suggest. When there are n possible actions, a mixed strategy is a point in n -dimensional space and the lines become hyperplanes. It's also possible for some pure strategies for the second player to be dominated by others, so that they are not optimal against *any* strategy for the first player. After removing all such strategies (which might have to be done repeatedly), the optimal choice at the root is the

⁷ It is a coincidence that these equations are the same as those for p ; the coincidence arises because $U_E(\text{one}, \text{two}) = U_E(\text{two}, \text{one}) = -3$. This also explains why the optimal strategy is the same for both players.



highest (or lowest) intersection point of the remaining hyperplanes. Finding this choice is an example of a **linear programming** problem: maximizing an objective function subject to linear constraints. Such problems can be solved by standard techniques in time polynomial in the number of actions (and in the number of bits used to specify the reward function, if you want to get technical).

The question remains, what should a rational agent actually *do* in playing a single game of Morra? The rational agent will have derived the fact that $[7/12: one; 5/12: two]$ is the maximin equilibrium strategy, and will assume that this is mutual knowledge with a rational opponent. The agent could use a 12-sided die or a random number generator to pick randomly according to this mixed strategy, in which case the expected payoff would be $-1/12$ for E . Or the agent could just decide to play *one*, or *two*. In either case, the expected payoff remains $-1/12$ for E . Curiously, unilaterally choosing a particular action does not harm one's expected payoff, but allowing the other agent to know that one has made such a unilateral decision *does* affect the expected payoff, because then the opponent can adjust his strategy accordingly.

Finding equilibria in non-zero-sum games is somewhat more complicated. The general approach has two steps: (1) Enumerate all possible subsets of actions that might form mixed strategies. For example, first try all strategy profiles where each player uses a single action, then those where each player uses either one or two actions, and so on. This is exponential in the number of actions, and so only applies to relatively small games. (2) For each strategy profile enumerated in (1), check to see if it is an equilibrium. This is done by solving a set of equations and inequalities that are similar to the ones used in the zero-sum case. For two players these equations are linear and can be solved with basic linear programming techniques, but for three or more players they are nonlinear and may be very difficult to solve.

17.5.2 Repeated games

REPEATED GAME

So far we have looked only at games that last a single move. The simplest kind of multiple-move game is the **repeated game**, in which players face the same choice repeatedly, but each time with knowledge of the history of all players' previous choices. A strategy profile for a repeated game specifies an action choice for each player at each time step for every possible history of previous choices. As with MDPs, payoffs are additive over time.

Let's consider the repeated version of the prisoner's dilemma. Will Alice and Bob work together and refuse to testify, knowing they will meet again? The answer depends on the details of the engagement. For example, suppose Alice and Bob know that they must play exactly 100 rounds of prisoner's dilemma. Then they both know that the 100th round will not be a repeated game—that is, its outcome can have no effect on future rounds—and therefore they will both choose the dominant strategy, *testify*, in that round. But once the 100th round is determined, the 99th round can have no effect on subsequent rounds, so it too will have a dominant strategy equilibrium at $(testify, testify)$. By induction, both players will choose *testify* on every round, earning a total jail sentence of 500 years each.

We can get different solutions by changing the rules of the interaction. For example, suppose that after each round there is a 99% chance that the players will meet again. Then the expected number of rounds is still 100, but neither player knows for sure which round

PERPETUAL
PUNISHMENT

will be the last. Under these conditions, more cooperative behavior is possible. For example, one equilibrium strategy is for each player to *refuse* unless the other player has ever played *testify*. This strategy could be called **perpetual punishment**. Suppose both players have adopted this strategy, and this is mutual knowledge. Then as long as neither player has played *testify*, then at any point in time the expected future total payoff for each player is

$$\sum_{t=0}^{\infty} 0.99^t \cdot (-1) = -100.$$

A player who deviates from the strategy and chooses *testify* will gain a score of 0 rather than -1 on the very next move, but from then on both players will play *testify* and the player's total expected future payoff becomes

$$0 + \sum_{t=1}^{\infty} 0.99^t \cdot (-5) = -495.$$

Therefore, at every step, there is no incentive to deviate from (*refuse*, *refuse*). Perpetual punishment is the “mutually assured destruction” strategy of the prisoner's dilemma: once either player decides to *testify*, it ensures that both players suffer a great deal. But it works as a deterrent only if the other player believes you have adopted this strategy—or at least that you might have adopted it.

TIT-FOR-TAT

Other strategies are more forgiving. The most famous, called **tit-for-tat**, calls for starting with *refuse* and then echoing the other player's previous move on all subsequent moves. So Alice would refuse as long as Bob refuses and would testify the move after Bob testified, but would go back to refusing if Bob did. Although very simple, this strategy has proven to be highly robust and effective against a wide variety of strategies.

We can also get different solutions by changing the agents, rather than changing the rules of engagement. Suppose the agents are finite-state machines with n states and they are playing a game with $m > n$ total steps. The agents are thus incapable of representing the number of remaining steps, and must treat it as an unknown. Therefore, they cannot do the induction, and are free to arrive at the more favorable (*refuse*, *refuse*) equilibrium. In this case, ignorance *is* bliss—or rather, having your opponent believe that you are ignorant is bliss. Your success in these repeated games depends on the other player's *perception* of you as a bully or a simpleton, and not on your actual characteristics.

17.5.3 Sequential games

EXTENSIVE FORM

In the general case, a game consists of a sequence of turns that need not be all the same. Such games are best represented by a game tree, which game theorists call the **extensive form**. The tree includes all the same information we saw in Section 5.1: an initial state S_0 , a function $\text{PLAYER}(s)$ that tells which player has the move, a function $\text{ACTIONS}(s)$ enumerating the possible actions, a function $\text{RESULT}(s, a)$ that defines the transition to a new state, and a partial function $\text{UTILITY}(s, p)$, which is defined only on terminal states, to give the payoff for each player.

To represent stochastic games, such as backgammon, we add a distinguished player, *chance*, that can take random actions. *Chance*'s “strategy” is part of the definition of the

game, specified as a probability distribution over actions (the other players get to choose their own strategy). To represent games with nondeterministic actions, such as billiards, we break the action into two pieces: the player's action itself has a deterministic result, and then *chance* has a turn to react to the action in its own capricious way. To represent simultaneous moves, as in the prisoner's dilemma or two-finger Morra, we impose an arbitrary order on the players, but we have the option of asserting that the earlier player's actions are not observable to the subsequent players: e.g., Alice must choose *refuse* or *testify* first, then Bob chooses, but Bob does not know what choice Alice made at that time (we can also represent the fact that the move is revealed later). However, we assume the players always remember all their *own* previous actions; this assumption is called **perfect recall**.

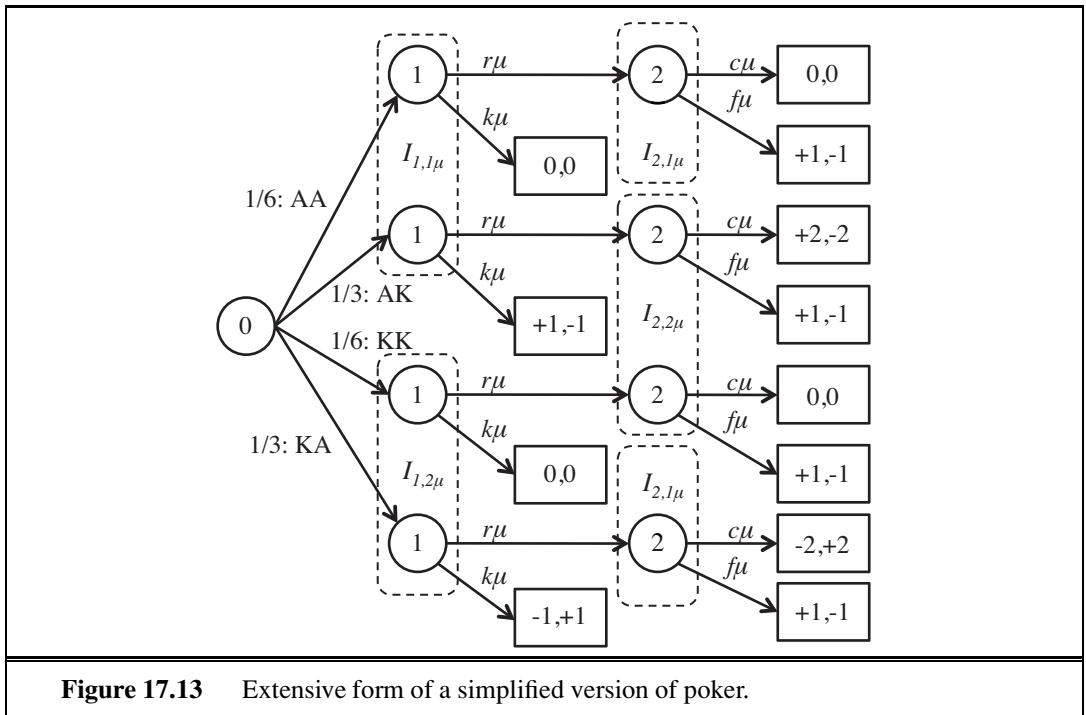
The key idea of extensive form that sets it apart from the game trees of Chapter 5 is the representation of partial observability. We saw in Section 5.6 that a player in a partially observable game such as Kriegspiel can create a game tree over the space of **belief states**. With that tree, we saw that in some cases a player can find a sequence of moves (a strategy) that leads to a forced checkmate regardless of what actual state we started in, and regardless of what strategy the opponent uses. However, the techniques of Chapter 5 could not tell a player what to do when there is no guaranteed checkmate. If the player's best strategy depends on the opponent's strategy and vice versa, then minimax (or alpha-beta) by itself cannot find a solution. The extensive form *does* allow us to find solutions because it represents the belief states (game theorists call them **information sets**) of *all* players at once. From that representation we can find equilibrium solutions, just as we did with normal-form games.

INFORMATION SETS

As a simple example of a sequential game, place two agents in the 4×3 world of Figure 17.1 and have them move simultaneously until one agent reaches an exit square, and gets the payoff for that square. If we specify that no movement occurs when the two agents try to move into the same square simultaneously (a common problem at many traffic intersections), then certain pure strategies can get stuck forever. Thus, agents need a mixed strategy to perform well in this game: randomly choose between moving ahead and staying put. This is exactly what is done to resolve packet collisions in Ethernet networks.

Next we'll consider a very simple variant of poker. The deck has only four cards, two aces and two kings. One card is dealt to each player. The first player then has the option to *raise* the stakes of the game from 1 point to 2, or to *check*. If player 1 checks, the game is over. If he raises, then player 2 has the option to *call*, accepting that the game is worth 2 points, or *fold*, conceding the 1 point. If the game does not end with a fold, then the payoff depends on the cards: it is zero for both players if they have the same card; otherwise the player with the king pays the stakes to the player with the ace.

The extensive-form tree for this game is shown in Figure 17.13. Nonterminal states are shown as circles, with the player to move inside the circle; player 0 is *chance*. Each action is depicted as an arrow with a label, corresponding to a *raise*, *check*, *call*, or *fold*, or, for *chance*, the four possible deals ("AK" means that player 1 gets an ace and player 2 a king). Terminal states are rectangles labeled by their payoff to player 1 and player 2. Information sets are shown as labeled dashed boxes; for example, $I_{1,1}$ is the information set where it is player 1's turn, and he knows he has an ace (but does not know what player 2 has). In information set $I_{2,1}$, it is player 2's turn and she knows that she has an ace and that player 1 has raised,



but does not know what card player 1 has. (Due to the limits of two-dimensional paper, this information set is shown as two boxes rather than one.)

One way to solve an extensive game is to convert it to a normal-form game. Recall that the normal form is a matrix, each row of which is labeled with a pure strategy for player 1, and each column by a pure strategy for player 2. In an extensive game a pure strategy for player i corresponds to an action for each information set involving that player. So in Figure 17.13, one pure strategy for player 1 is “raise when in $I_{1,1}$ (that is, when I have an ace), and check when in $I_{1,2}$ (when I have a king).” In the payoff matrix below, this strategy is called rk . Similarly, strategy cf for player 2 means “call when I have an ace and fold when I have a king.” Since this is a zero-sum game, the matrix below gives only the payoff for player 1; player 2 always has the opposite payoff:

	2:cc	2:cf	2:ff	2:fc
1:rr	0	-1/6	1	7/6
1:kr	-1/3	-1/6	5/6	2/3
1:rk	1/3	0	1/6	1/2
1:kk	0	0	0	0

This game is so simple that it has two pure-strategy equilibria, shown in bold: cf for player 2 and rk or kk for player 1. But in general we can solve extensive games by converting to normal form and then finding a solution (usually a mixed strategy) using standard linear programming methods. That works in theory. But if a player has I information sets and a actions per set, then that player will have a^I pure strategies. In other words, the size of the normal-form matrix is exponential in the number of information sets, so in practice the

approach works only for very small game trees, on the order of a dozen states. A game like Texas hold'em poker has about 10^{18} states, making this approach completely infeasible.

What are the alternatives? In Chapter 5 we saw how alpha-beta search could handle games of perfect information with huge game trees by generating the tree incrementally, by pruning some branches, and by heuristically evaluating nonterminal nodes. But that approach does not work well for games with imperfect information, for two reasons: first, it is harder to prune, because we need to consider mixed strategies that combine multiple branches, not a pure strategy that always chooses the best branch. Second, it is harder to heuristically evaluate a nonterminal node, because we are dealing with information sets, not individual states.

SEQUENCE FORM

Koller *et al.* (1996) come to the rescue with an alternative representation of extensive games, called the **sequence form**, that is only linear in the size of the tree, rather than exponential. Rather than represent strategies, it represents paths through the tree; the number of paths is equal to the number of terminal nodes. Standard linear programming methods can again be applied to this representation. The resulting system can solve poker variants with 25,000 states in a minute or two. This is an exponential speedup over the normal-form approach, but still falls far short of handling full poker, with 10^{18} states.

ABSTRACTION

If we can't handle 10^{18} states, perhaps we can simplify the problem by changing the game to a simpler form. For example, if I hold an ace and am considering the possibility that the next card will give me a pair of aces, then I don't care about the suit of the next card; any suit will do equally well. This suggests forming an **abstraction** of the game, one in which suits are ignored. The resulting game tree will be smaller by a factor of $4! = 24$. Suppose I can solve this smaller game; how will the solution to that game relate to the original game? If no player is going for a flush (or bluffing so), then the suits don't matter to any player, and the solution for the abstraction will also be a solution for the original game. However, if any player is contemplating a flush, then the abstraction will be only an approximate solution (but it is possible to compute bounds on the error).

There are many opportunities for abstraction. For example, at the point in a game where each player has two cards, if I hold a pair of queens, then the other players' hands could be abstracted into three classes: *better* (only a pair of kings or a pair of aces), *same* (pair of queens) or *worse* (everything else). However, this abstraction might be too coarse. A better abstraction would divide *worse* into, say, *medium pair* (nines through jacks), *low pair*, and *no pair*. These examples are abstractions of states; it is also possible to abstract actions. For example, instead of having a bet action for each integer from 1 to 1000, we could restrict the bets to 10^0 , 10^1 , 10^2 and 10^3 . Or we could cut out one of the rounds of betting altogether. We can also abstract over chance nodes, by considering only a subset of the possible deals. This is equivalent to the rollout technique used in Go programs. Putting all these abstractions together, we can reduce the 10^{18} states of poker to 10^7 states, a size that can be solved with current techniques.

Poker programs based on this approach can easily defeat novice and some experienced human players, but are not yet at the level of master players. Part of the problem is that the solution these programs approximate—the equilibrium solution—is optimal only against an opponent who also plays the equilibrium strategy. Against fallible human players it is important to be able to exploit an opponent's deviation from the equilibrium strategy. As

Gautam Rao (aka “The Count”), the world’s leading online poker player, said (Billings *et al.*, 2003), “You have a very strong program. Once you add opponent modeling to it, it will kill everyone.” However, good models of human fallability remain elusive.

In a sense, extensive game form is the one of the most complete representations we have seen so far: it can handle partially observable, multiagent, stochastic, sequential, dynamic environments—most of the hard cases from the list of environment properties on page 42. However, there are two limitations of game theory. First, it does not deal well with continuous states and actions (although there have been some extensions to the continuous case; for example, the theory of **Cournot competition** uses game theory to solve problems where two companies choose prices for their products from a continuous space). Second, game theory assumes the game is *known*. Parts of the game may be specified as unobservable to some of the players, but it must be known what parts are unobservable. In cases in which the players learn the unknown structure of the game over time, the model begins to break down. Let’s examine each source of uncertainty, and whether each can be represented in game theory.

Actions: There is no easy way to represent a game where the players have to discover what actions are available. Consider the game between computer virus writers and security experts. Part of the problem is anticipating what action the virus writers will try next.

Strategies: Game theory is very good at representing the idea that the other players’ strategies are initially unknown—as long as we assume all agents are rational. The theory itself does not say what to do when the other players are less than fully rational. The notion of a **Bayes–Nash equilibrium** partially addresses this point: it is an equilibrium with respect to a player’s prior probability distribution over the other players’ strategies—in other words, it expresses a player’s beliefs about the other players’ likely strategies.

Chance: If a game depends on the roll of a die, it is easy enough to model a chance node with uniform distribution over the outcomes. But what if it is possible that the die is unfair? We can represent that with another chance node, higher up in the tree, with two branches for “die is fair” and “die is unfair,” such that the corresponding nodes in each branch are in the same information set (that is, the players don’t know if the die is fair or not). And what if we suspect the other opponent does know? Then we add *another* chance node, with one branch representing the case where the opponent does know, and one where he doesn’t.

Utilities: What if we don’t know our opponent’s utilities? Again, that can be modeled with a chance node, such that the other agent knows its own utilities in each branch, but we don’t. But what if we don’t know our *own* utilities? For example, how do I know if it is rational to order the Chef’s salad if I don’t know how much I will like it? We can model that with yet another chance node specifying an unobservable “intrinsic quality” of the salad.

Thus, we see that game theory is good at representing most sources of uncertainty—but at the cost of doubling the size of the tree every time we add another node; a habit which quickly leads to intractably large trees. Because of these and other problems, game theory has been used primarily to *analyze* environments that are at equilibrium, rather than to *control* agents within an environment. Next we shall see how it can help *design* environments.

COURNOT
COMPETITION

BAYES–NASH
EQUILIBRIUM

17.6 MECHANISM DESIGN

MECHANISM DESIGN

In the previous section, we asked, “Given a game, what is a rational strategy?” In this section, we ask, “Given that agents pick rational strategies, what game should we design?” More specifically, we would like to design a game whose solutions, consisting of each agent pursuing its own rational strategy, result in the maximization of some global utility function. This problem is called **mechanism design**, or sometimes **inverse game theory**. Mechanism design is a staple of economics and political science. Capitalism 101 says that if everyone tries to get rich, the total wealth of society will increase. But the examples we will discuss show that proper mechanism design is necessary to keep the invisible hand on track. For collections of agents, mechanism design allows us to construct smart systems out of a collection of more limited systems—even uncooperative systems—in much the same way that teams of humans can achieve goals beyond the reach of any individual.

MECHANISM
CENTER

Examples of mechanism design include auctioning off cheap airline tickets, routing TCP packets between computers, deciding how medical interns will be assigned to hospitals, and deciding how robotic soccer players will cooperate with their teammates. Mechanism design became more than an academic subject in the 1990s when several nations, faced with the problem of auctioning off licenses to broadcast in various frequency bands, lost hundreds of millions of dollars in potential revenue as a result of poor mechanism design. Formally, a **mechanism** consists of (1) a language for describing the set of allowable strategies that agents may adopt, (2) a distinguished agent, called the **center**, that collects reports of strategy choices from the agents in the game, and (3) an outcome rule, known to all agents, that the center uses to determine the payoffs to each agent, given their strategy choices.

17.6.1 Auctions

AUCTION

Let’s consider **auctions** first. An auction is a mechanism for selling some goods to members of a pool of bidders. For simplicity, we concentrate on auctions with a single item for sale. Each bidder i has a utility value v_i for having the item. In some cases, each bidder has a **private value** for the item. For example, the first item sold on eBay was a broken laser pointer, which sold for \$14.83 to a collector of broken laser pointers. Thus, we know that the collector has $v_i \geq \$14.83$, but most other people would have $v_j \ll \$14.83$. In other cases, such as auctioning drilling rights for an oil tract, the item has a **common value**—the tract will produce some amount of money, X , and all bidders value a dollar equally—but there is uncertainty as to what the actual value of X is. Different bidders have different information, and hence different estimates of the item’s true value. In either case, bidders end up with their own v_i . Given v_i , each bidder gets a chance, at the appropriate time or times in the auction, to make a bid b_i . The highest bid, b_{max} wins the item, but the price paid need not be b_{max} ; that’s part of the mechanism design.

ASCENDING-BID
ENGLISH AUCTION

The best-known auction mechanism is the **ascending-bid**,⁸ or **English auction**, in which the center starts by asking for a minimum (or **reserve**) bid b_{min} . If some bidder is

⁸ The word “auction” comes from the Latin *augere*, to increase.

willing to pay that amount, the center then asks for $b_{min} + d$, for some increment d , and continues up from there. The auction ends when nobody is willing to bid anymore; then the last bidder wins the item, paying the price he bid.

EFFICIENT

How do we know if this is a good mechanism? One goal is to maximize expected revenue for the seller. Another goal is to maximize a notion of global utility. These goals overlap to some extent, because one aspect of maximizing global utility is to ensure that the winner of the auction is the agent who values the item the most (and thus is willing to pay the most). We say an auction is **efficient** if the goods go to the agent who values them most. The ascending-bid auction is usually both efficient and revenue maximizing, but if the reserve price is set too high, the bidder who values it most may not bid, and if the reserve is set too low, the seller loses net revenue.

COLLUSION

Probably the most important things that an auction mechanism can do is encourage a sufficient number of bidders to enter the game and discourage them from engaging in **collusion**. Collusion is an unfair or illegal agreement by two or more bidders to manipulate prices. It can happen in secret backroom deals or tacitly, within the rules of the mechanism.

For example, in 1999, Germany auctioned ten blocks of cell-phone spectrum with a simultaneous auction (bids were taken on all ten blocks at the same time), using the rule that any bid must be a minimum of a 10% raise over the previous bid on a block. There were only two credible bidders, and the first, Mannesman, entered the bid of 20 million deutschmark on blocks 1-5 and 18.18 million on blocks 6-10. Why 18.18M? One of T-Mobile's managers said they "interpreted Mannesman's first bid as an offer." Both parties could compute that a 10% raise on 18.18M is 19.99M; thus Mannesman's bid was interpreted as saying "we can each get half the blocks for 20M; let's not spoil it by bidding the prices up higher." And in fact T-Mobile bid 20M on blocks 6-10 and that was the end of the bidding. The German government got less than they expected, because the two competitors were able to use the bidding mechanism to come to a tacit agreement on how not to compete. From the government's point of view, a better result could have been obtained by any of these changes to the mechanism: a higher reserve price; a sealed-bid first-price auction, so that the competitors could not communicate through their bids; or incentives to bring in a third bidder. Perhaps the 10% rule was an error in mechanism design, because it facilitated the precise signaling from Mannesman to T-Mobile.

STRATEGY-PROOF

TRUTH-REVEALING

REVELATION
PRINCIPLE

In general, both the seller and the global utility function benefit if there are more bidders, although global utility can suffer if you count the cost of wasted time of bidders that have no chance of winning. One way to encourage more bidders is to make the mechanism easier for them. After all, if it requires too much research or computation on the part of the bidders, they may decide to take their money elsewhere. So it is desirable that the bidders have a **dominant strategy**. Recall that "dominant" means that the strategy works against all other strategies, which in turn means that an agent can adopt it without regard for the other strategies. An agent with a dominant strategy can just bid, without wasting time contemplating other agents' possible strategies. A mechanism where agents have a dominant strategy is called a **strategy-proof** mechanism. If, as is usually the case, that strategy involves the bidders revealing their true value, v_i , then it is called a **truth-revealing**, or **truthful**, auction; the term **incentive compatible** is also used. The **revelation principle** states that any mecha-

nism can be transformed into an equivalent truth-revealing mechanism, so part of mechanism design is finding these equivalent mechanisms.

It turns out that the ascending-bid auction has most of the desirable properties. The bidder with the highest value v_i gets the goods at a price of $b_o + d$, where b_o is the highest bid among all the other agents and d is the auctioneer's increment.⁹ Bidders have a simple dominant strategy: keep bidding as long as the current cost is below your v_i . The mechanism is not quite truth-revealing, because the winning bidder reveals only that his $v_i \geq b_o + d$; we have a lower bound on v_i but not an exact amount.

A disadvantage (from the point of view of the seller) of the ascending-bid auction is that it can discourage competition. Suppose that in a bid for cell-phone spectrum there is one advantaged company that everyone agrees would be able to leverage existing customers and infrastructure, and thus can make a larger profit than anyone else. Potential competitors can see that they have no chance in an ascending-bid auction, because the advantaged company can always bid higher. Thus, the competitors may not enter at all, and the advantaged company ends up winning at the reserve price.

Another negative property of the English auction is its high communication costs. Either the auction takes place in one room or all bidders have to have high-speed, secure communication lines; in either case they have to have the time available to go through several rounds of bidding. An alternative mechanism, which requires much less communication, is the **sealed-bid auction**. Each bidder makes a single bid and communicates it to the auctioneer, without the other bidders seeing it. With this mechanism, there is no longer a simple dominant strategy. If your value is v_i and you believe that the maximum of all the other agents' bids will be b_o , then you should bid $b_o + \epsilon$, for some small ϵ , if that is less than v_i . Thus, your bid depends on your estimation of the other agents' bids, requiring you to do more work. Also, note that the agent with the highest v_i might not win the auction. This is offset by the fact that the auction is more competitive, reducing the bias toward an advantaged bidder.

A small change in the mechanism for sealed-bid auctions produces the **sealed-bid second-price auction**, also known as a **Vickrey auction**.¹⁰ In such auctions, the winner pays the price of the *second*-highest bid, b_o , rather than paying his own bid. This simple modification completely eliminates the complex deliberations required for standard (or **first-price**) sealed-bid auctions, because the dominant strategy is now simply to bid v_i ; the mechanism is truth-revealing. Note that the utility of agent i in terms of his bid b_i , his value v_i , and the best bid among the other agents, b_o , is

$$u_i = \begin{cases} (v_i - b_o) & \text{if } b_i > b_o \\ 0 & \text{otherwise.} \end{cases}$$

To see that $b_i = v_i$ is a dominant strategy, note that when $(v_i - b_o)$ is positive, any bid that wins the auction is optimal, and bidding v_i in particular wins the auction. On the other hand, when $(v_i - b_o)$ is negative, any bid that loses the auction is optimal, and bidding v_i in

⁹ There is actually a small chance that the agent with highest v_i fails to get the goods, in the case in which $b_o < v_i < b_o + d$. The chance of this can be made arbitrarily small by decreasing the increment d .

¹⁰ Named after William Vickrey (1914–1996), who won the 1996 Nobel Prize in economics for this work and died of a heart attack three days later.

SEALED-BID
AUCTION

SEALED-BID
SECOND-PRICE
AUCTION
VICKREY AUCTION

particular loses the auction. So bidding v_i is optimal for all possible values of b_o , and in fact, v_i is the only bid that has this property. Because of its simplicity and the minimal computation requirements for both seller and bidders, the Vickrey auction is widely used in constructing distributed AI systems. Also, Internet search engines conduct over a billion auctions a day to sell advertisements along with their search results, and online auction sites handle \$100 billion a year in goods, all using variants of the Vickrey auction. Note that the expected value to the seller is b_o , which is the same expected return as the limit of the English auction as the increment d goes to zero. This is actually a very general result: the **revenue equivalence theorem** states that, with a few minor caveats, any auction mechanism where risk-neutral bidders have values v_i known only to themselves (but know a probability distribution from which those values are sampled), will yield the same expected revenue. This principle means that the various mechanisms are not competing on the basis of revenue generation, but rather on other qualities.

Although the second-price auction is truth-revealing, it turns out that extending the idea to multiple goods and using a next-price auction is not truth-revealing. Many Internet search engines use a mechanism where they auction k slots for ads on a page. The highest bidder wins the top spot, the second highest gets the second spot, and so on. Each winner pays the price bid by the next-lower bidder, with the understanding that payment is made only if the searcher actually clicks on the ad. The top slots are considered more valuable because they are more likely to be noticed and clicked on. Imagine that three bidders, b_1 , b_2 and b_3 , have valuations for a click of $v_1 = 200$, $v_2 = 180$, and $v_3 = 100$, and that $k = 2$ slots are available, where it is known that the top spot is clicked on 5% of the time and the bottom spot 2%. If all bidders bid truthfully, then b_1 wins the top slot and pays 180, and has an expected return of $(200 - 180) \times 0.05 = 1$. The second slot goes to b_2 . But b_1 can see that if she were to bid anything in the range 101–179, she would concede the top slot to b_2 , win the second slot, and yield an expected return of $(200 - 100) \times .02 = 2$. Thus, b_1 can double her expected return by bidding less than her true value in this case. In general, bidders in this multislot auction must spend a lot of energy analyzing the bids of others to determine their best strategy; there is no simple dominant strategy. Aggarwal *et al.* (2006) show that there is a unique truthful auction mechanism for this multislot problem, in which the winner of slot j pays the full price for slot j just for those additional clicks that are available at slot j and not at slot $j + 1$. The winner pays the price for the lower slot for the remaining clicks. In our example, b_1 would bid 200 truthfully, and would pay 180 for the additional $.05 - .02 = .03$ clicks in the top slot, but would pay only the cost of the bottom slot, 100, for the remaining $.02$ clicks. Thus, the total return to b_1 would be $(200 - 180) \times .03 + (200 - 100) \times .02 = 2.6$.

Another example of where auctions can come into play within AI is when a collection of agents are deciding whether to cooperate on a joint plan. Hunsberger and Grosz (2000) show that this can be accomplished efficiently with an auction in which the agents bid for roles in the joint plan.

17.6.2 Common goods

TRAGEDY OF THE
COMMONS

Now let's consider another type of game, in which countries set their policy for controlling air pollution. Each country has a choice: they can reduce pollution at a cost of -10 points for implementing the necessary changes, or they can continue to pollute, which gives them a net utility of -5 (in added health costs, etc.) and also contributes -1 points to every other country (because the air is shared across countries). Clearly, the dominant strategy for each country is "continue to pollute," but if there are 100 countries and each follows this policy, then each country gets a total utility of -104, whereas if every country reduced pollution, they would each have a utility of -10. This situation is called the **tragedy of the commons**: if nobody has to pay for using a common resource, then it tends to be exploited in a way that leads to a lower total utility for all agents. It is similar to the prisoner's dilemma: there is another solution to the game that is better for all parties, but there appears to be no way for rational agents to arrive at that solution.

EXTERNALITIES

The standard approach for dealing with the tragedy of the commons is to change the mechanism to one that charges each agent for using the commons. More generally, we need to ensure that all **externalities**—effects on global utility that are not recognized in the individual agents' transactions—are made explicit. Setting the prices correctly is the difficult part. In the limit, this approach amounts to creating a mechanism in which each agent is effectively required to maximize global utility, but can do so by making a local decision. For this example, a carbon tax would be an example of a mechanism that charges for use of the commons in a way that, if implemented well, maximizes global utility.

VICKREY-CLARKE-
GROVES
VCG

As a final example, consider the problem of allocating some common goods. Suppose a city decides it wants to install some free wireless Internet transceivers. However, the number of transceivers they can afford is less than the number of neighborhoods that want them. The city wants to allocate the goods efficiently, to the neighborhoods that would value them the most. That is, they want to maximize the global utility $V = \sum_i v_i$. The problem is that if they just ask each neighborhood council "how much do you value this free gift?" they would all have an incentive to lie, and report a high value. It turns out there is a mechanism, known as the **Vickrey-Clarke-Groves**, or **VCG**, mechanism, that makes it a dominant strategy for each agent to report its true utility and that achieves an efficient allocation of the goods. The trick is that each agent pays a tax equivalent to the loss in global utility that occurs because of the agent's presence in the game. The mechanism works like this:

1. The center asks each agent to report its value for receiving an item. Call this b_i .
2. The center allocates the goods to a subset of the bidders. We call this subset A , and use the notation $b_i(A)$ to mean the result to i under this allocation: b_i if i is in A (that is, i is a winner), and 0 otherwise. The center chooses A to maximize total reported utility $B = \sum_i b_i(A)$.
3. The center calculates (for each i) the sum of the reported utilities for all the winners except i . We use the notation $B_{-i} = \sum_{j \neq i} b_j(A)$. The center also computes (for each i) the allocation that would maximize total global utility if i were not in the game; call that sum W_{-i} .
4. Each agent i pays a tax equal to $W_{-i} - B_{-i}$.

In this example, the VCG rule means that each winner would pay a tax equal to the highest reported value among the losers. That is, if I report my value as 5, and that causes someone with value 2 to miss out on an allocation, then I pay a tax of 2. All winners should be happy because they pay a tax that is less than their value, and all losers are as happy as they can be, because they value the goods less than the required tax.

Why is it that this mechanism is truth-revealing? First, consider the payoff to agent i , which is the value of getting an item, minus the tax:

$$v_i(A) - (W_{-i} - B_{-i}) . \quad (17.14)$$

Here we distinguish the agent's true utility, v_i , from his reported utility b_i (but we are trying to show that a dominant strategy is $b_i = v_i$). Agent i knows that the center will maximize global utility using the reported values,

$$\sum_j b_j(A) = b_i(A) + \sum_{j \neq i} b_j(A)$$

whereas agent i wants the center to maximize (17.14), which can be rewritten as

$$v_i(A) + \sum_{j \neq i} b_j(A) - W_{-i} .$$

Since agent i cannot affect the value of W_{-i} (it depends only on the other agents), the only way i can make the center optimize what i wants is to report the true utility, $b_i = v_i$.

17.7 SUMMARY

This chapter shows how to use knowledge about the world to make decisions even when the outcomes of an action are uncertain and the rewards for acting might not be reaped until many actions have passed. The main points are as follows:

- Sequential decision problems in uncertain environments, also called **Markov decision processes**, or MDPs, are defined by a **transition model** specifying the probabilistic outcomes of actions and a **reward function** specifying the reward in each state.
- The utility of a state sequence is the sum of all the rewards over the sequence, possibly discounted over time. The solution of an MDP is a **policy** that associates a decision with every state that the agent might reach. An optimal policy maximizes the utility of the state sequences encountered when it is executed.
- The utility of a state is the expected utility of the state sequences encountered when an optimal policy is executed, starting in that state. The **value iteration** algorithm for solving MDPs works by iteratively solving the equations relating the utility of each state to those of its neighbors.
- **Policy iteration** alternates between calculating the utilities of states under the current policy and improving the current policy with respect to the current utilities.
- Partially observable MDPs, or POMDPs, are much more difficult to solve than are MDPs. They can be solved by conversion to an MDP in the continuous space of belief

states; both value iteration and policy iteration algorithms have been devised. Optimal behavior in POMDPs includes information gathering to reduce uncertainty and therefore make better decisions in the future.

- A decision-theoretic agent can be constructed for POMDP environments. The agent uses a **dynamic decision network** to represent the transition and sensor models, to update its belief state, and to project forward possible action sequences.
- **Game theory** describes rational behavior for agents in situations in which multiple agents interact simultaneously. Solutions of games are **Nash equilibria**—strategy profiles in which no agent has an incentive to deviate from the specified strategy.
- **Mechanism design** can be used to set the rules by which agents will interact, in order to maximize some global utility through the operation of individually rational agents. Sometimes, mechanisms exist that achieve this goal without requiring each agent to consider the choices made by other agents.

We shall return to the world of MDPs and POMDP in Chapter 21, when we study **reinforcement learning** methods that allow an agent to improve its behavior from experience in sequential, uncertain environments.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Richard Bellman developed the ideas underlying the modern approach to sequential decision problems while working at the RAND Corporation beginning in 1949. According to his autobiography (Bellman, 1984), he coined the exciting term “dynamic programming” to hide from a research-phobic Secretary of Defense, Charles Wilson, the fact that his group was doing mathematics. (This cannot be strictly true, because his first paper using the term (Bellman, 1952) appeared before Wilson became Secretary of Defense in 1953.) Bellman’s book, *Dynamic Programming* (1957), gave the new field a solid foundation and introduced the basic algorithmic approaches. Ron Howard’s Ph.D. thesis (1960) introduced policy iteration and the idea of average reward for solving infinite-horizon problems. Several additional results were introduced by Bellman and Dreyfus (1962). Modified policy iteration is due to van Nunen (1976) and Puterman and Shin (1978). Asynchronous policy iteration was analyzed by Williams and Baird (1993), who also proved the policy loss bound in Equation (17.9). The analysis of discounting in terms of stationary preferences is due to Koopmans (1972). The texts by Bertsekas (1987), Puterman (1994), and Bertsekas and Tsitsiklis (1996) provide a rigorous introduction to sequential decision problems. Papadimitriou and Tsitsiklis (1987) describe results on the computational complexity of MDPs.

Seminal work by Sutton (1988) and Watkins (1989) on reinforcement learning methods for solving MDPs played a significant role in introducing MDPs into the AI community, as did the later survey by Barto *et al.* (1995). (Earlier work by Werbos (1977) contained many similar ideas, but was not taken up to the same extent.) The connection between MDPs and AI planning problems was made first by Sven Koenig (1991), who showed how probabilistic STRIPS operators provide a compact representation for transition models (see also Wellman,

1990b). Work by Dean *et al.* (1993) and Tash and Russell (1994) attempted to overcome the combinatorics of large state spaces by using a limited search horizon and abstract states. Heuristics based on the value of information can be used to select areas of the state space where a local expansion of the horizon will yield a significant improvement in decision quality. Agents using this approach can tailor their effort to handle time pressure and generate some interesting behaviors such as using familiar “beaten paths” to find their way around the state space quickly without having to recompute optimal decisions at each point.

As one might expect, AI researchers have pushed MDPs in the direction of more expressive representations that can accommodate much larger problems than the traditional atomic representations based on transition matrices. The use of a dynamic Bayesian network to represent transition models was an obvious idea, but work on **factored MDPs** (Boutilier *et al.*, 2000; Koller and Parr, 2000; Guestrin *et al.*, 2003b) extends the idea to structured representations of the value function with provable improvements in complexity. **Relational MDPs** (Boutilier *et al.*, 2001; Guestrin *et al.*, 2003a) go one step further, using structured representations to handle domains with many related objects.

The observation that a partially observable MDP can be transformed into a regular MDP over belief states is due to Astrom (1965) and Aoki (1965). The first complete algorithm for the exact solution of POMDPs—essentially the value iteration algorithm presented in this chapter—was proposed by Edward Sondik (1971) in his Ph.D. thesis. (A later journal paper by Smallwood and Sondik (1973) contains some errors, but is more accessible.) Lovejoy (1991) surveyed the first twenty-five years of POMDP research, reaching somewhat pessimistic conclusions about the feasibility of solving large problems. The first significant contribution within AI was the Witness algorithm (Cassandra *et al.*, 1994; Kaelbling *et al.*, 1998), an improved version of POMDP value iteration. Other algorithms soon followed, including an approach due to Hansen (1998) that constructs a policy incrementally in the form of a finite-state automaton. In this policy representation, the belief state corresponds directly to a particular state in the automaton. More recent work in AI has focused on **point-based** value iteration methods that, at each iteration, generate conditional plans and α -vectors for a finite set of belief states rather than for the entire belief space. Lovejoy (1991) proposed such an algorithm for a fixed grid of points, an approach taken also by Bonet (2002). An influential paper by Pineau *et al.* (2003) suggested generating reachable points by simulating trajectories in a somewhat greedy fashion; Spaan and Vlassis (2005) observe that one need generate plans for only a small, randomly selected subset of points to improve on the plans from the previous iteration for all points in the set. Current point-based methods—such as point-based policy iteration (Ji *et al.*, 2007)—can generate near-optimal solutions for POMDPs with thousands of states. Because POMDPs are PSPACE-hard (Papadimitriou and Tsitsiklis, 1987), further progress may require taking advantage of various kinds of structure within a factored representation.

The online approach—using look-ahead search to select an action for the current belief state—was first examined by Satia and Lave (1973). The use of sampling at chance nodes was explored analytically by Kearns *et al.* (2000) and Ng and Jordan (2000). The basic ideas for an agent architecture using dynamic decision networks were proposed by Dean and Kanazawa (1989a). The book *Planning and Control* by Dean and Wellman (1991) goes

FACTORED MDP

RELATIONAL MDP

into much greater depth, making connections between DBN/DDN models and the classical control literature on filtering. Tatman and Shachter (1990) showed how to apply dynamic programming algorithms to DDN models. Russell (1998) explains various ways in which such agents can be scaled up and identifies a number of open research issues.

The roots of game theory can be traced back to proposals made in the 17th century by Christiaan Huygens and Gottfried Leibniz to study competitive and cooperative human interactions scientifically and mathematically. Throughout the 19th century, several leading economists created simple mathematical examples to analyze particular examples of competitive situations. The first formal results in game theory are due to Zermelo (1913) (who had, the year before, suggested a form of minimax search for games, albeit an incorrect one). Emile Borel (1921) introduced the notion of a mixed strategy. John von Neumann (1928) proved that every two-person, zero-sum game has a maximin equilibrium in mixed strategies and a well-defined value. Von Neumann's collaboration with the economist Oskar Morgenstern led to the publication in 1944 of the *Theory of Games and Economic Behavior*, the defining book for game theory. Publication of the book was delayed by the wartime paper shortage until a member of the Rockefeller family personally subsidized its publication.

In 1950, at the age of 21, John Nash published his ideas concerning equilibria in general (non-zero-sum) games. His definition of an equilibrium solution, although originating in the work of Cournot (1838), became known as Nash equilibrium. After a long delay because of the schizophrenia he suffered from 1959 onward, Nash was awarded the Nobel Memorial Prize in Economics (along with Reinhard Selten and John Harsanyi) in 1994. The Bayes–Nash equilibrium is described by Harsanyi (1967) and discussed by Kadane and Larkey (1982). Some issues in the use of game theory for agent control are covered by Binmore (1982).

The prisoner's dilemma was invented as a classroom exercise by Albert W. Tucker in 1950 (based on an example by Merrill Flood and Melvin Dresher) and is covered extensively by Axelrod (1985) and Poundstone (1993). Repeated games were introduced by Luce and Raiffa (1957), and games of partial information in extensive form by Kuhn (1953). The first practical algorithm for sequential, partial-information games was developed within AI by Koller *et al.* (1996); the paper by Koller and Pfeffer (1997) provides a readable introduction to the field and describe a working system for representing and solving sequential games.

The use of abstraction to reduce a game tree to a size that can be solved with Koller's technique is discussed by Billings *et al.* (2003). Bowling *et al.* (2008) show how to use importance sampling to get a better estimate of the value of a strategy. Waugh *et al.* (2009) show that the abstraction approach is vulnerable to making systematic errors in approximating the equilibrium solution, meaning that the whole approach is on shaky ground: it works for some games but not others. Korb *et al.* (1999) experiment with an opponent model in the form of a Bayesian network. It plays five-card stud about as well as experienced humans. (Zinkevich *et al.*, 2008) show how an approach that minimizes regret can find approximate equilibria for abstractions with 10^{12} states, 100 times more than previous methods.

Game theory and MDPs are combined in the theory of Markov games, also called stochastic games (Littman, 1994; Hu and Wellman, 1998). Shapley (1953) actually described the value iteration algorithm independently of Bellman, but his results were not widely appreciated, perhaps because they were presented in the context of Markov games. Evolu-

tionary game theory (Smith, 1982; Weibull, 1995) looks at strategy drift over time: if your opponent's strategy is changing, how should you react? Textbooks on game theory from an economics point of view include those by Myerson (1991), Fudenberg and Tirole (1991), Osborne (2004), and Osborne and Rubinstein (1994); Mailath and Samuelson (2006) concentrate on repeated games. From an AI perspective we have Nisan *et al.* (2007), Leyton-Brown and Shoham (2008), and Shoham and Leyton-Brown (2009).

The 2007 Nobel Memorial Prize in Economics went to Hurwicz, Maskin, and Myerson “for having laid the foundations of mechanism design theory” (Hurwicz, 1973). The tragedy of the commons, a motivating problem for the field, was presented by Hardin (1968). The revelation principle is due to Myerson (1986), and the revenue equivalence theorem was developed independently by Myerson (1981) and Riley and Samuelson (1981). Two economists, Milgrom (1997) and Klemperer (2002), write about the multibillion-dollar spectrum auctions they were involved in.

Mechanism design is used in multiagent planning (Hunsberger and Grosz, 2000; Stone *et al.*, 2009) and scheduling (Rassenti *et al.*, 1982). Varian (1995) gives a brief overview with connections to the computer science literature, and Rosenschein and Zlotkin (1994) present a book-length treatment with applications to distributed AI. Related work on distributed AI also goes under other names, including collective intelligence (Tumer and Wolpert, 2000; Segaran, 2007) and market-based control (Clearwater, 1996). Since 2001 there has been an annual Trading Agents Competition (TAC), in which agents try to make the best profit on a series of auctions (Wellman *et al.*, 2001; Arunachalam and Sadeh, 2005). Papers on computational issues in auctions often appear in the ACM Conferences on Electronic Commerce.

EXERCISES

17.1 For the 4×3 world shown in Figure 17.1, calculate which squares can be reached from (1,1) by the action sequence $[Up, Up, Right, Right, Right]$ and with what probabilities. Explain how this computation is related to the prediction task (see Section 15.2.1) for a hidden Markov model.

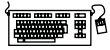
17.2 Select a specific member of the set of policies that are optimal for $R(s) > 0$ as shown in Figure 17.2(b), and calculate the fraction of time the agent spends in each state, in the limit, if the policy is executed forever. (*Hint:* Construct the state-to-state transition probability matrix corresponding to the policy and see Exercise 15.2.)

17.3 Suppose that we define the utility of a state sequence to be the *maximum* reward obtained in any state in the sequence. Show that this utility function does not result in stationary preferences between state sequences. Is it still possible to define a utility function on states such that MEU decision making gives optimal behavior?

17.4 Sometimes MDPs are formulated with a reward function $R(s, a)$ that depends on the action taken or with a reward function $R(s, a, s')$ that also depends on the outcome state.

- a. Write the Bellman equations for these formulations.

- b. Show how an MDP with reward function $R(s, a, s')$ can be transformed into a different MDP with reward function $R(s, a)$, such that optimal policies in the new MDP correspond exactly to optimal policies in the original MDP.
- c. Now do the same to convert MDPs with $R(s, a)$ into MDPs with $R(s)$.



17.5 For the environment shown in Figure 17.1, find all the threshold values for $R(s)$ such that the optimal policy changes when the threshold is crossed. You will need a way to calculate the optimal policy and its value for fixed $R(s)$. (*Hint*: Prove that the value of any fixed policy varies linearly with $R(s)$.)

17.6 Equation (17.7) on page 654 states that the Bellman operator is a contraction.

- a. Show that, for any functions f and g ,

$$|\max_a f(a) - \max_a g(a)| \leq \max_a |f(a) - g(a)|.$$

- b. Write out an expression for $|(BU_i - BU'_i)(s)|$ and then apply the result from (a) to complete the proof that the Bellman operator is a contraction.

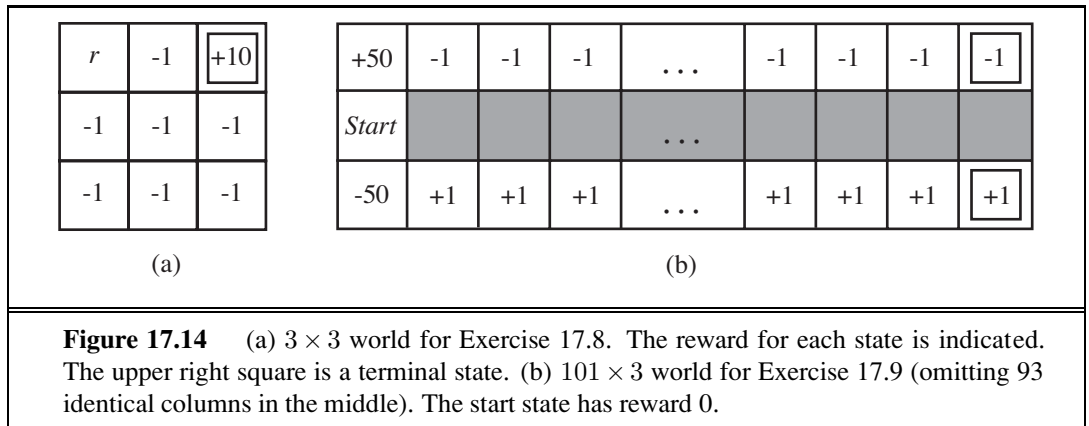
17.7 This exercise considers two-player MDPs that correspond to zero-sum, turn-taking games like those in Chapter 5. Let the players be A and B , and let $R(s)$ be the reward for player A in state s . (The reward for B is always equal and opposite.)

- a. Let $U_A(s)$ be the utility of state s when it is A 's turn to move in s , and let $U_B(s)$ be the utility of state s when it is B 's turn to move in s . All rewards and utilities are calculated from A 's point of view (just as in a minimax game tree). Write down Bellman equations defining $U_A(s)$ and $U_B(s)$.
- b. Explain how to do two-player value iteration with these equations, and define a suitable termination criterion.
- c. Consider the game described in Figure 5.17 on page 197. Draw the state space (rather than the game tree), showing the moves by A as solid lines and moves by B as dashed lines. Mark each state with $R(s)$. You will find it helpful to arrange the states (s_A, s_B) on a two-dimensional grid, using s_A and s_B as “coordinates.”
- d. Now apply two-player value iteration to solve this game, and derive the optimal policy.

17.8 Consider the 3×3 world shown in Figure 17.14(a). The transition model is the same as in the 4×3 Figure 17.1: 80% of the time the agent goes in the direction it selects; the rest of the time it moves at right angles to the intended direction.

Implement value iteration for this world for each value of r below. Use discounted rewards with a discount factor of 0.99. Show the policy obtained in each case. Explain intuitively why the value of r leads to each policy.

- a. $r = 100$
- b. $r = -3$
- c. $r = 0$
- d. $r = +3$



17.9 Consider the 101×3 world shown in Figure 17.14(b). In the start state the agent has a choice of two deterministic actions, *Up* or *Down*, but in the other states the agent has one deterministic action, *Right*. Assuming a discounted reward function, for what values of the discount γ should the agent choose *Up* and for which *Down*? Compute the utility of each action as a function of γ . (Note that this simple example actually reflects many real-world situations in which one must weigh the value of an immediate action versus the potential continual long-term consequences, such as choosing to dump pollutants into a lake.)

17.10 Consider an undiscounted MDP having three states, (1, 2, 3), with rewards -1 , -2 , 0, respectively. State 3 is a terminal state. In states 1 and 2 there are two possible actions: a and b . The transition model is as follows:

- In state 1, action a moves the agent to state 2 with probability 0.8 and makes the agent stay put with probability 0.2.
- In state 2, action a moves the agent to state 1 with probability 0.8 and makes the agent stay put with probability 0.2.
- In either state 1 or state 2, action b moves the agent to state 3 with probability 0.1 and makes the agent stay put with probability 0.9.

Answer the following questions:

- a. What can be determined *qualitatively* about the optimal policy in states 1 and 2?
- b. Apply policy iteration, showing each step in full, to determine the optimal policy and the values of states 1 and 2. Assume that the initial policy has action b in both states.
- c. What happens to policy iteration if the initial policy has action a in both states? Does discounting help? Does the optimal policy depend on the discount factor?



17.11 Consider the 4×3 world shown in Figure 17.1.

- a. Implement an environment simulator for this environment, such that the specific geography of the environment is easily altered. Some code for doing this is already in the online code repository.

- b. Create an agent that uses policy iteration, and measure its performance in the environment simulator from various starting states. Perform several experiments from each starting state, and compare the average total reward received per run with the utility of the state, as determined by your algorithm.
- c. Experiment with increasing the size of the environment. How does the run time for policy iteration vary with the size of the environment?

17.12 How can the value determination algorithm be used to calculate the expected loss experienced by an agent using a given set of utility estimates U and an estimated model P , compared with an agent using correct values?

17.13 Let the initial belief state b_0 for the 4×3 POMDP on page 658 be the uniform distribution over the nonterminal states, i.e., $\langle \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, 0, 0 \rangle$. Calculate the exact belief state b_1 after the agent moves *Left* and its sensor reports 1 adjacent wall. Also calculate b_2 assuming that the same thing happens again.

17.14 What is the time complexity of d steps of POMDP value iteration for a sensorless environment?

17.15 Consider a version of the two-state POMDP on page 661 in which the sensor is 90% reliable in state 0 but provides no information in state 1 (that is, it reports 0 or 1 with equal probability). Analyze, either qualitatively or quantitatively, the utility function and the optimal policy for this problem.

17.16 Show that a dominant strategy equilibrium is a Nash equilibrium, but not vice versa.

17.17 In the children's game of rock–paper–scissors each player reveals at the same time a choice of rock, paper, or scissors. Paper wraps rock, rock blunts scissors, and scissors cut paper. In the extended version rock–paper–scissors–fire–water, fire beats rock, paper, and scissors; rock, paper, and scissors beat water; and water beats fire. Write out the payoff matrix and find a mixed-strategy solution to this game.

17.18 The following payoff matrix, from Blinder (1983) by way of Bernstein (1996), shows a game between politicians and the Federal Reserve.

	Fed: contract	Fed: do nothing	Fed: expand
Pol: contract	$F = 7, P = 1$	$F = 9, P = 4$	$F = 6, P = 6$
Pol: do nothing	$F = 8, P = 2$	$F = 5, P = 5$	$F = 4, P = 9$
Pol: expand	$F = 3, P = 3$	$F = 2, P = 7$	$F = 1, P = 8$

Politicians can expand or contract fiscal policy, while the Fed can expand or contract monetary policy. (And of course either side can choose to do nothing.) Each side also has preferences for who should do what—neither side wants to look like the bad guys. The payoffs shown are simply the rank orderings: 9 for first choice through 1 for last choice. Find the Nash equilibrium of the game in pure strategies. Is this a Pareto-optimal solution? You might wish to analyze the policies of recent administrations in this light.

17.19 A Dutch auction is similar in an English auction, but rather than starting the bidding at a low price and increasing, in a Dutch auction the seller starts at a high price and gradually lowers the price until some buyer is willing to accept that price. (If multiple bidders accept the price, one is arbitrarily chosen as the winner.) More formally, the seller begins with a price p and gradually lowers p by increments of d until at least one buyer accepts the price. Assuming all bidders act rationally, is it true that for arbitrarily small d , a Dutch auction will always result in the bidder with the highest value for the item obtaining the item? If so, show mathematically why. If not, explain how it may be possible for the bidder with highest value for the item not to obtain it.

17.20 Imagine an auction mechanism that is just like an ascending-bid auction, except that at the end, the winning bidder, the one who bid b_{max} , pays only $b_{max}/2$ rather than b_{max} . Assuming all agents are rational, what is the expected revenue to the auctioneer for this mechanism, compared with a standard ascending-bid auction?

17.21 Teams in the National Hockey League historically received 2 points for winning a game and 0 for losing. If the game is tied, an overtime period is played; if nobody wins in overtime, the game is a tie and each team gets 1 point. But league officials felt that teams were playing too conservatively in overtime (to avoid a loss), and it would be more exciting if overtime produced a winner. So in 1999 the officials experimented in mechanism design: the rules were changed, giving a team that loses in overtime 1 point, not 0. It is still 2 points for a win and 1 for a tie.

- a. Was hockey a zero-sum game before the rule change? After?
- b. Suppose that at a certain time t in a game, the home team has probability p of winning in regulation time, probability $0.78 - p$ of losing, and probability 0.22 of going into overtime, where they have probability q of winning, $.9 - q$ of losing, and .1 of tying. Give equations for the expected value for the home and visiting teams.
- c. Imagine that it were legal and ethical for the two teams to enter into a pact where they agree that they will skate to a tie in regulation time, and then both try in earnest to win in overtime. Under what conditions, in terms of p and q , would it be rational for both teams to agree to this pact?
- d. Longley and Sankaran (2005) report that since the rule change, the percentage of games with a winner in overtime went up 18.2%, as desired, but the percentage of overtime games also went up 3.6%. What does that suggest about possible collusion or conservative play after the rule change?