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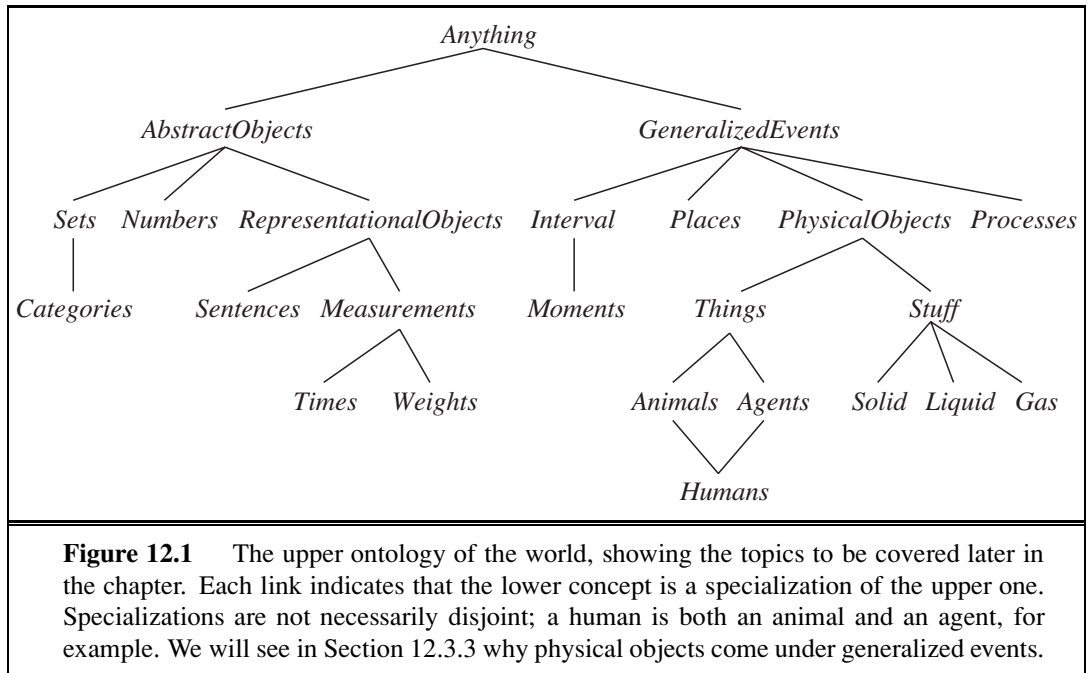
The previous chapters described the technology for knowledge-based agents: the syntax, semantics, and proof theory of propositional and first-order logic, and the implementation of agents that use these logics. In this chapter we address the question of what *content* to put into such an agent's knowledge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

ONTOLOGICAL
ENGINEERING
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

Figure 12.1   The upper ontology of the world, showing the topics to be covered later in the chapter. Each link indicates that the lower concept is a specialization of the upper one. Specializations are not necessarily disjoint; a human is both an animal and an agent, for example. We will see in Section 12.3.3 why physical objects come under generalized events.

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

SUBCATEGORY

a **subcategory** of $Balls$. We will use subcategory, subclass, and subset interchangeably.

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.
  $(x \in Basketballs) \implies Spherical(x)$

---

[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.
  $Orange(x) \land Round(x) \land Diameter(x) = 9.5'' \land x \in Balls \Rightarrow x \in Basketballs$
- 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

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

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

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

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

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

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,

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

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

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

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

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 cate-
gories. (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 compos-
ite 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 struc-
ture. 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
BUNCH   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

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

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

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

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

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

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

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

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:

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

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

$$x \in Typical(Tomatoes) \implies Red(x) \land Round(x) .$$

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

UNITS FUNCTION

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

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

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

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

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

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

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

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

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

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

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

12.2.3   Objects: Things and 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).

INDIVIDUATION

STUFF

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

COUNT NOUNS

MASS NOUN

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 w omit for now, any part of a butter-object is also a butter-object:

$$b \in Butter \land PartOf(p, b) \; \Rightarrow \; p \in Butter \; .$$

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

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.

EXTRINSIC

12.3   EVENTS

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
EVENT CALCULUS   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:

$$Happens(e, (t_1, t_2)) \land Initiates(e, f, t_1) \land \neg Clipped(f, (t_1, t)) \land t_1 < t \Rightarrow$$
$$T(f, t)$$
$$Happens(e, (t_1, t_2)) \land Terminates(e, f, t_1) \land \neg Restored(f, (t_1, t)) \land t_1 < t \Rightarrow$$
$$\neg T(f, t)$$

where *Clipped* and *Restored* are defined by

$$Clipped(f, (t_1, t_2)) \Leftrightarrow$$
$$\exists e, t, t_3 \ Happens(e, (t, t_3)) \land t_1 \leq t < t_2 \land Terminates(e, f, t)$$
$$Restored(f, (t_1, t_2)) \Leftrightarrow$$
$$\exists e, t, t_3 \ Happens(e, (t, t_3)) \land t_1 \leq t < t_2 \land Initiates(e, f, t)$$

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:

$$Initiates(e, HaveArrow(a), t) \Leftrightarrow e = Start$$
$$Terminates(e, HaveArrow(a), t) \Leftrightarrow e \in Shootings(a)$$

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:

$$(e \in Processes) \land Happens(e, (t_1, t_4)) \land (t_1 < t_2 < t_3 < t_4) \Rightarrow Happens(e, (t_2, t_3)).$$

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:

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

Next we invent a time scale and associate points on that scale with moments, giving us absolute times. The time scale is arbitrary; we 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.

$$Interval(i) \implies Duration(i) = (Time(End(i)) - Time(Begin(i))) \ .$$
$$Time(Begin(AD1900)) = Seconds(0) \ .$$
$$Time(Begin(AD2001)) = Seconds(3187324800) \ .$$
$$Time(End(AD2001)) = Seconds(3218860800) \ .$$
$$Duration(AD2001) = Seconds(31536000) \ .$$

To make these numbers easier to read, we also introduce a function $Date$, which takes six arguments (hours, minutes, seconds, day, month, and year) and returns a time point:
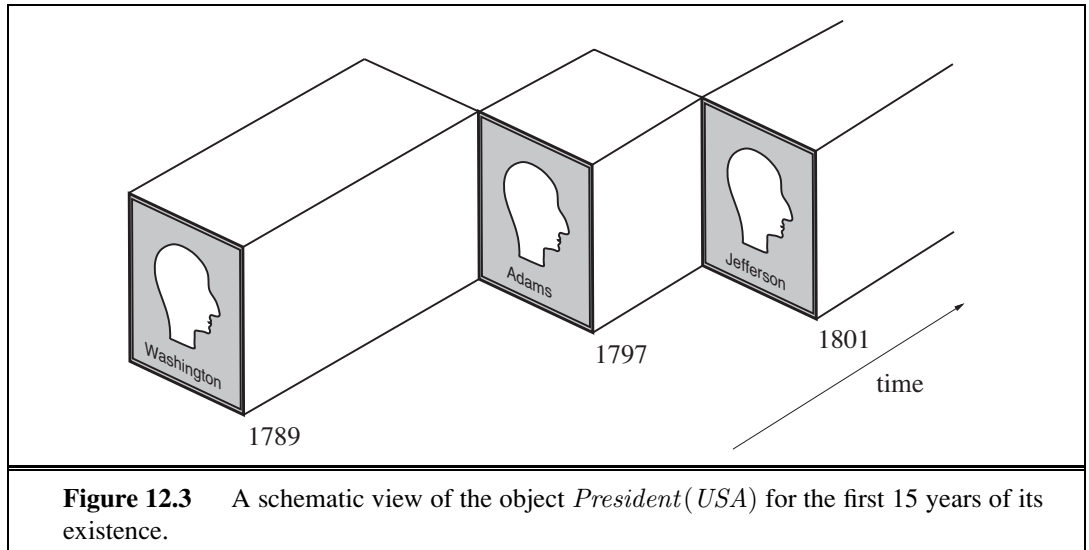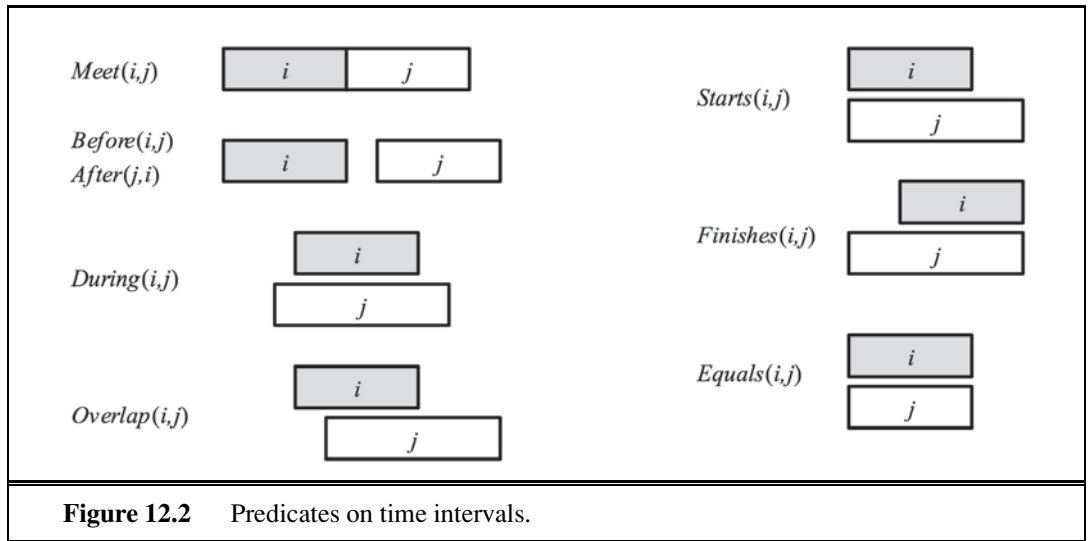
$$Time(Begin(AD2001)) = Date(0, 0, 0, 1, Jan, 2001)$$
$$Date(0, 20, 21, 24, 1, 1995) = Seconds(3000000000) \ .$$

Two intervals $Meet$ if the end time of the first equals the start time of the second. The complete set of interval relations, as proposed by Allen (1983), is shown graphically in Figure 12.2 and logically below:

$$
\begin{array}{lll}
Meet(i, j) & \iff & End(i) = Begin(j) \\
Before(i, j) & \iff & End(i) < Begin(j) \\
After(j, i) & \iff & Before(i, j) \\
During(i, j) & \iff & Begin(j) < Begin(i) < End(i) < End(j) \\
Overlap(i, j) & \iff & Begin(i) < Begin(j) < End(i) < End(j) \\
Begins(i, j) & \iff & Begin(i) = Begin(j) \\
Finishes(i, j) & \iff & End(i) = End(j) \\
Equals(i, j) & \iff & Begin(i) = Begin(j) \wedge End(i) = End(j)
\end{array}
$$

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:

$$Meets(ReignOf(GeorgeVI), ReignOf(ElizabethII)) \ .$$
$$Overlap(Fifties, ReignOf(Elvis)) \ .$$
$$Begin(Fifties) = Begin(AD1950) \ .$$
$$End(Fifties) = End(AD1959) \ .$$

Figure 12.2   Predicates on time intervals.

Figure 12.3   A schematic view of the object $President(USA)$ for the first 15 years of its existence.

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.

PROPOSITIONAL
ATTITUDE
        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 be 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:

$$(Superman = Clark) \wedge Knows(Lois, CanFly(Superman))$$
$$\models Knows(Lois, CanFly(Clark)) \ .$$

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.

       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 $Acc(\mathbf{K}_A, w_0, w_1)$. 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.
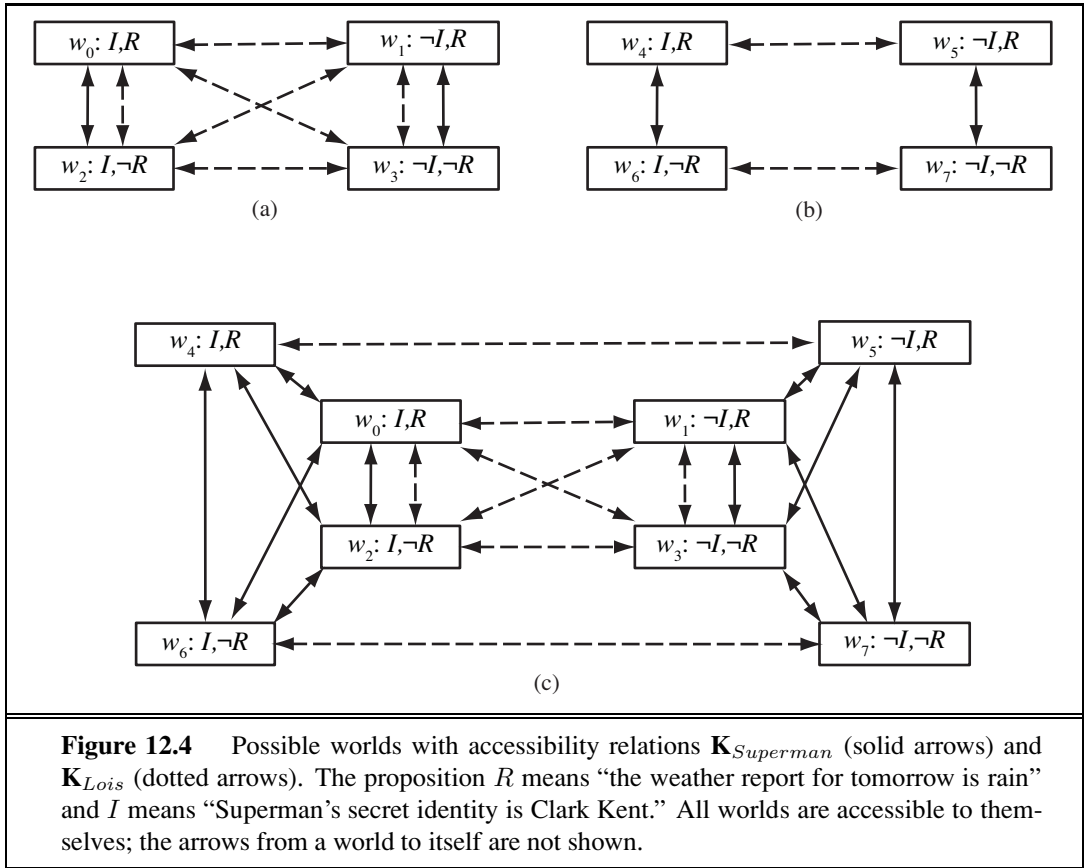
POSSIBLE WORLD
ACCESSIBILITY
RELATIONS

       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:

$$\mathbf{K}_{Lois}[\mathbf{K}_{Clark}Identity(Superman, Clark) \vee \mathbf{K}_{Clark}\neg Identity(Superman, Clark)]$$

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.

Figure 12.4   Possible worlds with accessibility relations $\mathbf{K}_{Superman}$ (solid arrows) and $\mathbf{K}_{Lois}$ (dotted arrows). The proposition R means "the weather report for tomorrow is rain" and I means "Superman's secret identity is Clark Kent." All worlds are accessible to themselves; the arrows from a world to itself are not shown.

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 does not know whether R is true, or he could 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:

$$\mathbf{K}_a P \;\Rightarrow\; \mathbf{K}_a(\mathbf{K}_a P) \,.$$

LOGICAL
OMNISCIENCE

We can define similar axioms for belief (often denoted by **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.

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

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

$$\forall x \ \ x \in Persons \ \Rightarrow \ [\forall y \ \ HasMother(x,y) \ \Rightarrow \ y \in FemalePersons] \,.$$
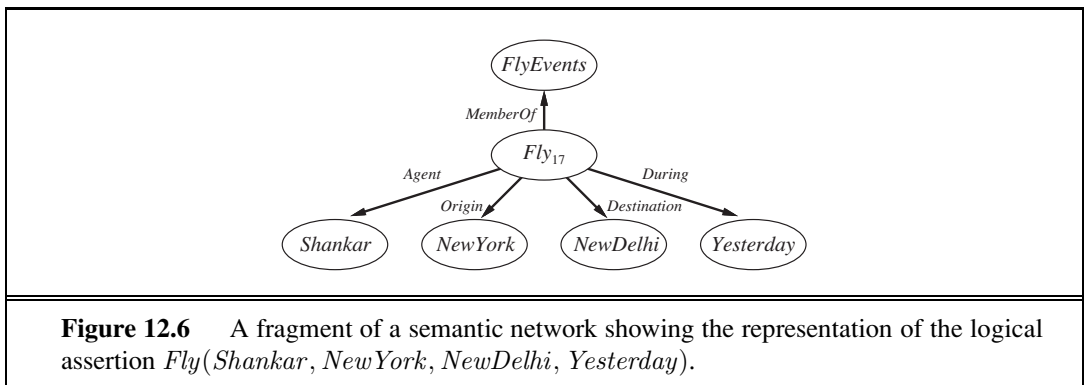
We might also want to assert that persons have two legs—that is,

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

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.

Figure 12.5   A semantic network with four objects (John, Mary, 1, and 2) and four categories. Relations are denoted by labeled links.

Figure 12.6   A fragment of a semantic network showing the representation of the logical assertion $Fly(Shankar, NewYork, NewDelhi, Yesterday)$.
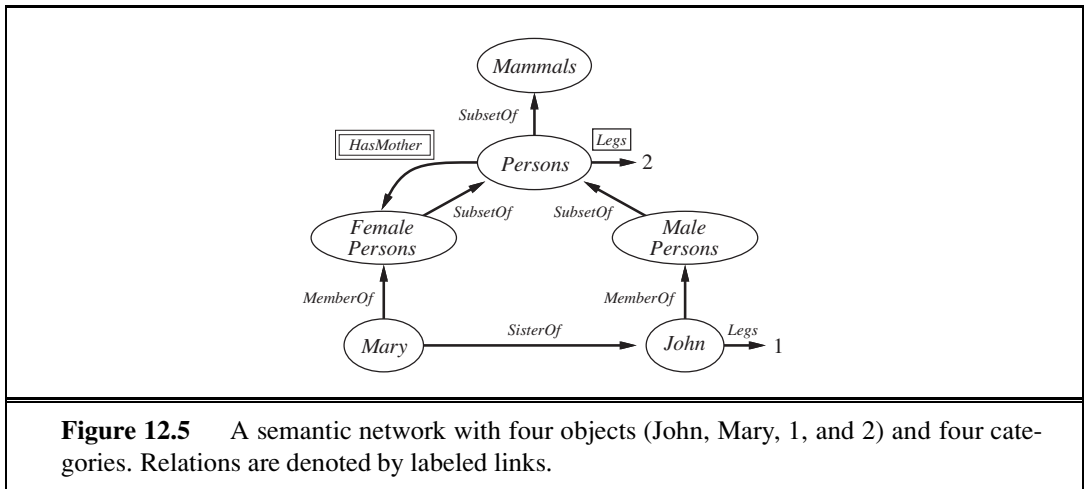
mechanism, compared with logical theorem proving, has been one of the main attractions of semantic networks.

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-

MULTIPLE
INHERITANCE

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

OVERRIDING

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.

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.

$$
\begin{aligned}
Concept \quad \rightarrow \quad & \textbf{Thing} \mid \; ConceptName \\
\mid \quad & \textbf{And}(Concept, \ldots) \\
\mid \quad & \textbf{All}(RoleName, Concept) \\
\mid \quad & \textbf{AtLeast}(Integer, RoleName) \\
\mid \quad & \textbf{AtMost}(Integer, RoleName) \\
\mid \quad & \textbf{Fills}(RoleName, IndividualName, \ldots) \\
\mid \quad & \textbf{SameAs}(Path, Path) \\
\mid \quad & \textbf{OneOf}(IndividualName, \ldots) \\
Path \quad \rightarrow \quad & [RoleName, \ldots]
\end{aligned}
$$

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

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

The equivalent in first-order logic would be

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

Notice that the description logic has an 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

$$
\begin{aligned}
And(&Man, AtLeast(3, Son), AtMost(2, Daughter), \\
&All(Son, And(Unemployed, Married, All(Spouse, Doctor))), \\
&All(Daughter, And(Professor, Fills(Department, Physics, Math)))) \, .
\end{aligned}
$$

We leave it as an exercise to translate this into first-order logic.

Perhaps the most important aspect of description logics is their emphasis on tractability of inference. A problem instance is solved by describing it and then asking if it is subsumed by one of several possible solution categories. In standard first-order logic systems, predicting the solution time is often impossible. It is frequently left to the user to engineer the representation to detour around sets of sentences that seem to be causing the system to take several weeks to solve a problem. The thrust in description logics, on the other hand, is to ensure that subsumption-testing can be solved in time polynomial in the size of the descriptions.[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

$$Bird(x) \land \neg Abnormal_1(x) \Rightarrow Flies(x) \ .$$

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:

$$Republican(Nixon) \land Quaker(Nixon) \ .$$
$$Republican(x) \land \neg Abnormal_2(x) \Rightarrow \neg Pacifist(x) \ .$$
$$Quaker(x) \land \neg Abnormal_3(x) \Rightarrow Pacifist(x) \ .$$

If we circumscribe $Abnormal_2$ and $Abnormal_3$, there are two preferred models: one in which $Abnormal_2(Nixon)$ and $Pacifist(Nixon)$ hold and one in which $Abnormal_3(Nixon)$ and $\neg Pacifist(Nixon)$ hold. Thus, the circumscriptive reasoner remains properly agnostic as to whether Nixon 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

$$P : J_1, \ldots, J_n/C$$

where P is called the prerequisite, C is the conclusion, and J_i are the justifications—if any one of them can be proven false, then the conclusion cannot be drawn. Any variable that

---

[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:

$$Republican(Nixon) \land Quaker(Nixon) .$$
$$Republican(x) : \neg Pacifist(x)/\neg Pacifist(x) .$$
$$Quaker(x) : Pacifist(x)/Pacifist(x) .$$

EXTENSION
To interpret what the default rules mean, we define the notion of an **extension** of a default theory to be a maximal set of consequences of the theory. That is, an extension S consists of the original known facts and a set of conclusions from the default rules, such that no additional conclusions can be drawn from S and the justifications of every default conclusion in S are consistent with S. As in the case of the preferred models in circumscription, we have two possible extensions for the Nixon diamond: one wherein he is a pacifist and one wherein he is not. Prioritized schemes exist in which some default rules can be given precedence over others, allowing some ambiguities to be resolved.

Since 1980, when nonmonotonic logics were first proposed, a great deal of progress has been made in understanding their mathematical properties. 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 informa-
BELIEF REVISION
tion. 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 TELL(KB, $\neg P$). To avoid creating a contradiction, we must first execute RETRACT(KB, P). This sounds easy enough.

---

[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.

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.

TRUTH
MAINTENANCE
SYSTEM

   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 RETRACT(KB, P_i) is made, the system reverts to the state just before P_i was added, thereby removing both P_i and any inferences that were derived from P_i. The sentences P_{i+1} through P_n can then be added again. This is simple, and it guarantees that the knowledge base will be consistent, but retracting P_i requires retracting and reasserting $n - i$ sentences as well as 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 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 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 $Site(Swimming, Pitesti)$, $Site(Athletics, Bucharest)$, and $Site(Equestrian, Arad)$. A great deal of reasoning must then be done to work out the logistical consequences and hence the desirability of this selection. If we want to consider $Site(Athletics, Sibiu)$ instead, the TMS avoids the need to start again from scratch. Instead, we simply retract $Site(Athletics, Bucharest)$ and assert $Site(Athletics, Sibiu)$ and the TMS takes care of the necessary revisions. Inference chains generated from the choice of Bucharest can be reused with Sibiu, provided that the conclusions are the same.

ATMS

An assumption-based truth maintenance system, or **ATMS**, 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 ba-

ASSUMPTION

sis for proving that P must be the case. But explanations can also include **assumptions**—sentences that are not known to be true, but would suffice to prove P if they were true. For example, one might not have enough information to prove that one's car won't start, but a reasonable explanation might include the assumption that the battery is dead. This, combined with knowledge of how cars operate, explains the observed nonbehavior. In most cases, we will prefer an explanation E that is minimal, meaning that there is no proper subset of E that is also an explanation. An ATMS can generate explanations for the "car won't start" problem by making assumptions (such as "gas in car" or "battery dead") in any order we like, even if some assumptions are contradictory. Then we look at the label for the sentence "car won't start" to read off the sets of assumptions that would justify the sentence.

The exact algorithms used to implement truth maintenance systems are a little complicated, and we do not cover them here. The computational complexity of the truth maintenance problem is at least as great as that of propositional inference—that is, NP-hard. Therefore, you should not expect truth maintenance to be a panacea. When used carefully, however, a TMS can provide a substantial increase in the ability of a logical system to handle complex environments and hypotheses.

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 `<a href="`*url*`">`*Books*`</a>` 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...</a`"
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:

$$Amazon \in OnlineStores \land Homepage(Amazon, \text{``amazon.com''}) \,.$$
$$Ebay \in OnlineStores \land Homepage(Ebay, \text{``ebay.com''}) \,.$$
$$ExampleStore \in OnlineStores \land Homepage(ExampleStore, \text{``example.com''}) \,.$$

These stores classify their goods into product categories, and provide links to the major cat-
egories 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:

$$Relevant(page, query) \iff$$
$$\exists\, store, home \quad store \in OnlineStores \land Homepage(store, home)$$
$$\land \exists\, url, url_2 \quad RelevantChain(home, url_2, query) \land Link(url_2, url)$$
$$\land page = Contents(url) \,.$$

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:

$$RelevantChain(start, end, query) \iff (start = end)$$
$$\lor (\exists\, u, text \quad LinkText(start, u, text) \land RelevantCategoryName(query, text)$$
$$\land RelevantChain(u, end, query)) \,.$$

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

$Books \subset Products$
$MusicRecordings \subset Products$
$\quad MusicCDs \subset MusicRecordings$
$Electronics \subset Products$
$\quad DigitalCameras \subset Electronics$
$\quad StereoEquipment \subset Electronics$
$\quad Computers \subset Electronics$
$\qquad DesktopComputers \subset Computers$
$\qquad LaptopComputers \subset Computers$
$\dots$

(a)

$Name(\text{“books”}, Books)$
$Name(\text{“music”}, MusicRecordings)$
$\quad Name(\text{“CDs”}, MusicCDs)$
$Name(\text{“electronics”}, Electronics)$
$\quad Name(\text{“digital cameras”}, DigitalCameras)$
$\quad Name(\text{“stereos”}, StereoEquipment)$
$\quad Name(\text{“computers”}, Computers)$
$\qquad Name(\text{“desktops”}, DesktopComputers)$
$\qquad Name(\text{“laptops”}, LaptopComputers)$
$\qquad Name(\text{“notebooks”}, LaptopComputers)$
$\dots$

(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:

$$RelevantCategoryName(query, text) \Leftrightarrow$$
$$\exists c_1, c_2 \ Name(query, c_1) \wedge Name(text, c_2) \wedge (c_1 \subseteq c_2 \vee c_2 \subseteq c_1) \ . \qquad (12.1)$$

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

$$Name(\text{“CDs”}, CertificatesOfDeposit)$$

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:

$\exists c, offer \quad c \in LaptopComputers \land offer \in ProductOffers \land$
$\quad Manufacturer(c, IBM) \land Model(c, ThinkBook970) \land$
$\quad ScreenSize(c, Inches(14)) \land ScreenType(c, ColorLCD) \land$
$\quad MemorySize(c, Gigabytes(2)) \land CPUSpeed(c, GHz(1.2)) \land$
$\quad OfferedProduct(offer, c) \land Store(offer, GenStore) \land$
$\quad URL(offer, \text{"example.com/computers/34356.html"}) \land$
$\quad Price(offer, \$(399)) \land Date(offer, Today) .$

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 Linne (1707–1778). The problems associated with natural kinds and inexact category boundaries have been addressed by Wittgenstein (1953), Quine (1953), Lakoff (1987), and Schwartz (1977), among others.

Interest in larger-scale ontologies is increasing, 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 $MadeOf(Butter_3, Butter)$ rather than $Butter_3 \in Butter$. This leads to a substance hierarchy in which, for example, $UnsaltedButter$ is a more specific substance than $Butter$. The position adopted in this chapter, in which substances are categories of objects, was championed by Richard Montague (1973). It has also been adopted in the CYC project. Copeland (1993) mounts a serious, but not invincible, attack. The alternative approach mentioned in the chapter, in which butter is one object consisting of all buttery objects in the universe, was proposed

MEREOLOGY
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

SYNTACTIC THEORY
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.
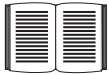
12.13 Investigate ways to extend the event calculus to handle *simultaneous* events. Is it possible to avoid a combinatorial explosion of axioms?

**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.
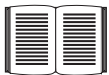
12.17 The two preceding exercises assume a fairly primitive notion of ownership. For example, the buyer starts by *owning* the dollar bills. This picture begins to break down when, for example, one's money is in the bank, because there is no longer any specific collection of dollar bills that one owns. 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 works 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:

$$
\begin{array}{lcl}
Description & \rightarrow & Category\ [Connector\ Modifier]* \\
Connector & \rightarrow & \text{``with''} \mid \text{``and''} \mid \text{``,''} \\
Modifier & \rightarrow & Attribute \mid Attribute\ Op\ Value \\
Op & \rightarrow & \text{``=''} \mid \text{``>''} \mid \text{``<''}
\end{array}
$$

Here, *Category* names a product category, *Attribute* is some feature such as "CPU" or "price," and *Value* is the target value for the attribute. So the query "computer with at least a 2.5 GHz CPU for under \$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 taxi!automated 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:

$$Toothache \ \Rightarrow \ Cavity \ .$$

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:

$$Toothache \ \Rightarrow \ Cavity \lor GumProblem \lor Abscess \ldots$$

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:

$$Cavity \ \Rightarrow \ Toothache \ .$$

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

- **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
PRACTICAL
IGNORANCE

- **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

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.

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

$$Decision\ theory = probability\ theory + utility\ theory\ .$$

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

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.

SAMPLE SPACE

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:

$$0 \le P(\omega) \le 1 \text{ for every } \omega \text{ and } \sum_{\omega \in \Omega} P(\omega) = 1 . \tag{13.1}$$

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.

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

EVENT

---

[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:

$$\text{For any proposition } \phi, \ P(\phi) = \sum_{\omega \in \phi} P(\omega) \ . \tag{13.2}$$

For example, when rolling fair dice, we have $P(Total = 11) = P((5,6)) + P((6,5)) = 1/36 + 1/36 = 1/18$. 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 $P(doubles) = 1/4$ 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 $P(Total = 11)$ and $P(doubles)$ 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 $P(doubles \mid Die_1 = 5)$, where the " $\mid$ " is pronounced "given." Similarly, if I am going to the dentist for a regular checkup, the probability $P(cavity) = 0.2$ might be of interest; but if I go to the dentist because I have a toothache, it's $P(cavity \mid toothache) = 0.6$ that matters. Note that the precedence of " $\mid$ " is such that any expression of the form $P(\ldots \mid \ldots)$ always means $P((\ldots)|(\ldots))$.

    It is important to understand that $P(cavity) = 0.2$ 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 $P(cavity \mid toothache) = 0.6$ 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 $P(cavity|toothache \wedge \neg cavity) = 0$.

    Mathematically speaking, conditional probabilities are defined in terms of unconditional probabilities as follows: for any propositions a and b, we have

$$P(a \mid b) = \frac{P(a \wedge b)}{P(b)} \ , \tag{13.3}$$

which holds whenever $P(b) > 0$. For example,

$$P(doubles \mid Die_1 = 5) = \frac{P(doubles \wedge Die_1 = 5)}{P(Die_1 = 5)} \ .$$

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**:

$$P(a \wedge b) = P(a \mid b)P(b) ,$$

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 Die_1 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 $\{2, \ldots, 12\}$ and the domain of Die_1 is $\{1, \ldots, 6\}$. 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:

$$P(cavity \mid \neg toothache \wedge teen) = 0.1 .$$

Sometimes we will want to talk about the probabilities of *all* the possible values of a random variable. We could write:

$$P(Weather = sunny) = 0.6$$
$$P(Weather = rain) = 0.1$$
$$P(Weather = cloudy) = 0.29$$
$$P(Weather = snow) = 0.01 ,$$

but as an abbreviation we will allow

$$\mathbf{P}(Weather) = \langle 0.6, 0.1, 0.29, 0.01 \rangle ,$$

where the bold **P** indicates that the result is a vector of numbers, and where we assume a pre-defined ordering $\langle sunny, rain, cloudy, snow \rangle$ on the domain of *Weather*. We say that the **P** statement defines a **probability distribution** for the random variable *Weather*. The **P** notation is also used for conditional distributions: $\mathbf{P}(X \mid Y)$ gives the values of $P(X = x_i \mid Y = y_j)$ 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

$$P(NoonTemp = x) = Uniform_{[18C,26C]}(x)$$

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:

$$P(x) = \lim_{dx \to 0} P(x \leq X \leq x + dx)/dx .$$

For *NoonTemp* we have

$$P(NoonTemp = x) = Uniform_{[18C,26C]}(x) = \begin{cases} \frac{1}{8C} & \text{if } 18C \leq x \leq 26C \\ 0 & \text{otherwise} \end{cases} ,$$

where C stands for centigrade (not for a constant). In $P(NoonTemp = 20.18C) = \frac{1}{8C}$, 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.

In addition to distributions on single variables, we need notation for distributions on multiple variables. Commas are used for this. For example, $\mathbf{P}(Weather, Cavity)$ 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; $\mathbf{P}(sunny, Cavity)$ would be a two-element vector giving the probabilities of a sunny day with a cavity and a sunny day with no cavity. The **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:

$$\mathbf{P}(Weather, Cavity) = \mathbf{P}(Weather \mid Cavity)\mathbf{P}(Cavity) ,$$

instead of as these $4 \times 2 = 8$ equations (using abbreviations W and C):

$$P(W = sunny \wedge C = true) = P(W = sunny | C = true) \, P(C = true)$$
$$P(W = rain \wedge C = true) = P(W = rain | C = true) \, P(C = true)$$
$$P(W = cloudy \wedge C = true) = P(W = cloudy | C = true) \, P(C = true)$$
$$P(W = snow \wedge C = true) = P(W = snow | C = true) \, P(C = true)$$
$$P(W = sunny \wedge C = false) = P(W = sunny | C = false) \, P(C = false)$$
$$P(W = rain \wedge C = false) = P(W = rain | C = false) \, P(C = false)$$
$$P(W = cloudy \wedge C = false) = P(W = cloudy | C = false) \, P(C = false)$$
$$P(W = snow \wedge C = false) = P(W = snow | C = false) \, P(C = false) \ .$$

As a degenerate case, $\mathbf{P}(sunny, cavity)$ 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 $P(sunny, cavity)$ or $P(sunny \wedge cavity)$. We will sometimes use $\mathbf{P}$ notation to derive results about individual P values, and when we say "$\mathbf{P}(sunny) = 0.6$" it is really an abbreviation for "$\mathbf{P}(sunny)$ is the one-element vector $\langle 0.6 \rangle$, which means that $P(sunny) = 0.6$."

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 $\mathbf{P}(Cavity, Toothache, Weather)$. 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.

FULL JOINT PROBABILITY DISTRIBUTION

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**:

$$P(a \lor b) = P(a) + P(b) - P(a \land b) . \tag{13.4}$$

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 $a \lor b$ holds; but summing the two sets of cases counts their intersection twice, so we need to subtract $P(a \land b)$. 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 $\neg(A \land B)$, 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)?

$$\begin{aligned} P(a) &= 0.4 & P(a \land b) &= 0.0 \\ P(b) &= 0.3 & P(a \lor b) &= 0.8 . \end{aligned} \tag{13.5}$$

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.

	toothache		¬toothache	
	catch	¬catch	catch	¬catch
cavity	0.108	0.012	0.072	0.008
¬cavity	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* ∨ *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*:

MARGINAL
PROBABILITY

$$P(cavity) = 0.108 + 0.012 + 0.072 + 0.008 = 0.2 .$$

MARGINALIZATION

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**:

$$\mathbf{P}(\mathbf{Y}) = \sum_{\mathbf{z} \in \mathbf{Z}} \mathbf{P}(\mathbf{Y}, \mathbf{z}) , \qquad (13.6)$$

where $\sum_{\mathbf{z} \in \mathbf{Z}}$ means to sum over all the possible combinations of values of the set of variables **Z**. We sometimes abbreviate this as $\sum_{\mathbf{z}}$, leaving **Z** implicit. We just used the rule as

$$\mathbf{P}(Cavity) = \sum_{\mathbf{z} \in \{Catch, Toothache\}} \mathbf{P}(Cavity, \mathbf{z}) . \qquad (13.7)$$

A variant of this rule involves conditional probabilities instead of joint probabilities, using the product rule:

$$\mathbf{P}(\mathbf{Y}) = \sum_{\mathbf{z}} \mathbf{P}(\mathbf{Y} \,|\, \mathbf{z}) P(\mathbf{z}) . \qquad (13.8)$$

CONDITIONING

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.

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:

$$P(cavity \mid toothache) \;=\; \frac{P(cavity \wedge toothache)}{P(toothache)}$$

$$=\; \frac{0.108 + 0.012}{0.108 + 0.012 + 0.016 + 0.064} = 0.6 \;.$$

Just to check, we can also compute the probability that there is no cavity, given a toothache:

$$P(\neg cavity \mid toothache) \;=\; \frac{P(\neg cavity \wedge toothache)}{P(toothache)}$$

$$=\; \frac{0.016 + 0.064}{0.108 + 0.012 + 0.016 + 0.064} = 0.4 \;.$$

The two values sum to 1.0, as they should. Notice that in these two calculations the term $1/P(toothache)$ remains constant, no matter which value of $Cavity$ we calculate. In fact, it can be viewed as a **normalization** constant for the distribution $\mathbf{P}(Cavity \mid toothache)$, 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:

NORMALIZATION

$$\mathbf{P}(Cavity \mid toothache) = \alpha\, \mathbf{P}(Cavity, toothache)$$
$$= \; \alpha\,[\mathbf{P}(Cavity, toothache, catch) + \mathbf{P}(Cavity, toothache, \neg catch)]$$
$$= \; \alpha\,[\langle 0.108, 0.016 \rangle + \langle 0.012, 0.064 \rangle] = \alpha\, \langle 0.12, 0.08 \rangle = \langle 0.6, 0.4 \rangle \;.$$

In other words, we can calculate $\mathbf{P}(Cavity \mid toothache)$ even if we don't know the value of $P(toothache)$! We temporarily forget about the factor $1/P(toothache)$ and add up the values for $cavity$ and $\neg cavity$, 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 $P(toothache)$) 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

$$\mathbf{P}(X \mid \mathbf{e}) = \alpha\, \mathbf{P}(X, \mathbf{e}) = \alpha \sum_{\mathbf{y}} \mathbf{P}(X, \mathbf{e}, \mathbf{y}) \;, \tag{13.9}$$

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 $\mathbf{P}(Toothache, Catch, Cavity, Weather)$, 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 $P(toothache, catch, cavity, cloudy)$ and $P(toothache, catch, cavity)$ related? We can use the product rule:

$$P(toothache, catch, cavity, cloudy)$$
$$= P(cloudy \mid toothache, catch, cavity)P(toothache, catch, cavity) \ .$$

Now, unless one is in the deity business, one should not imagine that one's dental problems influence the weather. 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:

$$P(cloudy \mid toothache, catch, cavity) = P(cloudy) \ . \tag{13.10}$$

From this, we can deduce

$$P(toothache, catch, cavity, cloudy) = P(cloudy)P(toothache, catch, cavity) \ .$$

A similar equation exists for *every entry* in $\mathbf{P}(Toothache, Catch, Cavity, Weather)$. In fact, we can write the general equation

$$\mathbf{P}(Toothache, Catch, Cavity, Weather) = \mathbf{P}(Toothache, Catch, Cavity)\mathbf{P}(Weather) \ .$$

Thus, the 32-element table for four variables can be constructed from one 8-element table and one 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

$$P(a \mid b) = P(a) \quad \text{or} \quad P(b \mid a) = P(b) \quad \text{or} \quad P(a \wedge b) = P(a)P(b) \ . \tag{13.11}$$

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

Figure 13.4   Two examples of factoring a large joint distribution into smaller distributions, using absolute independence. (a) Weather and dental problems are independent. (b) Coin flips are independent.
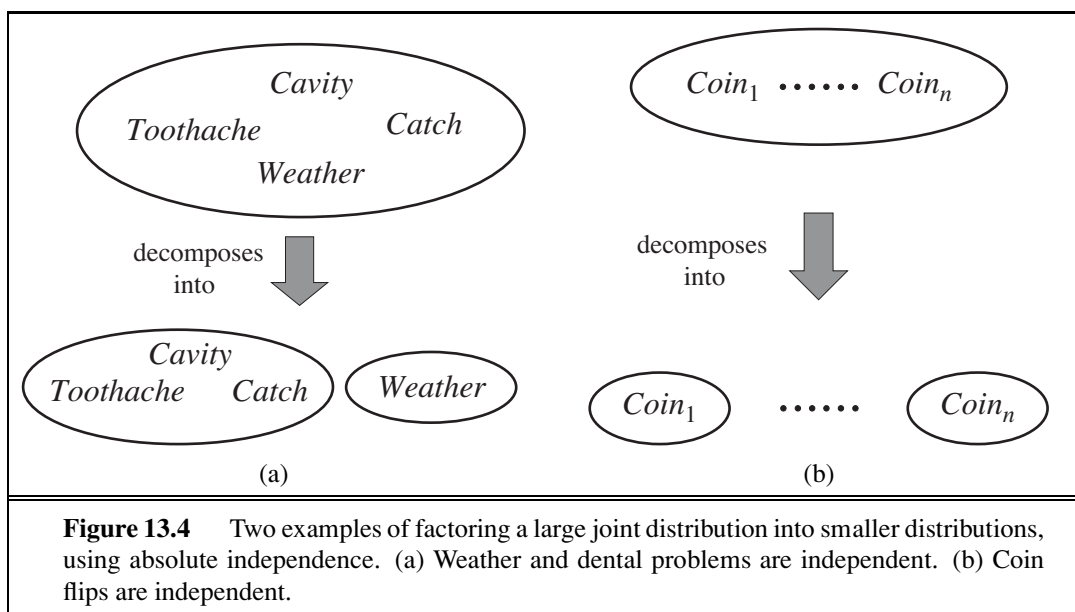
into independent subsets, then the full joint distribution can be *factored* into separate joint distributions on those subsets. For example, the full joint distribution on the outcome of n independent coin flips, $\mathbf{P}(C_1, \ldots, C_n)$, has 2^n entries, but it can be represented as the product 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:

$$P(a \wedge b) = P(a\,|\,b)P(b) \qquad \text{and} \qquad P(a \wedge b) = P(b\,|\,a)P(a) \ .$$

Equating the two right-hand sides and dividing by $P(a)$, we get

$$P(b\,|\,a) = \frac{P(a\,|\,b)P(b)}{P(a)} \ . \tag{13.12}$$

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:

$$\mathbf{P}(Y \mid X) = \frac{\mathbf{P}(X \mid Y)\mathbf{P}(Y)}{\mathbf{P}(X)} \, ,$$

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**:

$$\mathbf{P}(Y \mid X, \mathbf{e}) = \frac{\mathbf{P}(X \mid Y, \mathbf{e})\mathbf{P}(Y \mid \mathbf{e})}{\mathbf{P}(X \mid \mathbf{e})} \, . \tag{13.13}$$

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 $P(b \mid a)$ in terms of three terms: $P(a \mid b)$, $P(b)$, 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

$$P(cause \mid effect) = \frac{P(effect \mid cause)P(cause)}{P(effect)} \, .$$

CAUSAL

DIAGNOSTIC

The conditional probability $P(effect \mid cause)$ quantifies the relationship in the **causal** direction, whereas $P(cause \mid effect)$ describes the **diagnostic** direction. In a task such as medical diagnosis, we often have conditional probabilities on causal relationships (that is, the doctor knows $P(symptoms \mid disease)$) and want to derive a diagnosis, $P(disease \mid symptoms)$. 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

$$\begin{aligned} P(s \mid m) &= 0.7 \\ P(m) &= 1/50000 \\ P(s) &= 0.01 \\ P(m \mid s) &= \frac{P(s \mid m)P(m)}{P(s)} = \frac{0.7 \times 1/50000}{0.01} = 0.0014 \, . \end{aligned} \tag{13.14}$$

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

$$\mathbf{P}(M \mid s) = \alpha \left\langle P(s \mid m)P(m), P(s \mid \neg m)P(\neg m) \right\rangle .$$

Thus, to use this approach we need to estimate $P(s \mid \neg m)$ 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

$$\mathbf{P}(Y \mid X) = \alpha \, \mathbf{P}(X \mid Y)\mathbf{P}(Y) , \qquad\qquad\qquad (13.15)$$

where α is the normalization constant needed to make the entries in $\mathbf{P}(Y \mid X)$ 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 $P(m \mid s)$ directly from statistical observation of patients before the epidemic will have no idea how to update the value, but the doctor who computes $P(m \mid s)$ from the other three values will see that $P(m \mid s)$ should go up proportionately with $P(m)$. Most important, the causal information $P(s \mid m)$ 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 $P(effect \mid cause)$. 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:

$$\mathbf{P}(Cavity \mid toothache \wedge catch) = \alpha \left\langle 0.108, 0.016 \right\rangle \approx \left\langle 0.871, 0.129 \right\rangle .$$

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:

$$\mathbf{P}(Cavity \mid toothache \wedge catch)$$
$$= \alpha \, \mathbf{P}(toothache \wedge catch \mid Cavity) \, \mathbf{P}(Cavity) . \qquad\qquad (13.16)$$

For this reformulation to work, we need to know the conditional probabilities of the conjunction $toothache \wedge catch$ 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

$$\mathbf{P}(toothache \wedge catch \mid Cavity) = \mathbf{P}(toothache \mid Cavity)\mathbf{P}(catch \mid Cavity) . \quad (13.17)$$

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:

$$\mathbf{P}(Cavity \mid toothache \wedge catch)$$
$$= \alpha\,\mathbf{P}(toothache \mid Cavity)\,\mathbf{P}(catch \mid Cavity)\,\mathbf{P}(Cavity) . \quad (13.18)$$

Now the information requirements are the same as for inference, using each piece of evidence separately: the prior probability $\mathbf{P}(Cavity)$ 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*:

$$\mathbf{P}(Toothache, Catch \mid Cavity) = \mathbf{P}(Toothache \mid Cavity)\mathbf{P}(Catch \mid Cavity) . \quad (13.19)$$

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:

$$\mathbf{P}(Toothache, Catch, Cavity)$$
$$= \mathbf{P}(Toothache, Catch \mid Cavity)\mathbf{P}(Cavity) \quad \text{(product rule)}$$
$$= \mathbf{P}(Toothache \mid Cavity)\mathbf{P}(Catch \mid Cavity)\mathbf{P}(Cavity) \quad \text{(using 13.19)}.$$

(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

---

[5] We assume that the patient and dentist are distinct individuals.

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.

SEPARATION

The dentistry example illustrates a commonly occurring pattern in which a single cause directly influences a number of effects, all of which are conditionally independent, given the cause. The full joint distribution can be written as

$$\mathbf{P}(Cause, Effect_1, \ldots, Effect_n) = \mathbf{P}(Cause) \prod_i \mathbf{P}(Effect_i \mid Cause) \,.$$

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.

Figure 13.5   (a) After finding a breeze in both [1,2] and [2,1], the agent is stuck—there is no safe place to explore. (b) Division of the squares into *Known*, *Frontier*, and *Other*, for a query about [1,3].

- We also have Boolean variables B_{ij} that are true iff square $[i, j]$ is breezy; we include these variables only for the observed squares—in this case, [1,1], [1,2], and [2,1].

The next step is to specify the full joint distribution, $\mathbf{P}(P_{1,1}, \ldots, P_{4,4}, B_{1,1}, B_{1,2}, B_{2,1})$. Applying the product rule, we have

$$\mathbf{P}(P_{1,1}, \ldots, P_{4,4}, B_{1,1}, B_{1,2}, B_{2,1}) = \\ \mathbf{P}(B_{1,1}, B_{1,2}, B_{2,1} \mid P_{1,1}, \ldots, P_{4,4})\mathbf{P}(P_{1,1}, \ldots, P_{4,4}) \ .$$

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,

$$\mathbf{P}(P_{1,1}, \ldots, P_{4,4}) = \prod_{i,j \, = \, 1,1}^{4,4} \mathbf{P}(P_{i,j}) \ . \tag{13.20}$$

For a particular configuration with exactly n pits, $P(P_{1,1}, \ldots, P_{4,4}) = 0.2^n \times 0.8^{16-n}$.

   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

$$\mathbf{P}(P_{1,3} \mid known, b) = \alpha \sum_{unknown} \mathbf{P}(P_{1,3}, unknown, known, b) \ .$$

The full joint probabilities have already been specified, so we are done—that is, unless we care about computation. There are 12 unknown squares; hence the summation contains $2^{12} = 4096$ terms. In general, the summation grows exponentially with the number of squares.

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:
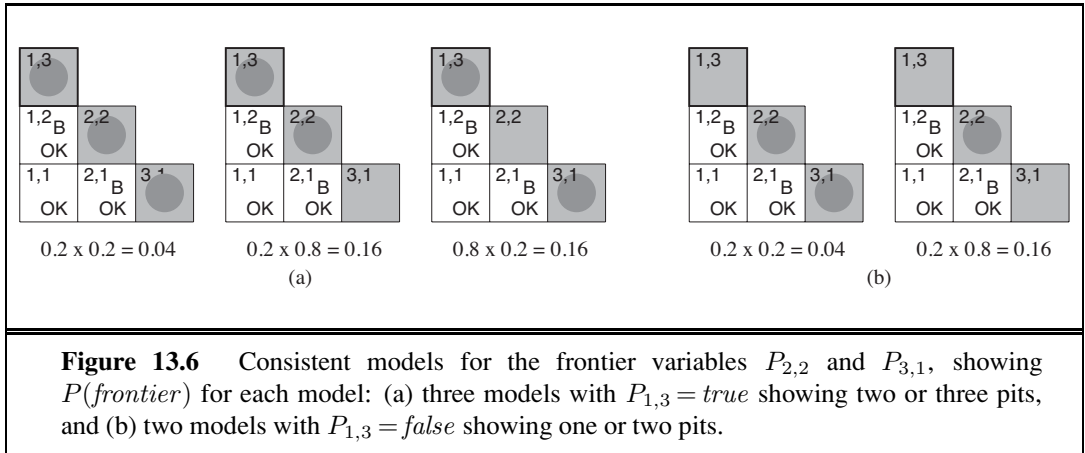
$$\mathbf{P}(P_{1,3} \mid known, b)$$
$$= \alpha \sum_{unknown} \mathbf{P}(P_{1,3}, known, b, unknown) \qquad \text{(by Equation (13.9))}$$
$$= \alpha \sum_{unknown} \mathbf{P}(b \mid P_{1,3}, known, unknown)\mathbf{P}(P_{1,3}, known, unknown)$$
$$\qquad \text{(by the product rule)}$$
$$= \alpha \sum_{frontier} \sum_{other} \mathbf{P}(b \mid known, P_{1,3}, frontier, other)\mathbf{P}(P_{1,3}, known, frontier, other)$$
$$= \alpha \sum_{frontier} \sum_{other} \mathbf{P}(b \mid known, P_{1,3}, frontier)\mathbf{P}(P_{1,3}, known, frontier, other) \ ,$$

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:

$$\mathbf{P}(P_{1,3} \mid known, b)$$
$$= \alpha \sum_{frontier} \mathbf{P}(b \mid known, P_{1,3}, frontier) \sum_{other} \mathbf{P}(P_{1,3}, known, frontier, other) \ .$$

By independence, as in Equation (13.20), the prior term can be factored, and then the terms can be reordered:

$$\mathbf{P}(P_{1,3} \mid known, b)$$
$$= \alpha \sum_{frontier} \mathbf{P}(b \mid known, P_{1,3}, frontier) \sum_{other} \mathbf{P}(P_{1,3})P(known)P(frontier)P(other)$$
$$= \alpha \, P(known)\mathbf{P}(P_{1,3}) \sum_{frontier} \mathbf{P}(b \mid known, P_{1,3}, frontier)P(frontier) \sum_{other} P(other)$$
$$= \alpha' \, \mathbf{P}(P_{1,3}) \sum_{frontier} \mathbf{P}(b \mid known, P_{1,3}, frontier)P(frontier) \ ,$$

Figure 13.6   Consistent models for the frontier variables $P_{2,2}$ and $P_{3,1}$, showing $P(frontier)$ for each model: (a) three models with $P_{1,3} = true$ showing two or three pits, and (b) two models with $P_{1,3} = false$ showing one or two pits.

where the last step folds $P(known)$ into the normalizing constant and uses the fact that $\sum_{other} P(other)$ equals 1.

Now, there are just four terms in the summation over the 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 $\mathbf{P}(b \mid known, P_{1,3}, frontier)$ 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—$P(frontier)$—are shown in Figure 13.6. We have

$$\mathbf{P}(P_{1,3} \mid known, b) = \alpha' \langle 0.2(0.04 + 0.16 + 0.16),\ 0.8(0.04 + 0.16) \rangle \approx \langle 0.31, 0.69 \rangle .$$

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).

<div style="margin-left:0">PRINCIPLE OF INDIFFERENCE

PRINCIPLE OF INSUFFICIENT REASON</div>

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**.

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 $P(a \mid b \wedge a) = 1$.

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 $P(a \mid b, c) = P(b \mid a, c)$, then $P(a \mid c) = P(b \mid c)$
   b. If $P(a \mid b, c) = P(a)$, then $P(b \mid c) = P(b)$
   c. If $P(a \mid b) = P(a)$, then $P(a \mid b, c) = P(a \mid c)$

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.

ATOMIC EVENT

   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.

   a. How many atomic events are there in the joint probability distribution (i.e., how many five-card hands are there)?
   b. What is the probability of each atomic event?
   c. What is the probability of being dealt a royal straight flush? Four of a kind?

13.8 Given the full joint distribution shown in Figure 13.3, calculate the following:

   a. $\mathbf{P}(toothache)$.
   b. $\mathbf{P}(Cavity)$.
   c. $\mathbf{P}(Toothache \mid cavity)$.
   d. $\mathbf{P}(Cavity \mid toothache \vee catch)$.

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

   a. Compute the expected "payback" percentage of the machine. In other words, for each coin played, what is the expected coin return?
   b. Compute the probability that playing the slot machine once will result in a win.
   c. 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:

   $$\mathbf{P}(X, Y \mid \mathbf{e}) = \mathbf{P}(X \mid Y, \mathbf{e})\mathbf{P}(Y \mid \mathbf{e}) \ .$$

   b. Prove the conditionalized version of Bayes' rule in Equation (13.13).

13.17   Show that the statement of conditional independence

   $$\mathbf{P}(X, Y \mid Z) = \mathbf{P}(X \mid Z)\mathbf{P}(Y \mid Z)$$

is equivalent to each of the statements

   $$\mathbf{P}(X \mid Y, Z) = \mathbf{P}(X \mid Z) \quad \text{and} \quad \mathbf{P}(B \mid X, Z) = \mathbf{P}(Y \mid Z) \ .$$

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 $P(s \mid \neg m)$, and use it to calculate unnormalized values for $P(m \mid s)$ and $P(\neg m \mid s)$ (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 $\mathbf{P}(P_{1,1}, \ldots, P_{4,4})$ 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 BAYESIAN NETWORK 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.

---

[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 $\mathbf{P}(X_i \mid Parents(X_i))$ 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
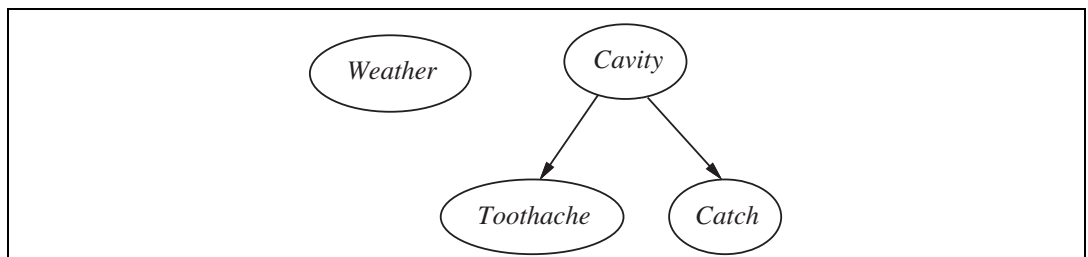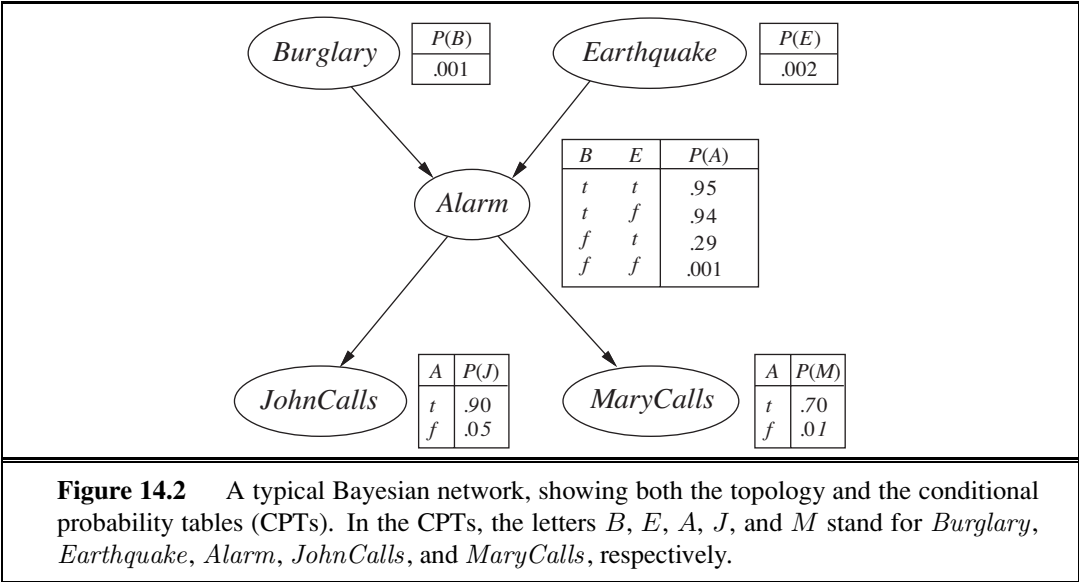

Figure 14.1   A simple Bayesian network in which *Weather* is independent of the other three variables and *Toothache* and *Catch* are conditionally independent, given *Cavity*.

Figure 14.2   A typical Bayesian network, showing both the topology and the conditional probability tables (CPTs). In the CPTs, the letters B, E, A, J, and M stand for *Burglary*, *Earthquake*, *Alarm*, *JohnCalls*, and *MaryCalls*, respectively.

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 $\mathbf{P}(X_i \mid Parents(X_i))$; this is a true statement, but until we assign semantics to the network as a whole, we should think of them just as numbers $\theta(X_i \mid Parents(X_i))$.

A generic entry in the joint distribution is the probability of a conjunction of particular assignments to each variable, such as $P(X_1 = x_1 \wedge \ldots \wedge X_n = x_n)$. We use the notation $P(x_1, \ldots, x_n)$ as an abbreviation for this. The value of this entry is given by the formula

$$P(x_1, \ldots, x_n) = \prod_{i=1}^{n} \theta(x_i \mid parents(X_i)) \, , \qquad (14.1)$$

where $parents(X_i)$ denotes the values of $Parents(X_i)$ that appear in $x_1, \ldots, x_n$. 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 $\theta(X_i \mid Parents(X_i))$ are exactly the conditional probabilities $\mathbf{P}(X_i \mid Parents(X_i))$ implied by the joint distribution (see Exercise 14.2). Hence, we can rewrite Equation (14.1) as

$$P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i \mid parents(X_i)) \, . \qquad (14.2)$$

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):

$$P(j, m, a, \neg b, \neg e) = P(j \mid a)P(m \mid a)P(a \mid \neg b \wedge \neg e)P(\neg b)P(\neg e)$$
$$= 0.90 \times 0.70 \times 0.001 \times 0.999 \times 0.998 = 0.000628 \ .$$

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):

$$P(x_1, \ldots, x_n) = P(x_n \mid x_{n-1}, \ldots, x_1)P(x_{n-1}, \ldots, x_1) \ .$$

Then we repeat the process, reducing each conjunctive probability to a conditional probability and a smaller conjunction. We end up with one big product:

$$P(x_1, \ldots, x_n) = P(x_n \mid x_{n-1}, \ldots, x_1)P(x_{n-1} \mid x_{n-2}, \ldots, x_1) \ \cdots \ P(x_2 \mid x_1)P(x_1)$$
$$= \prod_{i=1}^{n} P(x_i \mid x_{i-1}, \ldots, x_1) \ .$$

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,

$$\mathbf{P}(X_i \mid X_{i-1}, \ldots, X_1) = \mathbf{P}(X_i \mid Parents(X_i)) \ , \tag{14.3}$$

provided that $Parents(X_i) \subseteq \{X_{i-1}, \ldots, X_1\}$. 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, $\{X_1, \ldots, X_n\}$. 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 $X_1, \ldots, X_{i-1}$, 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 $X_1, \ldots, X_{i-1}$ that *directly influence* X_i. For example, suppose we have completed the network in Figure 14.2 except for the choice of parents for *MaryCalls*. *MaryCalls* is certainly influenced by whether there is a *Burglary* or an *Earthquake*, but not *directly* influenced. Intuitively, our knowledge of the domain tells us that these events influence Mary's calling behavior only through their effect on the alarm. Also, given the state of the alarm, whether John calls has no influence on Mary's calling. Formally speaking, we believe that the following conditional independence statement holds:

$$\mathbf{P}(MaryCalls \mid JohnCalls, Alarm, Earthquake, Burglary) = \mathbf{P}(MaryCalls \mid Alarm) \ .$$

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.*
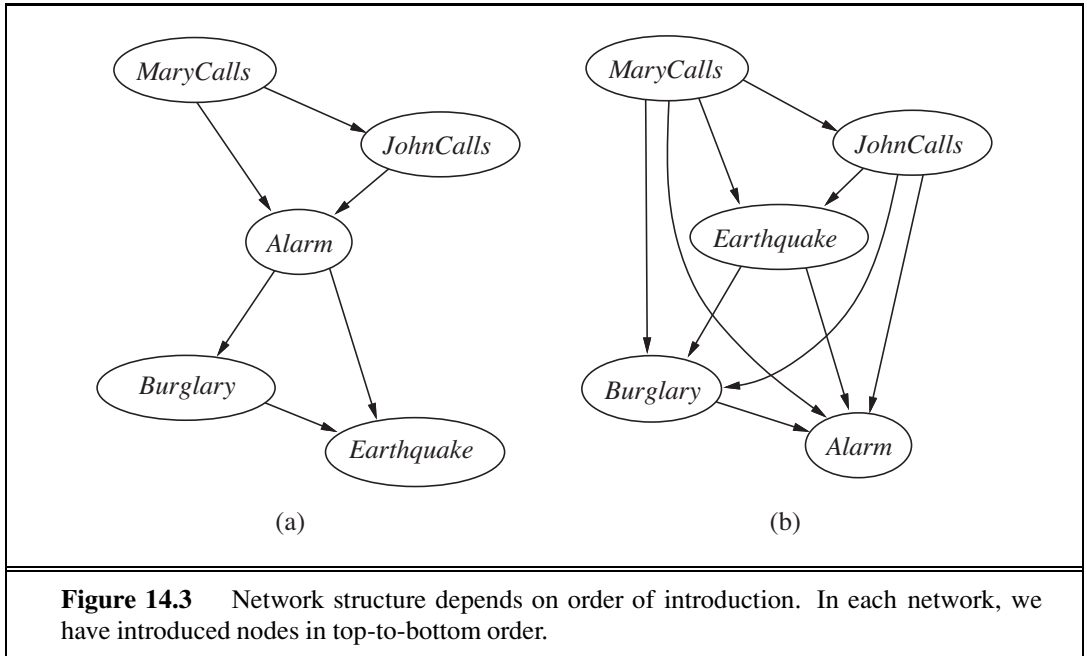
Compactness and node ordering

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

Figure 14.3   Network structure depends on order of introduction. In each network, we have introduced nodes in top-to-bottom order.

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:

$$\mathbf{P}(Burglary \mid Alarm, JohnCalls, MaryCalls) = \mathbf{P}(Burglary \mid Alarm) \ .$$

  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.

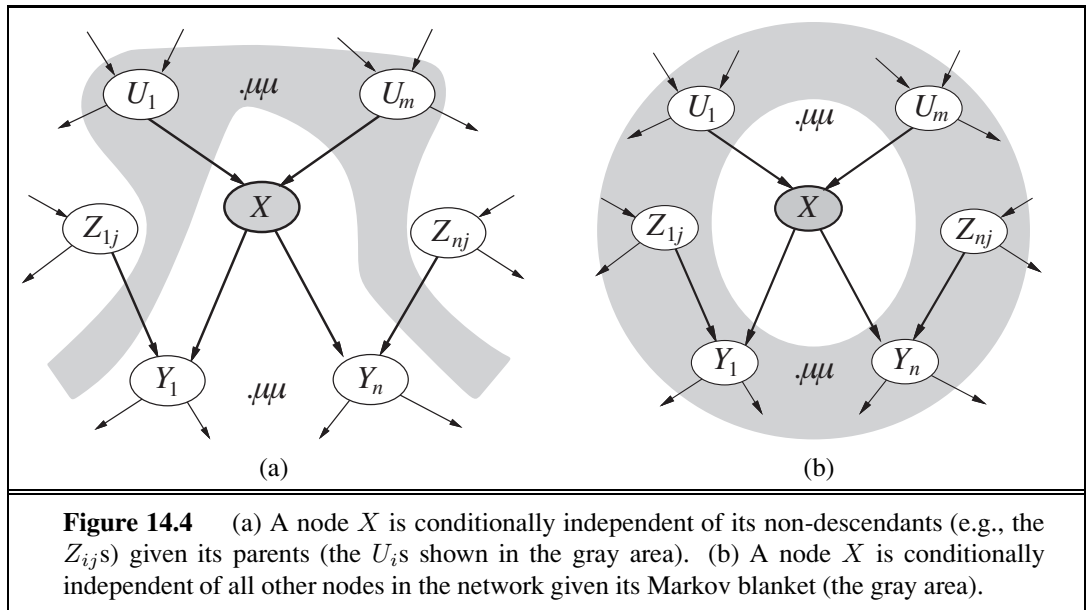### 14.2.2   Conditional independence relations in Bayesian networks

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 semantics[2] 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 $\theta(X_i \mid Parents(X_i))$ as specifications of conditional probabilities $\mathbf{P}(X_i \mid Parents(X_i))$, 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).

DESCENDANT

MARKOV BLANKET

---

[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.

Figure 14.4 (a) A node X is conditionally independent of its non-descendants (e.g., the Z_{ij}s) given its parents (the U_is shown in the gray area). (b) A node X is conditionally independent of all other nodes in the network given its Markov blanket (the gray area).

14.3 EFFICIENT REPRESENTATION OF CONDITIONAL DISTRIBUTIONS

Even if the maximum number of parents k is smallish, filling in the CPT for a node requires up to $O(2^k)$ numbers and perhaps a great deal of experience with all the possible conditioning cases. In fact, this is a worst-case scenario in which the relationship between the parents and the child is completely arbitrary. Usually, such relationships are describable by a **canonical distribution** that fits some standard pattern. In such cases, the complete table can be specified by naming the pattern and perhaps supplying a few parameters—much easier than supplying an exponential number of parameters.

CANONICAL
DISTRIBUTION

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.

DETERMINISTIC
NODES

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

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:

$$q_{\text{cold}} = P(\neg fever \mid cold, \neg flu, \neg malaria) = 0.6 \ ,$$
$$q_{\text{flu}} = P(\neg fever \mid \neg cold, flu, \neg malaria) = 0.2 \ ,$$
$$q_{\text{malaria}} = P(\neg fever \mid \neg cold, \neg flu, malaria) = 0.1 \ .$$

Then, from this information and the noisy-OR assumptions, the entire CPT can be built. The general rule is that

$$P(x_i \mid parents(X_i)) = 1 - \prod_{\{j : X_j \,=\, true\}} q_j \ ,$$

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:

Cold	*Flu*	*Malaria*	$P(Fever)$	$P(\neg Fever)$
F	F	F	0.0	1.0
F	F	T	0.9	**0.1**
F	T	F	0.8	**0.2**
F	T	T	0.98	$0.02 = 0.2 \times 0.1$
T	F	F	0.4	**0.6**
T	F	T	0.94	$0.06 = 0.6 \times 0.1$
T	T	F	0.88	$0.12 = 0.6 \times 0.2$
T	T	T	0.988	$0.012 = 0.6 \times 0.2 \times 0.1$

In general, noisy logical relationships in which a variable depends on k parents can be described using $O(k)$ parameters instead of $O(2^k)$ for the full conditional probability table. This makes assessment and learning much easier. For example, the CPCS network (Pradhan *et al.*, 1994) uses noisy-OR and noisy-MAX distributions to model relationships among diseases and symptoms in internal medicine. With 448 nodes and 906 links, it requires only 8,254 values instead of 133,931,430 for a network with full CPTs.

Bayesian nets with continuous variables

Many real-world problems involve continuous quantities, such as height, mass, temperature, and money; in fact, much of statistics deals with random variables whose domains are continuous. By definition, continuous variables have an infinite number of possible values, so it is impossible to specify conditional probabilities explicitly for each value. One possible way to

DISCRETIZATION
handle continuous variables is to avoid them by using **discretization**—that is, dividing up the

Figure 14.5   A simple network with discrete variables (*Subsidy* and *Buys*) and continuous variables (*Harvest* and *Cost*).
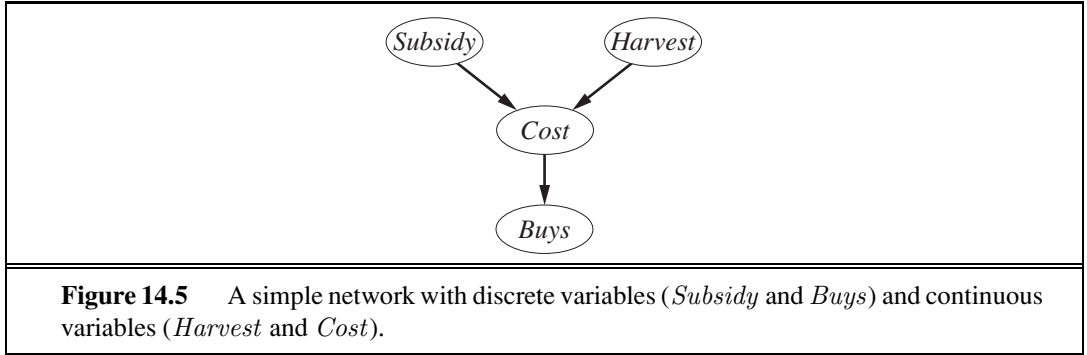
possible values into a fixed set of intervals. For example, temperatures could be divided into ($<0^{\circ}$C), (0°C-100°C), and ($>100^{\circ}$C). Discretization is sometimes an adequate solution, but often results in a considerable loss of accuracy and very large CPTs. The most com-
mon solution is 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.
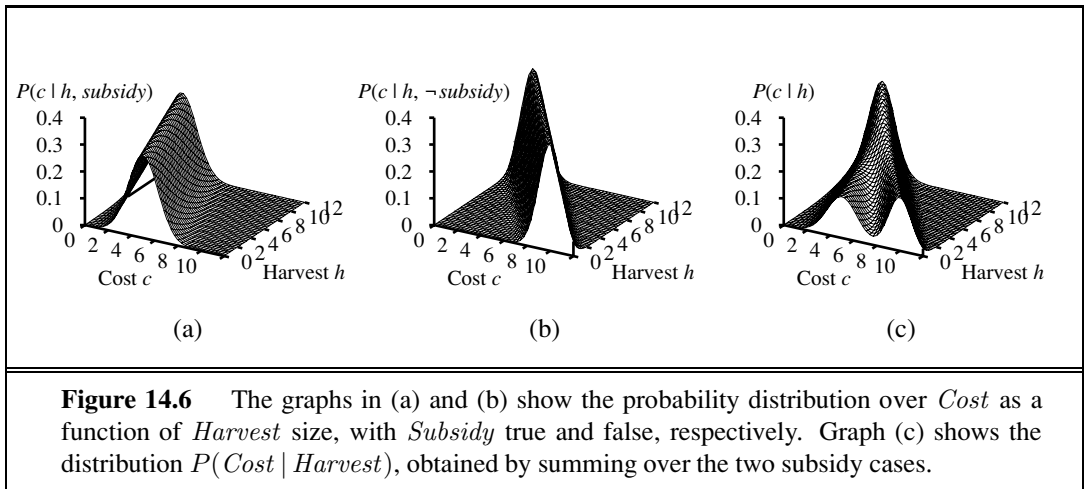
PARAMETER

NONPARAMETRIC

    A network with both discrete and continuous variables is called a **hybrid Bayesian network**. To specify a hybrid network, we have to specify two new kinds of distributions: the conditional distribution for a continuous variable given discrete or continuous parents; and the conditional distribution for a discrete variable given continuous parents. Consider the simple example in Figure 14.5, in which a customer buys some fruit depending on its cost, which depends in turn on the size of the harvest and whether the government's subsidy scheme is operating. The variable *Cost* is continuous and has continuous and discrete parents; the variable *Buys* is discrete and has a continuous parent.

HYBRID BAYESIAN
NETWORK

    For the *Cost* variable, we need to specify **P**(*Cost* | *Harvest*, *Subsidy*). The discrete parent is handled by enumeration—that is, by specifying both $P(Cost \,|\, Harvest, subsidy)$ and $P(Cost \,|\, Harvest, \neg subsidy)$. To handle *Harvest*, we specify how the distribution over the cost c depends on the continuous value h of *Harvest*. In other words, we specify the *parameters* of the cost distribution as a function of h. The most common choice is the **linear Gaussian** distribution, in which the child has a Gaussian distribution whose mean μ varies linearly with the value of the parent and whose standard deviation σ is fixed. We need two distributions, one for *subsidy* and one for $\neg subsidy$, with different parameters:

LINEAR GAUSSIAN

$$P(c \,|\, h, subsidy) \;=\; N(a_t h + b_t, \sigma_t^2)(c) = \frac{1}{\sigma_t \sqrt{2\pi}} \, e^{-\frac{1}{2}\left(\frac{c-(a_t h+b_t)}{\sigma_t}\right)^2}$$

$$P(c \,|\, h, \neg subsidy) \;=\; N(a_f h + b_f, \sigma_f^2)(c) = \frac{1}{\sigma_f \sqrt{2\pi}} \, e^{-\frac{1}{2}\left(\frac{c-(a_f h+b_f)}{\sigma_f}\right)^2} .$$

For this example, then, the conditional distribution for *Cost* is specified by naming the linear Gaussian distribution and providing the parameters a_t, b_t, σ_t, a_f, b_f, and σ_f. Figures 14.6(a)

Figure 14.6   The graphs in (a) and (b) show the probability distribution over *Cost* as a function of *Harvest* size, with *Subsidy* true and false, respectively. Graph (c) shows the distribution $P(Cost \mid Harvest)$, obtained by summing over the two subsidy cases.

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 $P(c \mid h)$, 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.

CONDITIONAL
GAUSSIAN

Now we turn to the distributions for discrete variables with continuous parents. Consider, for example, the *Buys* node in Figure 14.5. It seems reasonable to assume that the customer will buy if the cost is low and will not buy if it is high and that the probability of buying varies smoothly in some intermediate region. In other words, the conditional distribution is like a "soft" threshold function. One way to make soft thresholds is to use the *integral* of the standard normal distribution:

$$\Phi(x) = \int_{-\infty}^{x} N(0,1)(x)dx .$$

Then the probability of *Buys* given *Cost* might be

$$P(buys \mid Cost = c) = \Phi((-c + \mu)/\sigma) ,$$

which means that the cost threshold occurs around μ, the width of the threshold region is proportional to σ, and the probability of buying decreases as cost increases. This **probit distri-**

---

[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.

Figure 14.7   (a) A normal (Gaussian) distribution for the cost threshold, centered on $\mu = 6.0$ with standard deviation $\sigma = 1.0$. (b) Logit and probit distributions for the probability of *buys* given *cost*, for the parameters $\mu = 6.0$ and $\sigma = 1.0$.
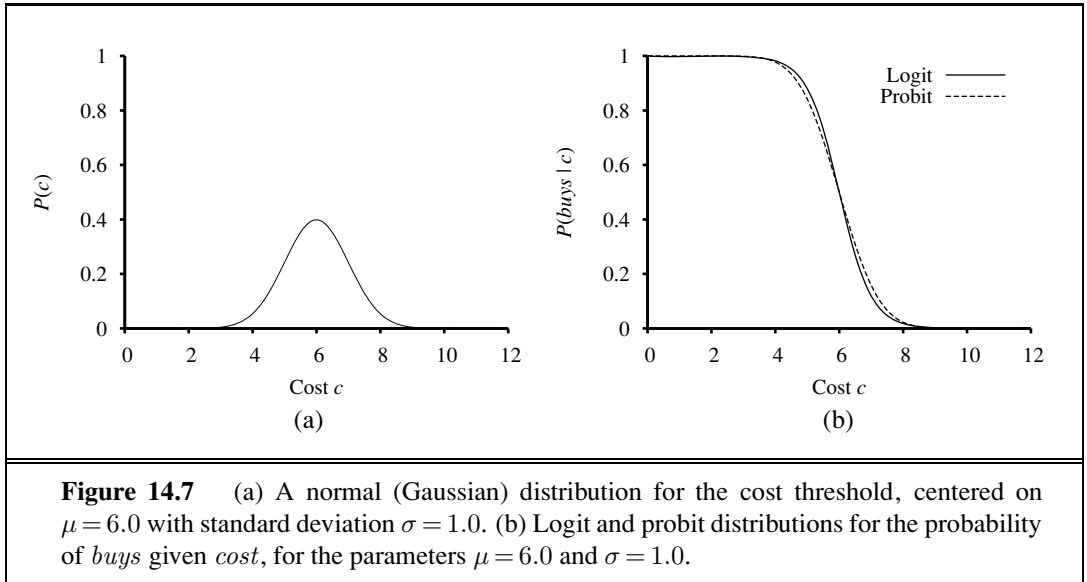
PROBIT
DISTRIBUTION

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:

$$P(buys \mid Cost = c) = \frac{1}{1 + exp(-2\frac{-c+\mu}{\sigma})} \ .$$

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

HIDDEN VARIABLE

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 $E_1, \ldots, E_m$, and $\mathbf{e}$ is a particular observed event; $\mathbf{Y}$ will denotes the nonevidence, nonquery variables $Y_1, \ldots, Y_l$ (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 $\mathbf{P}(X \mid \mathbf{e})$.

In the burglary network, we might observe the event in which $JohnCalls = true$ and $MaryCalls = true$. We could then ask for, say, the probability that a burglary has occurred:

$$\mathbf{P}(Burglary \mid JohnCalls = true, MaryCalls = true) = \langle 0.284, 0.716 \rangle .$$

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 $\mathbf{P}(Burglary \mid JohnCalls = true, MaryCalls = true)$. 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 have[4]

$$\mathbf{P}(B \mid j, m) = \alpha \, \mathbf{P}(B, j, m) = \alpha \sum_{e} \sum_{a} \mathbf{P}(B, j, m, e, a,) .$$

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

$$P(b \mid j, m) = \alpha \, P(b) \sum_{e} P(e) \sum_{a} P(a \mid b, e) P(j \mid a) P(m \mid a) . \tag{14.4}$$

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 $P(b \mid j, m) = \alpha \times 0.00059224$. The corresponding computation for $\neg b$ yields $\alpha \times 0.0014919$; hence,

$$\mathbf{P}(B \mid j, m) = \alpha \langle 0.00059224, 0.0014919 \rangle \approx \langle 0.284, 0.716 \rangle \ .$$

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 $P(j \mid a)P(m \mid a)$ and $P(j \mid \neg a)P(m \mid \neg a)$ 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

$$\mathbf{P}(B \mid j, m) = \alpha \underbrace{\mathbf{P}(B)}_{\mathbf{f}_1(B)} \sum_e \underbrace{P(e)}_{\mathbf{f}_2(E)} \sum_a \underbrace{\mathbf{P}(a \mid B, e)}_{\mathbf{f}_3(A,B,E)} \underbrace{P(j \mid a)}_{\mathbf{f}_4(A)} \underbrace{P(m \mid a)}_{\mathbf{f}_5(A)} \ .$$

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 $P(j \mid a)$ and $P(m \mid a)$ depend just on A because J and M are fixed by the query. They are therefore two-element vectors:

$$\mathbf{f}_4(A) = \begin{pmatrix} P(j \mid a) \\ P(j \mid \neg a) \end{pmatrix} = \begin{pmatrix} 0.90 \\ 0.05 \end{pmatrix} \qquad \mathbf{f}_5(A) = \begin{pmatrix} P(m \mid a) \\ P(m \mid \neg a) \end{pmatrix} = \begin{pmatrix} 0.70 \\ 0.01 \end{pmatrix} \ .$$

$\mathbf{f}_3(A, B, E)$ will be a $2 \times 2 \times 2$ matrix, which is hard to show on the printed page. (The "first" element is given by $P(a \mid b, e) = 0.95$ and the "last" by $P(\neg a \mid \neg b, \neg e) = 0.999$.) In terms of factors, the query expression is written as

$$\mathbf{P}(B \mid j, m) = \alpha \, \mathbf{f}_1(B) \times \sum_e \mathbf{f}_2(E) \times \sum_a \mathbf{f}_3(A, B, E) \times \mathbf{f}_4(A) \times \mathbf{f}_5(A)$$

VARIABLE
ELIMINATION

FACTOR
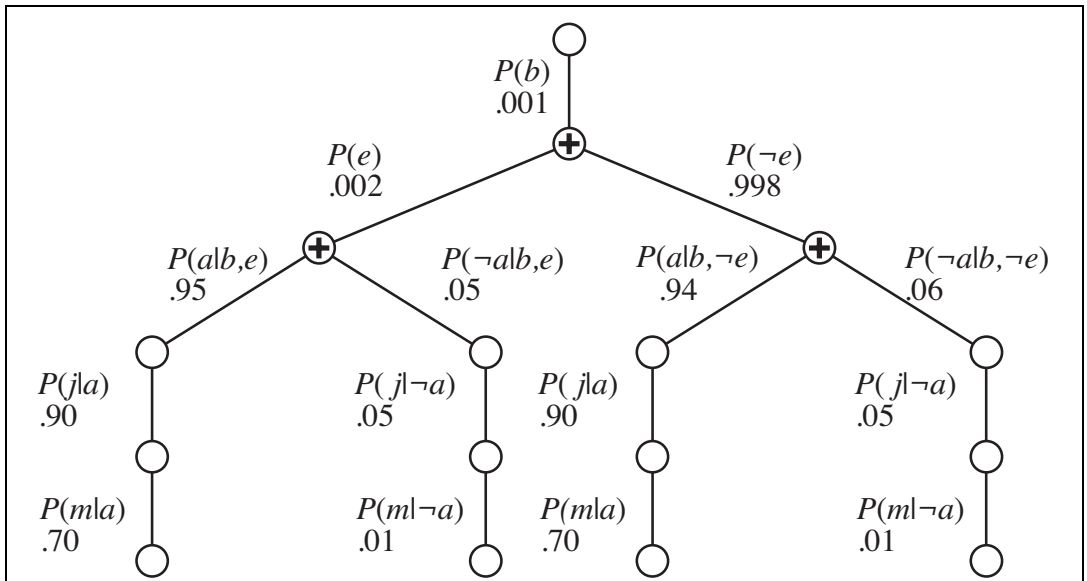
Figure 14.8   The structure of the expression shown in Equation (14.4). The evaluation proceeds top down, multiplying values along each path and summing at the "+" nodes. Notice the repetition of the paths for j and m.

---

function ENUMERATION-ASK(X, **e**, bn) **returns** a distribution over X
  inputs: X, the query variable
          e, observed values for variables **E**
          bn, a Bayes net with variables $\{X\} \cup \mathbf{E} \cup \mathbf{Y}$ / * **Y** = *hidden variables* * /

  $\mathbf{Q}(X) \leftarrow$ a distribution over X, initially empty
  for each value x_i of X **do**
      $\mathbf{Q}(x_i) \leftarrow$ ENUMERATE-ALL(bn.VARS, $\mathbf{e}_{x_i}$)
          where $\mathbf{e}_{x_i}$ is **e** extended with $X = x_i$
  return NORMALIZE($\mathbf{Q}(X)$)

---

function ENUMERATE-ALL($vars$, **e**) **returns** a real number
  if EMPTY?($vars$) **then return** 1.0
  $Y \leftarrow$ FIRST($vars$)
  if Y has value y in **e**
      then return $P(y \mid parents(Y)) \times$ ENUMERATE-ALL(REST($vars$), **e**)
      else return $\sum_y P(y \mid parents(Y)) \times$ ENUMERATE-ALL(REST($vars$), $\mathbf{e}_y$)
          where $\mathbf{e}_y$ is **e** extended with $Y = y$

Figure 14.9   The enumeration algorithm for answering queries on Bayesian networks.

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

$$\mathbf{f}(X_1 \dots X_j, Y_1 \dots Y_k, Z_1 \dots Z_l) = \mathbf{f}_1(X_1 \dots X_j, Y_1 \dots Y_k) \, \mathbf{f}_2(Y_1 \dots Y_k, Z, \dots Z_l).$$

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

$$\mathbf{f}(B,C) \;=\; \sum_a \mathbf{f}_3(A,B,C) = \mathbf{f}_3(a,B,C) + \mathbf{f}_3(\neg a, B, C)$$

$$= \begin{pmatrix} .06 & .24 \\ .42 & .28 \end{pmatrix} + \begin{pmatrix} .18 & .72 \\ .06 & .04 \end{pmatrix} = \begin{pmatrix} .24 & .96 \\ .48 & .32 \end{pmatrix} \; .$$

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

$$\mathbf{P}(B \,|\, j, m) = \alpha \, \mathbf{f}_1(B) \times \sum_a \mathbf{f}_4(A) \times \mathbf{f}_5(A) \times \sum_e \mathbf{f}_2(E) \times \mathbf{f}_3(A,B,E) \, ,$$

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, **e**, bn) **returns** a distribution over X
   inputs: X, the query variable
           e, observed values for variables **E**
           bn, a Bayesian network specifying joint distribution $\mathbf{P}(X_1, \ldots, X_n)$

   $factors \leftarrow [\,]$
   for each var **in** ORDER(bn.VARS) **do**
      $factors \leftarrow [\text{MAKE-FACTOR}(var, \mathbf{e}) | factors]$
      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*
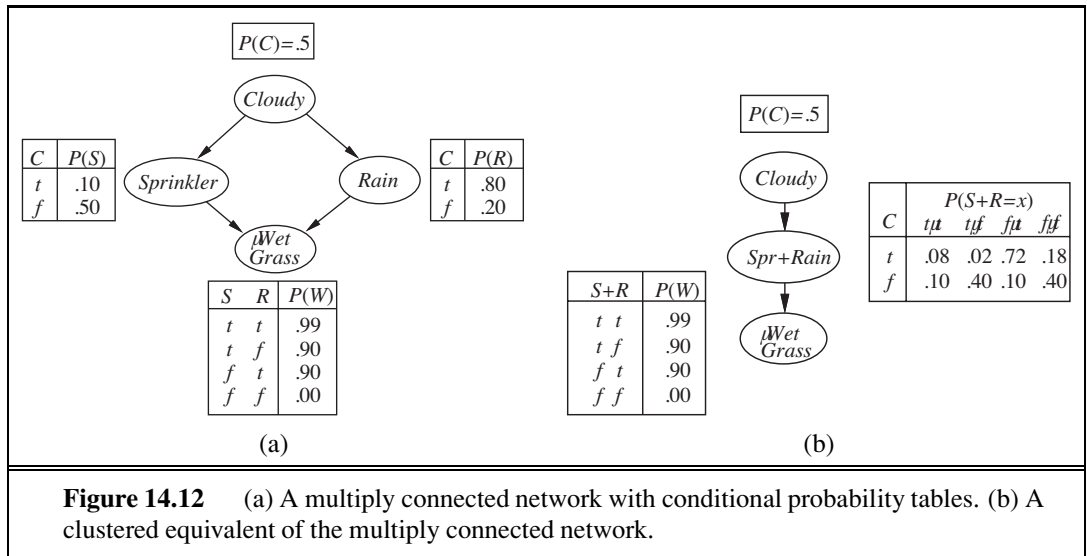
SINGLY CONNECTED

POLYTREE

MULTIPLY CONNECTED

$P(C)=.5$

Cloudy

C	P(S)
t	.10
f	.50

Sprinkler   Rain

C	P(R)
t	.80
f	.20

Wet Grass

S	R	P(W)
t	t	.99
t	f	.90
f	t	.90
f	f	.00

(a)

$P(C)=.5$

Cloudy

Spr+Rain

Wet Grass

S+R	P(W)
t t	.99
t f	.90
f t	.90
f f	.00

C	P(S+R=x)			
	tt	tf	ft	ff
t	.08	.02	.72	.18
f	.10	.40	.10	.40

(b)

Figure 14.12   (a) A multiply connected network with conditional probability tables. (b) A clustered equivalent of the multiply connected network.

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.

CLUSTERING

JOIN TREE

The basic idea of clustering is to join individual nodes of the network to form cluster nodes in such a way that the resulting network is a polytree. For example, the multiply connected network shown in Figure 14.12(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

Given the intractability of exact inference in large, multiply connected networks, it is essential to consider approximate inference methods. This section describes randomized sampling

MONTE CARLO   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 $\mathbf{P}(X_1, \ldots, X_n)$

   $\mathbf{x} \leftarrow$ an event with n elements
   foreach variable X_i **in** $X_1, \ldots, X_n$ **do**
        $\mathbf{x}[i] \leftarrow$ a random sample from $\mathbf{P}(X_i \mid parents(X_i))$
   return 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 $S_{PS}(x_1, \ldots, x_n)$ be the probability that a specific event is generated by the PRIOR-SAMPLE algorithm. *Just looking at the sampling process*, we have

$$S_{PS}(x_1 \ldots x_n) = \prod_{i=1}^{n} P(x_i \mid parents(X_i))$$

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

$$S_{PS}(x_1 \ldots x_n) = P(x_1 \ldots x_n) \,.$$

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 $N_{PS}(x_1, \ldots, x_n)$ be the number of times the specific event $x_1, \ldots, x_n$ 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:

$$\lim_{N \to \infty} \frac{N_{PS}(x_1, \ldots, x_n)}{N} = S_{PS}(x_1, \ldots, x_n) = P(x_1, \ldots, x_n) \,. \tag{14.5}$$

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 $x_1, \ldots, x_m$, where $m \leq n$, as follows:

$$P(x_1, \ldots, x_m) \approx N_{PS}(x_1, \ldots, x_m)/N \,. \tag{14.6}$$

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

CONSISTENT

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 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 $P(X \mid \mathbf{e})$. 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 $\hat{P}(X = x \mid \mathbf{e})$ is obtained by counting how often $X = x$ occurs in the remaining samples.

Let $\hat{\mathbf{P}}(X \mid \mathbf{e})$ be the estimated distribution that the algorithm returns. From the definition of the algorithm, we have

$$\hat{\mathbf{P}}(X \mid \mathbf{e}) = \alpha \, \mathbf{N}_{PS}(X, \mathbf{e}) = \frac{\mathbf{N}_{PS}(X, \mathbf{e})}{N_{PS}(\mathbf{e})} \ .$$

From Equation (14.6), this becomes

$$\hat{\mathbf{P}}(X \mid \mathbf{e}) \approx \frac{\mathbf{P}(X, \mathbf{e})}{P(\mathbf{e})} = \mathbf{P}(X \mid \mathbf{e}) \ .$$

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 $\mathbf{P}(Rain \mid Sprinkler = true)$, 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,

$$\mathbf{P}(Rain \mid Sprinkler = true) \approx \text{NORMALIZE}(\langle 8, 19 \rangle) = \langle 0.296, 0.704 \rangle \ .$$

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 $\mathbf{P}(Rain \mid RedSkyAtNight = true)$, 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 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

---

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**
    $\mathbf{x} \leftarrow$ 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 $\mathbf{P}(Rain \mid Cloudy = true, WetGrass = true)$ 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
$$w \leftarrow w \times P(Cloudy = true) = 0.5 \ .$$

2. *Sprinkler* is not an evidence variable, so sample from $\mathbf{P}(Sprinkler \mid Cloudy = true) = \langle 0.1, 0.9 \rangle$; suppose this returns *false*.

3. Similarly, sample from $\mathbf{P}(Rain \mid Cloudy = true) = \langle 0.8, 0.2 \rangle$; suppose this returns *true*.

4. *WetGrass* is an evidence variable with value *true*. Therefore, we set
$$w \leftarrow w \times P(WetGrass = true \mid Sprinkler = false, Rain = true) = 0.45 \ .$$

Here WEIGHTED-SAMPLE returns the event $[true, false, true, true]$ 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 $\mathbf{P}(X_1, \ldots, X_n)$
        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**
    $\mathbf{x}, w \leftarrow$ 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** $X_1, \ldots, X_n$ **do**
    if X_i is an evidence variable with value x_i in $\mathbf{e}$
      then $w \leftarrow w \times P(X_i = x_i \mid parents(X_i))$
      else $\mathbf{x}[i] \leftarrow$ a random sample from $\mathbf{P}(X_i \mid parents(X_i))$
  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:

$$S_{WS}(\mathbf{z}, \mathbf{e}) = \prod_{i=1}^{l} P(z_i \mid parents(Z_i)) . \tag{14.7}$$

Notice that $Parents(Z_i)$ 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 $P(\mathbf{z} \mid \mathbf{e})$, 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 $P(\mathbf{z} \mid \mathbf{e})$, 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_is):

$$w(\mathbf{z}, \mathbf{e}) = \prod_{i=1}^{m} P(e_i \,|\, parents(E_i)) \,. \tag{14.8}$$

Multiplying Equations (14.7) and (14.8), we see that the *weighted* probability of a sample has the particularly convenient form

$$
\begin{aligned}
S_{WS}(\mathbf{z}, \mathbf{e})w(\mathbf{z}, \mathbf{e}) &= \prod_{i=1}^{l} P(z_i \,|\, parents(Z_i)) \prod_{i=1}^{m} P(e_i \,|\, parents(E_i)) \\
&= P(\mathbf{z}, \mathbf{e})
\end{aligned}
\tag{14.9}
$$

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

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

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 $\mathbf{P}(Rain \mid Sprinkler = true, WetGrass = true)$ 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 $[true, true, false, true]$. 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 $\mathbf{P}(Cloudy \mid Sprinkler = true, Rain = false)$. (Shortly, we will show how to calculate this distribution.) Suppose the result is $Cloudy = false$. Then the new current state is $[false, true, false, true]$.

2. $Rain$ is sampled, given the current values of its Markov blanket variables: in this case, we sample from $\mathbf{P}(Rain \mid Cloudy = false, Sprinkler = true, WetGrass = true)$. Suppose this yields $Rain = true$. The new current state is $[false, true, true, true]$.

Each state visited during this process is a sample that contributes to the estimate for the query variable $Rain$. If the process visits 20 states where $Rain$ is true and 60 states where $Rain$ is false, then the answer to the query is NORMALIZE($\langle 20, 60 \rangle$) $= \langle 0.25, 0.75 \rangle$. The complete algorithm is shown in Figure 14.16.

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.

---

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.

Let $q(\mathbf{x} \to \mathbf{x}')$ 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 $\mathbf{x}'$:

$$\pi_{t+1}(\mathbf{x}') = \sum_{\mathbf{x}} \pi_t(\mathbf{x})q(\mathbf{x} \to \mathbf{x}') \ .$$

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

$$\pi(\mathbf{x}') = \sum_{\mathbf{x}} \pi(\mathbf{x})q(\mathbf{x} \to \mathbf{x}') \qquad \text{for all } \mathbf{x}' \ . \tag{14.10}$$

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,

$$\pi(\mathbf{x})q(\mathbf{x} \to \mathbf{x}') = \pi(\mathbf{x}')q(\mathbf{x}' \to \mathbf{x}) \qquad \text{for all } \mathbf{x}, \ \mathbf{x}' \ . \tag{14.11}$$

When these equations hold, we say that $q(\mathbf{x} \to \mathbf{x}')$ 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

$$\sum_{\mathbf{x}} \pi(\mathbf{x})q(\mathbf{x} \to \mathbf{x}') = \sum_{\mathbf{x}} \pi(\mathbf{x}')q(\mathbf{x}' \to \mathbf{x}) = \pi(\mathbf{x}')\sum_{\mathbf{x}} q(\mathbf{x}' \to \mathbf{x}) = \pi(\mathbf{x}')$$

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 $P(\mathbf{x} \mid \mathbf{e})$, (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 $\overline{\mathbf{X}_i}$ to be these other variables (except the evidence variables); their values in the current state are $\overline{\mathbf{x}_i}$. If we sample a new value x'_i for X_i conditionally on all the other variables, including the evidence, we have

$$q_i(\mathbf{x} \rightarrow \mathbf{x}') = q_i((x_i, \overline{\mathbf{x}_i}) \rightarrow (x'_i, \overline{\mathbf{x}_i})) = P(x'_i \mid \overline{\mathbf{x}_i}, \mathbf{e}) \ .$$

Now we show that the transition probability for each step of the Gibbs sampler is in detailed balance with the true posterior:

$$\pi(\mathbf{x})q_i(\mathbf{x} \rightarrow \mathbf{x}') = P(\mathbf{x} \mid \mathbf{e})P(x'_i \mid \overline{\mathbf{x}_i}, \mathbf{e}) = P(x_i, \overline{\mathbf{x}_i} \mid \mathbf{e})P(x'_i \mid \overline{\mathbf{x}_i}, \mathbf{e})$$

$$= \ P(x_i \mid \overline{\mathbf{x}_i}, \mathbf{e})P(\overline{\mathbf{x}_i} \mid \mathbf{e})P(x'_i \mid \overline{\mathbf{x}_i}, \mathbf{e}) \qquad \text{(using the chain rule on the first term)}$$

$$= \ P(x_i \mid \overline{\mathbf{x}_i}, \mathbf{e})P(x'_i, \overline{\mathbf{x}_i} \mid \mathbf{e}) \qquad\qquad \text{(using the chain rule backward)}$$

$$= \ \pi(\mathbf{x}')q_i(\mathbf{x}' \rightarrow \mathbf{x}) \ .$$

We can think of the loop "**for each** Z_i in **Z do**" in Figure 14.16 as defining one large transition probability q that is the sequential composition $q_1 \circ q_2 \circ \cdots \circ q_n$ 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 $P(\mathbf{x} \mid \mathbf{e})$ 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 $\mathbf{P}(X_i \mid \overline{\mathbf{x}_i}, \mathbf{e})$—in a Bayesian network. Recall from page 517 that a variable is independent of all other variables given its Markov blanket; hence,

$$P(x'_i \mid \overline{\mathbf{x}_i}, \mathbf{e}) = P(x'_i \mid mb(X_i)) \ ,$$

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:

$$P(x'_i \mid mb(X_i)) = \alpha \, P(x'_i \mid parents(X_i)) \times \prod_{Y_j \in Children(X_i)} P(y_j \mid parents(Y_j)) \ . \ (14.12)$$

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.

Figure 14.17   (a) Bayes net for a single customer C_1 recommending a single book B_1. $Honest(C_1)$ is Boolean, while the other variables have integer values from 1 to 5. (b) Bayes net with two customers and two books.
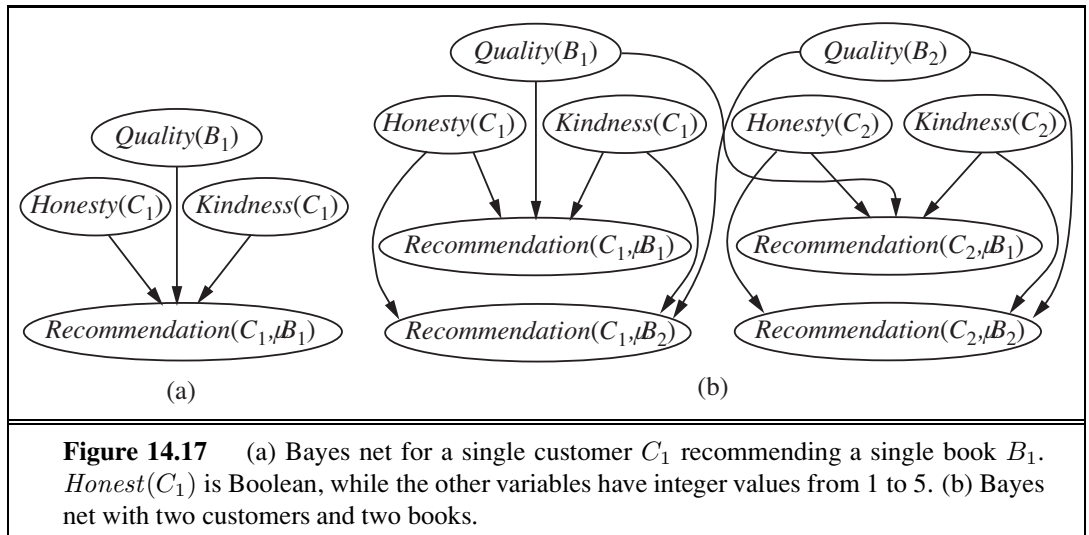
14.6   RELATIONAL AND FIRST-ORDER PROBABILITY MODELS

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.
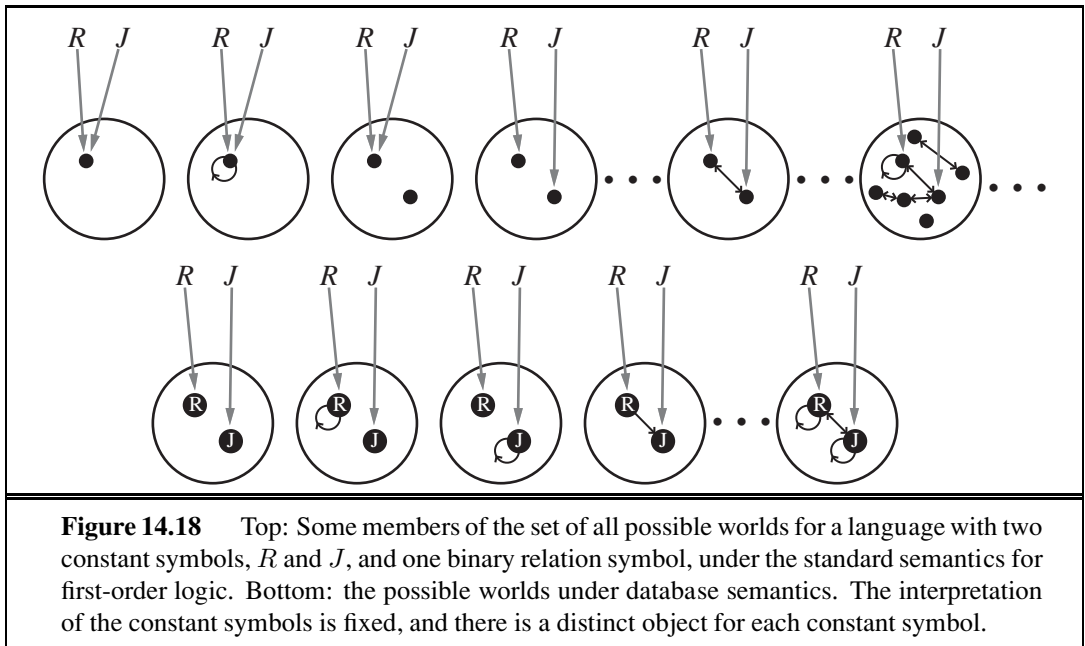
Let us suppose, for a moment, that we have figured out how to do this. Then, as usual (see page 485), we can obtain the probability of any first-order logical sentence ϕ as a sum over the possible worlds where it is true:

$$P(\phi) = \sum_{\omega:\phi \text{ is true in } \omega} P(\omega) . \tag{14.13}$$

Conditional probabilities $P(\phi \mid \mathbf{e})$ 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

Figure 14.18   Top: Some members of the set of all possible worlds for a language with two constant symbols, R and J, and one binary relation symbol, under the standard semantics for first-order logic. Bottom: the possible worlds under database semantics. The interpretation of the constant symbols is fixed, and there is a distinct object for each constant symbol.

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!

RELATIONAL PROBABILITY MODEL

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.

SIBYL

SIBYL ATTACK

EXISTENCE UNCERTAINTY

IDENTITY UNCERTAINTY

---

[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

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:

$Honest : Customer \rightarrow \{true, false\} Kindness : Customer \rightarrow \{1, 2, 3, 4, 5\}$
$Quality : Book \rightarrow \{1, 2, 3, 4, 5\}$
$Recommendation : Customer \times Book \rightarrow \{1, 2, 3, 4, 5\}$

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]

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 $Honest(c) = false$:

$Recommendation(c, b) \sim$ **if** $Honest(c)$ **then**
$HonestRecCPT(Kindness(c), Quality(b))$
else $\langle 0.4, 0.1, 0.0, 0.1, 0.4 \rangle$.

---

[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.

Figure 14.19   Fragment of the equivalent Bayes net when $Author(B_2)$ is unknown.
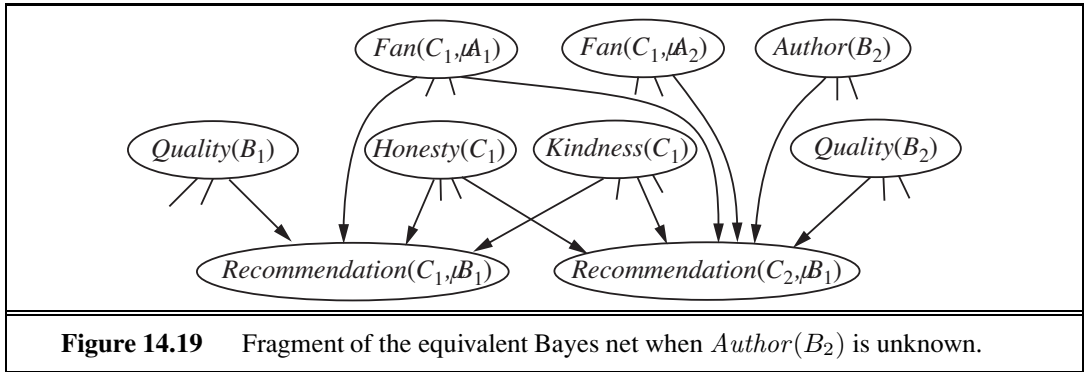
This kind of dependency may look like an ordinary if–then–else statement on a programming language, but there is a key difference: the inference engine *doesn't necessarily know the value of the conditional test*!

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:

$$Recommendation(c, b) \sim \quad \textbf{if } Honest(c) \textbf{ then}$$
$$\textbf{if } Fan(c, Author(b)) \textbf{ then } Exactly(5)$$
$$\textbf{else } HonestRecCPT(Kindness(c), Quality(b))$$
$$\textbf{else } \langle 0.4, 0.1, 0.0, 0.1, 0.4 \rangle$$

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 MULTIPLEXER 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 **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.

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.

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.

UNROLLING

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 **propositionalization** 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 distribution[9] as follows:

$$\# \, Customer \sim LogNormal[6.9, 2.3^2]() \ .$$

We expect honest customers to have just one ID, whereas dishonest customers might have anywhere between 10 and 1000 IDs:

$$\# \, LoginID(\, Owner = c) \sim \quad \textbf{if } Honest(c) \textbf{ then } Exactly(1)$$
$$\textbf{else } LogNormal[6.9, 2.3^2]() \ .$$

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 $LogNormal[\mu, \sigma^2](x)$ 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

$$Rain \mapsto WetGrass \qquad \text{and} \qquad WetGrass \mapsto Rain .$$

These two rules form a feedback loop: evidence for $Rain$ increases the belief in $WetGrass$, which in turn increases the belief in $Rain$ even more. Clearly, uncertain reasoning systems need to keep track of the paths along which evidence is propagated.

Intercausal reasoning (or explaining away) is also tricky. Consider what happens when we have the two rules

$$Sprinkler \mapsto WetGrass \qquad \text{and} \qquad WetGrass \mapsto Rain .$$

Suppose we see that the sprinkler is on. Chaining forward through our rules, this increases the belief that the grass will be wet, which in turn increases the belief that it is raining. But this is ridiculous: the fact that the sprinkler is on explains away the wet grass and should *reduce* the belief in rain. A truth-functional system acts as if it also believes $Sprinkler \mapsto Rain$.

Given these difficulties, how 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 CERTAINTY FACTOR 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–SHAFER
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

belief function, written $Bel(X)$.

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.

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

MASS

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 $\mathbf{P}(Flip \mid Bias)$ simply defines how the bias operates. If $\mathbf{P}(Bias)$ is symmetric about 0.5, then our prior probability for the flip is

$$P(Flip = heads) = \int_0^1 P(Bias = x)P(Flip = heads \mid Bias = x)\, dx = 0.5 \ .$$

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

$$T(A \wedge B) = \min(T(A), T(B))$$
$$T(A \vee B) = \max(T(A), T(B))$$
$$T(\neg A) = 1 - T(A) \ .$$

Fuzzy logic is therefore a truth-functional system—a fact that causes serious difficulties. For example, suppose that $T(Tall(Nate)) = 0.6$ and $T(Heavy(Nate)) = 0.4$. Then we have $T(Tall(Nate) \wedge Heavy(Nate)) = 0.4$, which seems reasonable, but we also get the result $T(Tall(Nate) \wedge \neg Tall(Nate)) = 0.4$, which does not. Clearly, the problem arises from the inability of a truth-functional approach to take into account the correlations or anticorrelations among the component propositions.

FUZZY CONTROL   **Fuzzy control** is a methodology for constructing control systems in which the mapping between real-valued input and output parameters is represented by fuzzy rules. Fuzzy control has been very successful in commercial products such as automatic transmissions, video

cameras, and electric shavers. Critics (see, e.g., Elkan, 1993) argue that these applications are successful because they have small rule bases, no chaining of inferences, and tunable parameters that can be adjusted to improve the system's performance. The fact that they are implemented with fuzzy operators might be incidental to their success; the key is simply to provide a concise and intuitive way to specify a smoothly interpolated, real-valued function.

There have been attempts to provide an explanation of fuzzy logic in terms of probability theory. One idea is to view assertions such as "Nate is Tall" as discrete observations made concerning a continuous hidden variable, Nate's actual $Height$. The probability model specifies $P(\text{Observer says Nate is tall} \mid Height)$, perhaps using a **probit distribution** as described on page 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

$$P(\text{Observer says Nate is tall and heavy} \mid Height, Weight)$$

allows for interactions between height and weight in the causing of the observation. Thus, someone who is eight feet tall and weighs 190 pounds is very unlikely to be called "tall and heavy," even though "eight feet" counts as "tall" and "190 pounds" counts as "heavy."

Fuzzy predicates can also be given a probabilistic interpretation in terms of **random sets**—that is, random variables whose possible values are sets of objects. For example, $Tall$ is a random set whose possible values are sets of people. The probability $P(Tall = S_1)$, where S_1 is some particular set of people, is the probability that exactly that set would be identified as "tall" by an observer. Then the probability that "Nate is tall" is the sum of the probabilities of all the sets of which Nate is a member.

RANDOM SET

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 polytree 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 CONVINCE (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.

MARKOV NETWORK

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.

NONSERIAL DYNAMIC PROGRAMMING

The inclusion of continuous random variables in Bayesian networks was considered by Pearl (1988) and Shachter and Kenley (1989); these papers discussed networks containing only continuous variables with linear Gaussian distributions. The inclusion of discrete variables has been investigated by Lauritzen and Wermuth (1989) and implemented in the

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** $\boldsymbol{\lambda}$ that are adjusted to minimize a distance function D between the original and the reduced problem, often by solving the system of equations $\partial D/\partial \boldsymbol{\lambda} = 0$. In many cases, strict upper and lower bounds can be obtained. Variational methods have long been used in statistics (Rustagi, 1976). In statistical physics, the **mean-field** method is a particular variational approximation in which the individual variables making up the model are assumed

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 $\forall x \quad P(Hungry(x)) > 0.2$ rules out distributions in which any object is hungry with probability less than 0.2; thus, it entails the sentence $\forall x \quad P(Hungry(x)) > 0.1$. 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 mdoels. (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.

POSSIBILITY 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 logicists was that the numerical calculations that probability theory was thought to require 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 $\theta(X_i \mid Parents(X_i))$. This exercise asks you to derive the equivalence between the parameters and the conditional probabilities $\mathbf{P}(X_i \mid Parents(X_i))$ 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 $P(z \mid y)$ 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 $\theta(X)$, $\theta(Y \mid X)$, and $\theta(Z \mid Y)$.

   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 $\sum_{x_i} \theta(x_i \mid parents(X_i)) = 1$, show that the resulting expression reduces to $\theta(x \mid y)$.

   d. Generalize this derivation to show that $\theta(X_i \mid Parents(X_i)) = \mathbf{P}(X_i \mid Parents(X_i))$ for any Bayesian network.

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.

   a. 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.)

   b. Under what circumstances can the total number remain constant?

   c. 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:

   $$\mathbf{P}(Y \mid \mathbf{U}, \mathbf{V}, \mathbf{W}) = \sum_x \mathbf{P}(Y \mid \mathbf{V}, \mathbf{W}, x)\mathbf{P}(x \mid \mathbf{U}, \mathbf{V})$$

   $$\mathbf{P}(X \mid \mathbf{U}, \mathbf{V}, \mathbf{W}, Y) = \mathbf{P}(Y \mid X, \mathbf{V}, \mathbf{W})\mathbf{P}(X \mid \mathbf{U}, \mathbf{V})/\mathbf{P}(Y \mid \mathbf{U}, \mathbf{V}, \mathbf{W}) .$$

   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.

   a. If no evidence is observed, are *Burglary* and *Earthquake* independent? Prove this from the numerical semantics and from the topological semantics.

   b. 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.

   a. Prove that removing the node Y from the network will not affect the posterior distribution for any other unobserved variable in the network.

   b. 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.

   a. Which of the three networks in Figure 14.20 claim that $\mathbf{P}(G_{father}, G_{mother}, G_{child}) = \mathbf{P}(G_{father})\mathbf{P}(G_{mother})\mathbf{P}(G_{child})$?

   b. Which of the three networks make independence claims that are consistent with the hypothesis about the inheritance of handedness?

Figure 14.20   Three possible structures for a Bayesian network describing genetic inheritance of handedness.
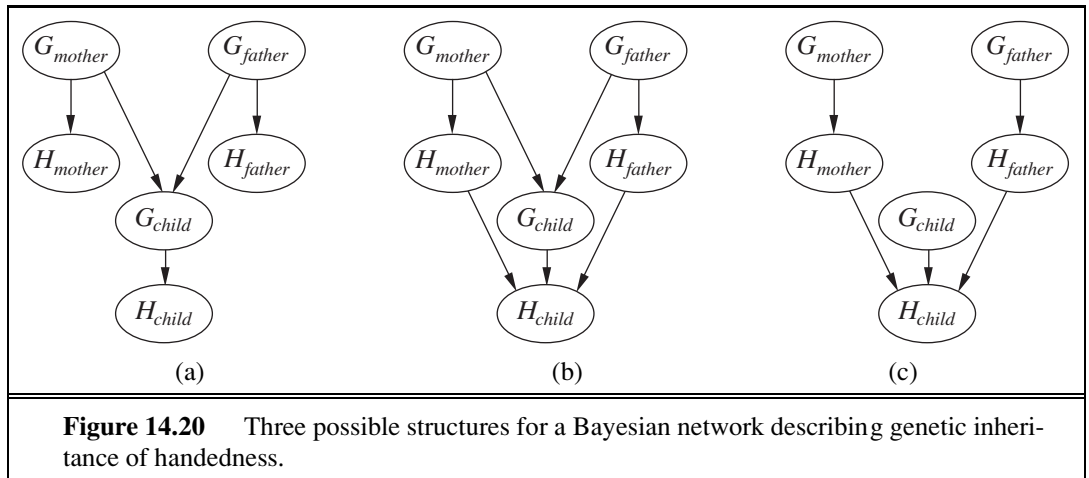
c. Which of the three networks is the best description of the hypothesis?

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).

Figure 14.21   A Bayesian network describing some features of a car's electrical system and engine. Each variable is Boolean, and the *true* value indicates that the corresponding aspect of the vehicle is in working order.
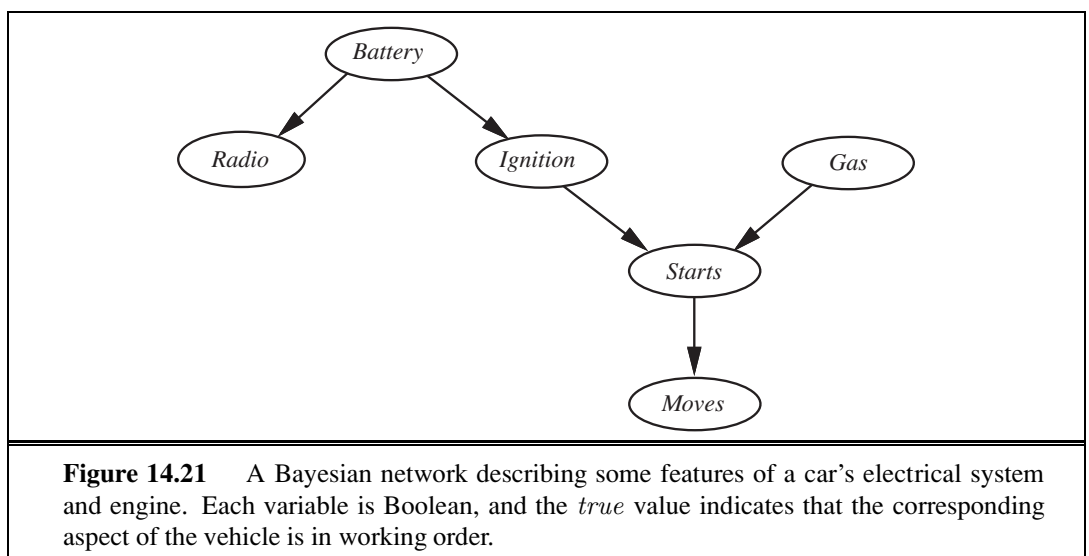
14.8 Consider the network for car diagnosis shown in Figure 14.21.

    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 $\mathbf{P}(X_2 \mid X_1)$ be a linear Gaussian distribution. Show that the joint distribution $P(X_1, X_2)$ is a multivariate Gaussian, and calculate its covariance matrix.

    b. Prove by induction that the joint distribution for a general linear Gaussian network on $X_1, \ldots, X_n$ is also a multivariate Gaussian.

14.10 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.
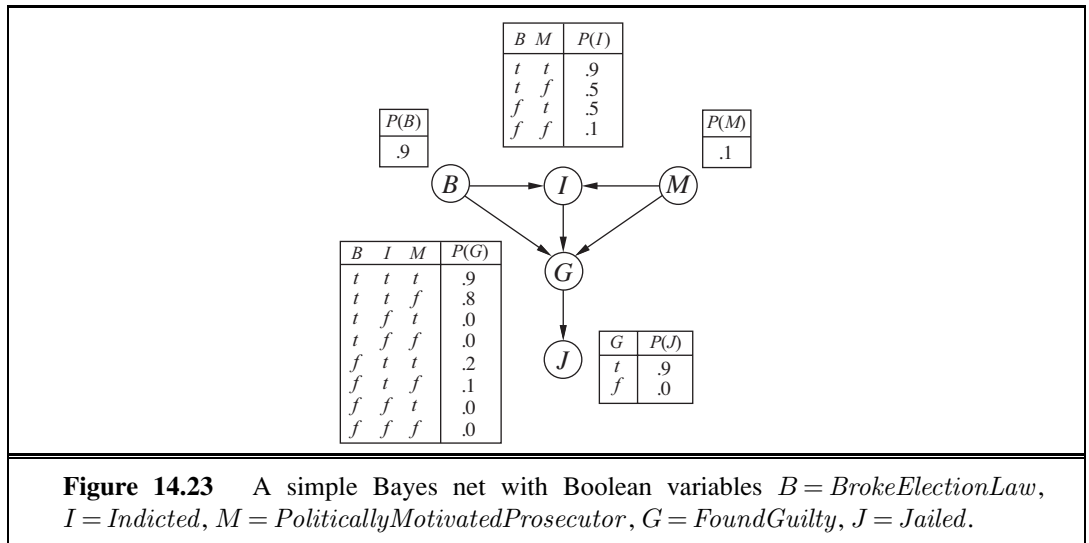
Figure 14.22   Three possible networks for the telescope problem.
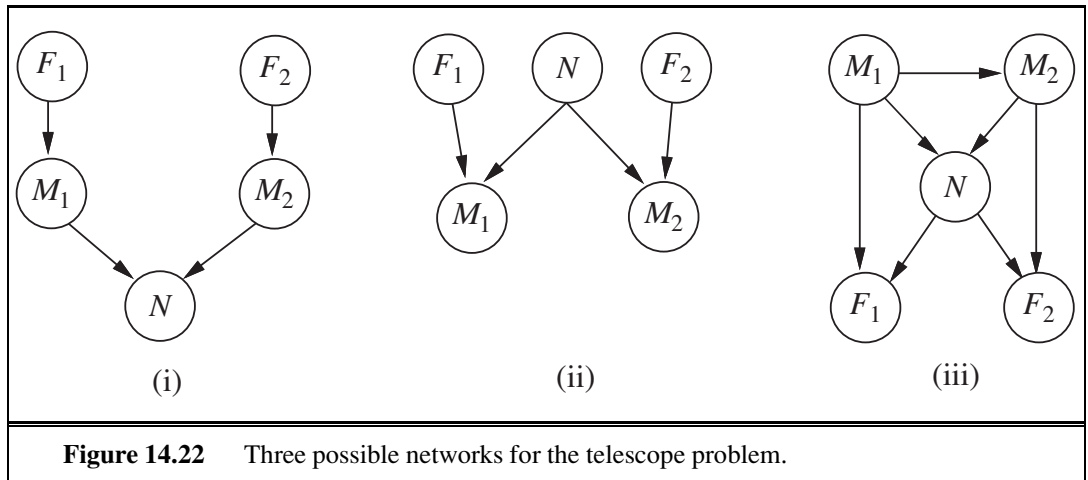
14.12   Two astronomers in different parts of the world make measurements M_1 and M_2 of the number of stars N in some small region of the sky, using their telescopes. Normally, there is a small possibility e of error by up to one star in each direction. Each telescope can also (with a much smaller probability 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.

    a. Which of these Bayesian networks are correct (but not necessarily efficient) representations of the preceding information?

    b. Which is the best network? Explain.

    c. Write out a conditional distribution for $\mathbf{P}(M_1 \mid N)$, 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.

    d. 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?

    e. What is the *most likely* number of stars, given these observations? Explain how to compute this, or if it is not possible to compute, explain what additional information is needed and how it would affect the result.

14.13   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.

    a. Which of the following are asserted by the network *structure*?

      (i) $\mathbf{P}(B, I, M) = \mathbf{P}(B)\mathbf{P}(I)\mathbf{P}(M)$.

     (ii) $\mathbf{P}(J \mid G) = \mathbf{P}(J \mid G, I)$.

    (iii) $\mathbf{P}(M \mid G, B, I) = \mathbf{P}(M \mid G, B, I, J)$.

Figure 14.23   A simple Bayes net with Boolean variables $B = BrokeElectionLaw$, $I = Indicted$, $M = PoliticallyMotivatedProsecutor$, $G = FoundGuilty$, $J = Jailed$.

b. Calculate the value of $P(b, i, \neg m, g, j)$.

c. Calculate the probability that someone goes to jail given that they broke the law, have been indicted, and face a politically motivated prosecutor.

d. 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?

e. Suppose we want to add the variable $P = PresidentialPardon$ 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).

a. Section 14.4 applies variable elimination to the query

$$\mathbf{P}(Burglary \mid JohnCalls = true, MaryCalls = true) \, .$$

Perform the calculations indicated and check that the answer is correct.

b. Count the number of arithmetic operations performed, and compare it with the number performed by the enumeration algorithm.

c. Suppose a network has the form of a *chain*: a sequence of Boolean variables $X_1, \ldots, X_n$ where $Parents(X_i) = \{X_{i-1}\}$ for $i = 2, \ldots, n$. What is the complexity of computing $\mathbf{P}(X_1 \mid X_n = true)$ using enumeration? Using variable elimination?

d. Prove that the complexity of running variable elimination on a polytree network is linear in the size of the tree for any variable ordering consistent with the network structure.

14.16   Investigate the complexity of exact inference in general Bayesian networks:

a. Prove that any 3-SAT problem can be reduced to exact inference in a Bayesian network constructed to represent the particular problem and hence that exact inference is NP-

hard. (*Hint*: Consider a network with one variable for each proposition symbol, one for each clause, and one for the conjunction of clauses.)

b. The problem of counting the number of satisfying assignments for a 3-SAT problem is #P-complete. Show that exact inference is at least as hard as this.

14.17   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 $i \in \{1, \ldots, k\}$. The **cumulative distribution** of X gives the probability that $X \in \{x_1, \ldots, x_j\}$ 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?

CUMULATIVE
DISTRIBUTION

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 $\mathbf{P}(Rain \mid Sprinkler = true, WetGrass = true)$ 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 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.

METROPOLIS–
HASTINGS

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

we are interested in sampling from $\pi(\mathbf{x}) = P(\mathbf{x} \mid \mathbf{e})$.) Like simulated annealing, Metropolis–Hastings operates in two stages. First, it samples a new state $\mathbf{x}'$ from a **proposal distribution** $q(\mathbf{x}' \mid \mathbf{x})$, given the current state $\mathbf{x}$. Then, it probabilistically accepts or rejects $\mathbf{x}'$ according to the **acceptance probability**

$$\alpha(\mathbf{x}' \mid \mathbf{x}) = \min \left(1, \frac{\pi(\mathbf{x}')q(\mathbf{x} \mid \mathbf{x}')}{\pi(\mathbf{x})q(\mathbf{x}' \mid \mathbf{x})} \right) \ .$$

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.
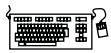
14.21   Three soccer teams A, B, and C, play each other once. Each match is between two teams, and can be won, drawn, or lost. Each team has a fixed, unknown degree of quality—an integer ranging from 0 to 3—and the outcome of a match depends probabilistically on the difference in quality between the 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?
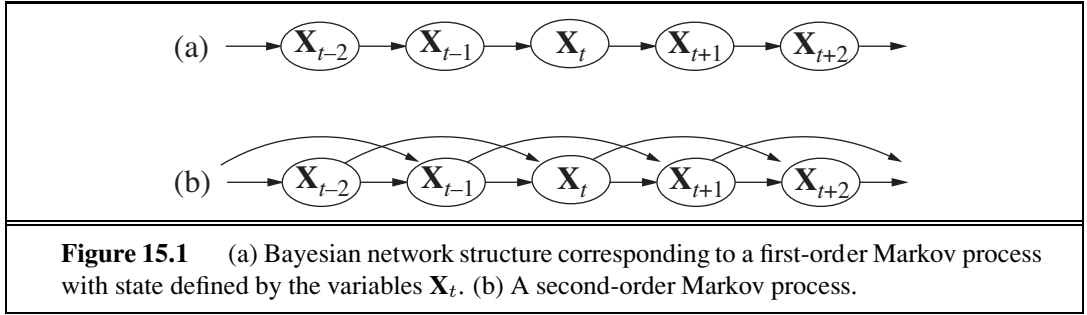
15.1.1   States and observations

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, . . . and evidence variables U_1, U_2, ... 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.

Figure 15.1   (a) Bayesian network structure corresponding to a first-order Markov process with state defined by the variables $\mathbf{X}_t$. (b) A second-order Markov process.

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

$$\mathbf{P}(\mathbf{X}_t \,|\, \mathbf{X}_{0:t-1}) = \mathbf{P}(\mathbf{X}_t \,|\, \mathbf{X}_{t-1}) \,. \tag{15.1}$$

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:

$$\mathbf{P}(\mathbf{E}_t \,|\, \mathbf{X}_{0:t}, \mathbf{E}_{0:t-1}) = \mathbf{P}(\mathbf{E}_t \,|\, \mathbf{X}_t) \,. \tag{15.2}$$
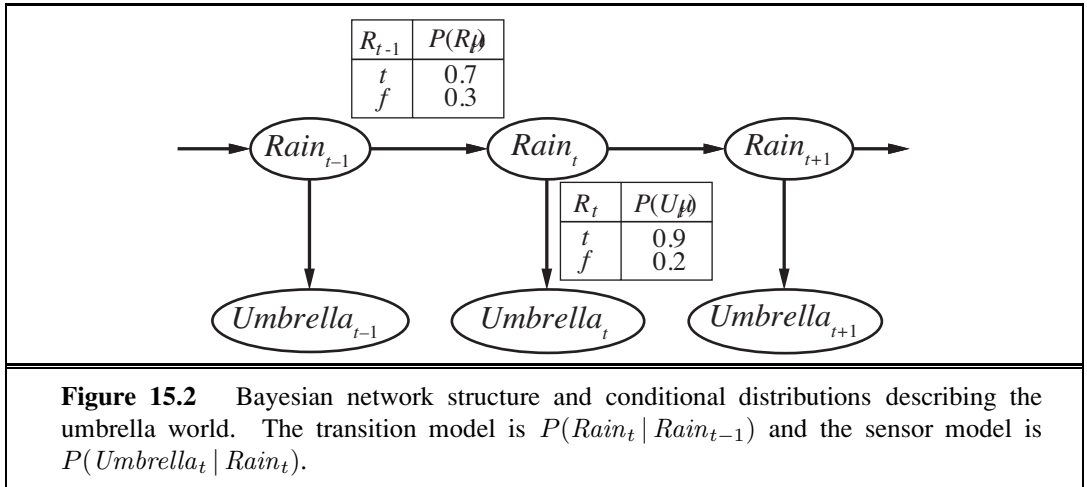
Thus, $\mathbf{P}(\mathbf{E}_t \,|\, \mathbf{X}_t)$ is our sensor model (sometimes called the **observation model**). Figure 15.2 shows both the transition model and the sensor model for the umbrella example. Notice the

MARKOV
ASSUMPTION

MARKOV PROCESS
FIRST-ORDER
MARKOV PROCESS

STATIONARY
PROCESS

SENSOR MARKOV
ASSUMPTION

Figure 15.2   Bayesian network structure and conditional distributions describing the umbrella world. The transition model is $P(Rain_t \mid Rain_{t-1})$ and the sensor model is $P(Umbrella_t \mid Rain_t)$.

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,

$$\mathbf{P}(\mathbf{X}_{0:t}, \mathbf{E}_{1:t}) = \mathbf{P}(\mathbf{X}_0) \prod_{i=1}^{t} \mathbf{P}(\mathbf{X}_i \mid \mathbf{X}_{i-1}) \, \mathbf{P}(\mathbf{E}_i \mid \mathbf{X}_i) \,. \tag{15.3}$$

The three terms on the right-hand side are the initial state model $\mathbf{P}(\mathbf{X}_0)$, the transition model $\mathbf{P}(\mathbf{X}_i \mid \mathbf{X}_{i-1})$, and the sensor model $\mathbf{P}(\mathbf{E}_i \mid \mathbf{X}_i)$.

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. Filtering[2] is also called **state estimation**. In our example, we wish to compute $\mathbf{P}(\mathbf{X}_t \mid \mathbf{e}_{1:t})$. 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 $\mathbf{P}(\mathbf{X}_{t+k} \mid \mathbf{e}_{1:t})$ 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**: 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 $\mathbf{P}(\mathbf{X}_k \mid \mathbf{e}_{1:t})$ 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 $\operatorname{argmax}_{\mathbf{x}_{1:t}} P(\mathbf{x}_{1:t} \mid \mathbf{e}_{1:t})$. 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 $\mathbf{e}_{t+1}$,

$$\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{e}_{1:t+1}) = f(\mathbf{e}_{t+1}, \mathbf{P}(\mathbf{X}_t \mid \mathbf{e}_{1:t})) \,,$$

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:

$$\mathbf{P}(\mathbf{X}_{t+1} \,|\, \mathbf{e}_{1:t+1}) = \mathbf{P}(\mathbf{X}_{t+1} \,|\, \mathbf{e}_{1:t}, \mathbf{e}_{t+1}) \quad \text{(dividing up the evidence)}$$

$$= \alpha \, \mathbf{P}(\mathbf{e}_{t+1} \,|\, \mathbf{X}_{t+1}, \mathbf{e}_{1:t}) \, \mathbf{P}(\mathbf{X}_{t+1} \,|\, \mathbf{e}_{1:t}) \quad \text{(using Bayes' rule)}$$

$$= \alpha \, \mathbf{P}(\mathbf{e}_{t+1} \,|\, \mathbf{X}_{t+1}) \, \mathbf{P}(\mathbf{X}_{t+1} \,|\, \mathbf{e}_{1:t}) \quad \text{(by the sensor Markov assumption).} \qquad (15.4)$$

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 $\mathbf{X}_t$:

$$\mathbf{P}(\mathbf{X}_{t+1} \,|\, \mathbf{e}_{1:t+1}) = \alpha \, \mathbf{P}(\mathbf{e}_{t+1} \,|\, \mathbf{X}_{t+1}) \sum_{\mathbf{x}_t} \mathbf{P}(\mathbf{X}_{t+1} \,|\, \mathbf{x}_t, \mathbf{e}_{1:t}) P(\mathbf{x}_t \,|\, \mathbf{e}_{1:t})$$

$$= \alpha \, \mathbf{P}(\mathbf{e}_{t+1} \,|\, \mathbf{X}_{t+1}) \sum_{\mathbf{x}_t} \mathbf{P}(\mathbf{X}_{t+1} \,|\, \mathbf{x}_t) P(\mathbf{x}_t \,|\, \mathbf{e}_{1:t}) \quad \text{(Markov assumption).} \qquad (15.5)$$

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 $U_1 = true$. The prediction from $t = 0$ to $t = 1$ is

$$\mathbf{P}(R_1) = \sum_{r_0} \mathbf{P}(R_1 \,|\, r_0) P(r_0)$$

$$= \langle 0.7, 0.3 \rangle \times 0.5 + \langle 0.3, 0.7 \rangle \times 0.5 = \langle 0.5, 0.5 \rangle \,.$$

Then the update step simply multiplies by the probability of the evidence for $t = 1$ and normalizes, as shown in Equation (15.4):

$$\mathbf{P}(R_1 \,|\, u_1) = \alpha \, \mathbf{P}(u_1 \,|\, R_1) \mathbf{P}(R_1) = \alpha \, \langle 0.9, 0.2 \rangle \langle 0.5, 0.5 \rangle$$

$$= \alpha \, \langle 0.45, 0.1 \rangle \approx \langle 0.818, 0.182 \rangle \,.$$

- On day 2, the umbrella appears, so $U_2 = true$. The prediction from $t = 1$ to $t = 2$ is

$$\mathbf{P}(R_2 \,|\, u_1) = \sum_{r_1} \mathbf{P}(R_2 \,|\, r_1) P(r_1 \,|\, u_1)$$

$$= \langle 0.7, 0.3 \rangle \times 0.818 + \langle 0.3, 0.7 \rangle \times 0.182 \approx \langle 0.627, 0.373 \rangle \,,$$

and updating it with the evidence for $t = 2$ gives

$$\mathbf{P}(R_2 \,|\, u_1, u_2) = \alpha \, \mathbf{P}(u_2 \,|\, R_2) \mathbf{P}(R_2 \,|\, u_1) = \alpha \, \langle 0.9, 0.2 \rangle \langle 0.627, 0.373 \rangle$$

$$= \alpha \, \langle 0.565, 0.075 \rangle \approx \langle 0.883, 0.117 \rangle \,.$$

Intuitively, the probability of rain increases from day 1 to day 2 because rain persists. Exercise 15.2(a) asks you to investigate this tendency further.

The task of **prediction** can be seen simply as filtering without the addition of new evidence. In fact, the filtering process already incorporates a one-step prediction, and it is easy to derive the following recursive computation for predicting the state at $t + k + 1$ from a prediction for $t + k$:

$$\mathbf{P}(\mathbf{X}_{t+k+1} \,|\, \mathbf{e}_{1:t}) = \sum_{\mathbf{x}_{t+k}} \mathbf{P}(\mathbf{X}_{t+k+1} \,|\, \mathbf{x}_{t+k}) P(\mathbf{x}_{t+k} \,|\, \mathbf{e}_{1:t}) \,. \tag{15.6}$$

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.

MIXING TIME

In addition to filtering and prediction, we can use a forward recursion to compute the **likelihood** of the evidence sequence, $P(\mathbf{e}_{1:t})$. This is a useful quantity if we want to compare different temporal models that might have produced the same evidence sequence (e.g., two different models for the persistence of rain). For this recursion, we use a likelihood message $\boldsymbol{\ell}_{1:t}(\mathbf{X}_t) = \mathbf{P}(\mathbf{X}_t, \mathbf{e}_{1:t})$. It is a simple exercise to show that the message calculation is identical to that for filtering:

$$\boldsymbol{\ell}_{1:t+1} = \text{FORWARD}(\boldsymbol{\ell}_{1:t}, \mathbf{e}_{t+1}) \,.$$

Having computed $\boldsymbol{\ell}_{1:t}$, we obtain the actual likelihood by summing out $\mathbf{X}_t$:

$$L_{1:t} = P(\mathbf{e}_{1:t}) = \sum_{\mathbf{x}_t} \boldsymbol{\ell}_{1:t}(\mathbf{x}_t) \,. \tag{15.7}$$

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 underflow problems with floating-point arithmetic. This is an important problem in practice, but we shall not go into solutions here.

Figure 15.3   Smoothing computes $\mathbf{P}(\mathbf{X}_k \mid \mathbf{e}_{1:t})$, the posterior distribution of the state at some past time k given a complete sequence of observations from 1 to t.
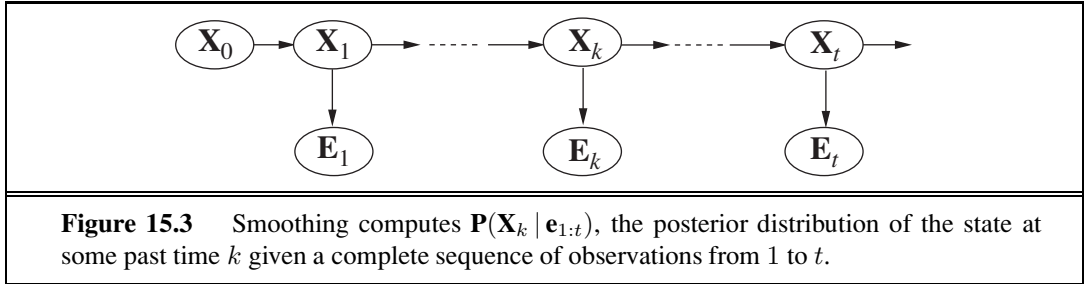
15.2.2 Smoothing

As we said earlier, smoothing is the process of computing the distribution over past states given evidence up to the present; that is, $\mathbf{P}(\mathbf{X}_k \mid \mathbf{e}_{1:t})$ for $0 \le k < t$. (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,

$$
\begin{aligned}
\mathbf{P}(\mathbf{X}_k \mid \mathbf{e}_{1:t}) &= \mathbf{P}(\mathbf{X}_k \mid \mathbf{e}_{1:k}, \mathbf{e}_{k+1:t}) \\
&= \alpha\, \mathbf{P}(\mathbf{X}_k \mid \mathbf{e}_{1:k}) \mathbf{P}(\mathbf{e}_{k+1:t} \mid \mathbf{X}_k, \mathbf{e}_{1:k}) \quad \text{(using Bayes' rule)} \\
&= \alpha\, \mathbf{P}(\mathbf{X}_k \mid \mathbf{e}_{1:k}) \mathbf{P}(\mathbf{e}_{k+1:t} \mid \mathbf{X}_k) \quad \text{(using conditional independence)} \\
&= \alpha\, \mathbf{f}_{1:k} \times \mathbf{b}_{k+1:t} \,.
\end{aligned} \tag{15.8}
$$

where "$\times$" represents pointwise multiplication of vectors. Here we have defined a "backward" message $\mathbf{b}_{k+1:t} = \mathbf{P}(\mathbf{e}_{k+1:t} \mid \mathbf{X}_k)$, 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:

$$
\begin{aligned}
\mathbf{P}(\mathbf{e}_{k+1:t} \mid \mathbf{X}_k) &= \sum_{\mathbf{x}_{k+1}} \mathbf{P}(\mathbf{e}_{k+1:t} \mid \mathbf{X}_k, \mathbf{x}_{k+1}) \mathbf{P}(\mathbf{x}_{k+1} \mid \mathbf{X}_k) \quad \text{(conditioning on } \mathbf{X}_{k+1}) \\
&= \sum_{\mathbf{x}_{k+1}} P(\mathbf{e}_{k+1:t} \mid \mathbf{x}_{k+1}) \mathbf{P}(\mathbf{x}_{k+1} \mid \mathbf{X}_k) \quad \text{(by conditional independence)} \\
&= \sum_{\mathbf{x}_{k+1}} P(\mathbf{e}_{k+1}, \mathbf{e}_{k+2:t} \mid \mathbf{x}_{k+1}) \mathbf{P}(\mathbf{x}_{k+1} \mid \mathbf{X}_k) \\
&= \sum_{\mathbf{x}_{k+1}} P(\mathbf{e}_{k+1} \mid \mathbf{x}_{k+1}) P(\mathbf{e}_{k+2:t} \mid \mathbf{x}_{k+1}) \mathbf{P}(\mathbf{x}_{k+1} \mid \mathbf{X}_k) \,,
\end{aligned} \tag{15.9}
$$

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 $\mathbf{b}_{t+1:t} = \mathbf{P}(\mathbf{e}_{t+1:t} \mid \mathbf{X}_t) = \mathbf{P}(\ \mid \mathbf{X}_t)\mathbf{1}$, 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

$$\mathbf{P}(R_1 \mid u_1, u_2) = \alpha\, \mathbf{P}(R_1 \mid u_1)\, \mathbf{P}(u_2 \mid R_1)\ . \tag{15.10}$$

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):

$$
\begin{aligned}
\mathbf{P}(u_2 \mid R_1) &= \sum_{r_2} P(u_2 \mid r_2)P(\ \mid r_2)\mathbf{P}(r_2 \mid R_1) \\
&= (0.9 \times 1 \times \langle 0.7, 0.3 \rangle) + (0.2 \times 1 \times \langle 0.3, 0.7 \rangle) = \langle 0.69, 0.41 \rangle\ .
\end{aligned}
$$

Plugging this into Equation (15.10), we find that the smoothed estimate for rain on day 1 is

$$\mathbf{P}(R_1 \mid u_1, u_2) = \alpha\, \langle 0.818, 0.182 \rangle \times \langle 0.69, 0.41 \rangle \approx \langle 0.883, 0.117 \rangle\ .$$

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.

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 $1, \ldots, t$
          prior, the prior distribution on the initial state, $\mathbf{P}(\mathbf{X}_0)$
   local variables: **fv**, a vector of forward messages for steps $0, \ldots, t$
               b, a representation of the backward message, initially all 1s
               sv, a vector of smoothed estimates for steps $1, \ldots, t$

   $\mathbf{fv}[0] \leftarrow prior$
   for $i = 1$ **to** t **do**
      $\mathbf{fv}[i] \leftarrow$ FORWARD($\mathbf{fv}[i-1], \mathbf{ev}[i]$)
   for $i = t$ **downto** 1 **do**
      $\mathbf{sv}[i] \leftarrow$ NORMALIZE($\mathbf{fv}[i] \times \mathbf{b}$)
      $\mathbf{b} \leftarrow$ BACKWARD($\mathbf{b}, \mathbf{ev}[i]$)
   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.
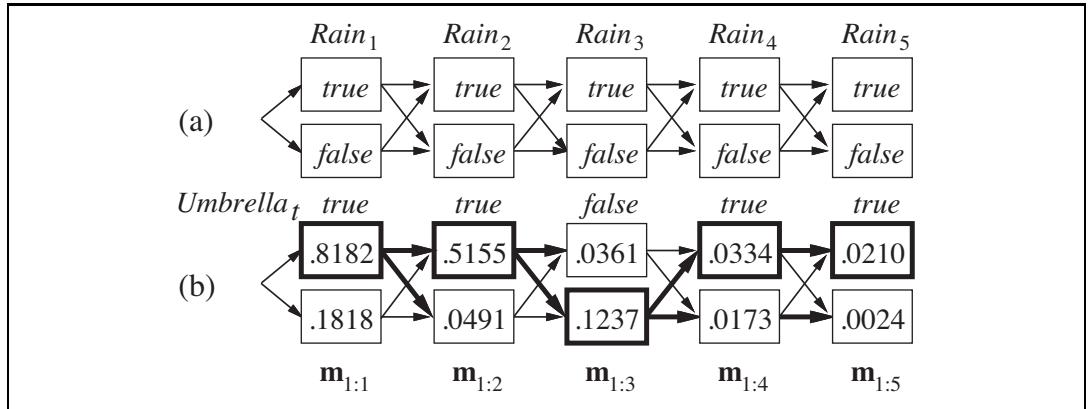
Figure 15.5   (a) Possible state sequences for $Rain_t$ can be viewed as paths through a graph of the possible states at each time step. (States are shown as rectangles to avoid confusion with nodes in a Bayes net.) (b) Operation of the Viterbi algorithm for the umbrella observation sequence $[true, true, false, true, true]$. For each t, we have shown the values of the message $\mathbf{m}_{1:t}$, which gives the probability of the best sequence reaching each state at time t. Also, for each state, the bold arrow leading into it indicates its best predecessor as measured by the product of the preceding sequence probability and the transition probability. Following the bold arrows back from the most likely state in $\mathbf{m}_{1:5}$ gives the most likely sequence.

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 $\mathbf{x}_{t+1}$ and most likely paths to each state $\mathbf{x}_t$*. We can write this relationship as an equation connecting the probabilities of the paths:

$$\max_{\mathbf{x}_1...\mathbf{x}_t} \mathbf{P}(\mathbf{x}_1, \ldots, \mathbf{x}_t, \mathbf{X}_{t+1} \mid \mathbf{e}_{1:t+1})$$

$$= \alpha\, \mathbf{P}(\mathbf{e}_{t+1} \mid \mathbf{X}_{t+1}) \max_{\mathbf{x}_t} \left(\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{x}_t) \max_{\mathbf{x}_1...\mathbf{x}_{t-1}} P(\mathbf{x}_1, \ldots, \mathbf{x}_{t-1}, \mathbf{x}_t \mid \mathbf{e}_{1:t}) \right) . \quad (15.11)$$

Equation (15.11) is *identical* to the filtering equation (15.5) except that

1. The forward message $\mathbf{f}_{1:t} = \mathbf{P}(\mathbf{X}_t \mid \mathbf{e}_{1:t})$ is replaced by the message

$$\mathbf{m}_{1:t} = \max_{\mathbf{x}_1 \dots \mathbf{x}_{t-1}} \mathbf{P}(\mathbf{x}_1, \dots, \mathbf{x}_{t-1}, \mathbf{X}_t \mid \mathbf{e}_{1:t}) \,,$$

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 **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 $1, \ldots, S$, where S is the number of possible states. The transition model $\mathbf{P}(X_t \mid X_{t-1})$ becomes an $S \times S$ matrix $\mathbf{T}$, where

$$\mathbf{T}_{ij} = P(X_t = j \mid X_{t-1} = i) \ .$$

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

$$\mathbf{T} = \mathbf{P}(X_t \mid X_{t-1}) = \begin{pmatrix} 0.7 & 0.3 \\ 0.3 & 0.7 \end{pmatrix} \ .$$

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 $P(e_t \mid X_t = i)$ for each state i. For mathematical convenience we place these values into an $S \times S$ diagonal matrix, $\mathbf{O}_t$ whose ith diagonal entry is $P(e_t \mid X_t = i)$ and whose other entries are 0. For example, on day 1 in the umbrella world of Figure 15.5, $U_1 = true$, and on day 3, $U_3 = false$, so, from Figure 15.2, we have

$$\mathbf{O}_1 = \begin{pmatrix} 0.9 & 0 \\ 0 & 0.2 \end{pmatrix}; \qquad \mathbf{O}_3 = \begin{pmatrix} 0.1 & 0 \\ 0 & 0.8 \end{pmatrix} \ .$$

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

$$\mathbf{f}_{1:t+1} = \alpha \, \mathbf{O}_{t+1} \mathbf{T}^{\top} \mathbf{f}_{1:t} \tag{15.12}$$

and the backward equation (15.9) becomes

$$\mathbf{b}_{k+1:t} = \mathbf{T} \mathbf{O}_{k+1} \mathbf{b}_{k+2:t} \ . \tag{15.13}$$

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**
    $\mathbf{f} \leftarrow$ FORWARD($\mathbf{f}, e_t$)
    remove e_{t-d-1} from the beginning of $e_{t-d:t}$
    $\mathbf{O}_{t-d} \leftarrow$ diagonal matrix containing $\mathbf{P}(e_{t-d}|X_{t-d})$
    $\mathbf{B} \leftarrow \mathbf{O}_{t-d}^{-1}\mathbf{T}^{-1}\mathbf{B}\mathbf{T}\mathbf{O}_t$
  else $\mathbf{B} \leftarrow \mathbf{B}\mathbf{T}\mathbf{O}_t$
  $t \leftarrow t + 1$
  if $t > d$ **then return** NORMALIZE($\mathbf{f} \times \mathbf{B1}$) **else return** null

---

Figure 15.6   An algorithm for smoothing with a fixed time lag of d steps, implemented as an online algorithm that outputs the new smoothed estimate given the observation for a new time step. Notice that the final output NORMALIZE($\mathbf{f} \times \mathbf{B1}$) is just $\alpha\,\mathbf{f} \times \mathbf{b}$, by Equation (15.14).

---

b and **f** together, using them to compute the smoothed estimate at each step. Since only one copy of each message is needed, the storage requirements are constant (i.e., independent of t, the length of the sequence). There 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

$$\mathbf{b}_{t-d+1:t} = \left(\prod_{i=t-d+1}^{t} \mathbf{TO}_i \right) \mathbf{b}_{t+1:t} = \mathbf{B}_{t-d+1:t}\mathbf{1} \,, \tag{15.14}$$

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:

$$\mathbf{b}_{t-d+2:t+1} = \left(\prod_{i=t-d+2}^{t+1} \mathbf{TO}_i \right) \mathbf{b}_{t+2:t+1} = \mathbf{B}_{t-d+2:t+1}\mathbf{1} \,. \tag{15.15}$$

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:

$$\mathbf{B}_{t-d+2:t+1} = \mathbf{O}_{t-d+1}^{-1}\mathbf{T}^{-1}\mathbf{B}_{t-d+1:t}\mathbf{TO}_{t+1} \,. \tag{15.16}$$

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.

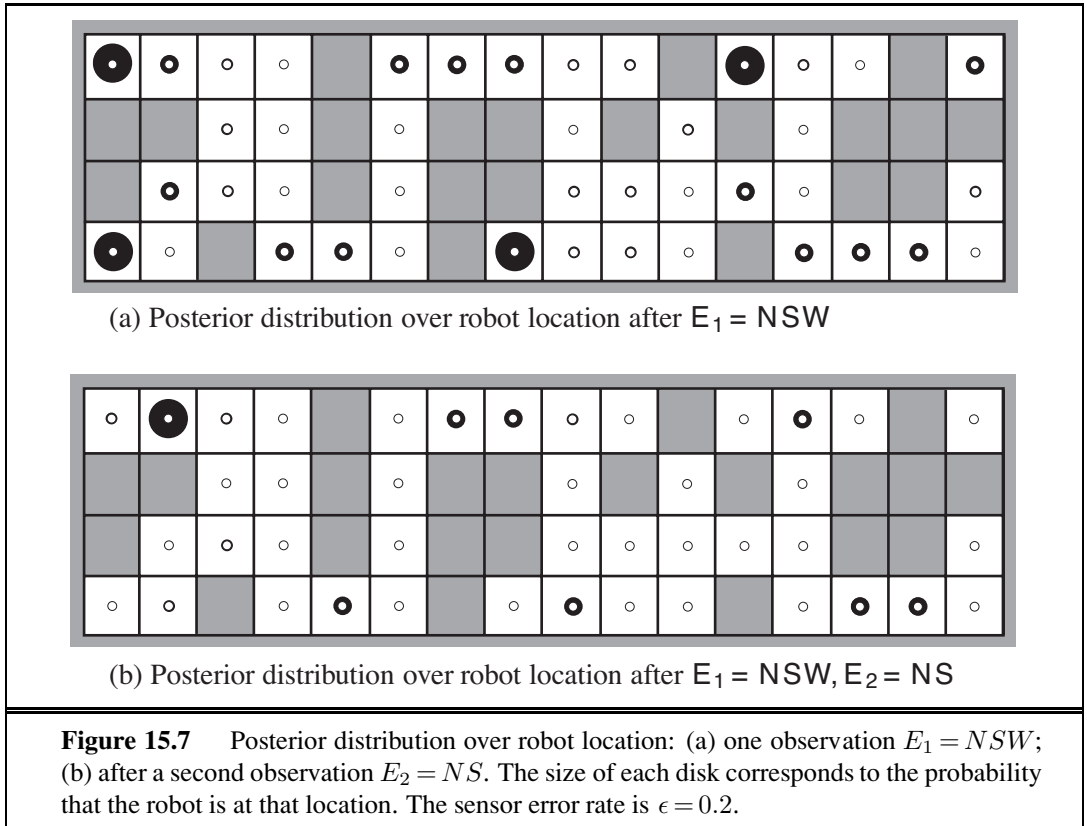### 15.3.2   Hidden Markov model example: Localization

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 $\{s_1, \ldots, s_n\}$. Let 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
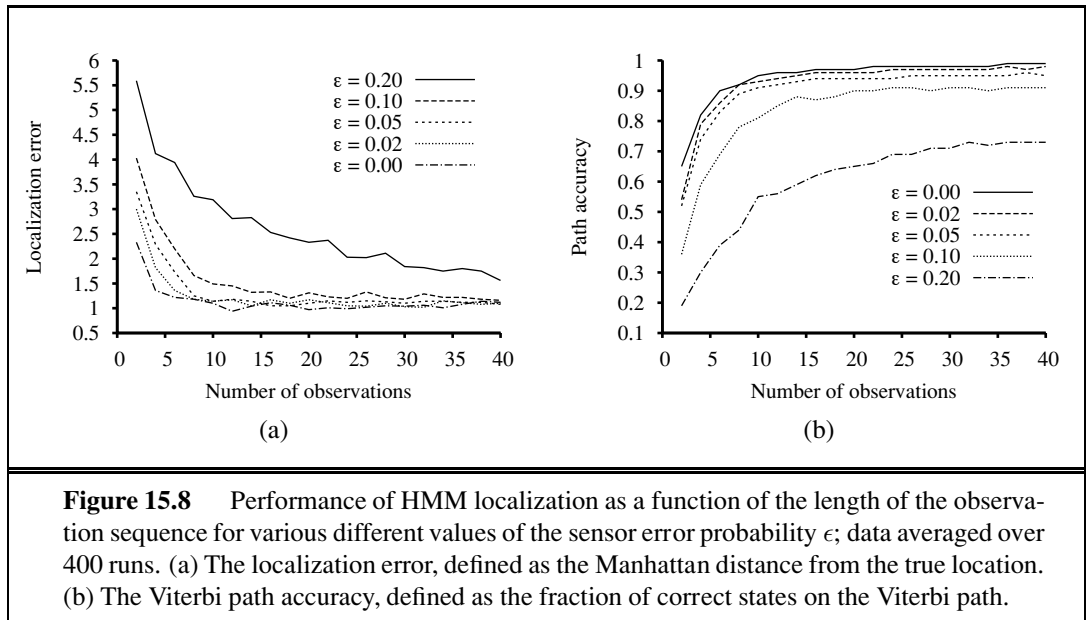
(a) Posterior distribution over robot location after $E_1 = NSW$

(b) Posterior distribution over robot location after $E_1 = NSW, E_2 = NS$

Figure 15.7   Posterior distribution over robot location: (a) one observation $E_1 = NSW$; (b) after a second observation $E_2 = NS$. The size of each disk corresponds to the probability that the robot is at that location. The sensor error rate is $\epsilon = 0.2$.

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 $\mathbf{P}(X_1 \mid E_1 = NSW)$ and $\mathbf{P}(X_2 \mid E_1 = NSW, E_2 = NS)$. 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

Figure 15.8   Performance of HMM localization as a function of the length of the observation sequence for various different values of the sensor error probability ϵ; data averaged over 400 runs. (a) The localization error, defined as the Manhattan distance from the true location. (b) The Viterbi path accuracy, defined as the fraction of correct states on the Viterbi path.

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

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.

KALMAN FILTERING

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 $X_{t+\Delta} = X_t + \dot{X}\,\Delta$. 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 $\boldsymbol{\Sigma}$.
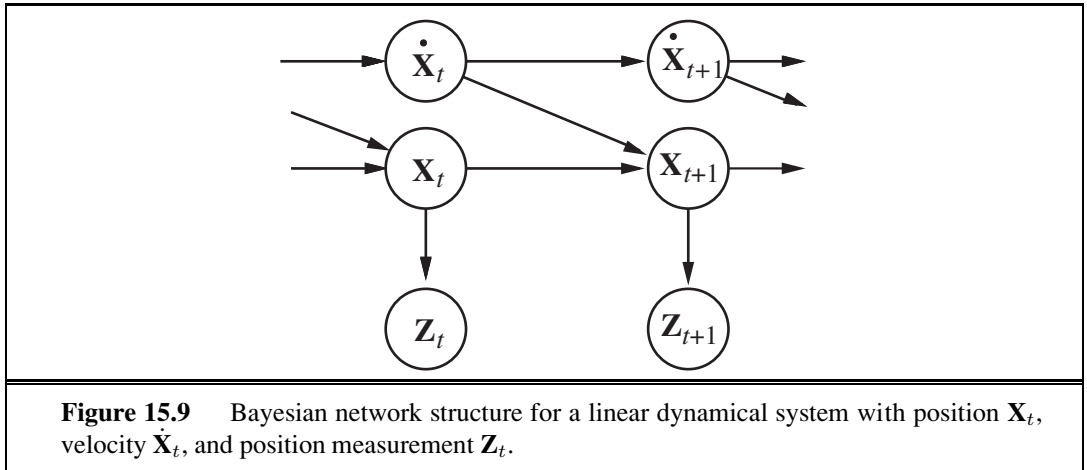
MULTIVARIATE
GAUSSIAN

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

$$\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{e}_{1:t}) = \int_{\mathbf{x}_t} \mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{x}_t) P(\mathbf{x}_t \mid \mathbf{e}_{1:t})\, d\mathbf{x}_t \qquad (15.17)$$

is also a Gaussian distribution.

Figure 15.9   Bayesian network structure for a linear dynamical system with position $\mathbf{X}_t$, velocity $\dot{\mathbf{X}}_t$, and position measurement $\mathbf{Z}_t$.

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

$$\mathbf{P}(\mathbf{X}_{t+1}\,|\,\mathbf{e}_{1:t+1}) = \alpha\,\mathbf{P}(\mathbf{e}_{t+1}\,|\,\mathbf{X}_{t+1})\mathbf{P}(\mathbf{X}_{t+1}\,|\,\mathbf{e}_{1:t}) \tag{15.18}$$

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:

$$P(x_{t+1} \mid x_t) = \alpha\, e^{-\frac{1}{2}\left(\frac{(x_{t+1} - x_t)^2}{\sigma_x^2}\right)}.$$

The sensor model assumes Gaussian noise with variance σ_z^2:

$$P(z_t \mid x_t) = \alpha\, e^{-\frac{1}{2}\left(\frac{(z_t - x_t)^2}{\sigma_z^2}\right)}.$$

Now, given the prior $\mathbf{P}(X_0)$, the one-step predicted distribution comes from Equation (15.17):

$$
\begin{aligned}
P(x_1) &= \int_{-\infty}^{\infty} P(x_1 \mid x_0) P(x_0)\, dx_0 = \alpha \int_{-\infty}^{\infty} e^{-\frac{1}{2}\left(\frac{(x_1 - x_0)^2}{\sigma_x^2}\right)} e^{-\frac{1}{2}\left(\frac{(x_0 - \mu_0)^2}{\sigma_0^2}\right)} dx_0 \\
&= \alpha \int_{-\infty}^{\infty} e^{-\frac{1}{2}\left(\frac{\sigma_0^2(x_1 - x_0)^2 + \sigma_x^2(x_0 - \mu_0)^2}{\sigma_0^2 \sigma_x^2}\right)} dx_0 .
\end{aligned}
$$

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

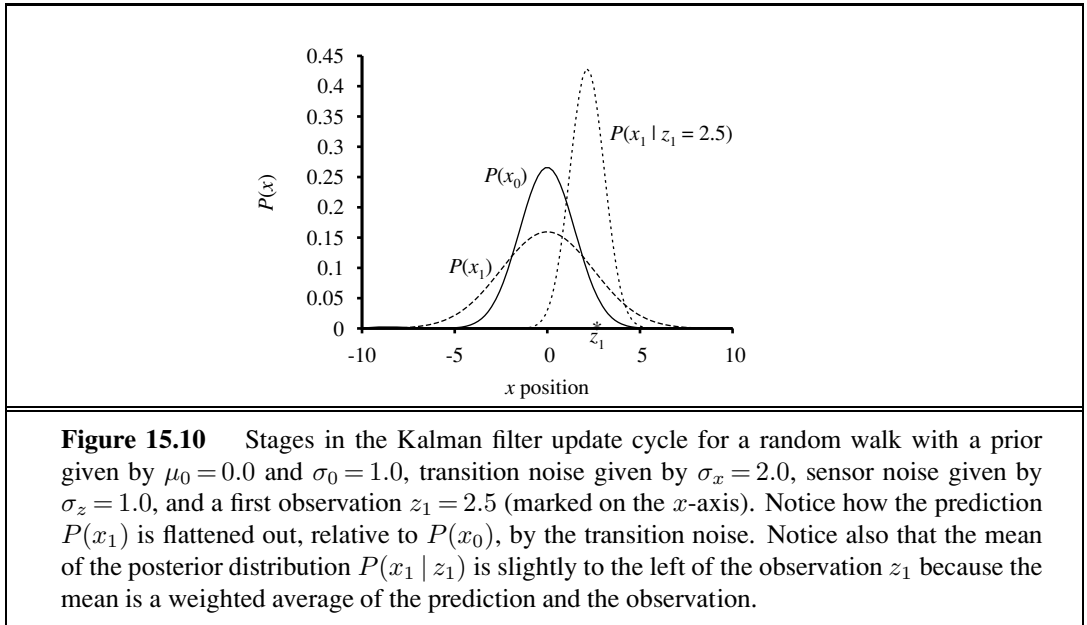$$P(x_1) = \alpha\, e^{-\frac{1}{2}\left(\frac{(x_1 - \mu_0)^2}{\sigma_0^2 + \sigma_x^2}\right)}.$$

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

$$
\begin{aligned}
P(x_1 \mid z_1) &= \alpha\, P(z_1 \mid x_1) P(x_1) \\
&= \alpha\, e^{-\frac{1}{2}\left(\frac{(z_1 - x_1)^2}{\sigma_z^2}\right)} e^{-\frac{1}{2}\left(\frac{(x_1 - \mu_0)^2}{\sigma_0^2 + \sigma_x^2}\right)} .
\end{aligned}
$$

Once again, we combine the exponents and complete the square (Exercise 15.11), obtaining

$$P(x_1 \mid z_1) = \alpha\, e^{-\frac{1}{2}\left(\frac{(x_1 - \frac{(\sigma_0^2 + \sigma_x^2)z_1 + \sigma_z^2 \mu_0}{\sigma_0^2 + \sigma_x^2 + \sigma_z^2})^2}{(\sigma_0^2 + \sigma_x^2)\sigma_z^2 / (\sigma_0^2 + \sigma_x^2 + \sigma_z^2)}\right)} . \tag{15.19}$$

COMPLETING THE SQUARE

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 $P(x_1 \mid z_1)$ 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:

$$\mu_{t+1} = \frac{(\sigma_t^2 + \sigma_x^2)z_{t+1} + \sigma_z^2 \mu_t}{\sigma_t^2 + \sigma_x^2 + \sigma_z^2} \qquad \text{and} \qquad \sigma_{t+1}^2 = \frac{(\sigma_t^2 + \sigma_x^2)\sigma_z^2}{\sigma_t^2 + \sigma_x^2 + \sigma_z^2} . \qquad (15.20)$$

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

$$N(\boldsymbol{\mu}, \boldsymbol{\Sigma})(\mathbf{x}) = \alpha \, e^{-\frac{1}{2}\left((\mathbf{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right)} .$$

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

$$
\begin{aligned}
P(\mathbf{x}_{t+1} \mid \mathbf{x}_t) &= N(\mathbf{F}\mathbf{x}_t, \boldsymbol{\Sigma}_x)(\mathbf{x}_{t+1}) \\
P(\mathbf{z}_t \mid \mathbf{x}_t) &= N(\mathbf{H}\mathbf{x}_t, \boldsymbol{\Sigma}_z)(\mathbf{z}_t) ,
\end{aligned}
\tag{15.21}
$$

where $\mathbf{F}$ and $\boldsymbol{\Sigma}_x$ are matrices describing the linear transition model and transition noise covariance, and $\mathbf{H}$ and $\boldsymbol{\Sigma}_z$ are the corresponding matrices for the sensor model. Now the update equations for the mean and covariance, in their full, hairy horribleness, are

$$
\begin{aligned}
\boldsymbol{\mu}_{t+1} &= \mathbf{F}\boldsymbol{\mu}_t + \mathbf{K}_{t+1}(\mathbf{z}_{t+1} - \mathbf{H}\mathbf{F}\boldsymbol{\mu}_t) \\
\boldsymbol{\Sigma}_{t+1} &= (\mathbf{I} - \mathbf{K}_{t+1}\mathbf{H})(\mathbf{F}\boldsymbol{\Sigma}_t\mathbf{F}^\top + \boldsymbol{\Sigma}_x) ,
\end{aligned}
\tag{15.22}
$$

KALMAN GAIN
MATRIX

where $\mathbf{K}_{t+1} = (\mathbf{F}\boldsymbol{\Sigma}_t\mathbf{F}^\top + \boldsymbol{\Sigma}_x)\mathbf{H}^\top(\mathbf{H}(\mathbf{F}\boldsymbol{\Sigma}_t\mathbf{F}^\top + \boldsymbol{\Sigma}_x)\mathbf{H}^\top + \boldsymbol{\Sigma}_z)^{-1}$ 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 $\boldsymbol{\mu}$. The term $\mathbf{F}\boldsymbol{\mu}_t$ is the *predicted* state at $t + 1$, so $\mathbf{H}\mathbf{F}\boldsymbol{\mu}_t$ is the *predicted* observation. Therefore, the term $\mathbf{z}_{t+1} - \mathbf{H}\mathbf{F}\boldsymbol{\mu}_t$ 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 $\boldsymbol{\Sigma}_t$ 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 $\mathbf{F}$, $\boldsymbol{\Sigma}_x$, $\mathbf{H}$, and $\boldsymbol{\Sigma}_z$ 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.
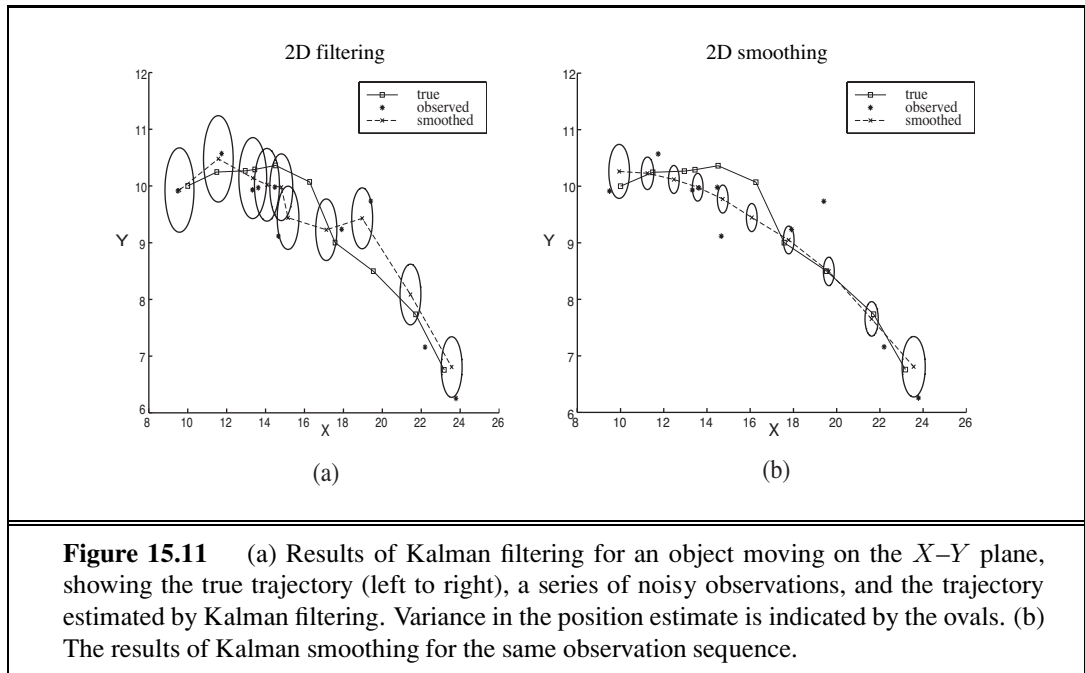
Figure 15.11   (a) Results of Kalman filtering for an object moving on the X–Y plane, showing the true trajectory (left to right), a series of noisy observations, and the trajectory estimated by Kalman filtering. Variance in the position estimate is indicated by the ovals. (b) The results of Kalman smoothing for the same observation sequence.
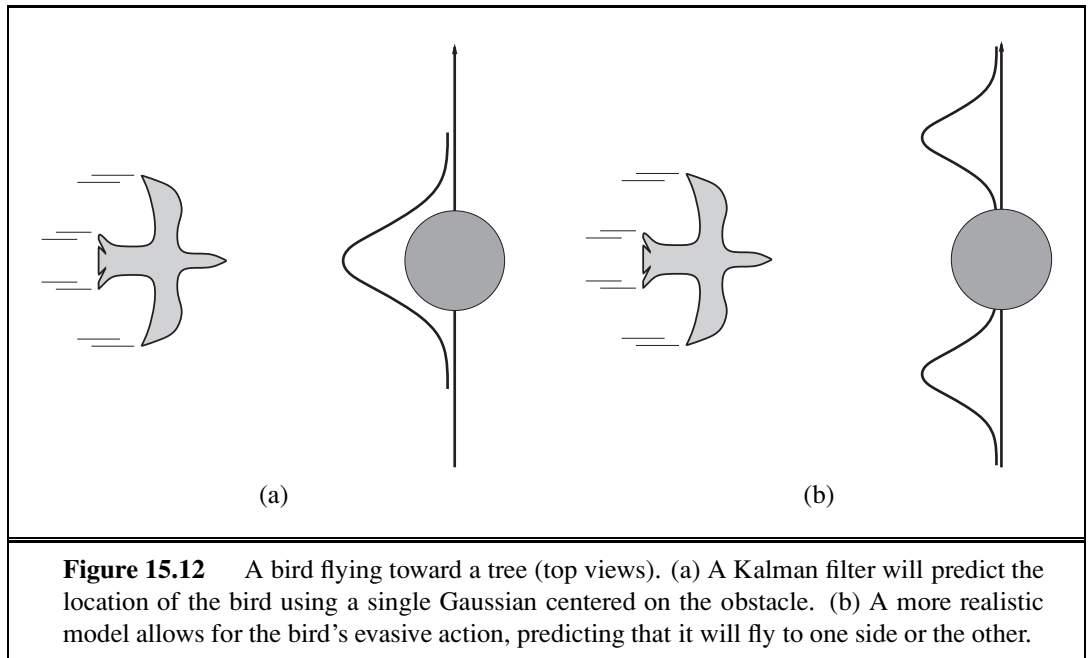
EXTENDED KALMAN
FILTER (EKF)

NONLINEAR

       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.

SWITCHING KALMAN
FILTER

       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-

Figure 15.12   A bird flying toward a tree (top views). (a) A Kalman filter will predict the
location of the bird using a single Gaussian centered on the obstacle. (b) A more realistic
model allows for the bird's evasive action, predicting that it will fly to one side or the other.

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 net-
work 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
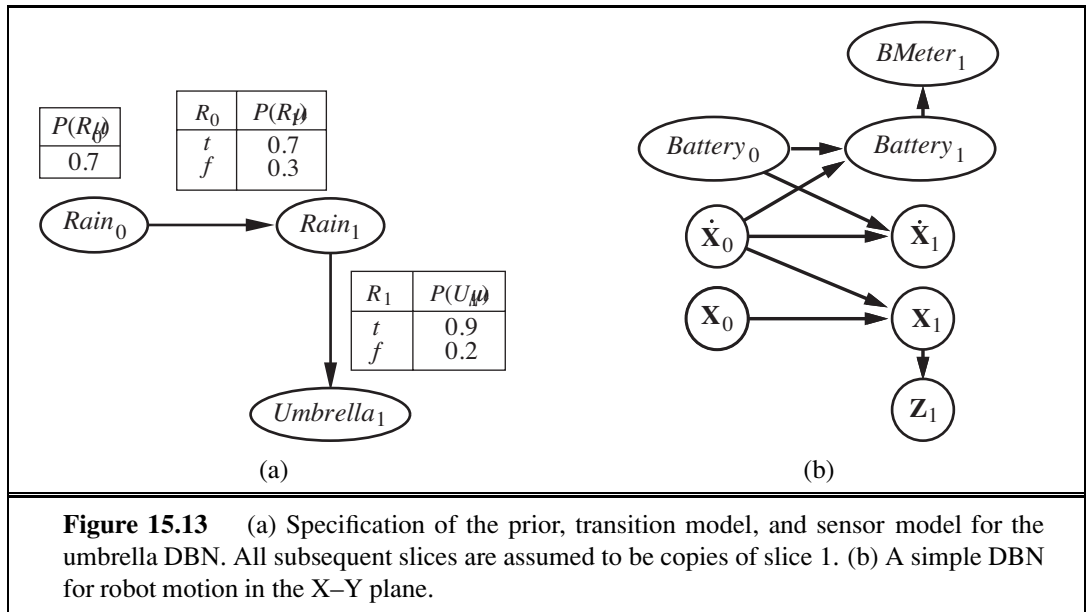
$P(R_0)$
0.7

R_0	$P(R_1)$
t	0.7
f	0.3

R_1	$P(U_1)$
t	0.9
f	0.2

(a)   (b)

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 $\mathbf{P}(BMeter_t \mid Battery_t)$ 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.

GAUSSIAN ERROR MODEL

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.

TRANSIENT FAILURE

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

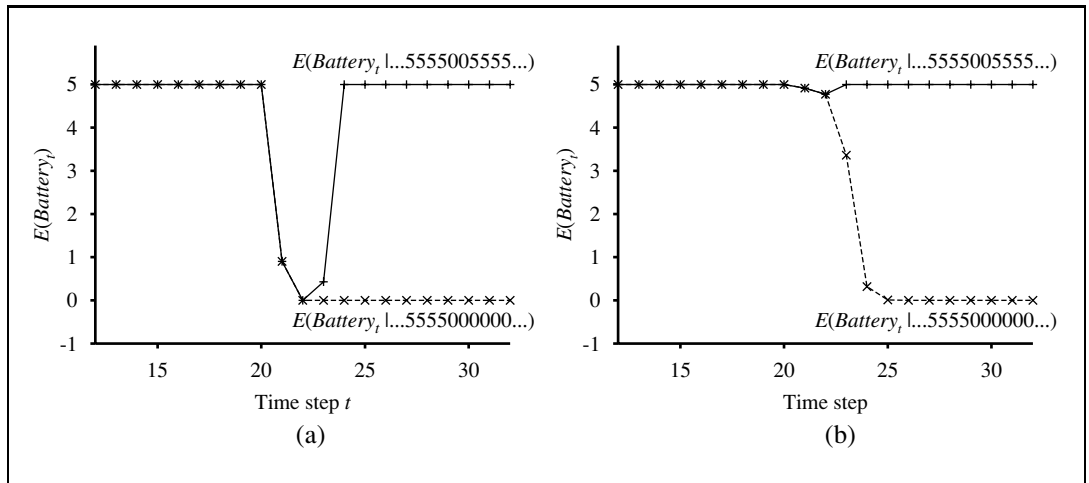TRANSIENT FAILURE
MODEL

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.
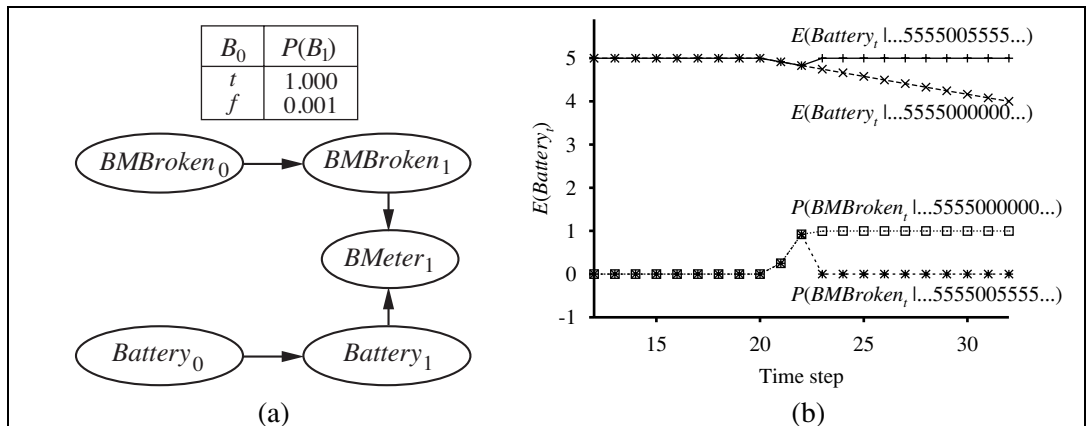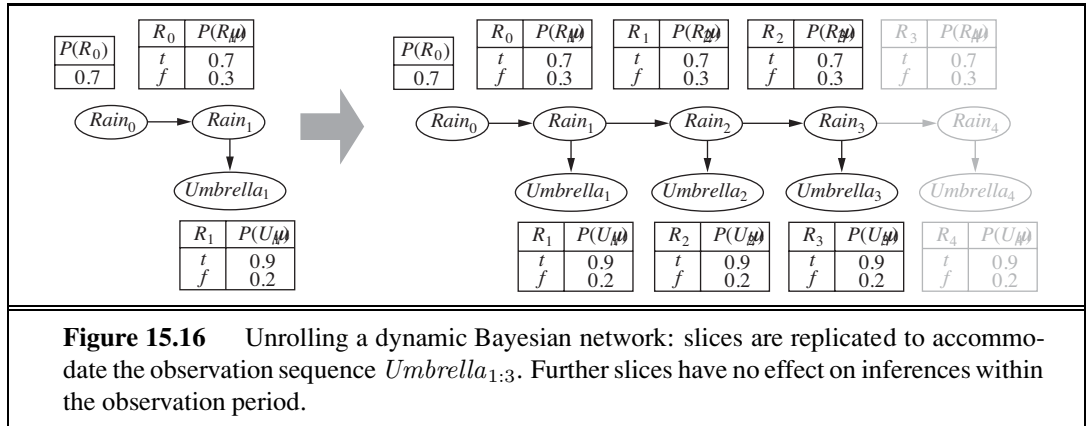

Figure 15.15 (a) A DBN fragment showing the sensor status variable required for modeling persistent failure of the battery sensor. (b) Upper curves: trajectories of the expected value of $Battery_t$ for the "transient failure" and "permanent failure" observations sequences. Lower curves: probability 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.

$P(R_0)$		R_0	$P(R_1)$
0.7		t	0.7
		f	0.3

$Rain_0 \rightarrow Rain_1$

$Umbrella_1$

R_1	$P(U_1)$
t	0.9
f	0.2

$P(R_0)$		R_0	$P(R_1)$		R_1	$P(R_2)$		R_2	$P(R_3)$		R_3	$P(R_4)$
0.7		t	0.7		t	0.7		t	0.7		t	0.7
		f	0.3		f	0.3		f	0.3		f	0.3

$Rain_0 \rightarrow Rain_1 \rightarrow Rain_2 \rightarrow Rain_3 \rightarrow Rain_4$

$Umbrella_1 \quad Umbrella_2 \quad Umbrella_3 \quad Umbrella_4$

R_1	$P(U_1)$		R_2	$P(U_2)$		R_3	$P(U_3)$		R_4	$P(U_4)$
t	0.9		t	0.9		t	0.9		t	0.9
f	0.2		f	0.2		f	0.2		f	0.2

Figure 15.16   Unrolling a dynamic Bayesian network: slices are replicated to accommodate the observation sequence $Umbrella_{1:3}$. Further slices have no effect on inferences within the observation period.

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 $E(Battery_t \mid \ldots)$.

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(**e**, N, dbn) **returns** a set of samples for the next time step
   inputs: **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 $\mathbf{P}(\mathbf{X}_1 \mid \mathbf{X}_0 = S[i])$ /* step 1 */
      $W[i] \leftarrow \mathbf{P}(\mathbf{e} \mid \mathbf{X}_1 = S[i])$ /* 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.
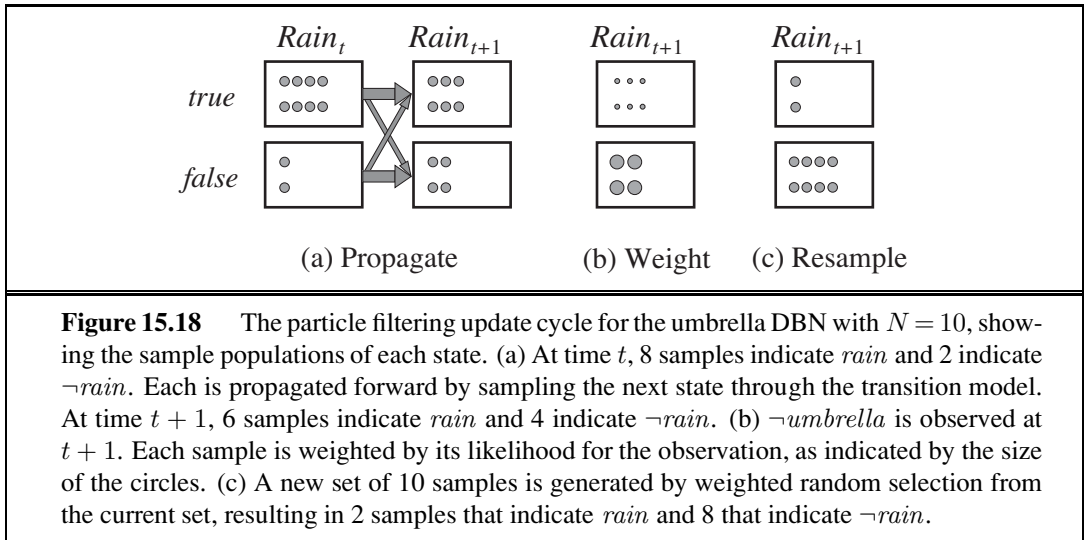

Figure 15.18   The particle filtering update cycle for the umbrella DBN with $N = 10$, showing the sample populations of each state. (a) At time t, 8 samples indicate $rain$ and 2 indicate $\neg rain$. Each is propagated forward by sampling the next state through the transition model. At time $t + 1$, 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 $\mathbf{f}_{1:t} = \mathbf{P}(\mathbf{X}_t \mid \mathbf{e}_{1:t})$ at time t. Writing $N(\mathbf{x}_t \mid \mathbf{e}_{1:t})$ for the number of samples occupying state $\mathbf{x}_t$ after observations $\mathbf{e}_{1:t}$ have been processed, we therefore have

$$N(\mathbf{x}_t \mid \mathbf{e}_{1:t})/N = P(\mathbf{x}_t \mid \mathbf{e}_{1:t}) \qquad\qquad (15.23)$$

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:
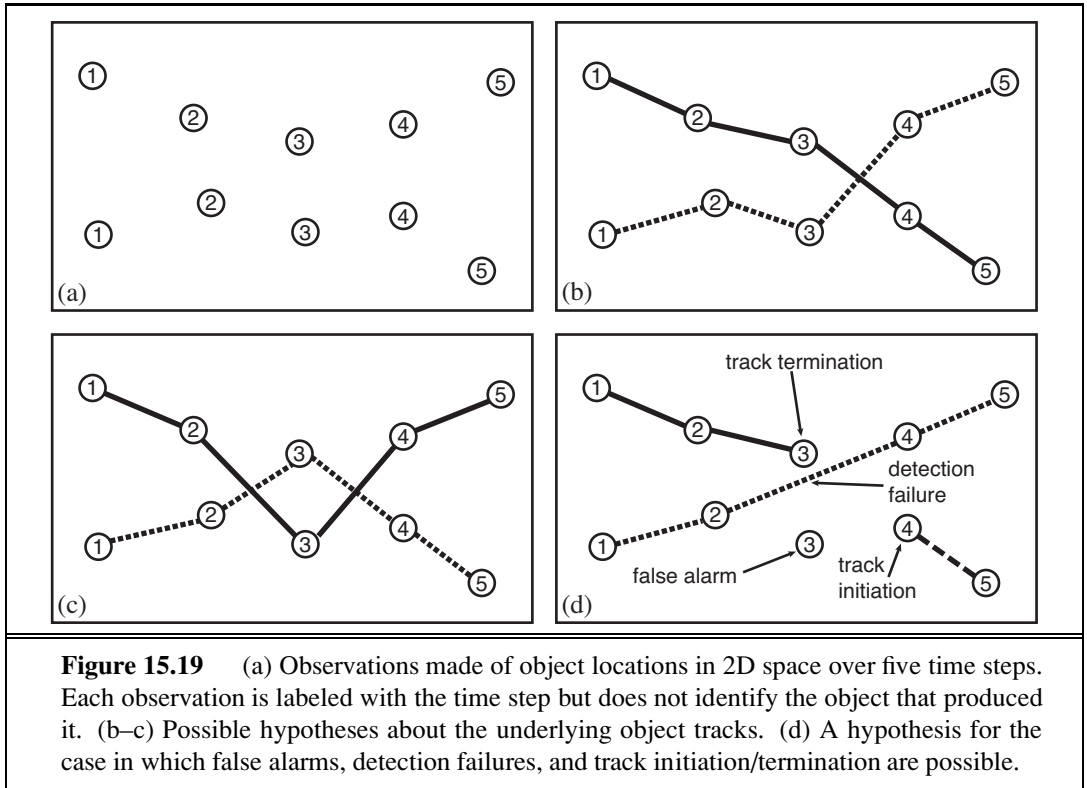
$$
\begin{aligned}
N(\mathbf{x}_{t+1} \,|\, \mathbf{e}_{1:t+1})/N &= \alpha\, W(\mathbf{x}_{t+1} \,|\, \mathbf{e}_{1:t+1}) \\
&= \alpha\, P(\mathbf{e}_{t+1} \,|\, \mathbf{x}_{t+1}) N(\mathbf{x}_{t+1} \,|\, \mathbf{e}_{1:t}) \\
&= \alpha\, P(\mathbf{e}_{t+1} \,|\, \mathbf{x}_{t+1}) \sum_{\mathbf{x}_t} P(\mathbf{x}_{t+1} \,|\, \mathbf{x}_t) N(\mathbf{x}_t \,|\, \mathbf{e}_{1:t}) \\
&= \alpha\, NP(\mathbf{e}_{t+1} \,|\, \mathbf{x}_{t+1}) \sum_{\mathbf{x}_t} P(\mathbf{x}_{t+1} \,|\, \mathbf{x}_t) P(\mathbf{x}_t \,|\, \mathbf{e}_{1:t}) \quad \text{(by 15.23)} \\
&= \alpha' P(\mathbf{e}_{t+1} \,|\, \mathbf{x}_{t+1}) \sum_{\mathbf{x}_t} P(\mathbf{x}_{t+1} \,|\, \mathbf{x}_t) P(\mathbf{x}_t \,|\, \mathbf{e}_{1:t}) \\
&= P(\mathbf{x}_{t+1} \,|\, \mathbf{e}_{1:t+1}) \quad \text{(by 15.5).}
\end{aligned}
$$

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.

DATA ASSOCIATION

Figure 15.19   (a) Observations made of object locations in 2D space over five time steps. Each observation is labeled with the time step but does not identify the object that produced it. (b–c) Possible hypotheses about the underlying object tracks. (d) A hypothesis for the case in which false alarms, detection failures, and track initiation/termination are possible.

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 the 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:

$$P(x_{0:t}^A, x_{0:t}^B, e_{1:t}^1, e_{1:t}^2) =$$
$$P(x_0^A)P(x_0^B) \prod_{i=1}^{t} P(x_i^A \,|\, x_{i-1}^A)P(x_i^B \,|\, x_{i-1}^B)\, P(e_i^1, e_i^2 \,|\, x_i^A, x_i^B) \,. \qquad (15.24)$$

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" ad "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 $P(e_i^1, e_i^2 \mid x_i^A, x_i^B)$ on ω_t and then simplify:

$$
\begin{aligned}
P(e_i^1, e_i^2 \mid x_i^A, x_i^B) &= \sum_{\omega_i} P(e_i^1, e_i^2 \mid x_i^A, x_i^B, \omega_i) P(\omega_i \mid x_i^A, x_i^B) \\
&= \sum_{\omega_i} P(e_i^{\omega_i(A)} \mid x_i^A) P(e_i^{\omega_i(B)} \mid x_i^B) P(\omega_i \mid x_i^A, x_i^B) \\
&= \frac{1}{2} \sum_{\omega_i} P(e_i^{\omega_i(A)} \mid x_i^A) P(e_i^{\omega_i(B)} \mid x_i^B) \,.
\end{aligned}
$$

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.
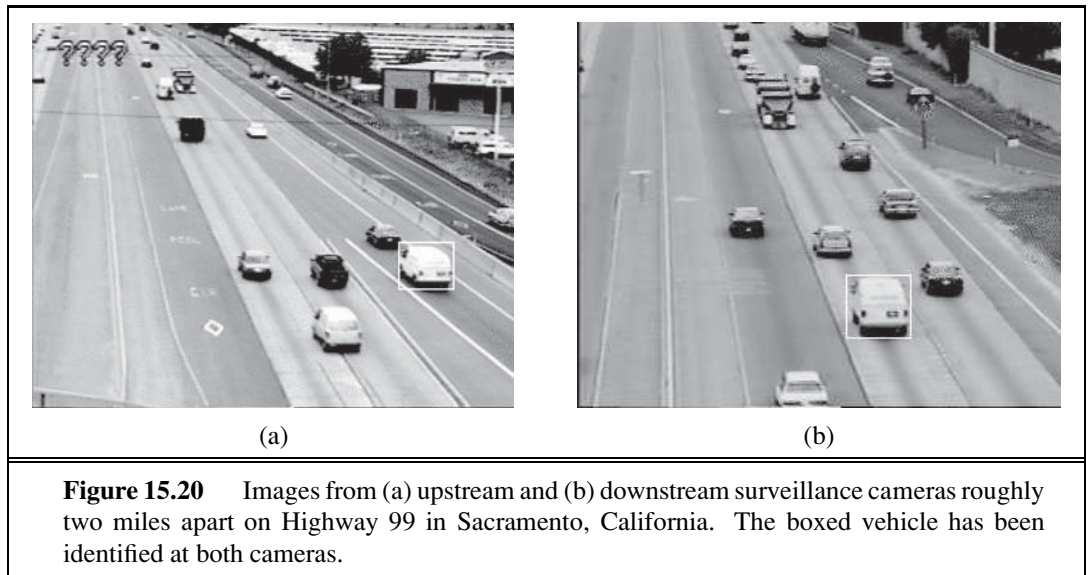
As for all probability models, inference means summing out the variables other than the query and the evidence. For filtering in HMMs and DBNs, we were able to sum out the state variables from 1 to $t-1$ by a simple dynamic programming trick; for Kalman filters, we took advantage of special properties 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 $P(x_t^A \mid e_{1:t}^1, e_{1:t}^2)$ 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.

NEAREST-NEIGHBOR
FILTER

HUNGARIAN
ALGORITHM

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

Figure 15.20   Images from (a) upstream and (b) downstream surveillance cameras roughly two miles apart on Highway 99 in Sacramento, California. The boxed vehicle has been identified at both cameras.

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 Thorvold 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
REVERSAL

RAO-
BLACKWELLIZED
PARTICLE FILTER

SEQUENTIAL MONTE
CARLO

DECAYED MCMC

ASSUMED-DENSITY
FILTER

FACTORED
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 Jenö 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 $k = 1 \ldots 20$ 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 $k = 1, \ldots, t$. 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 $k = 1, \ldots, h$ 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
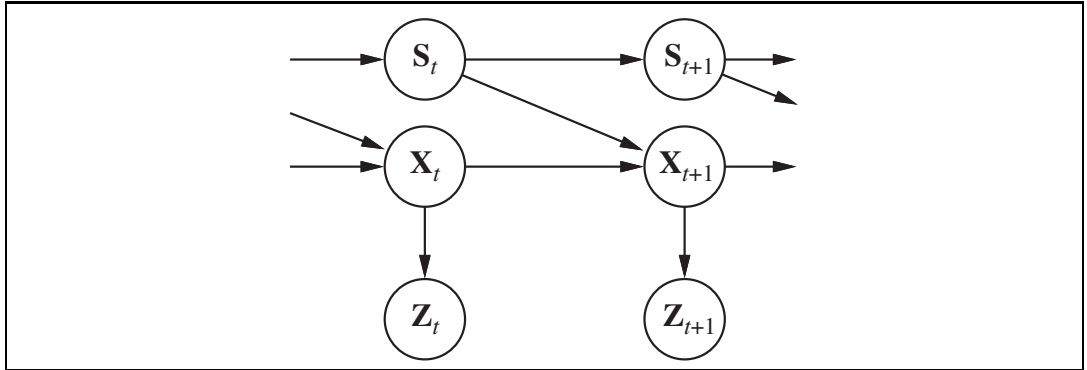
Figure 15.21   A Bayesian network representation of a switching Kalman filter.   The switching variable S_t is a discrete state variable whose value determines the transition model for the continuous state variables $\mathbf{X}_t$. For any discrete state i, the transition model $\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{X}_t, S_t = i)$ 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.

   a. 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.

   b. 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.

   c. 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).

   a. Plot the value of σ_t^2 as a function of t, given various values for σ_x^2 and σ_z^2.

   b. Show that the update has a fixed point σ^2 such that $\sigma_t^2 \to \sigma^2$ as $t \to \infty$, and calculate the value of σ^2.

   c. Give a qualitative explanation for what happens as $\sigma_x^2 \to 0$ and as $\sigma_z^2 \to 0$.

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 $P(EnoughSleep_t|\mathbf{e}_{1:t})$ for each of $t = 1, 2, 3$.
   b. Smoothing: Compute $P(EnoughSleep_t|\mathbf{e}_{1:3})$ 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 $t \to \infty$ 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 **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:

$$EU(a|\mathbf{e}) = \sum_{s'} P(\text{RESULT}(a) = s' \,|\, a, \mathbf{e}) \, U(s') \,. \qquad (16.1)$$

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:

$$action = \underset{a}{\operatorname{argmax}} \, EU(a|\mathbf{e})$$

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

LOTTERY

as a **lottery**—think of each action as a ticket. A lottery L with possible outcomes $S_1, \ldots, S_n$ that occur with probabilities $p_1, \ldots, p_n$ is written

$$L = [p_1, S_1; \ p_2, S_2; \ \ldots \ p_n, S_n] \,.$$

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:

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.

$$U([p_1, S_1; \ldots; p_n, S_n]) = \sum_i p_i U(S_i) \ .$$

---

[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."
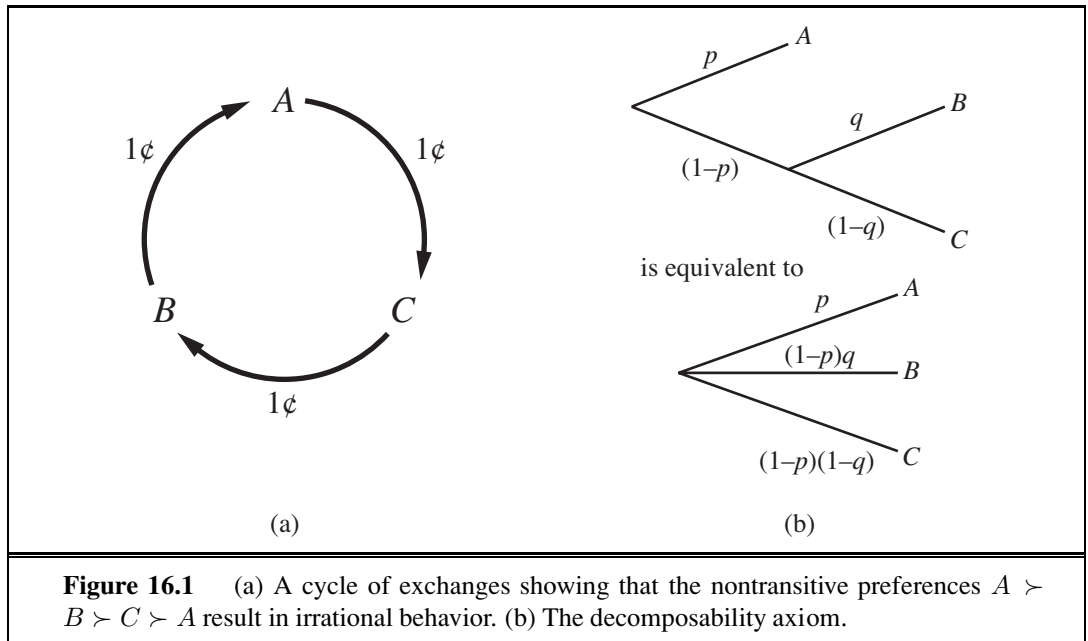
Figure 16.1   (a) A cycle of exchanges showing that the nontransitive preferences $A \succ B \succ C \succ A$ result in irrational behavior. (b) The decomposability axiom.

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

$$U'(S) = aU(S) + b \,, \tag{16.2}$$

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.

VALUE FUNCTION
ORDINAL UTILITY
FUNCTION

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 $U(S) = u_\bot$. **Normalized utilities** use a scale with $u_\bot = 0$ and $u_\top = 1$.

NORMALIZED UTILITIES

Given a utility scale between $u_\top$ and $u_\bot$, we can assess the utility of any particular prize S by asking the agent to choose between S and a **standard lottery** $[p, u_\top; (1-p), u_\bot]$. 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, $u_\bot$ 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.

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.

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.

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?

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

$$EU(Accept) = \tfrac{1}{2}U(S_k) + \tfrac{1}{2}U(S_{k+2,500,000}) ,$$
$$EU(Decline) = U(S_{k+1,000,000}) .$$

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
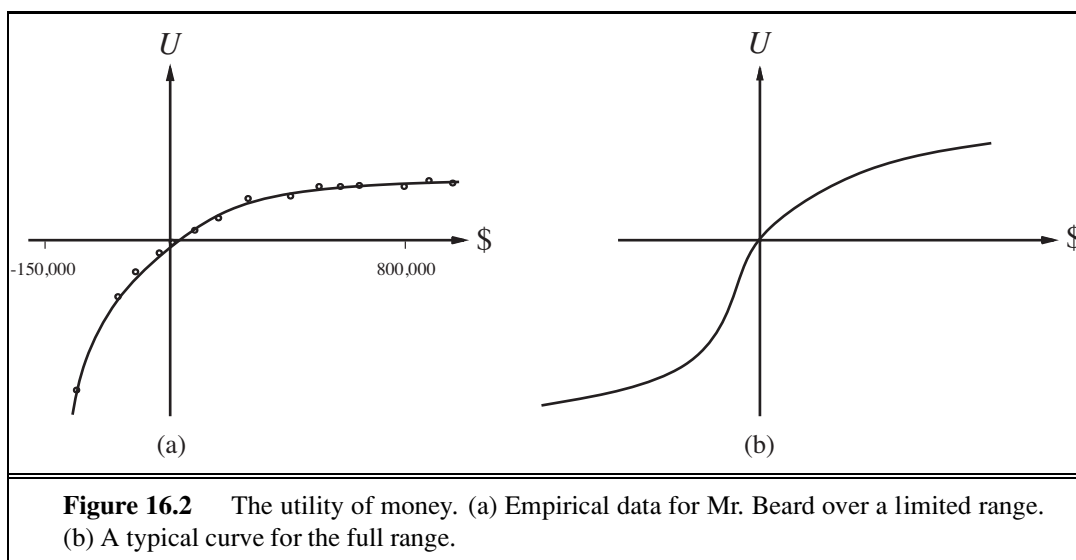
Figure 16.2   The utility of money. (a) Empirical data for Mr. Beard over a limited range. (b) A typical curve for the full range.

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

INSURANCE
PREMIUM

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.

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

UNBIASED

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.

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.

Figure 16.3   Plot of the error in each of k utility estimates and of the distribution of the maximum of k estimates for $k = 3$, 10, and 30.
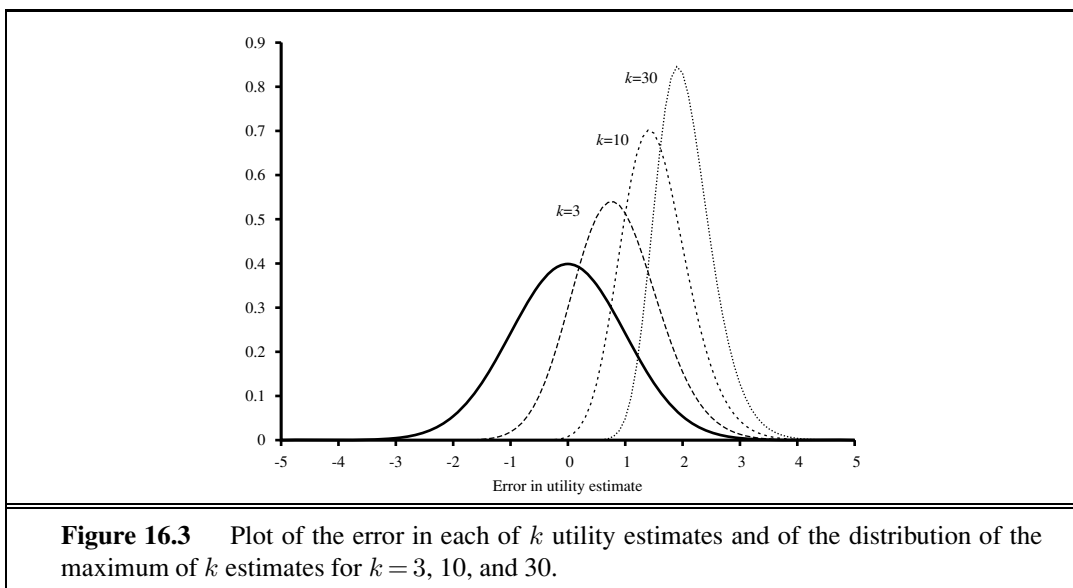
With more choices, extremely optimistic estimates are more likely to arise: for $k = 30$, the disappointment will be around twice the standard deviation in the estimates.

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

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 $U(\$3000) > 0.8\,U(\$4000)$, 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 .

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 received that \$27,000,000 from the Nigerian bank account of a previously unknown deceased relative.

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.

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.

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

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**.

MULTIATTRIBUTE
UTILITY THEORY

We will call the attributes $\mathbf{X} = X_1, \ldots, X_n$; a complete vector of assignments will be $\mathbf{x} = \langle x_1, \ldots, x_n \rangle$, 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.
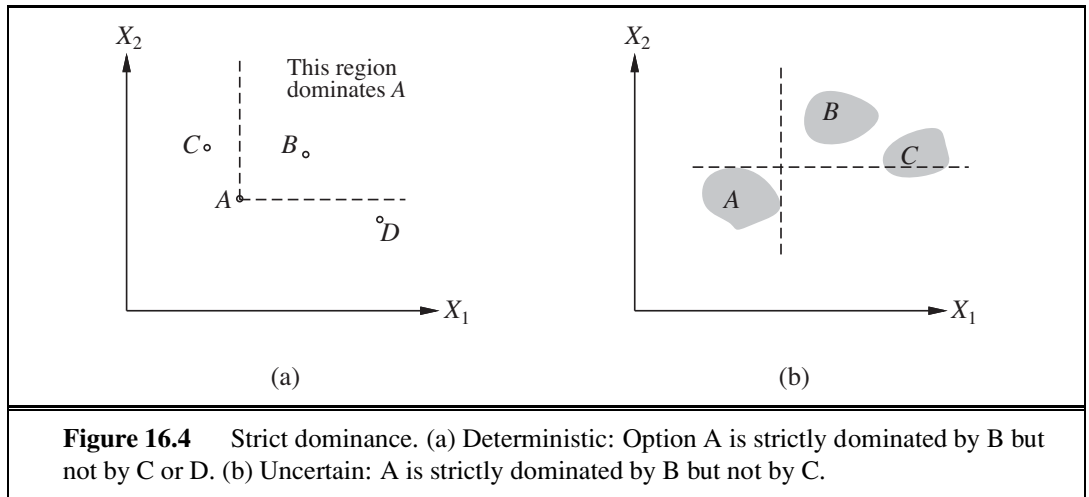
Figure 16.4   Strict dominance. (a) Deterministic: Option A is strictly dominated by B but not by C or D. (b) Uncertain: A is strictly dominated by B but not by C.
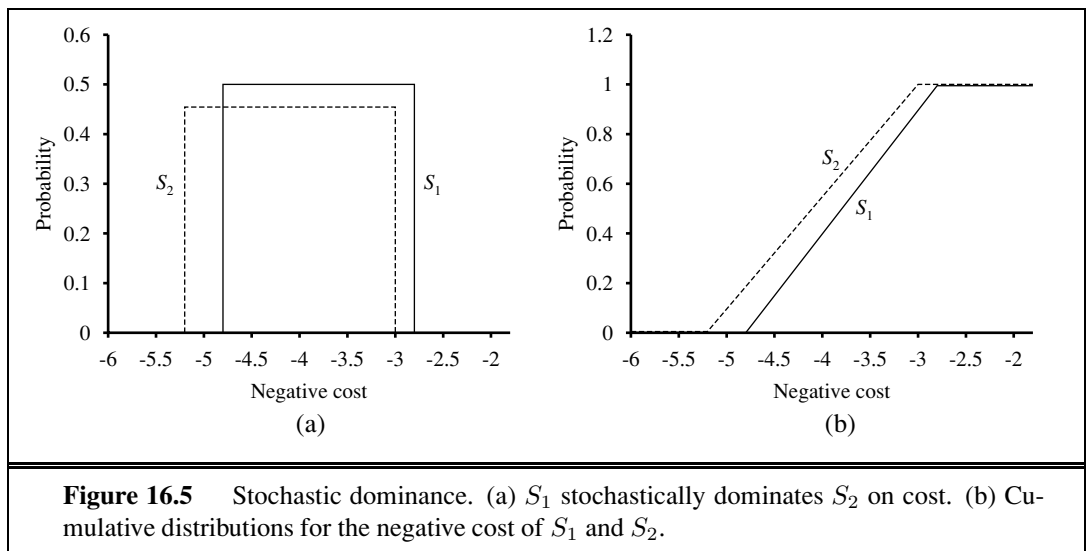

Figure 16.5   Stochastic dominance. (a) S_1 stochastically dominates S_2 on cost. (b) Cumulative distributions for the negative cost of S_1 and S_2.

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

$$\forall\, x \quad \int\limits_{-\infty}^{x} p_1(x')\ dx' \leq \int\limits_{-\infty}^{x} p_2(x')\ dx' \ .$$

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 $U(x_1, \ldots, x_n)$, 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

REPRESENTATION
THEOREM

$$U(x_1, \ldots, x_n) = F[f_1(x_1), \ldots, f_n(x_n)] \ ,$$

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.

Preferences without uncertainty

Let us begin with the deterministic case. Remember that for deterministic environments the agent has a value function $V(x_1, \ldots, x_n)$; 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-

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

MUTUAL PREFERENTIAL INDEPENDENCE

set of attributes $\{Noise, Cost, Deaths\}$ 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 $X_1, \ldots, X_n$ are mutually preferentially independent, then the agent's preference behavior can be described as maximizing the function*

$$V(x_1, \ldots, x_n) = \sum_i V_i(x_i) \,,$$

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

$$V(noise, cost, deaths) = -noise \times 10^4 - cost - deaths \times 10^{12} \,.$$

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

MUTUALLY UTILITY
INDEPENDENT

MULTIPLICATIVE
UTILITY FUNCTION

        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.

        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 $U_i(x_i)$:

$$U = k_1 U_1 + k_2 U_2 + k_3 U_3 + k_1 k_2 U_1 U_2 + k_2 k_3 U_2 U_3 + k_3 k_1 U_3 U_1$$
$$+ \, k_1 k_2 k_3 U_1 U_2 U_3 \; .$$

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

Figure 16.6   A simple decision network for the airport-siting problem.
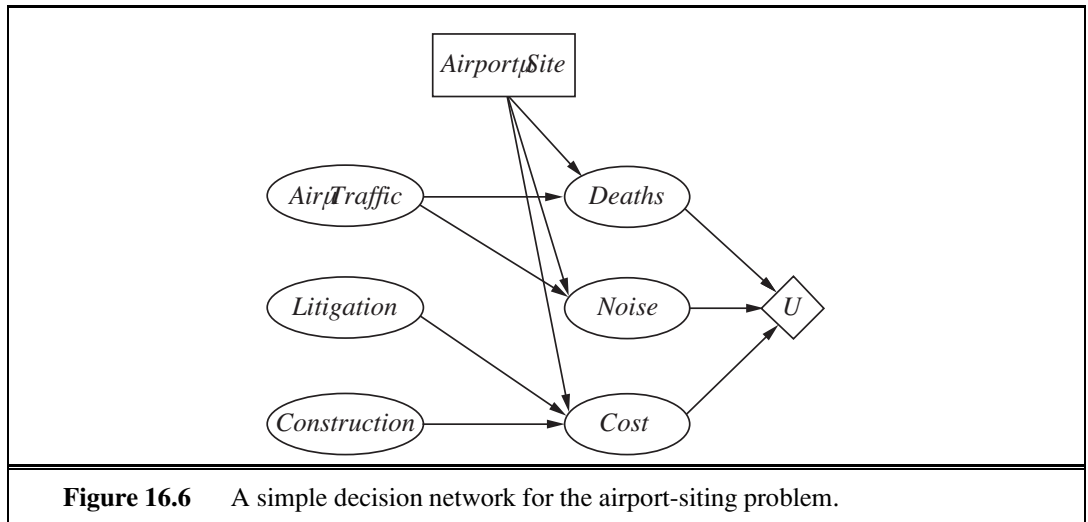
actions. In this case, the $AirportSite$ action can take on a different value for each site under consideration. The choice influences the cost, safety, and noise that will result. In this chapter, we assume that we are dealing with a single decision node. Chapter 17 deals with cases in which more than one decision must be made.
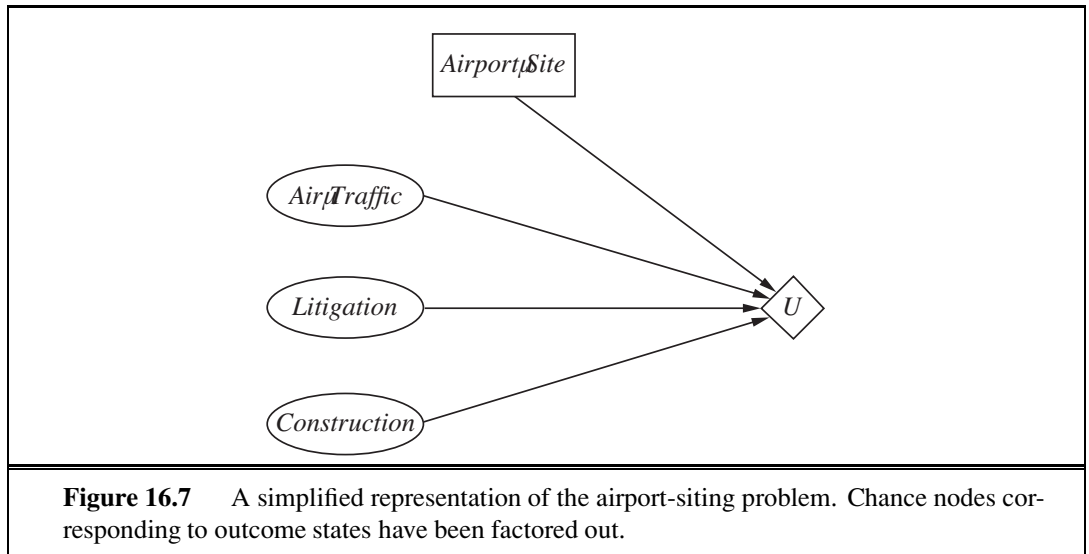
UTILITY NODES

- **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.

Figure 16.7   A simplified representation of the airport-siting problem. Chance nodes corresponding to outcome states have been factored out.

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

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.

INFORMATION VALUE
THEORY

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 **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 **e** we must average over all possible values e_{jk} that we might discover for E_j, using our *current* beliefs about its value:
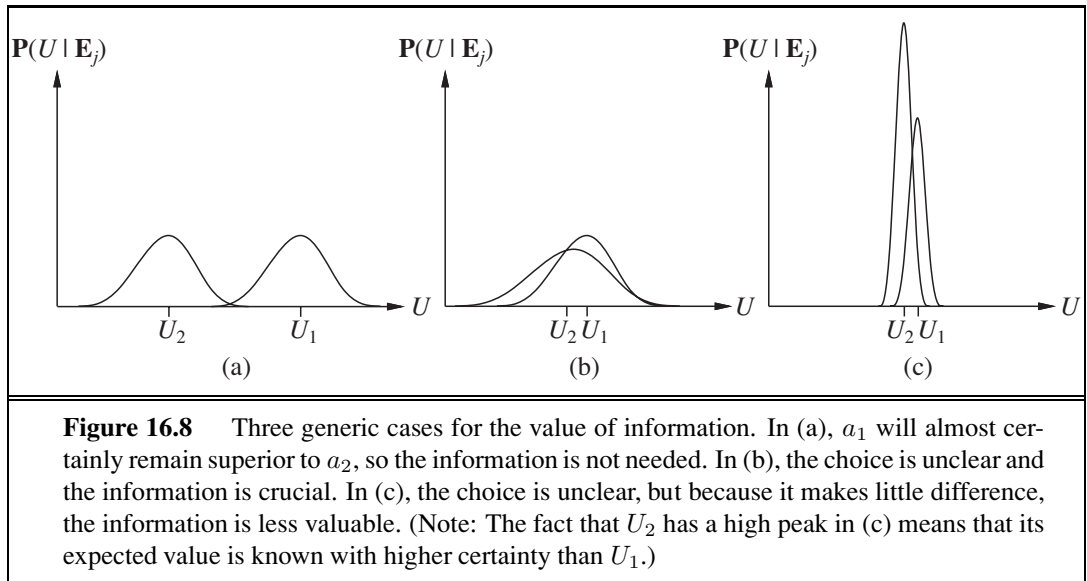
$$VPI_\mathbf{e}(E_j) = \left(\sum_k P(E_j = e_{jk}|\mathbf{e})\; EU(\alpha_{e_{jk}}|\mathbf{e}, E_j = e_{jk}) \right) - EU(\alpha|\mathbf{e}) \ .$$

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 *Thermometer* = 37; this gives us imperfect information about the true *Temperature*, and the uncertainty due to measurement error is encoded in the sensor model **P**(*Thermometer* | *Temperature*). See Exercise 16.17 for another example.

Figure 16.8   Three generic cases for the value of information. In (a), a_1 will almost certainly remain superior to a_2, so the information is not needed. In (b), the choice is unclear and the information is crucial. In (c), the choice is unclear, but because it makes little difference, the information is less valuable. (Note: The fact that U_2 has a high peak in (c) means that its expected value is known with higher certainty than U_1.)

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:

$$\forall \, \mathbf{e}, E_j \quad VPI_{\mathbf{e}}(E_j) \geq 0 \, .$$

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,

$$VPI_{\mathbf{e}}(E_j, E_k) \neq VPI_{\mathbf{e}}(E_j) + VPI_{\mathbf{e}}(E_k) \qquad \text{(in general)} .$$

VPI is, however, order independent. That is,

$$VPI_{\mathbf{e}}(E_j, E_k) = VPI_{\mathbf{e}}(E_j) + VPI_{\mathbf{e},e_j}(E_k) = VPI_{\mathbf{e}}(E_k) + VPI_{\mathbf{e},e_k}(E_j) .$$

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.

MYOPIC   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.)

---

function INFORMATION-GATHERING-AGENT(*percept*) **returns** an *action*
   persistent: D, a decision network

   integrate *percept* into D
   $j \leftarrow$ 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

DECISION MAKER

DECISION ANALYST

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.

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).

AORTIC
COARCTATION

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.

GOLD STANDARD

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,

Figure 16.10   Influence diagram for aortic coarctation (courtesy of Peter Lucas).
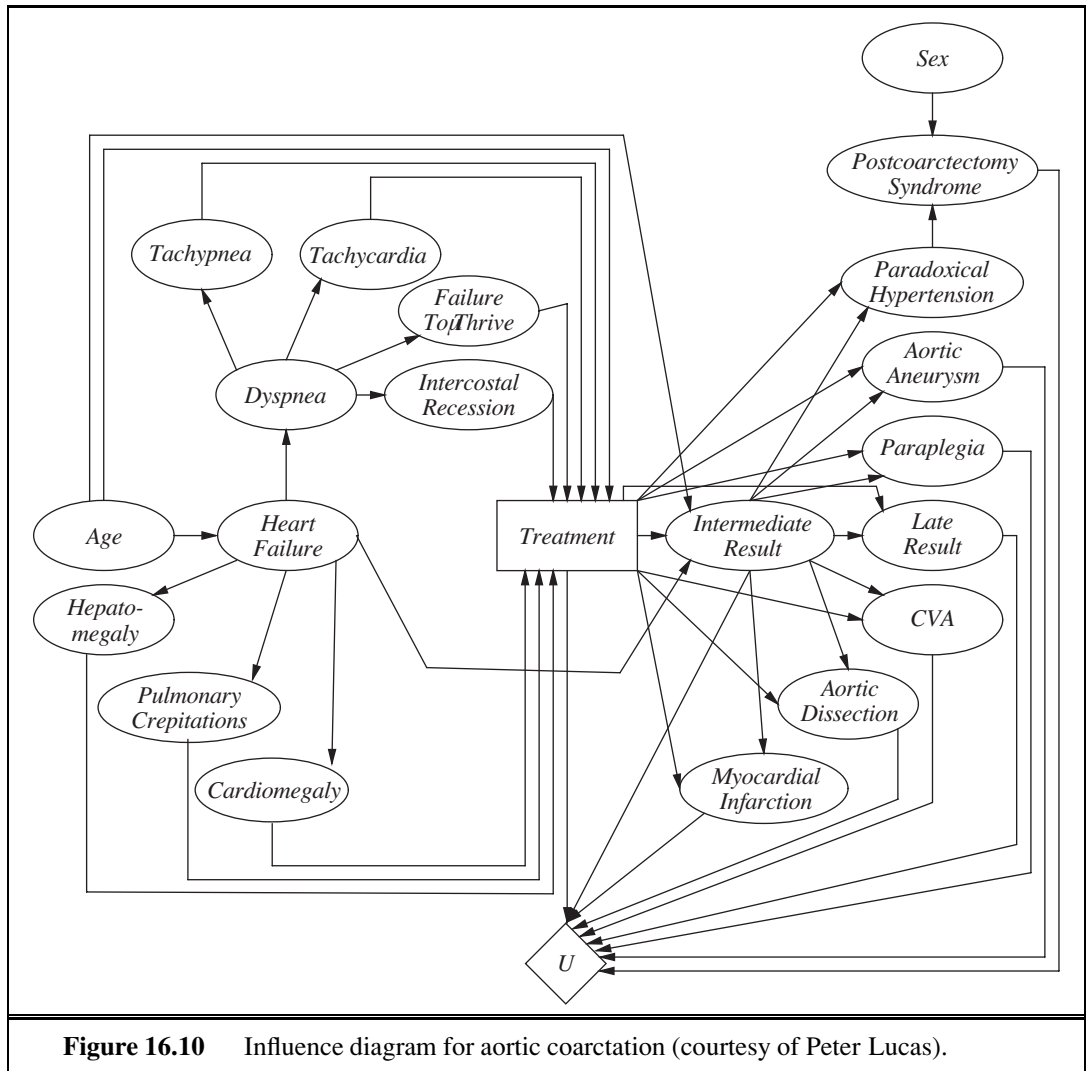
and asking them for their diagnosis and recommended treatment plan. We then see how well the system matches their recommendations. If it does poorly, we try to isolate the parts that are going wrong and fix them. It can be useful to run the system "backward." Instead of presenting the system with symptoms 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 $P(tachycardia \mid dyspnea)$, 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.

REGRESSION TO THE
MEAN

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 nth toss, you win 2^n dollars.

  a. Show that the expected monetary value of this game is infinite.
  b. How much would you, personally, pay to play the game?
  c. 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$). What is the expected utility of the game under this assumption?
  d. What is the maximum amount that it would be rational to pay to play the game, assuming that one's initial wealth is \$$k$?
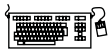
16.4   Write a computer program to automate the process in Exercise 16.9. Try your program out on several people of different net worth and political outlook. Comment on the consistency of your results, both for an individual and across individuals.

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.

Figure 16.11   Three proposed Bayes nets for the Surprise Candy problem, Exercise 16.5.
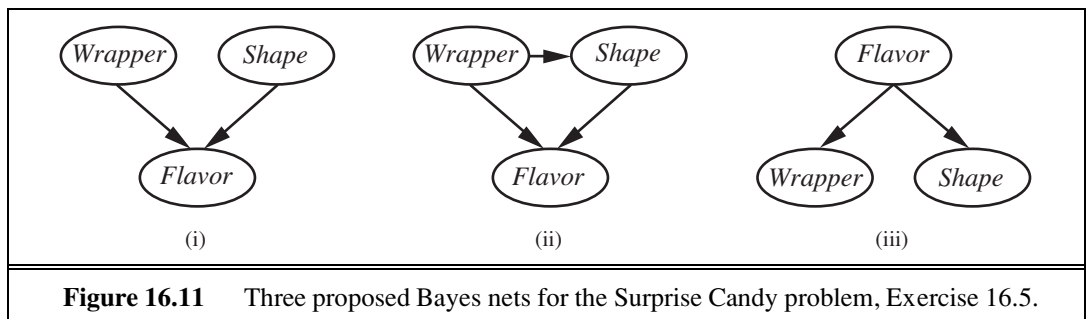
  a. Which network(s) can correctly represent $\mathbf{P}(Flavor, Wrapper, Shape)$?
  b. Which network is the best representation for this problem?

    c. Does network (i) assert that $\mathbf{P}(Wrapper|Shape) = \mathbf{P}(Wrapper)$?

    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 $X_1, \ldots, X_k$ be independently distributed according to the same probability density function $f(x)$. Prove that the density function for $\max\{X_1, \ldots, X_k\}$ 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 Marry 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:

$$P(p|b, m) = 0.9 \qquad P(m|b) = 0.9$$
$$P(p|b, \neg m) = 0.5 \quad P(m|\neg b) = 0.7$$
$$P(p|\neg b, m) = 0.8$$
$$P(p|\neg b, \neg m) = 0.3$$

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, depend-
ing 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:
      $P(pass(c_1, t_1)|q^+(c_1)) = 0.8$
      $P(pass(c_1, t_1)|q^-(c_1)) = 0.35$
      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 ex-
      ample, not wanting to know the sex of their baby when an ultrasound is done.

SUBMODULARITY   **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 in-
      formation, 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.

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 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.
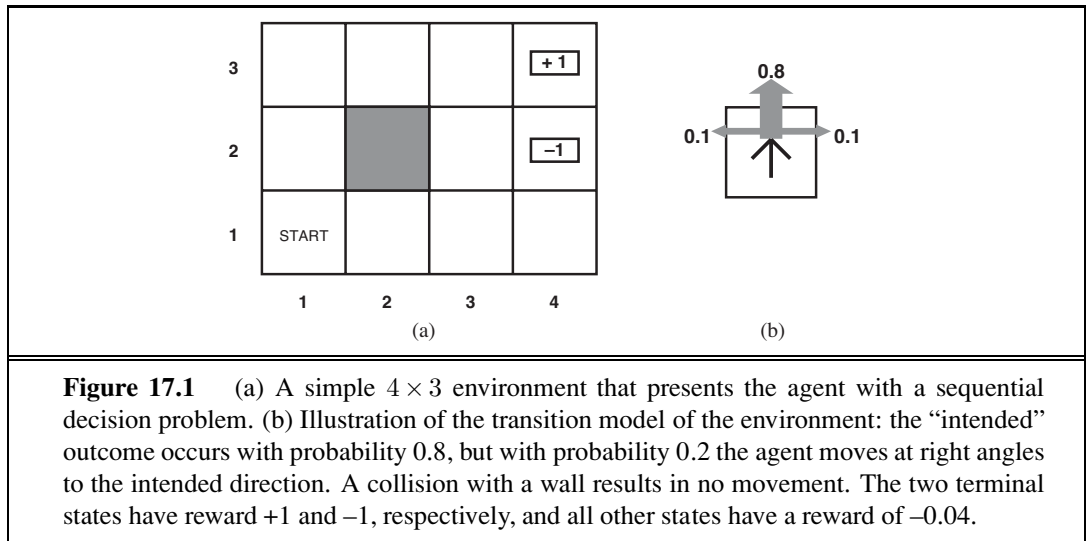
Figure 17.1   (a) A simple 4×3 environment that presents the agent with a sequential decision problem. (b) Illustration of the transition model of the environment: the "intended" outcome occurs with probability 0.8, but with probability 0.2 the agent moves at right angles to the intended direction. A collision with a wall results in no movement. The two terminal states have reward +1 and –1, respectively, and all other states have a reward of –0.04.

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 $P(s' \mid s, a)$ 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 $P(s' \mid s, a)$ 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, REWARD 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.

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 ACTIONS(s) of actions in each state; a transition model $P(s' \mid s, a)$; and a reward function $R(s)$.[1]

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.
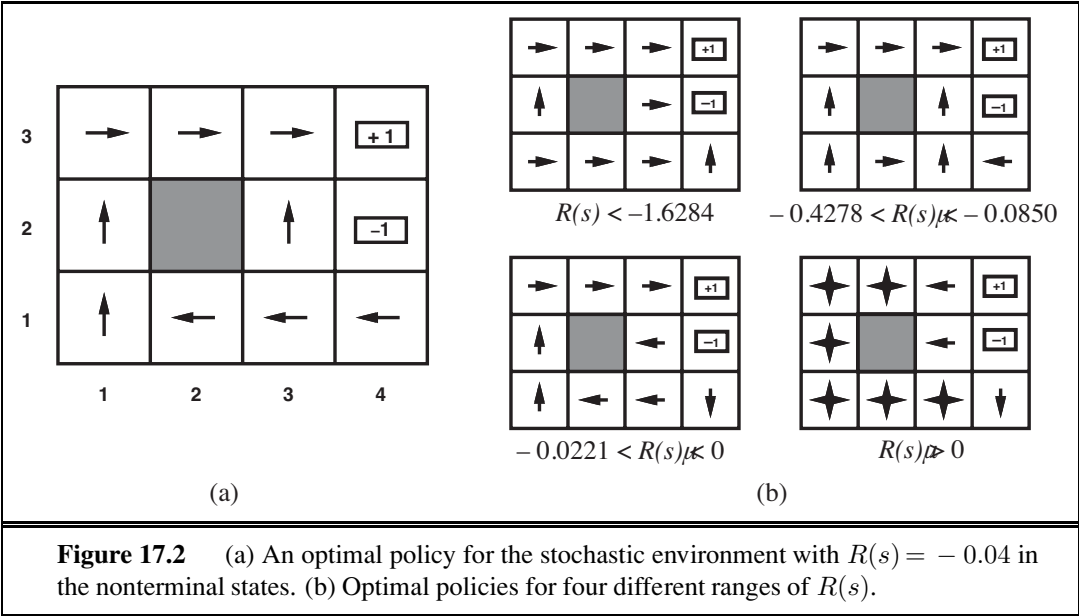
Each time a given policy is executed starting from the initial state, the stochastic nature of the environment may lead to a different environment history. The quality of a policy is therefore measured by the *expected* utility of the possible environment histories generated by that policy. An **optimal 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),

MARKOV DECISION
PROCESS

POLICY

OPTIMAL POLICY

---

[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.

Figure 17.2   (a) An optimal policy for the stochastic environment with $R(s) = -0.04$ in the nonterminal states. (b) Optimal policies for four different ranges of $R(s)$.

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 $U_h([s_0, s_1, \ldots, s_n])$. 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.

FINITE HORIZON

INFINITE HORIZON

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, $U_h([s_0, s_1, \ldots, s_{N+k}]) = U_h([s_0, s_1, \ldots, s_N])$ 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,*

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.

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 $[s_0, s_1, s_2 \ldots]$. 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 $[s_0, s_1, s_2, \ldots]$ and $[s'_0, s'_1, s'_2, \ldots]$ 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 $[s_1, s_2, \ldots]$ and $[s'_1, s'_2, \ldots]$. 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:

1. **Additive rewards**: The utility of a state sequence is

$$U_h([s_0, s_1, s_2, \ldots]) = R(s_0) + R(s_1) + R(s_2) + \cdots .$$

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).

2. **Discounted rewards**: The utility of a state sequence is

$$U_h([s_0, s_1, s_2, \ldots]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots ,$$

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

$$U_h([s_0, s_1, s_2, \ldots]) = \sum_{t=0}^{\infty} \gamma^t R(s_t) \leq \sum_{t=0}^{\infty} \gamma^t R_{\max} = R_{\max}/(1-\gamma) \, , \qquad (17.1)$$

   using the standard formula for the sum of an infinite geometric series.

PROPER POLICY

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.

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 $S_1, S_2, \ldots$, is determined by the initial state s, the policy π, and the transition model for the environment.

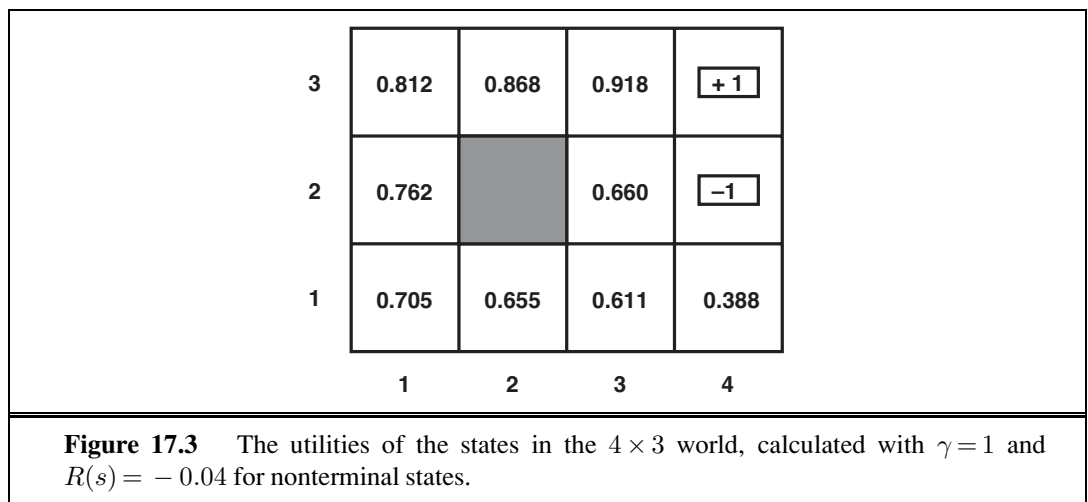The expected utility obtained by executing π starting in s is given by

$$U^\pi(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t)\right] \, , \qquad (17.2)$$

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:

$$\pi_s^* = \underset{\pi}{\operatorname{argmax}} \, U^\pi(s) \, . \qquad (17.3)$$

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.

Figure 17.3   The utilities of the states in the 4×3 world, calculated with $\gamma = 1$ and $R(s) = -0.04$ for nonterminal states.

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:

$$\pi^*(s) = \operatorname*{argmax}_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U(s') \,. \tag{17.4}$$

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

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U(s') \,. \tag{17.5}$$

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

$$
\begin{aligned}
U(1,1) = -0.04 + \gamma \max[\ & 0.8U(1,2) + 0.1U(2,1) + 0.1U(1,1), & (Up) \\
& 0.9U(1,1) + 0.1U(1,2), & (Left) \\
& 0.9U(1,1) + 0.1U(2,1), & (Down) \\
& 0.8U(2,1) + 0.1U(1,2) + 0.1U(1,1)\ ]. & (Right)
\end{aligned}
$$

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 ith iteration. The iteration BELLMAN UPDATE   step, called a **Bellman update**, looks like this:

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U_i(s') \,, \tag{17.6}$$

function VALUE-ITERATION(mdp, ϵ) **returns** a utility function
   inputs: mdp, an MDP with states S, actions $A(s)$, transition model $P(s' \mid s, a)$,
              rewards $R(s)$, discount γ
       ϵ, the maximum error allowed in the utility of any state
   local variables: U, U', vectors of utilities for states in S, initially zero
                  δ, 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**
$$U'[s] \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\ U[s']$$
        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).
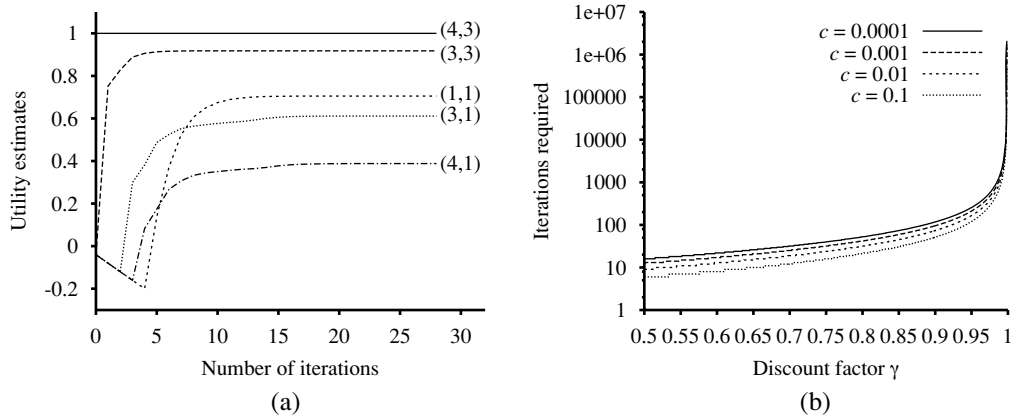

Figure 17.5   (a) Graph showing the evolution of the utilities of selected states using value iteration. (b) The number of value iterations k required to guarantee an error of at most $\epsilon = c \cdot R_{\max}$, for different values of c, as a function of the discount factor γ.

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 ith 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*

$$||B\,U_i - B\,U_i'|| \le \gamma\,||U_i - U_i'|| \;. \tag{17.7}$$

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

$$||BU_i - U|| \le \gamma ||U_i - U|| .$$

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 $||U_0 - U|| \le 2R_{\max}/(1 - \gamma)$. 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 $\gamma^N \cdot 2R_{\max}/(1 - \gamma) \le \epsilon$. 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 $\epsilon/R_{\max}$. The good news is that, because of the exponentially fast convergence, N does not depend much on the ratio $\epsilon/R_{\max}$. 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,

$$\text{if} \quad ||U_{i+1} - U_i|| < \epsilon(1 - \gamma)/\gamma \quad \text{then} \quad ||U_{i+1} - U|| < \epsilon . \tag{17.8}$$

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:

POLICY LOSS

$$\text{if} \quad ||U_i - U|| < \epsilon \quad \text{then} \quad ||U^{\pi_i} - U|| < 2\epsilon\gamma/(1 - \gamma) . \tag{17.9}$$

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

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

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 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 ith iteration, the policy π_i specifies the action $\pi_i(s)$ in
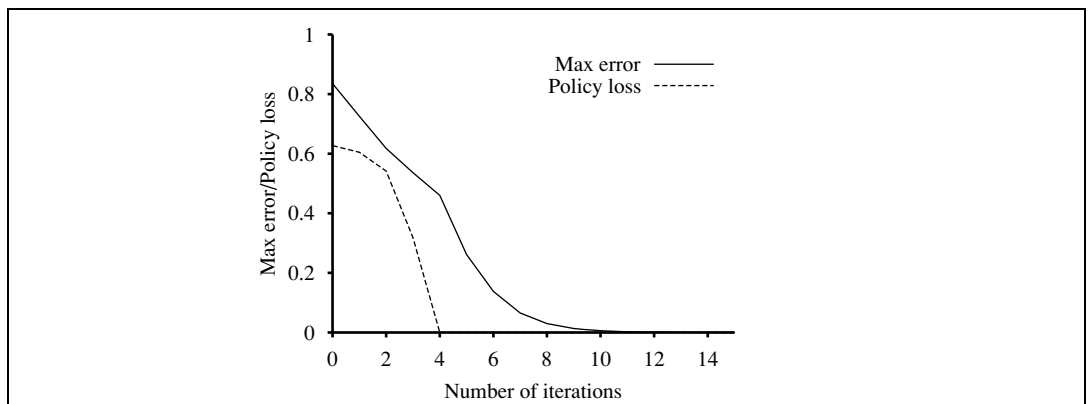

Figure 17.6   The maximum error $||U_i - U||$ of the utility estimates and the policy loss $||U^{\pi_i} - U||$, as a function of the number of iterations of value iteration.

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:

$$U_i(s) = R(s) + \gamma \sum_{s'} P(s' \mid s, \pi_i(s)) U_i(s') . \qquad (17.10)$$

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

$$U_i(1,1) = -0.04 + 0.8 U_i(1,2) + 0.1 U_i(1,1) + 0.1 U_i(2,1) ,$$
$$U_i(1,2) = -0.04 + 0.8 U_i(1,3) + 0.2 U_i(1,2) ,$$
$$\vdots$$

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

$$U_{i+1}(s) \leftarrow R(s) + \gamma \sum_{s'} P(s' \mid s, \pi_i(s)) U_i(s') ,$$

MODIFIED POLICY
ITERATION

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.

---

function POLICY-ITERATION(mdp) **returns** a policy
   inputs: mdp, an MDP with states S, actions $A(s)$, transition model $P(s' \mid s, a)$
   local variables: U, a vector of utilities for states in S, initially zero
                  π, a policy vector indexed by state, initially random

   repeat
      $U \leftarrow$ POLICY-EVALUATION(π, U, mdp)
      $unchanged? \leftarrow$ true
      for each state s **in** S **do**
         if $\displaystyle\max_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\ U[s'] > \sum_{s'} P(s' \mid s, \pi[s])\ U[s']$ **then do**
            $\pi[s] \leftarrow \displaystyle\operatorname*{argmax}_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\ U[s']$
            $unchanged? \leftarrow$ false
   until $unchanged?$
   return π

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.

ASYNCHRONOUS
POLICY ITERATION

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.

PARTIALLY
OBSERVABLE MDP

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

$$b'(s') = \alpha \, P(e \mid s') \sum_s P(s' \mid s, a) b(s) \,,$$

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

$$b' = \text{FORWARD}(b, a, e) \,. \tag{17.11}$$

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:

$$P(e|a, b) = \sum_{s'} P(e|a, s', b) P(s'|a, b)$$

$$= \sum_{s'} P(e \mid s') P(s'|a, b)$$

$$= \sum_{s'} P(e \mid s') \sum_{s} P(s' \mid s, a) b(s) .$$

Let us write the probability of reaching b' from b, given action a, as $P(b' \mid b, a))$. Then that gives us

$$P(b' \mid b, a) = P(b'|a, b) = \sum_{e} P(b'|e, a, b) P(e|a, b)$$

$$= \sum_{e} P(b'|e, a, b) \sum_{s'} P(e \mid s') \sum_{s} P(s' \mid s, a) b(s) , \qquad (17.12)$$

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, $P(b' \mid b, a)$ 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.
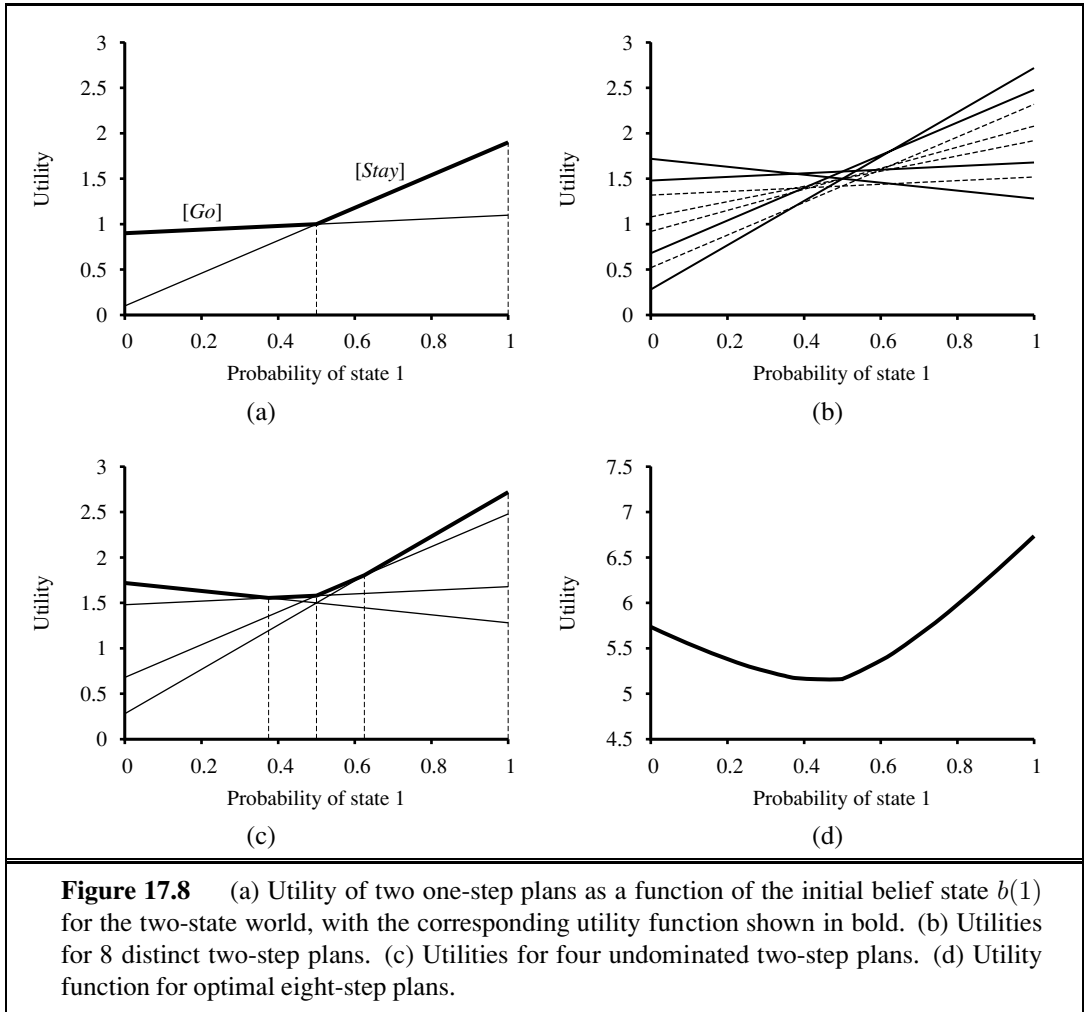
The advantage of a two-state world is that the belief space can be viewed as one-dimensional, because the two probabilities must sum to 1. In Figure 17.8(a), the x-axis represents the belief state, defined by $b(1)$, the probability of being in state 1. Now let us consider the one-step plans $[Stay]$ and $[Go]$, each of which receives the reward for the current state followed by the (discounted) reward for the state reached after the action:

$$\alpha_{[Stay]}(0) = R(0) + \gamma(0.9R(0) + 0.1R(1)) = 0.1$$
$$\alpha_{[Stay]}(1) = R(1) + \gamma(0.9R(1) + 0.1R(0)) = 1.9$$
$$\alpha_{[Go]}(0) = R(0) + \gamma(0.9R(1) + 0.1R(0)) = 0.9$$
$$\alpha_{[Go]}(1) = R(1) + \gamma(0.9R(0) + 0.1R(1)) = 1.1$$

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:

$[Stay; \textbf{if } Percept = 0 \textbf{ then } Stay \textbf{ else } Stay]$
$[Stay; \textbf{if } Percept = 0 \textbf{ then } Stay \textbf{ else } Go]\ldots$

Figure 17.8   (a) Utility of two one-step plans as a function of the initial belief state $b(1)$ for the two-state world, with the corresponding utility function shown in bold. (b) Utilities for 8 distinct two-step plans. (c) Utilities for four undominated two-step plans. (d) Utility function for optimal eight-step plans.

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 DOMINATED PLAN   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

$$\alpha_p(s) = R(s) + \gamma \left(\sum_{s'} P(s' \,|\, s, a) \sum_{e} P(e \,|\, s') \alpha_{p.e}(s') \right) . \tag{17.13}$$

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 $P(s' \mid s, a)$,
                 sensor model $P(e \mid s)$, rewards $R(s)$, discount γ
        ϵ, the maximum error allowed in the utility of any state
   local variables: U, U', sets of plans p with associated utility vectors α_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)
      $U' \leftarrow$ REMOVE-DOMINATED-PLANS(U')
   until MAX-DIFFERENCE(U, U') $< \epsilon(1 - \gamma)/\gamma$
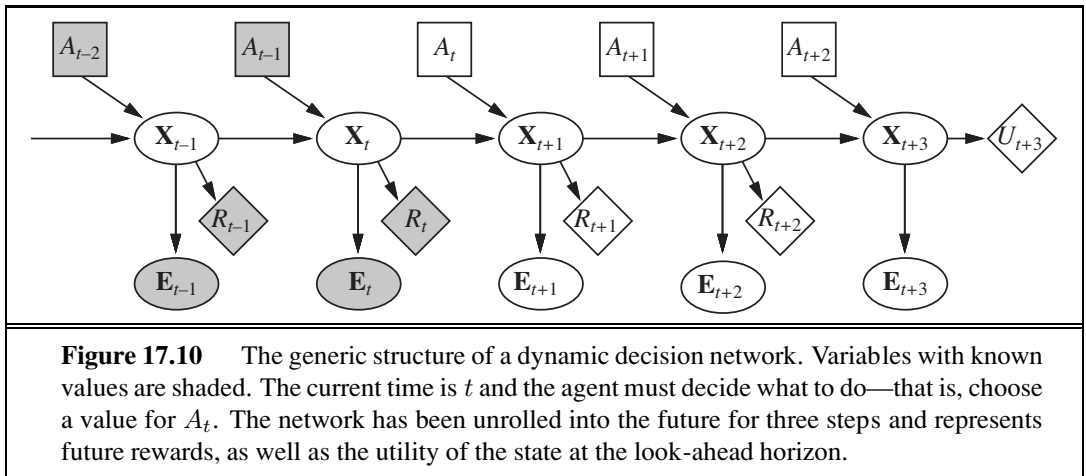   **return** U

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.

Figure 17.10   The generic structure of a dynamic decision network. Variables with known values are shaded. The current time is t and the agent must decide what to do—that is, choose a value for A_t. The network has been unrolled into the future for three steps and represents future rewards, as well as the utility of the state at the look-ahead horizon.

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

Figure 17.11   Part of the look-ahead solution of the DDN in Figure 17.10. Each decision will be taken in the belief state indicated.
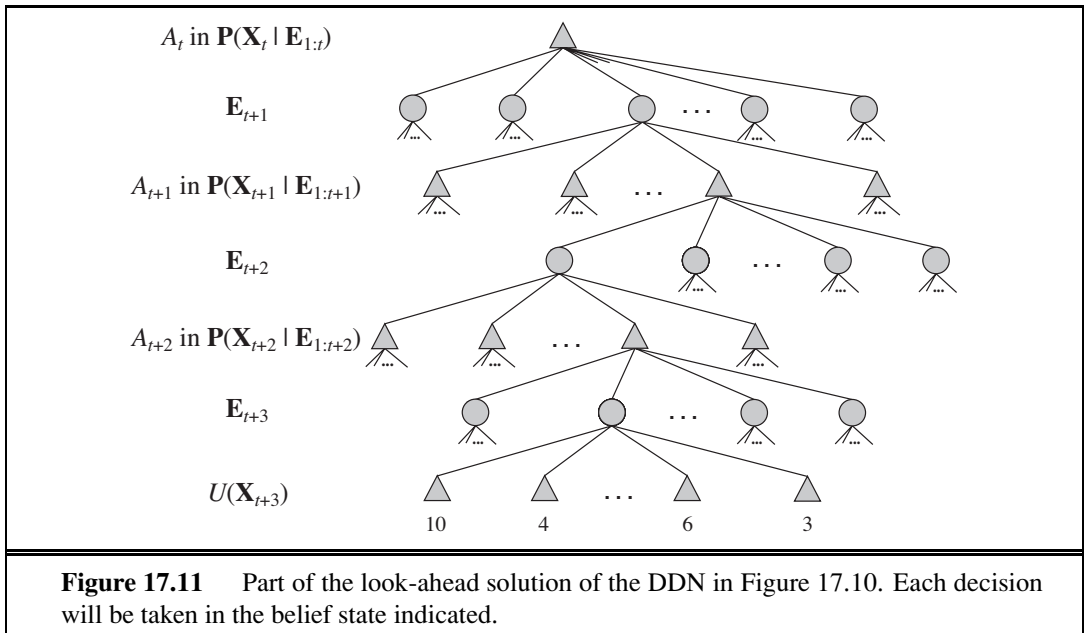
$\mathbf{E}$ and rewards R are all unknown. Notice that the network includes nodes for the *rewards* for $\mathbf{X}_{t+1}$ and $\mathbf{X}_{t+2}$, but the *utility* for $\mathbf{X}_{t+3}$. This is because the agent must maximize the (discounted) sum of all future rewards, and $U(\mathbf{X}_{t+3})$ represents 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 $i = 0, 1, 2, \ldots$. 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 $\mathbf{E}_{t+1}, \ldots, \mathbf{E}_{t+i}$, 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

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 GAME THEORY 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

• **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.

ACTION

• **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.

PAYOFF FUNCTION

STRATEGIC FORM

• 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.

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.

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:

PRISONER'S DILEMMA

	$Alice{:}testify$	$Alice{:}refuse$
$Bob{:}testify$	$A = -5, B = -5$	$A = -10, B = 0$
$Bob{:}refuse$	$A = 0, B = -10$	$A = -1, B = -1$

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."

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.

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 STRONG DOMINATION

WEAK DOMINATION

PARETO OPTIMAL

PARETO DOMINATED

DOMINANT STRATEGY EQUILIBRIUM EQUILIBRIUM

---

[5] Pareto optimality is named after the economist Vilfredo Pareto (1848–1923).

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.

NASH EQUILIBRIUM

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 is 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:

	$Acme$:$bluray$	$Acme$:dvd
$Best$:$bluray$	$A = +9, B = +9$	$A = -4, B = -1$
$Best$:dvd	$A = -3, B = -1$	$A = +5, B = +5$

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

MAXIMIN

Neumann's method is called the the **maximin** technique, and it works as follows:

- 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

$$U_{E,O} \leq U \leq U_{O,E} \qquad \text{or in this case,} \qquad -3 \leq U \leq 2 \ .$$

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, $[p: one; (1-p): two]$, its expected utility is a linear combination $(p \cdot u_{one} + (1-p) \cdot u_{two})$ of
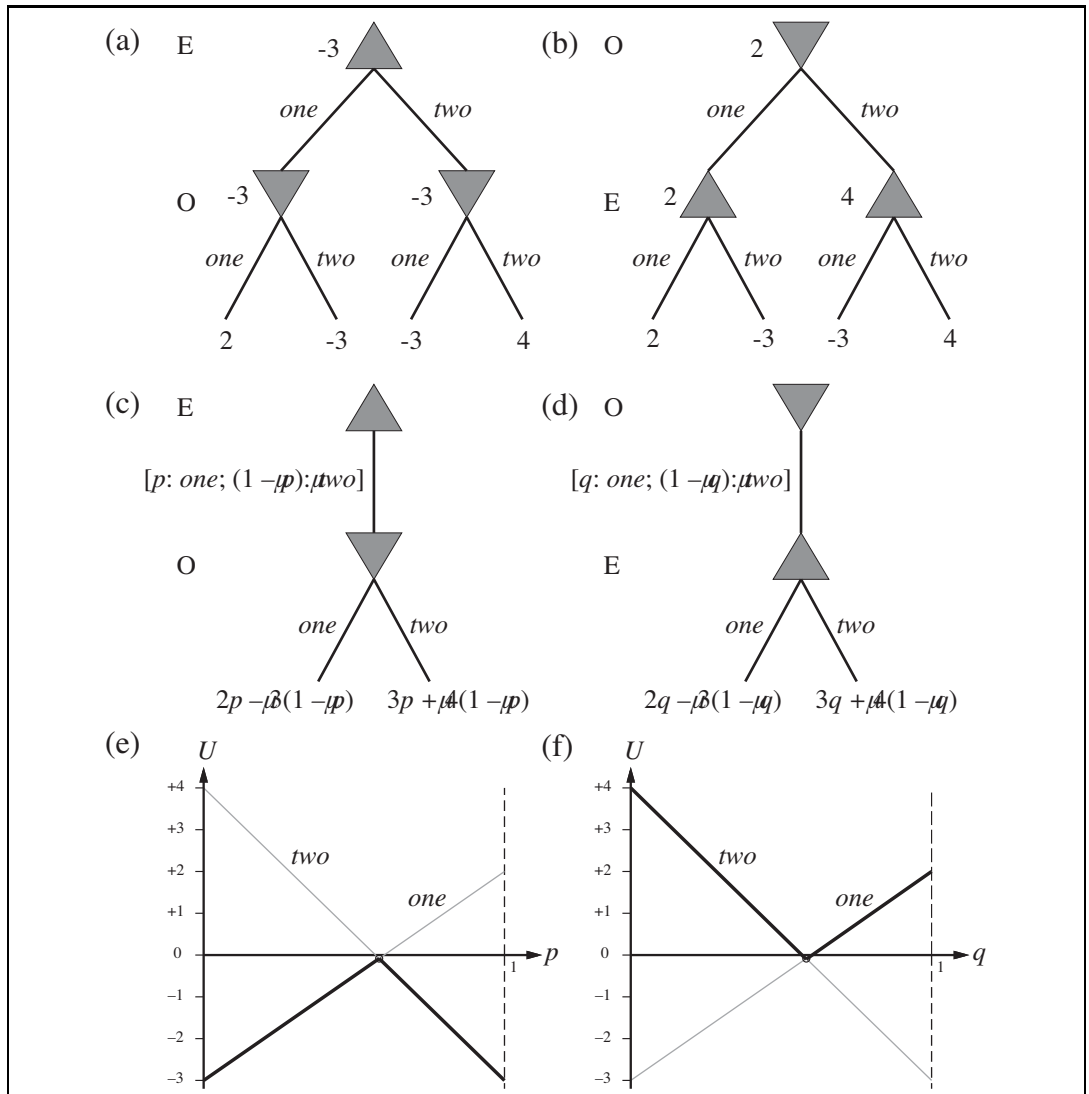
[6] or a constant—see page 162.

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 $[p: one; (1-p): two]$ 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

$$5p - 3 = 4 - 7p \qquad \Rightarrow \qquad p = 7/12 \ .$$

  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 $[q: one; (1-q): two]$ 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:

$$5q - 3 = 4 - 7q \qquad \Rightarrow \qquad q = 7/12 \ .$$

  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 $[7/12: one; 5/12: two]$, 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.

MAXIMIN
EQUILIBRIUM

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 $U_E(one, two) = U_E(two, one) = -3$. 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

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

PERPETUAL
PUNISHMENT

$$\sum_{t=0}^{\infty} 0.99^t \cdot (-1) = -100 \ .$$

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 PLAYER(s) that tells which player has the move, a function ACTIONS(s) enumerating the possible actions, a function RESULT(s, a) that defines the transition to a new state, and a partial function 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,

Figure 17.13   Extensive form of a simplified version of poker.
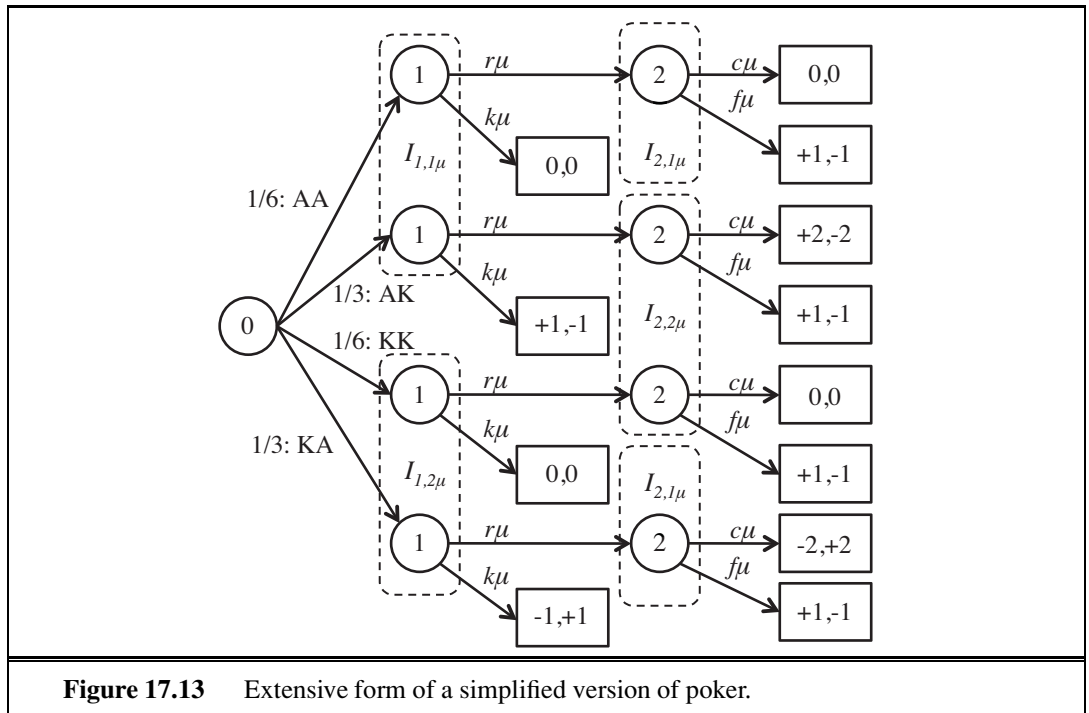
but does not know what card player 1 has. (Due to the limits of two-dimensional paper, this information set is shown as two boxes rather than one.)

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.

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).

ABSTRACTION

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.

COURNOT
COMPETITION

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.

BAYES–NASH
EQUILIBRIUM

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.

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.

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.

EFFICIENT

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.

COLLUSION

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.

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-

STRATEGY-PROOF

TRUTH-REVEALING

REVELATION
PRINCIPLE

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.

SEALED-BID AUCTION

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

SEALED-BID SECOND-PRICE AUCTION VICKREY AUCTION

$$u_i = \begin{cases} (v_i - b_o) & \text{if } b_i > b_o \\ 0 & \text{otherwise.} \end{cases}$$

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.

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.

REVENUE
EQUIVALENCE
THEOREM

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

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.

TRAGEDY OF THE COMMONS

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.

EXTERNALITIES

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:

VICKREY-CLARKE-GROVES

VCG

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:

$$v_i(A) - (W_{-i} - B_{-i}) . \tag{17.14}$$

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

FACTORED MDP

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**

RELATIONAL MDP

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

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 Reinhart 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 $R(s)$.
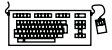
17.5 For the environment shown in Figure 17.1, find all the threshold values for $R(s)$ such that the optimal policy changes when the threshold is crossed. You will need a way to calculate the optimal policy and its value for fixed $R(s)$. (*Hint*: Prove that the value of any fixed policy varies linearly 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 $|(B\,U_i - B\,U_i')(s)|$ and then apply the result from (a) to complete the proof that the Bellman operator is a contraction.
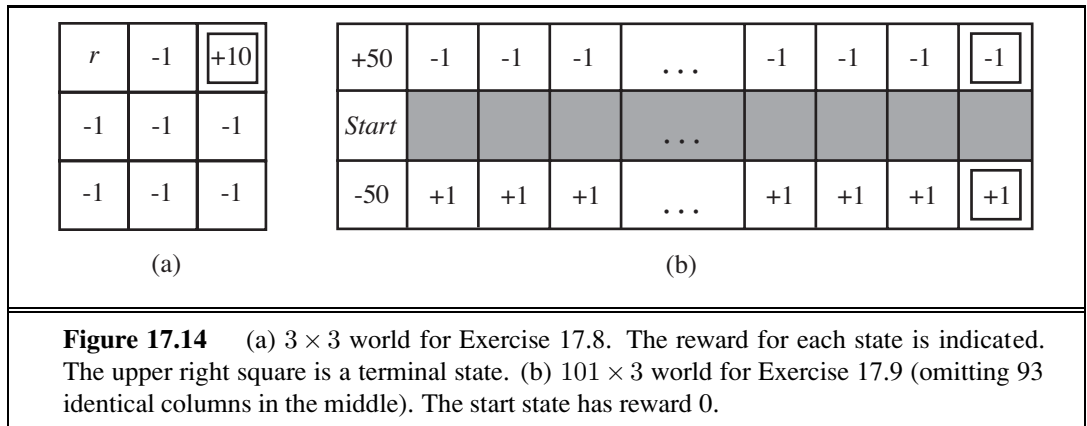
17.7 This exercise considers two-player MDPs that correspond to zero-sum, turn-taking games like those in Chapter 5. Let the players be A and B, and let $R(s)$ be the reward for player A in state s. (The reward for B is always equal and opposite.)

    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. $r = +3$

Figure 17.14   (a) 3×3 world for Exercise 17.8. The reward for each state is indicated. The upper right square is a terminal state. (b) 101×3 world for Exercise 17.9 (omitting 93 identical columns in the middle). The start state has reward 0.

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?

18 LEARNING FROM EXAMPLES

In which we describe agents that can improve their behavior through diligent study of their own experiences.

An agent is **learning** if it improves its performance on future tasks after making observations about the world. Learning can range from the trivial, as exhibited by jotting down a phone number, to the profound, as exhibited by Albert Einstein, who inferred a new theory of the universe. In this chapter we will concentrate on one class of learning problem, which seems restricted but actually has vast applicability: from a collection of input–output pairs, learn a function that predicts the output for new inputs.

Why would we want an agent to learn? If the design of the agent can be improved, why wouldn't the designers just program in that improvement to begin with? There are three main reasons. First, the designers cannot anticipate all possible situations that the agent might find itself in. For example, a robot designed to navigate mazes must learn the layout of each new maze it encounters. Second, the designers cannot anticipate all changes over time; a program designed to predict tomorrow's stock market prices must learn to adapt when conditions change from boom to bust. Third, sometimes human programmers have no idea how to program a solution themselves. For example, most people are good at recognizing the faces of family members, but even the best programmers are unable to program a computer to accomplish that task, except by using learning algorithms. This chapter first gives an overview of the various forms of learning, then describes one popular approach, decision-tree learning, in Section 18.3, followed by a theoretical analysis of learning in Sections 18.4 and 18.5. We look at various learning systems used in practice: linear models, nonlinear models (in particular, neural networks), nonparametric models, and support vector machines. Finally we show how ensembles of models can outperform a single model.

18.1 FORMS OF LEARNING

Any component of an agent can be improved by learning from data. The improvements, and the techniques used to make them, depend on four major factors:

- Which *component* is to be improved.

- What *prior knowledge* the agent already has.
- What *representation* is used for the data and the component.
- What *feedback* is available to learn from.

Components to be learned

Chapter 2 described several agent designs. The components of these agents include:

1. A direct mapping from conditions on the current state to actions.
2. A means to infer relevant properties of the world from the percept sequence.
3. Information about the way the world evolves and about the results of possible actions the agent can take.
4. Utility information indicating the desirability of world states.
5. Action-value information indicating the desirability of actions.
6. Goals that describe classes of states whose achievement maximizes the agent's utility.

Each of these components can be learned. Consider, for example, an agent training to become a taxi driver. Every time the instructor shouts "Brake!" the agent might learn a condition–action rule for when to brake (component 1); the agent also learns every time the instructor does not shout. By seeing many camera images that it is told contain buses, it can learn to recognize them (2). By trying actions and observing the results—for example, braking hard on a wet road—it can learn the effects of its actions (3). Then, when it receives no tip from passengers who have been thoroughly shaken up during the trip, it can learn a useful component of its overall utility function (4).

Representation and prior knowledge

We have seen several examples of representations for agent components: propositional and first-order logical sentences for the components in a logical agent; Bayesian networks for the inferential components of a decision-theoretic agent, and so on. Effective learning algorithms have been devised for all of these representations. This chapter (and most of current machine learning research) covers inputs that form a **factored representation**—a vector of attribute values—and outputs that can be either a continuous numerical value or a discrete value. Chapter 19 covers functions and prior knowledge composed of first-order logic sentences, and Chapter 20 concentrates on Bayesian networks.

There is another way to look at the various types of learning. We say that learning a (possibly incorrect) general function or rule from specific input–output pairs is called **inductive learning**. We will see in Chapter 19 that we can also do **analytical** or **deductive learning**: going from a known general rule to a new rule that is logically entailed, but is useful because it allows more efficient processing.

INDUCTIVE
LEARNING
DEDUCTIVE
LEARNING

Feedback to learn from

There are three *types of feedback* that determine the three main types of learning:

In **unsupervised learning** the agent learns patterns in the input even though no explicit feedback is supplied. The most common unsupervised learning task is **clustering**: detecting

UNSUPERVISED
LEARNING
CLUSTERING

potentially useful clusters of input examples. For example, a taxi agent might gradually develop a concept of "good traffic days" and "bad traffic days" without ever being given labeled examples of each by a teacher.

REINFORCEMENT
LEARNING

In **reinforcement learning** the agent learns from a series of reinforcements—rewards or punishments. For example, the lack of a tip at the end of the journey gives the taxi agent an indication that it did something wrong. The two points for a win at the end of a chess game tells the agent it did something right. It is up to the agent to decide which of the actions prior to the reinforcement were most responsible for it.

SUPERVISED
LEARNING

In **supervised learning** the agent observes some example input–output pairs and learns a function that maps from input to output. In component 1 above, the inputs are percepts and the output are provided by a teacher who says "Brake!" or "Turn left." In component 2, the inputs are camera images and the outputs again come from a teacher who says "that's a bus." In 3, the theory of braking is a function from states and braking actions to stopping distance in feet. In this case the output value is available directly from the agent's percepts (after the fact); the environment is the teacher.

SEMI-SUPERVISED
LEARNING

In practice, these distinction are not always so crisp. In **semi-supervised learning** we are given a few labeled examples and must make what we can of a large collection of un-labeled examples. Even the labels themselves may not be the oracular truths that we hope for. Imagine that you are trying to build a system to guess a person's age from a photo. You gather some labeled examples by snapping pictures of people and asking their age. That's supervised learning. But in reality some of the people lied about their age. It's not just that there is random noise in the data; rather the inaccuracies are systematic, and to uncover them is an unsupervised learning problem involving images, self-reported ages, and true (un-known) ages. Thus, both noise and lack of labels create a continuum between supervised and unsupervised learning.

18.2 SUPERVISED LEARNING

The task of supervised learning is this:

TRAINING SET

Given a **training set** of N example input–output pairs
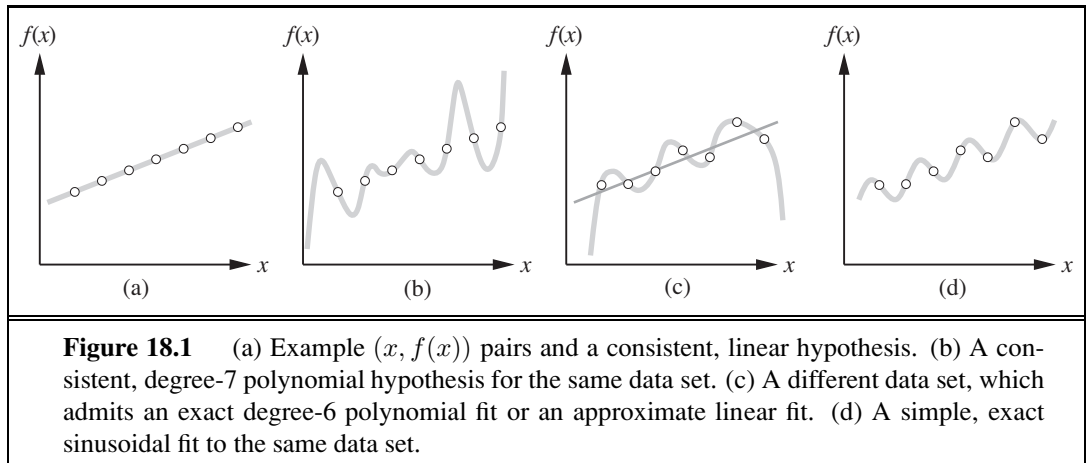
$$(x_1, y_1), (x_2, y_2), \ldots (x_N, y_N) \,,$$

where each y_j was generated by an unknown function $y = f(x)$,
discover a function h that approximates the true function f.

HYPOTHESIS

Here x and y can be any value; they need not be numbers. The function h is a **hypothesis**.[1] Learning is a search through the space of possible hypotheses for one that will perform well, even on new examples beyond the training set. To measure the accuracy of a hypothesis we

TEST SET

give it a **test set** of examples that are distinct from the training set. We say a hypothesis

---

[1] A note on notation: except where noted, we will use j to index the N examples; x_j will always be the input and y_j the output. In cases where the input is specifically a vector of attribute values (beginning with Section 18.3), we will use $\mathbf{x}_j$ for the jth example and we will use i to index the n attributes of each example. The elements of $\mathbf{x}_j$ are written $x_{j,1}, x_{j,2}, \ldots, x_{j,n}$.

Figure 18.1   (a) Example $(x, f(x))$ pairs and a consistent, linear hypothesis. (b) A consistent, degree-7 polynomial hypothesis for the same data set. (c) A different data set, which admits an exact degree-6 polynomial fit or an approximate linear fit. (d) A simple, exact sinusoidal fit to the same data set.

GENERALIZATION

generalizes well if it correctly predicts the value of y for novel examples. Sometimes the function f is stochastic—it is not strictly a function of x, and what we have to learn is a conditional probability distribution, $\mathbf{P}(Y \mid x)$.

When the output y is one of a finite set of values (such as *sunny, cloudy* or *rainy*), the learning problem is called **classification**, and is called Boolean or binary classification if there are only two values. When y is a number (such as tomorrow's temperature), the learning problem is called **regression**. (Technically, solving a regression problem is finding a conditional expectation or average value of y, because the probability that we have found *exactly* the right real-valued number for y is 0.)

CLASSIFICATION

REGRESSION

Figure 18.1 shows a familiar example: fitting a function of a single variable to some data points. The examples are points in the (x, y) plane, where $y = f(x)$. We don't know what f is, but we will approximate it with a function h selected from a **hypothesis space**, $\mathcal{H}$, which for this example we will take to be the set of polynomials, such as x^5+3x^2+2. Figure 18.1(a) shows some data with an exact fit by a straight line (the polynomial $0.4x + 3$). The line is called a **consistent** hypothesis because it agrees with all the data. Figure 18.1(b) shows a high-degree polynomial that is also consistent with the same data. This illustrates a fundamental problem in inductive learning: *how do we choose from among multiple consistent hypotheses?* One answer is to prefer the *simplest* hypothesis consistent with the data. This principle is called **Ockham's razor**, after the 14th-century English philosopher William of Ockham, who used it to argue sharply against all sorts of complications. Defining simplicity is not easy, but it seems clear that a degree-1 polynomial is simpler than a degree-7 polynomial, and thus (a) should be preferred to (b). We will make this intuition more precise in Section 18.4.3.

HYPOTHESIS SPACE

CONSISTENT

☞

OCKHAM'S RAZOR

Figure 18.1(c) shows a second data set. There is no consistent straight line for this data set; in fact, it requires a degree-6 polynomial for an exact fit. There are just 7 data points, so a polynomial with 7 parameters does not seem to be finding any pattern in the data and we do not expect it to generalize well. A straight line that is not consistent with any of the data points, but might generalize fairly well for unseen values of x, is also shown in (c). *In general, there is a tradeoff between complex hypotheses that fit the training data well and simpler hypotheses that may generalize better.* In Figure 18.1(d) we expand the

☞

REALIZABLE

hypothesis space $\mathcal{H}$ to allow polynomials over both x and $\sin(x)$, and find that the data in (c) can be fitted exactly by a simple function of the form $ax + b + c\sin(x)$. This shows the importance of the choice of hypothesis space. We say that a learning problem is **realizable** if the hypothesis space contains the true function. Unfortunately, we cannot always tell whether a given learning problem is realizable, because the true function is not known.

In some cases, an analyst looking at a problem is willing to make more fine-grained distinctions about the hypothesis space, to say—even before seeing any data—not just that a hypothesis is possible or impossible, but rather how probable it is. Supervised learning can be done by choosing the hypothesis h^* that is most probable given the data:

$$h^* = \underset{h \in \mathcal{H}}{\operatorname{argmax}} \, P(h|data) \ .$$

By Bayes' rule this is equivalent to

$$h^* = \underset{h \in \mathcal{H}}{\operatorname{argmax}} \, P(data|h) \, P(h) \ .$$

Then we can say that the prior probability $P(h)$ is high for a degree-1 or -2 polynomial, lower for a degree-7 polynomial, and especially low for degree-7 polynomials with large, sharp spikes as in Figure 18.1(b). We allow unusual-looking functions when the data say we really need them, but we discourage them by giving them a low prior probability.

Why not let $\mathcal{H}$ be the class of all Java programs, or Turing machines? After all, every computable function can be represented by some Turing machine, and that is the best we can do. One problem with this idea is that it does not take into account the computational complexity of learning. *There is a tradeoff between the expressiveness of a hypothesis space and the complexity of finding a good hypothesis within that space.* For example, fitting a straight line to data is an easy computation; fitting high-degree polynomials is somewhat harder; and fitting Turing machines is in general undecidable. A second reason to prefer simple hypothesis spaces is that presumably we will want to use h after we have learned it, and computing $h(x)$ when h is a linear function is guaranteed to be fast, while computing an arbitrary Turing machine program is not even guaranteed to terminate. For these reasons, most work on learning has focused on simple representations.

We will see that the expressiveness–complexity tradeoff is not as simple as it first seems: it is often the case, as we saw with first-order logic in Chapter 8, that an expressive language makes it possible for a *simple* hypothesis to fit the data, whereas restricting the expressiveness of the language means that any consistent hypothesis must be very complex. For example, the rules of chess can be written in a page or two of first-order logic, but require thousands of pages when written in propositional logic.

18.3   LEARNING DECISION TREES

Decision tree induction is one of the simplest and yet most successful forms of machine learning. We first describe the representation—the hypothesis space—and then show how to learn a good hypothesis.

18.3.1   The decision tree representation

DECISION TREE

A **decision tree** represents a function that takes as input a vector of attribute values and returns a "decision"—a single output value. The input and output values can be discrete or continuous. For now we will concentrate on problems where the inputs have discrete values and the output has exactly two possible values; this is Boolean classification, where each

POSITIVE

example input will be classified as true (a **positive** example) or false (a **negative** example).

NEGATIVE

A decision tree reaches its decision by performing a sequence of tests. Each internal node in the tree corresponds to a test of the value of one of the input attributes, A_i, and the branches from the node are labeled with the possible values of the attribute, $A_i = v_{ik}$. Each leaf node in the tree specifies a value to be returned by the function. The decision tree representation is natural for humans; indeed, many "How To" manuals (e.g., for car repair) are written entirely as a single decision tree stretching over hundreds of pages.

GOAL PREDICATE

As an example, we will build a decision tree to decide whether to wait for a table at a restaurant. The aim here is to learn a definition for the **goal predicate** $WillWait$. First we list the attributes that we will consider as part of the input:

1. $Alternate$: whether there is a suitable alternative restaurant nearby.
2. Bar: whether the restaurant has a comfortable bar area to wait in.
3. Fri/Sat: true on Fridays and Saturdays.
4. $Hungry$: whether we are hungry.
5. $Patrons$: how many people are in the restaurant (values are $None$, $Some$, and $Full$).
6. $Price$: the restaurant's price range ($, $$, $$$).
7. $Raining$: whether it is raining outside.
8. $Reservation$: whether we made a reservation.
9. $Type$: the kind of restaurant (French, Italian, Thai, or burger).
10. $WaitEstimate$: the wait estimated by the host (0–10 minutes, 10–30, 30–60, or >60).

Note that every variable has a small set of possible values; the value of $WaitEstimate$, for example, is not an integer, rather it is one of the four discrete values 0–10, 10–30, 30–60, or >60. The decision tree usually used by one of us (SR) for this domain is shown in Figure 18.2. Notice that the tree ignores the $Price$ and $Type$ attributes. Examples are processed by the tree starting at the root and following the appropriate branch until a leaf is reached. For instance, an example with $Patrons = Full$ and $WaitEstimate = 0$–10 will be classified as positive (i.e., yes, we will wait for a table).

18.3.2   Expressiveness of decision trees

A Boolean decision tree is logically equivalent to the assertion that the goal attribute is true if and only if the input attributes satisfy one of the paths leading to a leaf with value $true$. Writing this out in propositional logic, we have

$$Goal \iff (Path_1 \lor Path_2 \lor \cdots) ,$$

where each $Path$ is a conjunction of attribute-value tests required to follow that path. Thus, the whole expression is equivalent to disjunctive normal form (see page 283), which means

that any function in propositional logic can be expressed as a decision tree. As an example, the rightmost path in Figure 18.2 is

$$Path = (Patrons = Full \wedge WaitEstimate = 0\text{–}10) \, .$$

For a wide variety of problems, the decision tree format yields a nice, concise result. But some functions cannot be represented concisely. For example, the majority function, which returns true if and only if more than half of the inputs are true, requires an exponentially large decision tree. In other words, decision trees are good for some kinds of functions and bad for others. Is there *any* kind of representation that is efficient for *all* kinds of functions? Unfortunately, the answer is no. We can show this in a general way. Consider the set of all Boolean functions on n attributes. How many different functions are in this set? This is just the number of different truth tables that we can write down, because the function is defined by its truth table. A truth table over n attributes has 2^n rows, one for each combination of values of the attributes. We can consider the "answer" column of the table as a 2^n-bit number that defines the function. That means there are 2^{2^n} different functions (and there will be more than that number of trees, since more than one tree can compute the same function). This is a scary number. For example, with just the ten Boolean attributes of our restaurant problem there are 2^{1024} or about 10^{308} different functions to choose from, and for 20 attributes there are over $10^{300,000}$. We will need some ingenious algorithms to find good hypotheses in such a large space.

18.3.3   Inducing decision trees from examples

An example for a Boolean decision tree consists of an $(\mathbf{x}, y)$ pair, where $\mathbf{x}$ is a vector of values for the input attributes, and y is a single Boolean output value. A training set of 12 examples

Figure 18.2   A decision tree for deciding whether to wait for a table.
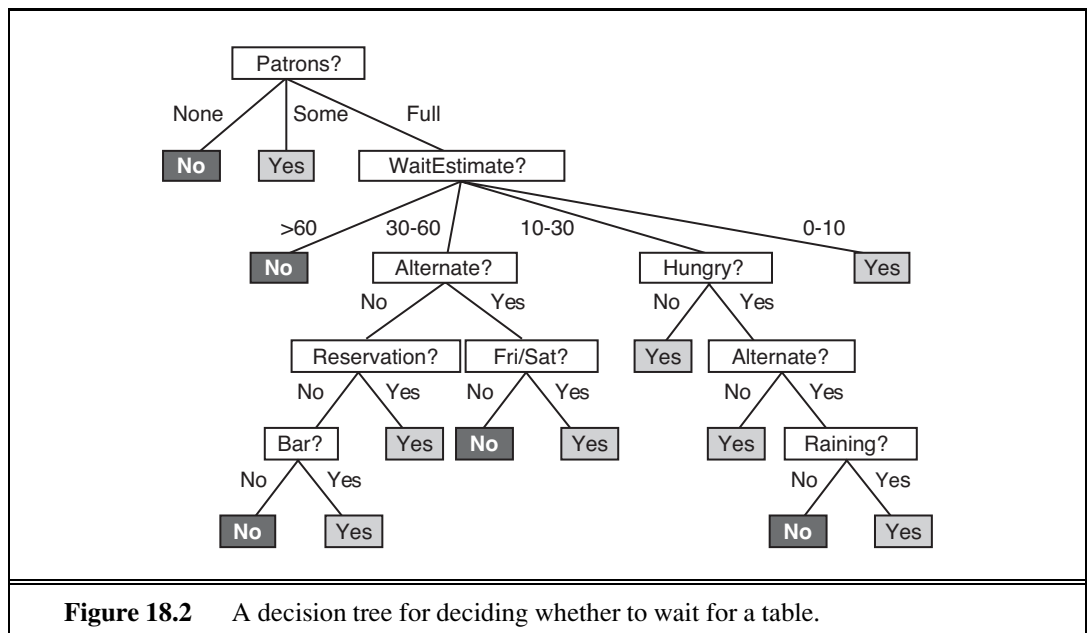
| Example | Input Attributes | | | | | | | | | | Goal |
	Alt	Bar	Fri	Hun	Pat	$Price$	$Rain$	Res	$Type$	Est	$WillWait$
$\mathbf{x}_1$	Yes	No	No	Yes	Some	\$\$\$	No	Yes	French	0–10	$y_1 = Yes$
$\mathbf{x}_2$	Yes	No	No	Yes	Full	\$	No	No	Thai	30–60	$y_2 = No$
$\mathbf{x}_3$	No	Yes	No	No	Some	\$	No	No	Burger	0–10	$y_3 = Yes$
$\mathbf{x}_4$	Yes	No	Yes	Yes	Full	\$	Yes	No	Thai	10–30	$y_4 = Yes$
$\mathbf{x}_5$	Yes	No	Yes	No	Full	\$\$\$	No	Yes	French	>60	$y_5 = No$
$\mathbf{x}_6$	No	Yes	No	Yes	Some	\$\$	Yes	Yes	Italian	0–10	$y_6 = Yes$
$\mathbf{x}_7$	No	Yes	No	No	None	\$	Yes	No	Burger	0–10	$y_7 = No$
$\mathbf{x}_8$	No	No	No	Yes	Some	\$\$	Yes	Yes	Thai	0–10	$y_8 = Yes$
$\mathbf{x}_9$	No	Yes	Yes	No	Full	\$	Yes	No	Burger	>60	$y_9 = No$
$\mathbf{x}_{10}$	Yes	Yes	Yes	Yes	Full	\$\$\$	No	Yes	Italian	10–30	$y_{10} = No$
$\mathbf{x}_{11}$	No	No	No	No	None	\$	No	No	Thai	0–10	$y_{11} = No$
$\mathbf{x}_{12}$	Yes	Yes	Yes	Yes	Full	\$	No	No	Burger	30–60	$y_{12} = Yes$

Figure 18.3   Examples for the restaurant domain.

is shown in Figure 18.3. The positive examples are the ones in which the goal $WillWait$ is true ($\mathbf{x}_1, \mathbf{x}_3, \ldots$); the negative examples are the ones in which it is false ($\mathbf{x}_2, \mathbf{x}_5, \ldots$).

We want a tree that is consistent with the examples and is as small as possible. Unfortunately, no matter how we measure size, it is an intractable problem to find the smallest consistent tree; there is no way to efficiently search through the 2^{2^n} trees. With some simple heuristics, however, we can find a good approximate solution: a small (but not smallest) consistent tree. The DECISION-TREE-LEARNING algorithm adopts a greedy divide-and-conquer strategy: always test the most important attribute first. This test divides the problem up into smaller subproblems that can then be solved recursively. By "most important attribute," we mean the one that makes the most difference to the classification of an example. That way, we hope to get to the correct classification with a small number of tests, meaning that all paths in the tree will be short and the tree as a whole will be shallow.

Figure 18.4(a) shows that $Type$ is a poor attribute, because it leaves us with four possible outcomes, each of which has the same number of positive as negative examples. On the other hand, in (b) we see that $Patrons$ is a fairly important attribute, because if the value is $None$ or $Some$, then we are left with example sets for which we can answer definitively (No and Yes, respectively). If the value is $Full$, we are left with a mixed set of examples. In general, after the first attribute test splits up the examples, each outcome is a new decision tree learning problem in itself, with fewer examples and one less attribute. There are four cases to consider for these recursive problems:

1. If the remaining examples are all positive (or all negative), then we are done: we can answer Yes or No. Figure 18.4(b) shows examples of this happening in the $None$ and $Some$ branches.

2. If there are some positive and some negative examples, then choose the best attribute to split them. Figure 18.4(b) shows $Hungry$ being used to split the remaining examples.

3. If there are no examples left, it means that no example has been observed for this com-
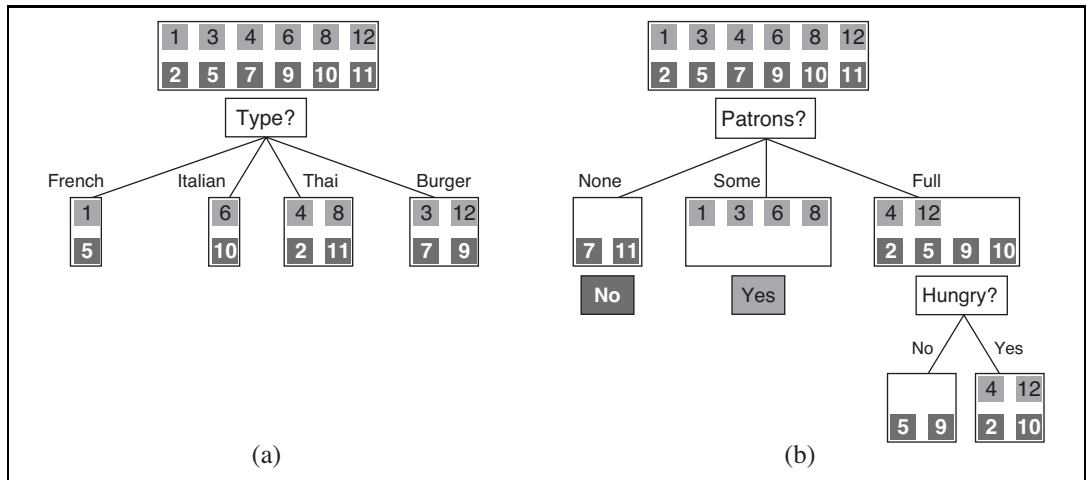
Figure 18.4   Splitting the examples by testing on attributes. At each node we show the positive (light boxes) and negative (dark boxes) examples remaining. (a) Splitting on *Type* brings us no nearer to distinguishing between positive and negative examples. (b) Splitting on *Patrons* does a good job of separating positive and negative examples. After splitting on *Patrons*, *Hungry* is a fairly good second test.

bination of attribute values, and we return a default value calculated from the plurality classification of all the examples that were used in constructing the node's parent. These are passed along in the variable *parent_examples*.

4. If there are no attributes left, but both positive and negative examples, it means that these examples have exactly the same description, but different classifications. This can NOISE happen because there is an error or **noise** in the data; because the domain is nondeterministic; or because we can't observe an attribute that would distinguish the examples. The best we can do is return the plurality classification of the remaining examples.

The DECISION-TREE-LEARNING algorithm is shown in Figure 18.5. Note that the set of examples is crucial for *constructing* the tree, but nowhere do the examples appear in the tree itself. A tree consists of just tests on attributes in the interior nodes, values of attributes on the branches, and output values on the leaf nodes. The details of the IMPORTANCE function are given in Section 18.3.4. The output of the learning algorithm on our sample training set is shown in Figure 18.6. The tree is clearly different from the original tree shown in Figure 18.2. One might conclude that the learning algorithm is not doing a very good job of learning the correct function. This would be the wrong conclusion to draw, however. The learning algorithm looks at the *examples*, not at the correct function, and in fact, its hypothesis (see Figure 18.6) not only is consistent with all the examples, but is considerably simpler than the original tree! The learning algorithm has no reason to include tests for *Raining* and *Reservation*, because it can classify all the examples without them. It has also detected an interesting and previously unsuspected pattern: the first author will wait for Thai food on weekends. It is also bound to make some mistakes for cases where it has seen no examples. For example, it has never seen a case where the wait is 0–10 minutes but the restaurant is full.

function DECISION-TREE-LEARNING(*examples*, *attributes*, *parent_examples*) **returns** a tree

   if *examples* is empty **then return** PLURALITY-VALUE(*parent_examples*)
   else if all *examples* have the same classification **then return** the classification
   else if *attributes* is empty **then return** PLURALITY-VALUE(*examples*)
   else
      $A \leftarrow \text{argmax}_{a \in attributes}$ IMPORTANCE(*a*, *examples*)
      tree ← a new decision tree with root test A
      for each value v_k of A **do**
         $exs \leftarrow \{e \,:\, e \in examples \text{ and } e.A = v_k\}$
         subtree ← DECISION-TREE-LEARNING(*exs*, *attributes* − *A*, *examples*)
         add a branch to *tree* with label $(A = v_k)$ and subtree *subtree*
      return *tree*

Figure 18.5   The decision-tree learning algorithm. The function IMPORTANCE is described in Section 18.3.4. The function PLURALITY-VALUE selects the most common output value among a set of examples, breaking ties randomly.
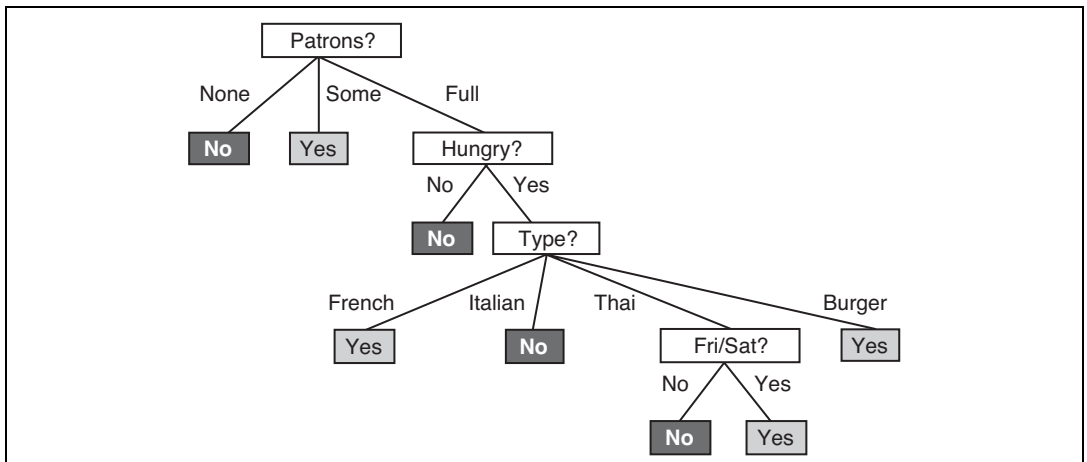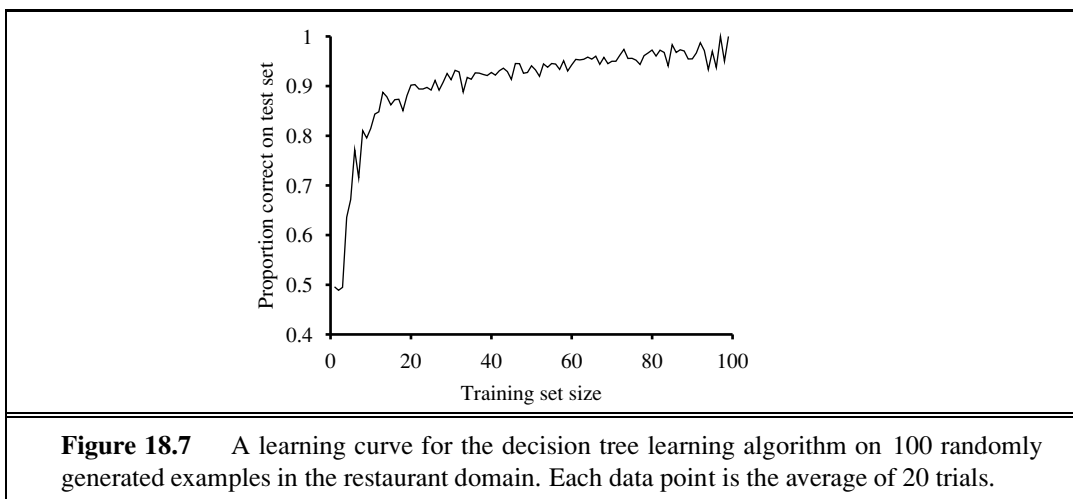

Figure 18.6   The decision tree induced from the 12-example training set.

In that case it says not to wait when *Hungry* is false, but I (SR) would certainly wait. With more training examples the learning program could correct this mistake.

    We note there is a danger of over-interpreting the tree that the algorithm selects. When there are several variables of similar importance, the choice between them is somewhat arbitrary: with slightly different input examples, a different variable would be chosen to split on first, and the whole tree would look completely different. The function computed by the tree would still be similar, but the structure of the tree can vary widely.

LEARNING CURVE   We can evaluate the accuracy of a learning algorithm with a **learning curve**, as shown in Figure 18.7. We have 100 examples at our disposal, which we split into a training set and

Figure 18.7   A learning curve for the decision tree learning algorithm on 100 randomly generated examples in the restaurant domain. Each data point is the average of 20 trials.

a test set. We learn a hypothesis h with the training set and measure its accuracy with the test set. We do this starting with a training set of size 1 and increasing one at a time up to size 99. For each size we actually repeat the process of randomly splitting 20 times, and average the results of the 20 trials. The curve shows that as the training set size grows, the accuracy increases. (For this reason, learning curves are also called **happy graphs**.) In this graph we reach 95% accuracy, and it looks like the curve might continue to increase with more data.

18.3.4   Choosing attribute tests

The greedy search used in decision tree learning is designed to approximately minimize the depth of the final tree. The idea is to pick the attribute that goes as far as possible toward providing an exact classification of the examples. A perfect attribute divides the examples into sets, each of which are all positive or all negative and thus will be leaves of the tree. The *Patrons* attribute is not perfect, but it is fairly good. A really useless attribute, such as *Type*, leaves the example sets with roughly the same proportion of positive and negative examples as the original set.

ENTROPY

All we need, then, is a formal measure of "fairly good" and "really useless" and we can implement the IMPORTANCE function of Figure 18.5. We will use the notion of information gain, which is defined in terms of **entropy**, the fundamental quantity in information theory (Shannon and Weaver, 1949).

Entropy is a measure of the uncertainty of a random variable; acquisition of information corresponds to a reduction in entropy. A random variable with only one value—a coin that always comes up heads—has no uncertainty and thus its entropy is defined as zero; thus, we gain no information by observing its value. A flip of a fair coin is equally likely to come up heads or tails, 0 or 1, and we will soon show that this counts as "1 bit" of entropy. The roll of a fair *four*-sided die has 2 bits of entropy, because it takes two bits to describe one of four equally probable choices. Now consider an unfair coin that comes up heads 99% of the time. Intuitively, this coin has less uncertainty than the fair coin—if we guess heads we'll be wrong only 1% of the time—so we would like it to have an entropy measure that is close to zero, but

positive. In general, the entropy of a random variable V with values v_k, each with probability $P(v_k)$, is defined as

$$\text{Entropy:} \quad H(V) = \sum_k P(v_k) \log_2 \frac{1}{P(v_k)} = -\sum_k P(v_k) \log_2 P(v_k) .$$

We can check that the entropy of a fair coin flip is indeed 1 bit:

$$H(Fair) = -(0.5 \log_2 0.5 + 0.5 \log_2 0.5) = 1 .$$

If the coin is loaded to give 99% heads, we get

$$H(Loaded) = -(0.99 \log_2 0.99 + 0.01 \log_2 0.01) \approx 0.08 \text{ bits.}$$

It will help to define $B(q)$ as the entropy of a Boolean random variable that is true with probability q:

$$B(q) = -(q \log_2 q + (1-q) \log_2 (1-q)) .$$

Thus, $H(Loaded) = B(0.99) \approx 0.08$. Now let's get back to decision tree learning. If a training set contains p positive examples and n negative examples, then the entropy of the goal attribute on the whole set is

$$H(Goal) = B\left(\frac{p}{p+n}\right) .$$

The restaurant training set in Figure 18.3 has $p = n = 6$, so the corresponding entropy is $B(0.5)$ or exactly 1 bit. A test on a single attribute A might give us only part of this 1 bit. We can measure exactly how much by looking at the entropy remaining *after* the attribute test.

An attribute A with d distinct values divides the training set E into subsets $E_1, \ldots, E_d$. Each subset E_k has p_k positive examples and n_k negative examples, so if we go along that branch, we will need an additional $B(p_k/(p_k + n_k))$ bits of information to answer the question. A randomly chosen example from the training set has the kth value for the attribute with probability $(p_k + n_k)/(p + n)$, so the expected entropy remaining after testing attribute A is

$$Remainder(A) = \sum_{k=1}^{d} \frac{p_k + n_k}{p+n} B(\frac{p_k}{p_k+n_k}) .$$

INFORMATION GAIN
The **information gain** from the attribute test on A is the expected reduction in entropy:

$$Gain(A) = B(\frac{p}{p+n}) - Remainder(A) .$$

In fact $Gain(A)$ is just what we need to implement the IMPORTANCE function. Returning to the attributes considered in Figure 18.4, we have

$$Gain(Patrons) = 1 - \left[\frac{2}{12}B(\frac{0}{2}) + \frac{4}{12}B(\frac{4}{4}) + \frac{6}{12}B(\frac{2}{6})\right] \approx 0.541 \text{ bits,}$$

$$Gain(Type) = 1 - \left[\frac{2}{12}B(\frac{1}{2}) + \frac{2}{12}B(\frac{1}{2}) + \frac{4}{12}B(\frac{2}{4}) + \frac{4}{12}B(\frac{2}{4})\right] = 0 \text{ bits,}$$

confirming our intuition that $Patrons$ is a better attribute to split on. In fact, $Patrons$ has the maximum gain of any of the attributes and would be chosen by the decision-tree learning algorithm as the root.

18.3.5   Generalization and overfitting

On some problems, the DECISION-TREE-LEARNING algorithm will generate a large tree when there is actually no pattern to be found. Consider the problem of trying to predict whether the roll of a die will come up as 6 or not. Suppose that experiments are carried out with various dice and that the attributes describing each training example include the color of the die, its weight, the time when the roll was done, and whether the experimenters had their fingers crossed. If the dice are fair, the right thing to learn is a tree with a single node that says "no," But the DECISION-TREE-LEARNING algorithm will seize on any pattern it can find in the input. If it turns out that there are 2 rolls of a 7-gram blue die with fingers crossed and they both come out 6, then the algorithm may construct a path that predicts 6 in

OVERFITTING
that case. This problem is called **overfitting**. A general phenomenon, overfitting occurs with all types of learners, even when the target function is not at all random. In Figure 18.1(b) and (c), we saw polynomial functions overfitting the data. Overfitting becomes more likely as the hypothesis space and the number of input attributes grows, and less likely as we increase the number of training examples.

DECISION TREE
PRUNING
For decision trees, a technique called **decision tree pruning** combats overfitting. Pruning works by eliminating nodes that are not clearly relevant. We start with a full tree, as generated by DECISION-TREE-LEARNING. We then look at a test node that has only leaf nodes as descendants. If the test appears to be irrelevant—detecting only noise in the data—then we eliminate the test, replacing it with a leaf node. We repeat this process, considering each test with only leaf descendants, until each one has either been pruned or accepted as is.

The question is, how do we detect that a node is testing an irrelevant attribute? Suppose we are at a node consisting of p positive and n negative examples. If the attribute is irrelevant, we would expect that it would split the examples into subsets that each have roughly the same proportion of positive examples as the whole set, $p/(p+n)$, and so the information gain will be close to zero.[2] Thus, the information gain is a good clue to irrelevance. Now the question is, how large a gain should we require in order to split on a particular attribute?

SIGNIFICANCE TEST

NULL HYPOTHESIS
We can answer this question by using a statistical **significance test**. Such a test begins by assuming that there is no underlying pattern (the so-called **null hypothesis**). Then the actual data are analyzed to calculate the extent to which they deviate from a perfect absence of pattern. If the degree of deviation is statistically unlikely (usually taken to mean a 5% probability or less), then that is considered to be good evidence for the presence of a significant pattern in the data. The probabilities are calculated from standard distributions of the amount of deviation one would expect to see in random sampling.

In this case, the null hypothesis is that the attribute is irrelevant and, hence, that the information gain for an infinitely large sample would be zero. We need to calculate the probability that, under the null hypothesis, a sample of size $v = n + p$ would exhibit the observed deviation from the expected distribution of positive and negative examples. We can measure the deviation by comparing the actual numbers of positive and negative examples in

---

[2] The gain will be strictly positive except for the unlikely case where all the proportions are *exactly* the same. (See Exercise 18.5.)

each subset, p_k and n_k, with the expected numbers, $\hat{p}_k$ and $\hat{n}_k$, assuming true irrelevance:

$$\hat{p}_k = p \times \frac{p_k + n_k}{p + n} \qquad\qquad \hat{n}_k = n \times \frac{p_k + n_k}{p + n} \;.$$

A convenient measure of the total deviation is given by

$$\Delta = \sum_{k=1}^{d} \frac{(p_k - \hat{p}_k)^2}{\hat{p}_k} + \frac{(n_k - \hat{n}_k)^2}{\hat{n}_k} \;.$$

Under the null hypothesis, the value of Δ is distributed according to the χ^2 (chi-squared) distribution with $v - 1$ degrees of freedom. We can use a χ^2 table or a standard statistical library routine to see if a particular Δ value confirms or rejects the null hypothesis. For example, consider the restaurant type attribute, with four values and thus three degrees of freedom. A value of $\Delta = 7.82$ or more would reject the null hypothesis at the 5% level (and a value of $\Delta = 11.35$ or more would reject at the 1% level). Exercise 18.8 asks you to extend the DECISION-TREE-LEARNING algorithm to implement this form of pruning, which is known as χ^2 **pruning**.

χ^2 PRUNING

With pruning, noise in the examples can be tolerated. Errors in the example's label (e.g., an example $(\mathbf{x}, Yes)$ that should be $(\mathbf{x}, No)$) give a linear increase in prediction error, whereas errors in the descriptions of examples (e.g., $Price = \$$ when it was actually $Price = \$\$$) have an asymptotic effect that gets worse as the tree shrinks down to smaller sets. Pruned trees perform significantly better than unpruned trees when the data contain a large amount of noise. Also, the pruned trees are often much smaller and hence easier to understand.

One final warning: You might think that χ^2 pruning and information gain look similar, so why not combine them using an approach called **early stopping**—have the decision tree algorithm stop generating nodes when there is no good attribute to split on, rather than going to all the trouble of generating nodes and then pruning them away. The problem with early stopping is that it stops us from recognizing situations where there is no one good attribute, but there are combinations of attributes that are informative. For example, consider the XOR function of two binary attributes. If there are roughly equal number of examples for all four combinations of input values, then neither attribute will be informative, yet the correct thing to do is to split on one of the attributes (it doesn't matter which one), and then at the second level we will get splits that are informative. Early stopping would miss this, but generate-and-then-prune handles it correctly.

EARLY STOPPING

18.3.6   Broadening the applicability of decision trees

In order to extend decision tree induction to a wider variety of problems, a number of issues must be addressed. We will briefly mention several, suggesting that a full understanding is best obtained by doing the associated exercises:

- **Missing data**: In many domains, not all the attribute values will be known for every example. The values might have gone unrecorded, or they might be too expensive to obtain. This gives rise to two problems: First, given a complete decision tree, how should one classify an example that is missing one of the test attributes? Second, how

should one modify the information-gain formula when some examples have unknown values for the attribute? These questions are addressed in Exercise 18.9.

- **Multivalued attributes**: When an attribute has many possible values, the information gain measure gives an inappropriate indication of the attribute's usefulness. In the extreme case, an attribute such as $ExactTime$ has a different value for every example, which means each subset of examples is a singleton with a unique classification, and the information gain measure would have its highest value for this attribute. But choosing this split first is unlikely to yield the best tree. One solution is to use the **gain ratio** (Exercise 18.10). Another possibility is to allow a Boolean test of the form $A = v_k$, that is, picking out just one of the possible values for an attribute, leaving the remaining values to possibly be tested later in the tree.

GAIN RATIO

- **Continuous and integer-valued input attributes**: Continuous or integer-valued attributes such as $Height$ and $Weight$, have an infinite set of possible values. Rather than generate infinitely many branches, decision-tree learning algorithms typically find the **split point** that gives the highest information gain. For example, at a given node in the tree, it might be the case that testing on $Weight > 160$ gives the most information. Efficient methods exist for finding good split points: start by sorting the values of the attribute, and then consider only split points that are between two examples in sorted order that have different classifications, while keeping track of the running totals of positive and negative examples on each side of the split point. Splitting is the most expensive part of real-world decision tree learning applications.

SPLIT POINT

- **Continuous-valued output attributes**: If we are trying to predict a numerical output value, such as the price of an apartment, then we need a **regression tree** rather than a classification tree. A regression tree has at each leaf a linear function of some subset of numerical attributes, rather than a single value. For example, the branch for two-bedroom apartments might end with a linear function of square footage, number of bathrooms, and average income for the neighborhood. The learning algorithm must decide when to stop splitting and begin applying linear regression (see Section 18.6) over the attributes.

REGRESSION TREE

A decision-tree learning system for real-world applications must be able to handle all of these problems. Handling continuous-valued variables is especially important, because both physical and financial processes provide numerical data. Several commercial packages have been built that meet these criteria, and they have been used to develop thousands of fielded systems. In many areas of industry and commerce, decision trees are usually the first method tried when a classification method is to be extracted from a data set. One important property of decision trees is that it is possible for a human to understand the reason for the output of the learning algorithm. (Indeed, this is a *legal requirement* for financial decisions that are subject to anti-discrimination laws.) This is a property not shared by some other representations, such as neural networks.

18.4   EVALUATING AND CHOOSING THE BEST HYPOTHESIS

STATIONARITY
ASSUMPTION

We want to learn a hypothesis that fits the future data best. To make that precise we need to define "future data" and "best." We make the **stationarity assumption**: that there is a probability distribution over examples that remains stationary over time. Each example data point (before we see it) is a random variable E_j whose observed value $e_j = (x_j, y_j)$ is sampled from that distribution, and is independent of the previous examples:

$$\mathbf{P}(E_j|E_{j-1}, E_{j-2}, \ldots) = \mathbf{P}(E_j) \ ,$$

and each example has an identical prior probability distribution:

$$\mathbf{P}(E_j) = \mathbf{P}(E_{j-1}) = \mathbf{P}(E_{j-2}) = \cdots \ .$$

I.I.D.

Examples that satisfy these assumptions are called *independent and identically distributed* or **i.i.d.**. An i.i.d. assumption connects the past to the future; without some such connection, all bets are off—the future could be anything. (We will see later that learning can still occur if there are *slow* changes in the distribution.)

ERROR RATE

The next step is to define "best fit." We define the **error rate** of a hypothesis as the proportion of mistakes it makes—the proportion of times that $h(x) \neq y$ for an (x, y) example. Now, just because a hypothesis h has a low error rate on the training set does not mean that it will generalize well. A professor knows that an exam will not accurately evaluate students if they have already seen the exam questions. Similarly, to get an accurate evaluation of a hypothesis, we need to test it on a set of examples it has not seen yet. The simplest approach is the one we have seen already: randomly split the available data into a training set from which the learning algorithm produces h and a test set on which the accuracy of h is evaluated. This

HOLDOUT
CROSS-VALIDATION

method, sometimes called **holdout cross-validation**, has the disadvantage that it fails to use all the available data; if we use half the data for the test set, then we are only training on half the data, and we may get a poor hypothesis. On the other hand, if we reserve only 10% of the data for the test set, then we may, by statistical chance, get a poor estimate of the actual accuracy.

K-FOLD
CROSS-VALIDATION

We can squeeze more out of the data and still get an accurate estimate using a technique called k-**fold cross-validation**. The idea is that each example serves double duty—as training data and test data. First we split the data into k equal subsets. We then perform k rounds of learning; on each round $1/k$ of the data is held out as a test set and the remaining examples are used as training data. The average test set score of the k rounds should then be a better estimate than a single score. Popular values for k are 5 and 10—enough to give an estimate that is statistically likely to be accurate, at a cost of 5 to 10 times longer computation time.

LEAVE-ONE-OUT
CROSS-VALIDATION

The extreme is $k = n$, also known as **leave-one-out cross-validation** or **LOOCV**.

LOOCV

Despite the best efforts of statistical methodologists, users frequently invalidate their

PEEKING

results by inadvertently **peeking** at the test data. Peeking can happen like this: A learning algorithm has various "knobs" that can be twiddled to tune its behavior—for example, various different criteria for choosing the next attribute in decision tree learning. The researcher generates hypotheses for various different settings of the knobs, measures their error rates on the test set, and reports the error rate of the best hypothesis. Alas, peeking has occurred! The

reason is that the hypothesis was selected *on the basis of its test set error rate*, so information about the test set has leaked into the learning algorithm.

Peeking is a consequence of using test-set performance to both *choose* a hypothesis and *evaluate* it. The way to avoid this is to *really* hold the test set out—lock it away until you are completely done with learning and simply wish to obtain an independent evaluation of the final hypothesis. (And then, if you don't like the results . . . you have to obtain, and lock away, a completely new test set if you want to go back and find a better hypothesis.) If the test set is locked away, but you still want to measure performance on unseen data as a way of selecting a good hypothesis, then divide the available data (without the test set) into a training set and a **validation set**. The next section shows how to use validation sets to find a good tradeoff between hypothesis complexity and goodness of fit.

VALIDATION SET

18.4.1   Model selection: Complexity versus goodness of fit

In Figure 18.1 (page 696) we showed that higher-degree polynomials can fit the training data better, but when the degree is too high they will overfit, and perform poorly on validation data. Choosing the degree of the polynomial is an instance of the problem of **model selection**. You can think of the task of finding the best hypothesis as two tasks: model selection defines the hypothesis space and then **optimization** finds the best hypothesis within that space.

MODEL SELECTION

OPTIMIZATION

In this section we explain how to select among models that are parameterized by *size*. For example, with polynomials we have $size = 1$ for linear functions, $size = 2$ for quadratics, and so on. For decision trees, the size could be the number of nodes in the tree. In all cases we want to find the value of the *size* parameter that best balances underfitting and overfitting to give the best test set accuracy.

An algorithm to perform model selection and optimization is shown in Figure 18.8. It is a **wrapper** that takes a learning algorithm as an argument (DECISION-TREE-LEARNING, for example). The wrapper enumerates models according to a parameter, *size*. For each size, it uses cross validation on *Learner* to compute the average error rate on the training and test sets. We start with the smallest, simplest models (which probably underfit the data), and iterate, considering more complex models at each step, until the models start to overfit. In Figure 18.9 we see typical curves: the training set error decreases monotonically (although there may in general be slight random variation), while the validation set error decreases at first, and then increases when the model begins to overfit. The cross-validation procedure picks the value of *size* with the lowest validation set error; the bottom of the U-shaped curve. We then generate a hypothesis of that *size*, using all the data (without holding out any of it). Finally, of course, we should evaluate the returned hypothesis on a separate test set.

WRAPPER

This approach requires that the learning algorithm accept a parameter, *size*, and deliver a hypothesis of that size. As we said, for decision tree learning, the size can be the number of nodes. We can modify DECISION-TREE-LEARNER so that it takes the number of nodes as an input, builds the tree breadth-first rather than depth-first (but at each level it still chooses the highest gain attribute first), and stops when it reaches the desired number of nodes.

---

function CROSS-VALIDATION-WRAPPER(*Learner*, *k*, *examples*) **returns** a hypothesis

    local variables: *errT*, an array, indexed by *size*, storing training-set error rates
                      errV, an array, indexed by *size*, storing validation-set error rates
    for *size* = 1 **to** ∞ **do**
        errT[*size*], *errV*[*size*] ← CROSS-VALIDATION(*Learner*, *size*, *k*, *examples*)
        if *errT* has converged **then do**
            best_size ← the value of *size* with minimum *errV*[*size*]
            return *Learner*(*best_size*, *examples*)

---

function CROSS-VALIDATION(*Learner*, *size*, *k*, *examples*) **returns** two values:
             average training set error rate, average validation set error rate

    fold_errT ← 0; *fold_errV* ← 0
    for *fold* = 1 *to k* **do**
        training_set, *validation_set* ← PARTITION(*examples*, *fold*, *k*)
        h ← *Learner*(*size*, *training_set*)
        fold_errT ← *fold_errT* + ERROR-RATE(*h*, *training_set*)
        fold_errV ← *fold_errV* + ERROR-RATE(*h*, *validation_set*)
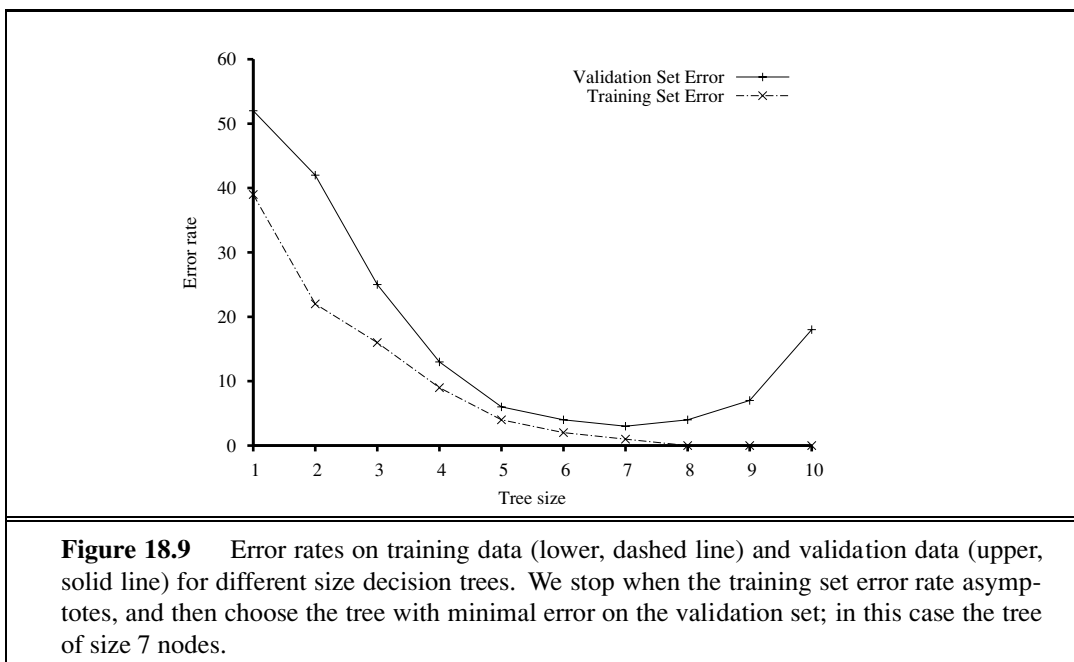   **return** *fold_errT/k*, *fold_errV/k*

Figure 18.8   An algorithm to select the model that has the lowest error rate on validation data by building models of increasing complexity, and choosing the one with best empirical error rate on validation data. Here *errT* means error rate on the training data, and *errV* means error rate on the validation data. *Learner*(*size*, *examples*) returns a hypothesis whose complexity is set by the parameter *size*, and which is trained on the *examples*. PARTITION(*examples*, *fold*, *k*) splits *examples* into two subsets: a validation set of size N/k and a training set with all the other examples. The split is different for each value of *fold*.

18.4.2   From error rates to loss

So far, we have been trying to minimize error rate. This is clearly better than maximizing error rate, but it is not the full story. Consider the problem of classifying email messages as spam or non-spam. It is worse to classify non-spam as spam (and thus potentially miss an important message) then to classify spam as non-spam (and thus suffer a few seconds of annoyance). So a classifier with a 1% error rate, where almost all the errors were classifying spam as non-spam, would be better than a classifier with only a 0.5% error rate, if most of those errors were classifying non-spam as spam. We saw in Chapter 16 that decision-makers should maximize expected utility, and utility is what learners should maximize as well. In machine learning it is traditional to express utilities by means of a **loss function**. The loss function $L(x, y, \hat{y})$ is defined as the amount of utility lost by predicting $h(x) = \hat{y}$ when the correct answer is $f(x) = y$:

LOSS FUNCTION

$$L(x, y, \hat{y}) \;=\; \textit{Utility}(\text{result of using } y \text{ given an input } x)$$
$$-\;\; \textit{Utility}(\text{result of using } \hat{y} \text{ given an input } x)$$

Figure 18.9   Error rates on training data (lower, dashed line) and validation data (upper, solid line) for different size decision trees. We stop when the training set error rate asymptotes, and then choose the tree with minimal error on the validation set; in this case the tree of size 7 nodes.

This is the most general formulation of the loss function. Often a simplified version is used, $L(y, \hat{y})$, that is independent of x. We will use the simplified version for the rest of this chapter, which means we can't say that it is worse to misclassify a letter from Mom than it is to misclassify a letter from our annoying cousin, but we can say it is 10 times worse to classify non-spam as spam than vice-versa:

$$L(spam, nospam) = 1, \quad L(nospam, spam) = 10 \,.$$

Note that $L(y, y)$ is always zero; by definition there is no loss when you guess exactly right. For functions with discrete outputs, we can enumerate a loss value for each possible misclassification, but we can't enumerate all the possibilities for real-valued data. If $f(x)$ is 137.035999, we would be fairly happy with $h(x) = 137.036$, but just how happy should we be? In general small errors are better than large ones; two functions that implement that idea are the absolute value of the difference (called the L_1 loss), and the square of the difference (called the L_2 loss). If we are content with the idea of minimizing error rate, we can use the $L_{0/1}$ loss function, which has a loss of 1 for an incorrect answer and is appropriate for discrete-valued outputs:

> Absolute value loss: $L_1(y, \hat{y}) = |y - \hat{y}|$
> Squared error loss: $L_2(y, \hat{y}) = (y - \hat{y})^2$
> 0/1 loss: $L_{0/1}(y, \hat{y}) = 0$ if $y = \hat{y}$, else 1

The learning agent can theoretically maximize its expected utility by choosing the hypothesis that minimizes expected loss over all input–output pairs it will see. It is meaningless to talk about this expectation without defining a prior probability distribution, $\mathbf{P}(X, Y)$ over examples. Let $\mathcal{E}$ be the set of all possible input–output examples. Then the expected **generalization loss** for a hypothesis h (with respect to loss function L) is

GENERALIZATION LOSS

$$GenLoss_L(h) = \sum_{(x,y)\in\mathcal{E}} L(y, h(x))\, P(x,y)\ ,$$

and the best hypothesis, h^*, is the one with the minimum expected generalization loss:

$$h^* = \underset{h\in\mathcal{H}}{\operatorname{argmin}}\ GenLoss_L(h)\ .$$

EMPIRICAL LOSS

Because $P(x, y)$ is not known, the learning agent can only *estimate* generalization loss with **empirical loss** on a set of examples, E:

$$EmpLoss_{L,E}(h) = \frac{1}{N} \sum_{(x,y)\in E} L(y, h(x))\ .$$

The estimated best hypothesis $\hat{h}^*$ is then the one with minimum empirical loss:

$$\hat{h}^* = \underset{h\in\mathcal{H}}{\operatorname{argmin}}\ EmpLoss_{L,E}(h)\ .$$

There are four reasons why $\hat{h}^*$ may differ from the true function, f: unrealizability, variance, noise, and computational complexity. First, f may not be realizable—may not be in $\mathcal{H}$—or may be present in such a way that other hypotheses are preferred. Second, a learning algorithm will return different hypotheses for different sets of examples, even if those sets are drawn from the same true function f, and those hypotheses will make different predictions on new examples. The higher the variance among the predictions, the higher the probability of significant error. Note that even when the problem is realizable, there will still be random variance, but that variance decreases towards zero as the number of training examples in-

NOISE

creases. Third, f may be nondeterministic or **noisy**—it may return different values for $f(x)$ each time x occurs. By definition, noise cannot be predicted; in many cases, it arises because the observed labels y are the result of attributes of the environment not listed in x. And finally, when $\mathcal{H}$ is complex, it can be computationally intractable to systematically search the whole hypothesis space. The best we can do is a local search (hill climbing or greedy search) that explores only part of the space. That gives us an approximation error. Combining the sources of error, we're left with an estimation of an approximation of the true function f.

SMALL-SCALE LEARNING

Traditional methods in statistics and the early years of machine learning concentrated on **small-scale learning**, where the number of training examples ranged from dozens to the low thousands. Here the generalization error mostly comes from the approximation error of not having the true f in the hypothesis space, and from estimation error of not having enough training examples to limit variance. In recent years there has been more emphasis on **large-**

LARGE-SCALE LEARNING

scale learning, often with millions of examples. Here the generalization error is dominated by limits of computation: there is enough data and a rich enough model that we could find an h that is very close to the true f, but the computation to find it is too complex, so we settle for a sub-optimal approximation.

18.4.3 Regularization

In Section 18.4.1, we saw how to do model selection with cross-validation on model size. An alternative approach is to search for a hypothesis that directly minimizes the weighted sum of

empirical loss and the complexity of the hypothesis, which we will call the total cost:

$$Cost(h) = EmpLoss(h) + \lambda\, Complexity(h)$$
$$\hat{h}^* = \underset{h \in \mathcal{H}}{\operatorname{argmin}}\, Cost(h)\,.$$

Here λ is a parameter, a positive number that serves as a conversion rate between loss and hypothesis complexity (which after all are not measured on the same scale). This approach combines loss and complexity into one metric, allowing us to find the best hypothesis all at once. Unfortunately we still need to do a cross-validation search to find the hypothesis that generalizes best, but this time it is with different values of λ rather than $size$. We select the value of λ that gives us the best validation set score.

REGULARIZATION   This process of explicitly penalizing complex hypotheses is called **regularization** (because it looks for a function that is more regular, or less complex). Note that the cost function requires us to make two choices: the loss function and the complexity measure, which is called a regularization function. The choice of regularization function depends on the hypothesis space. For example, a good regularization function for polynomials is the sum of the squares of the coefficients—keeping the sum small would guide us away from the wiggly polynomials in Figure 18.1(b) and (c). We will show an example of this type of regularization in Section 18.6.

FEATURE SELECTION   Another way to simplify models is to reduce the dimensions that the models work with. A process of **feature selection** can be performed to discard attributes that appear to be irrelevant. χ^2 pruning is a kind of feature selection.

          It is in fact possible to have the empirical loss and the complexity measured on the same scale, without the conversion factor λ: they can both be measured in bits. First encode the hypothesis as a Turing machine program, and count the number of bits. Then count the number of bits required to encode the data, where a correctly predicted example costs zero bits and the cost of an incorrectly predicted example depends on how large the error is.

MINIMUM
DESCRIPTION
LENGTH   The **minimum description length** or MDL hypothesis minimizes the total number of bits required. This works well in the limit, but for smaller problems there is a difficulty in that the choice of encoding for the program—for example, how best to encode a decision tree as a bit string—affects the outcome. In Chapter 20 (page 805), we describe a probabilistic interpretation of the MDL approach.

18.5   THE THEORY OF LEARNING

The main unanswered question in learning is this: How can we be sure that our learning algorithm has produced a hypothesis that will predict the correct value for previously unseen inputs? In formal terms, how do we know that the hypothesis h is close to the target function f if we don't know what f is? These questions have been pondered for several centuries. In more recent decades, other questions have emerged: how many examples do we need to get a good h? What hypothesis space should we use? If the hypothesis space is very complex, can we even find the best h, or do we have to settle for a local maximum in the

space of hypotheses? How complex should h be? How do we avoid overfitting? This section examines these questions.

We'll start with the question of how many examples are needed for learning. We saw from the learning curve for decision tree learning on the restaurant problem (Figure 18.7 on page 703) that improves with more training data. Learning curves are useful, but they are specific to a particular learning algorithm on a particular problem. Are there some more general principles governing the number of examples needed in general? Questions like this are addressed by **computational learning theory**, which lies at the intersection of AI, statistics, and theoretical computer science. The underlying principle is that *any hypothesis that is seriously wrong will almost certainly be "found out" with high probability after a small number of examples, because it will make an incorrect prediction. Thus, any hypothesis that is consistent with a sufficiently large set of training examples is unlikely to be seriously wrong: that is, it must be **probably approximately correct**.* Any learning algorithm that returns hypotheses that are probably approximately correct is called a **PAC learning** algorithm; we can use this approach to provide bounds on the performance of various learning algorithms.

PAC-learning theorems, like all theorems, are logical consequences of axioms. When a *theorem* (as opposed to, say, a political pundit) states something about the future based on the past, the axioms have to provide the "juice" to make that connection. For PAC learning, the juice is provided by the stationarity assumption introduced on page 708, which says that future examples are going to be drawn from the same fixed distribution $\mathbf{P}(E) = \mathbf{P}(X, Y)$ as past examples. (Note that we do not have to know what distribution that is, just that it doesn't change.) In addition, to keep things simple, we will assume that the true function f is deterministic and is a member of the hypothesis class $\mathcal{H}$ that is being considered.

The simplest PAC theorems deal with Boolean functions, for which the 0/1 loss is appropriate. The **error rate** of a hypothesis h, defined informally earlier, is defined formally here as the expected generalization error for examples drawn from the stationary distribution:

$$\text{error}(h) = GenLoss_{L_{0/1}}(h) = \sum_{x,y} L_{0/1}(y, h(x)) P(x, y) \;.$$

In other words, error(h) is the probability that h misclassifies a new example. This is the same quantity being measured experimentally by the learning curves shown earlier.

A hypothesis h is called **approximately correct** if error$(h) \leq \epsilon$, where ϵ is a small constant. We will show that we can find an N such that, after seeing N examples, with high probability, all consistent hypotheses will be approximately correct. One can think of an approximately correct hypothesis as being "close" to the true function in hypothesis space: it lies inside what is called the ϵ-**ball** around the true function f. The hypothesis space outside this ball is called $\mathcal{H}_{\text{bad}}$.

We can calculate the probability that a "seriously wrong" hypothesis $h_b \in \mathcal{H}_{\text{bad}}$ is consistent with the first N examples as follows. We know that error$(h_b) > \epsilon$. Thus, the probability that it agrees with a given example is at most $1 - \epsilon$. Since the examples are independent, the bound for N examples is

$$P(h_b \text{ agrees with } N \text{ examples}) \leq (1 - \epsilon)^N \;.$$

COMPUTATIONAL
LEARNING THEORY

PROBABLY
APPROXIMATELY
CORRECT
PAC LEARNING

ϵ-BALL

The probability that $\mathcal{H}_{\text{bad}}$ contains at least one consistent hypothesis is bounded by the sum of the individual probabilities:

$$P(\mathcal{H}_{\text{bad}} \text{ contains a consistent hypothesis}) \leq |\mathcal{H}_{\text{bad}}|(1-\epsilon)^N \leq |\mathcal{H}|(1-\epsilon)^N \,,$$

where we have used the fact that $|\mathcal{H}_{\text{bad}}| \leq |\mathcal{H}|$. We would like to reduce the probability of this event below some small number δ:

$$|\mathcal{H}|(1-\epsilon)^N \leq \delta \,.$$

Given that $1 - \epsilon \leq e^{-\epsilon}$, we can achieve this if we allow the algorithm to see

$$N \geq \frac{1}{\epsilon}\left(\ln\frac{1}{\delta} + \ln|\mathcal{H}|\right) \tag{18.1}$$

examples. Thus, if a learning algorithm returns a hypothesis that is consistent with this many examples, then with probability at least $1 - \delta$, it has error at most ϵ. In other words, it is probably approximately correct. The number of required examples, as a function of ϵ and δ, is called the **sample complexity** of the hypothesis space.

SAMPLE
COMPLEXITY

As we saw earlier, if $\mathcal{H}$ is the set of all Boolean functions on n attributes, then $|\mathcal{H}| = 2^{2^n}$. Thus, the sample complexity of the space grows as 2^n. Because the number of possible examples is also 2^n, this suggests that PAC-learning in the class of all Boolean functions requires seeing all, or nearly all, of the possible examples. A moment's thought reveals the reason for this: $\mathcal{H}$ contains enough hypotheses to classify any given set of examples in all possible ways. In particular, for any set of N examples, the set of hypotheses consistent with those examples contains equal numbers of hypotheses that predict x_{N+1} to be positive and hypotheses that predict x_{N+1} to be negative.

To obtain real generalization to unseen examples, then, it seems we need to restrict the hypothesis space $\mathcal{H}$ in some way; but of course, if we do restrict the space, we might eliminate the true function altogether. There are three ways to escape this dilemma. The first, which we will cover in Chapter 19, is to bring prior knowledge to bear on the problem. The second, which we introduced in Section 18.4.3, is to insist that the algorithm return not just any consistent hypothesis, but preferably a simple one (as is done in decision tree learning). In cases where finding simple consistent hypotheses is tractable, the sample complexity results are generally better than for analyses based only on consistency. The third escape, which we pursue next, is to focus on learnable subsets of the entire hypothesis space of Boolean functions. This approach relies on the assumption that the restricted language contains a hypothesis h that is close enough to the true function f; the benefits are that the restricted hypothesis space allows for effective generalization and is typically easier to search. We now examine one such restricted language in more detail.

18.5.1   PAC learning example: Learning decision lists

DECISION LISTS

We now show how to apply PAC learning to a new hypothesis space: **decision lists**. A decision list consists of a series of tests, each of which is a conjunction of literals. If a test succeeds when applied to an example description, the decision list specifies the value to be returned. If the test fails, processing continues with the next test in the list. Decision lists resemble decision trees, but their overall structure is simpler: they branch only in one

Figure 18.10   A decision list for the restaurant problem.
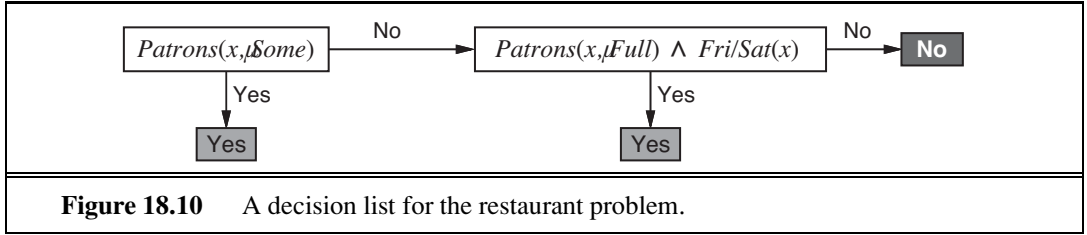
direction. In contrast, the individual tests are more complex. Figure 18.10 shows a decision list that represents the following hypothesis:

$$WillWait \Leftrightarrow (Patrons = Some) \lor (Patrons = Full \land Fri/Sat) .$$

If we allow tests of arbitrary size, then decision lists can represent any Boolean function (Exercise 18.14). On the other hand, if we restrict the size of each test to at most k literals, then it is possible for the learning algorithm to generalize successfully from a small number of examples. We call this language k-**DL**. The example in Figure 18.10 is in 2-DL. It is easy to show (Exercise 18.14) that k-DL includes as a subset the language k-**DT**, the set of all decision trees of depth at most k. It is important to remember that the particular language referred to by k-DL depends on the attributes used to describe the examples. We will use the notation k-DL(n) to denote a k-DL language using n Boolean attributes.

> k-DL
> k-DT

The first task is to show that k-DL is learnable—that is, that any function in k-DL can be approximated accurately after training on a reasonable number of examples. To do this, we need to calculate the number of hypotheses in the language. Let the language of tests— conjunctions of at most k literals using n attributes—be $Conj(n, k)$. Because a decision list is constructed of tests, and because each test can be attached to either a *Yes* or a *No* outcome or can be absent from the decision list, there are at most $3^{|Conj(n,k)|}$ distinct sets of component tests. Each of these sets of tests can be in any order, so

$$|k\text{-DL}(n)| \leq 3^{|Conj(n,k)|}|Conj(n,k)|! .$$

The number of conjunctions of k literals from n attributes is given by

$$|Conj(n,k)| = \sum_{i=0}^{k} \binom{2n}{i} = O(n^k) .$$

Hence, after some work, we obtain

$$|k\text{-DL}(n)| = 2^{O(n^k \log_2(n^k))} .$$

We can plug this into Equation (18.1) to show that the number of examples needed for PAC-learning a k-DL function is polynomial in n:

$$N \geq \frac{1}{\epsilon} \left(\ln\frac{1}{\delta} + O(n^k \log_2(n^k)) \right) .$$

Therefore, any algorithm that returns a consistent decision list will PAC-learn a k-DL function in a reasonable number of examples, for small k.

The next task is to find an efficient algorithm that returns a consistent decision list. We will use a greedy algorithm called DECISION-LIST-LEARNING that repeatedly finds a

---

function DECISION-LIST-LEARNING(*examples*) **returns** a decision list, or *failure*

    if *examples* is empty **then return** the trivial decision list *No*
    $t \leftarrow$ a test that matches a nonempty subset *examples_t* of *examples*
        such that the members of *examples_t* are all positive or all negative
    if there is no such t **then return** *failure*
    if the examples in *examples_t* are positive **then** $o \leftarrow$ *Yes* **else** $o \leftarrow$ *No*
    return a decision list with initial test t and outcome o and remaining tests given by
        DECISION-LIST-LEARNING(*examples* $-$ *examples_t*)

Figure 18.11   An algorithm for learning decision lists.
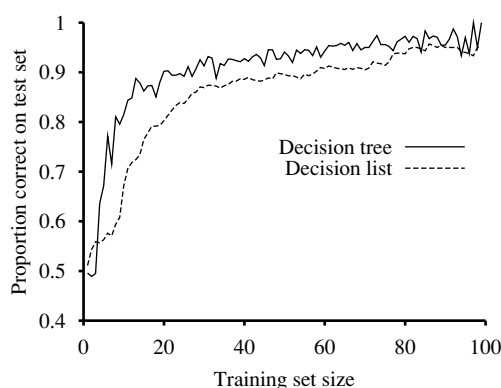

Figure 18.12   Learning curve for DECISION-LIST-LEARNING algorithm on the restaurant data. The curve for DECISION-TREE-LEARNING is shown for comparison.

test that agrees exactly with some subset of the training set. Once it finds such a test, it adds it to the decision list under construction and removes the corresponding examples. It then constructs the remainder of the decision list, using just the remaining examples. This is repeated until there are no examples left. The algorithm is shown in Figure 18.11.

    This algorithm does not specify the method for selecting the next test to add to the decision list. Although the formal results given earlier do not depend on the selection method, it would seem reasonable to prefer small tests that match large sets of uniformly classified examples, so that the overall decision list will be as compact as possible. The simplest strategy is to find the smallest test t that matches any uniformly classified subset, regardless of the size of the subset. Even this approach works quite well, as Figure 18.12 suggests.

18.6 REGRESSION AND CLASSIFICATION WITH LINEAR MODELS

LINEAR FUNCTION

Now it is time to move on from decision trees and lists to a different hypothesis space, one that has been used for hundred of years: the class of **linear functions** of continuous-valued
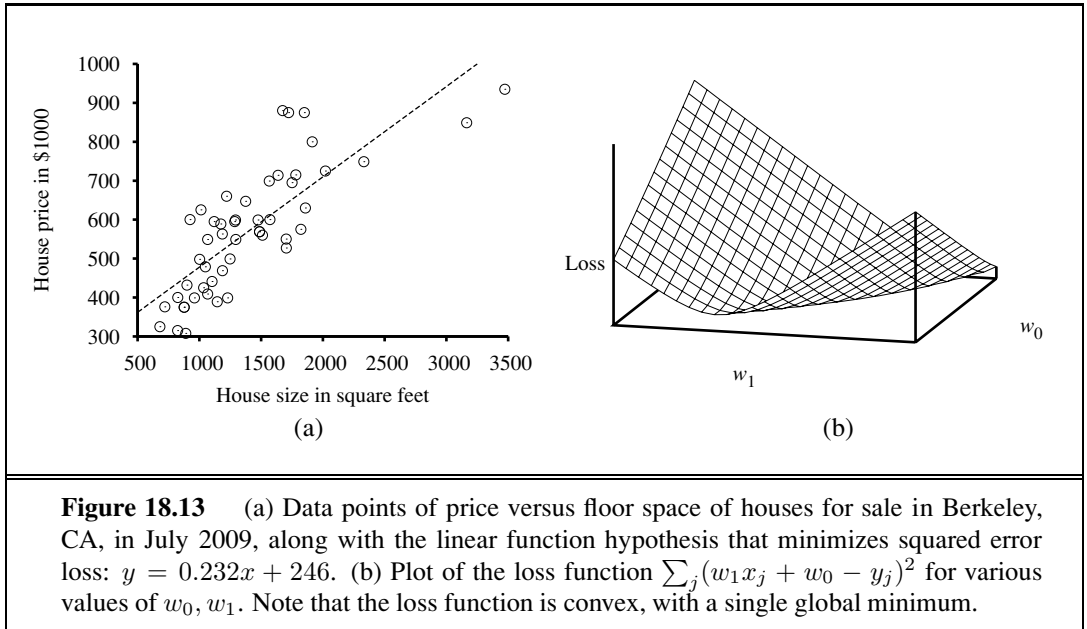
Figure 18.13   (a) Data points of price versus floor space of houses for sale in Berkeley, CA, in July 2009, along with the linear function hypothesis that minimizes squared error loss: $y = 0.232x + 246$. (b) Plot of the loss function $\sum_j (w_1 x_j + w_0 - y_j)^2$ for various values of w_0, w_1. Note that the loss function is convex, with a single global minimum.

inputs. We'll start with the simplest case: regression with a univariate linear function, otherwise known as "fitting a straight line." Section 18.6.2 covers the multivariate case. Sections 18.6.3 and 18.6.4 show how to turn linear functions into classifiers by applying hard and soft thresholds.

18.6.1   Univariate linear regression

A univariate linear function (a straight line) with input x and output y has the form $y = w_1 x + w_0$, where w_0 and w_1 are real-valued coefficients to be learned. We use the letter w because

WEIGHT   we think of the coefficients as **weights**; the value of y is changed by changing the relative weight of one term or another. We'll define **w** to be the vector $[w_0, w_1]$, and define

$$h_{\mathbf{w}}(x) = w_1 x + w_0 .$$

Figure 18.13(a) shows an example of a training set of n points in the x, y plane, each point representing the size in square feet and the price of a house offered for sale. The task of

LINEAR REGRESSION   finding the $h_{\mathbf{w}}$ that best fits these data is called **linear regression**. To fit a line to the data, all we have to do is find the values of the weights $[w_0, w_1]$ that minimize the empirical loss. It is traditional (going back to Gauss[3]) to use the squared loss function, L_2, summed over all the training examples:

$$Loss(h_{\mathbf{w}}) = \sum_{j=1}^{N} L_2(y_j, h_{\mathbf{w}}(x_j)) = \sum_{j=1}^{N} (y_j - h_{\mathbf{w}}(x_j))^2 = \sum_{j=1}^{N} (y_j - (w_1 x_j + w_0))^2 .$$

----

[3] Gauss showed that if the y_j values have normally distributed noise, then the most likely values of w_1 and w_0 are obtained by minimizing the sum of the squares of the errors.

We would like to find $\mathbf{w}^* = \text{argmin}_{\mathbf{w}} \, Loss(h_{\mathbf{w}})$. The sum $\sum_{j=1}^{N}(y_j - (w_1 x_j + w_0))^2$ is minimized when its partial derivatives with respect to w_0 and w_1 are zero:

$$\frac{\partial}{\partial w_0}\sum_{j=1}^{N}(y_j - (w_1 x_j + w_0))^2 = 0 \text{ and } \frac{\partial}{\partial w_1}\sum_{j=1}^{N}(y_j - (w_1 x_j + w_0))^2 = 0 . \quad (18.2)$$

These equations have a unique solution:

$$w_1 = \frac{N(\sum x_j y_j) - (\sum x_j)(\sum y_j)}{N(\sum x_j^2) - (\sum x_j)^2}; \quad w_0 = (\sum y_j - w_1(\sum x_j))/N . \quad (18.3)$$

For the example in Figure 18.13(a), the solution is $w_1 = 0.232$, $w_0 = 246$, and the line with those weights is shown as a dashed line in the figure.

Many forms of learning involve adjusting weights to minimize a loss, so it helps to have a mental picture of what's going on in **weight space**—the space defined by all possible settings of the weights. For univariate linear regression, the weight space defined by w_0 and w_1 is two-dimensional, so we can graph the loss as a function of w_0 and w_1 in a 3D plot (see Figure 18.13(b)). We see that the loss function is **convex**, as defined on page 133; this is true for *every* linear regression problem with an L_2 loss function, and implies that there are no local minima. In some sense that's the end of the story for linear models; if we need to fit lines to data, we apply Equation (18.3).[4]

WEIGHT SPACE

To go beyond linear models, we will need to face the fact that the equations defining minimum loss (as in Equation (18.2)) will often have no closed-form solution. Instead, we will face a general optimization search problem in a continuous weight space. As indicated in Section 4.2 (page 129), such problems can be addressed by a hill-climbing algorithm that follows the **gradient** of the function to be optimized. In this case, because we are trying to minimize the loss, we will use **gradient descent**. We choose any starting point in weight space—here, a point in the (w_0, w_1) plane—and then move to a neighboring point that is downhill, repeating until we converge on the minimum possible loss:

GRADIENT DESCENT

> $\mathbf{w} \leftarrow$ any point in the parameter space
>
> **loop** until convergence **do**
>
> > **for each** w_i **in w do**
> >
> > $$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} Loss(\mathbf{w}) \qquad\qquad (18.4)$$

The parameter α, which we called the **step size** in Section 4.2, is usually called the **learning rate** when we are trying to minimize loss in a learning problem. It can be a fixed constant, or it can decay over time as the learning process proceeds.

LEARNING RATE

For univariate regression, the loss function is a quadratic function, so the partial derivative will be a linear function. (The only calculus you need to know is that $\frac{\partial}{\partial x} x^2 = 2x$ and $\frac{\partial}{\partial x} x = 1$.) Let's first work out the partial derivatives—the slopes—in the simplified case of

---

[4]   With some caveats: the L_2 loss function is appropriate when there is normally-distributed noise that is independent of x; all results rely on the stationarity assumption; etc.

only one training example, (x, y):

$$\frac{\partial}{\partial w_i} Loss(\mathbf{w}) = \frac{\partial}{\partial w_i}(y - h_{\mathbf{w}}(x))^2$$

$$= 2(y - h_{\mathbf{w}}(x)) \times \frac{\partial}{\partial w_i}(y - h_{\mathbf{w}}(x))$$

$$= 2(y - h_{\mathbf{w}}(x)) \times \frac{\partial}{\partial w_i}(y - (w_1 x + w_0)) , \qquad (18.5)$$

applying this to both w_0 and w_1 we get:

$$\frac{\partial}{\partial w_0} Loss(\mathbf{w}) = -2(y - h_{\mathbf{w}}(x)) ; \qquad \frac{\partial}{\partial w_1} Loss(\mathbf{w}) = -2(y - h_{\mathbf{w}}(x)) \times x$$

Then, plugging this back into Equation (18.4), and folding the 2 into the unspecified learning rate α, we get the following learning rule for the weights:

$$w_0 \leftarrow w_0 + \alpha\, (y - h_{\mathbf{w}}(x)) ; \quad w_1 \leftarrow w_1 + \alpha\, (y - h_{\mathbf{w}}(x)) \times x$$

These updates make intuitive sense: if $h_{\mathbf{w}}(x) > y$, i.e., the output of the hypothesis is too large, reduce w_0 a bit, and reduce w_1 if x was a positive input but increase w_1 if x was a negative input.

The preceding equations cover one training example. For N training examples, we want to minimize the sum of the individual losses for each example. The derivative of a sum is the sum of the derivatives, so we have:

$$w_0 \leftarrow w_0 + \alpha \sum_j (y_j - h_{\mathbf{w}}(x_j)) ; \quad w_1 \leftarrow w_1 + \alpha \sum_j (y_j - h_{\mathbf{w}}(x_j)) \times x_j .$$

BATCH GRADIENT
DESCENT

These updates constitute the **batch gradient descent** learning rule for univariate linear regression. Convergence to the unique global minimum is guaranteed (as long as we pick α small enough) but may be very slow: we have to cycle through all the training data for every step, and there may be many steps.

STOCHASTIC
GRADIENT DESCENT

There is another possibility, called **stochastic gradient descent**, where we consider only a single training point at a time, taking a step after each one using Equation (18.5). Stochastic gradient descent can be used in an online setting, where new data are coming in one at a time, or offline, where we cycle through the same data as many times as is necessary, taking a step after considering each single example. It is often faster than batch gradient descent. With a fixed learning rate α, however, it does not guarantee convergence; it can oscillate around the minimum without settling down. In some cases, as we see later, a schedule of decreasing learning rates (as in simulated annealing) does guarantee convergence.

18.6.2   Multivariate linear regression

MULTIVARIATE
LINEAR REGRESSION

We can easily extend to **multivariate linear regression** problems, in which each example $\mathbf{x}_j$ is an n-element vector.[5] Our hypothesis space is the set of functions of the form

$$h_{sw}(\mathbf{x}_j) = w_0 + w_1 x_{j,1} + \cdots + w_n x_{j,n} = w_0 + \sum_i w_i x_{j,i} .$$

---

[5]   The reader may wish to consult Appendix A for a brief summary of linear algebra.

The w_0 term, the intercept, stands out as different from the others. We can fix that by inventing a dummy input attribute, $x_{j,0}$, which is defined as always equal to 1. Then h is simply the dot product of the weights and the input vector (or equivalently, the matrix product of the transpose of the weights and the input vector):

$$h_{sw}(\mathbf{x}_j) = \mathbf{w} \cdot \mathbf{x}_j = \mathbf{w}^\top \mathbf{x}_j = \sum_i w_i x_{j,i} .$$

The best vector of weights, $\mathbf{w}^*$, minimizes squared-error loss over the examples:

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_j L_2(y_j, \mathbf{w} \cdot \mathbf{x}_j) .$$

Multivariate linear regression is actually not much more complicated than the univariate case we just covered. Gradient descent will reach the (unique) minimum of the loss function; the update equation for each weight w_i is

$$w_i \leftarrow w_i + \alpha \sum_j x_{j,i}(y_j - h_\mathbf{w}(\mathbf{x}_j)) . \tag{18.6}$$

It is also possible to solve analytically for the $\mathbf{w}$ that minimizes loss. Let $\mathbf{y}$ be the vector of outputs for the training examples, and $\mathbf{X}$ be the **data matrix**, i.e., the matrix of inputs with one n-dimensional example per row. Then the solution

DATA MATRIX

$$\mathbf{w}^* = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$$

minimizes the squared error.

With univariate linear regression we didn't have to worry about overfitting. But with multivariate linear regression in high-dimensional spaces it is possible that some dimension that is actually irrelevant appears by chance to be useful, resulting in **overfitting**.

Thus, it is common to use **regularization** on multivariate linear functions to avoid overfitting. Recall that with regularization we minimize the total cost of a hypothesis, counting both the empirical loss and the complexity of the hypothesis:

$$Cost(h) = EmpLoss(h) + \lambda \, Complexity(h) .$$

For linear functions the complexity can be specified as a function of the weights. We can consider a family of regularization functions:

$$Complexity(h_\mathbf{w}) = L_q(\mathbf{w}) = \sum_i |w_i|^q .$$

As with loss functions,[6] with $q = 1$ we have L_1 regularization, which minimizes the sum of the absolute values; with $q = 2$, L_2 regularization minimizes the sum of squares. Which regularization function should you pick? That depends on the specific problem, but L_1 regularization has an important advantage: it tends to produce a **sparse model**. That is, it often sets many weights to zero, effectively declaring the corresponding attributes to be irrelevant—just as DECISION-TREE-LEARNING does (although by a different mechanism). Hypotheses that discard attributes can be easier for a human to understand, and may be less likely to overfit.

SPARSE MODEL

---

[6] It is perhaps confusing that L_1 and L_2 are used for both loss functions and regularization functions. They need not be used in pairs: you could use L_2 loss with L_1 regularization, or vice versa.
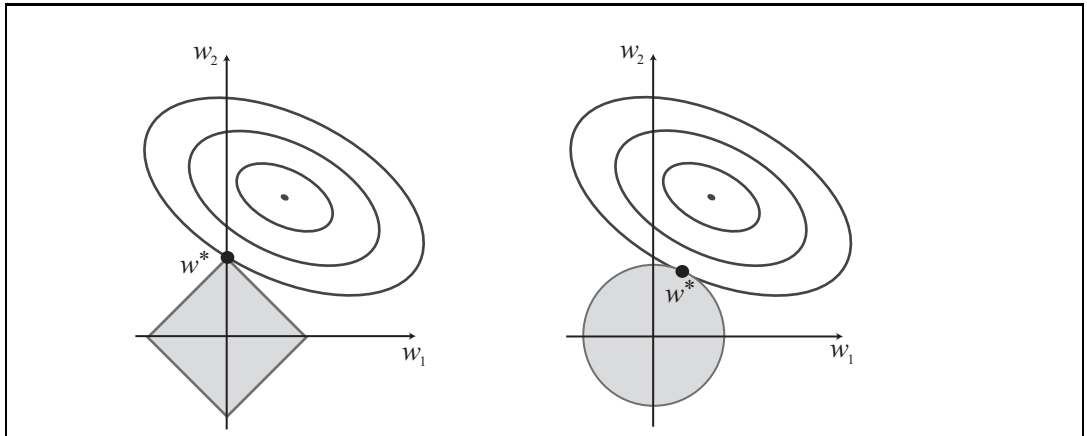
Figure 18.14   Why L_1 regularization tends to produce a sparse model. (a) With L_1 regularization (box), the minimal achievable loss (concentric contours) often occurs on an axis, meaning a weight of zero. (b) With L_2 regularization (circle), the minimal loss is likely to occur anywhere on the circle, giving no preference to zero weights.

Figure 18.14 gives an intuitive explanation of why L_1 regularization leads to weights of zero, while L_2 regularization does not. Note that minimizing $Loss(\mathbf{w}) + \lambda Complexity(\mathbf{w})$ is equivalent to minimizing $Loss(\mathbf{w})$ subject to the constraint that $Complexity(\mathbf{w}) \leq c$, for some constant c that is related to λ. Now, in Figure 18.14(a) the diamond-shaped box represents the set of points $\mathbf{w}$ in two-dimensional weight space that have L_1 complexity less than c; our solution will have to be somewhere inside this box. The concentric ovals represent contours of the loss function, with the minimum loss at the center. We want to find the point in the box that is closest to the minimum; you can see from the diagram that, for an arbitrary position of the minimum and its contours, it will be common for the corner of the box to find its way closest to the minimum, just because the corners are pointy. And of course the corners are the points that have a value of zero in some dimension. In Figure 18.14(b), we've done the same for the L_2 complexity measure, which represents a circle rather than a diamond. Here you can see that, in general, there is no reason for the intersection to appear on one of the axes; thus L_2 regularization does not tend to produce zero weights. The result is that the number of examples required to find a good h is linear in the number of irrelevant features for L_2 regularization, but only logarithmic with L_1 regularization. Empirical evidence on many problems supports this analysis.

Another way to look at it is that L_1 regularization takes the dimensional axes seriously, while L_2 treats them as arbitrary. The L_2 function is spherical, which makes it rotationally invariant: Imagine a set of points in a plane, measured by their x and y coordinates. Now imagine rotating the axes by 45°. You'd get a different set of (x', y') values representing the same points. If you apply L_2 regularization before and after rotating, you get exactly the same point as the answer (although the point would be described with the new (x', y') coordinates). That is appropriate when the choice of axes really is arbitrary—when it doesn't matter whether your two dimensions are distances north and east; or distances north-east and

south-east. With L_1 regularization you'd get a different answer, because the L_1 function is not rotationally invariant. That is appropriate when the axes are not interchangeable; it doesn't make sense to rotate "number of bathrooms" 45° towards "lot size."

18.6.3   Linear classifiers with a hard threshold

Linear functions can be used to do classification as well as regression. For example, Figure 18.15(a) shows data points of two classes: earthquakes (which are of interest to seismologists) and underground explosions (which are of interest to arms control experts). Each point is defined by two input values, x_1 and x_2, that refer to body and surface wave magnitudes computed from the seismic signal. Given these training data, the task of classification is to learn a hypothesis h that will take new (x_1, x_2) points and return either 0 for earthquakes or 1 for explosions.
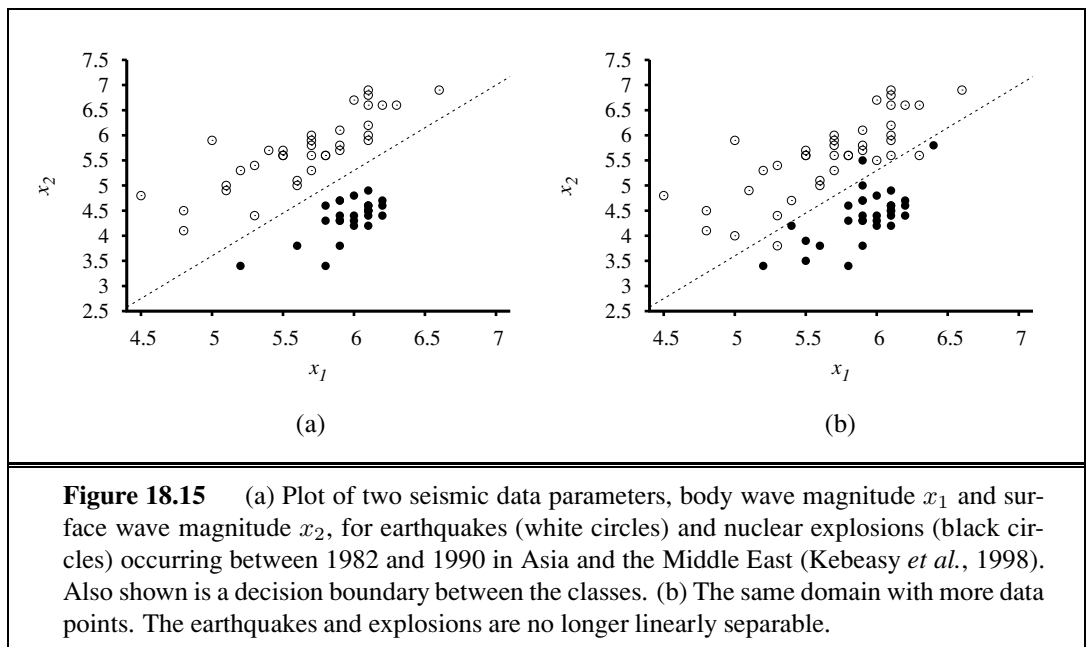

Figure 18.15   (a) Plot of two seismic data parameters, body wave magnitude x_1 and surface wave magnitude x_2, for earthquakes (white circles) and nuclear explosions (black circles) occurring between 1982 and 1990 in Asia and the Middle East (Kebeasy *et al.*, 1998). Also shown is a decision boundary between the classes. (b) The same domain with more data points. The earthquakes and explosions are no longer linearly separable.

DECISION
BOUNDARY

LINEAR SEPARATOR

LINEAR
SEPARABILITY

A **decision boundary** is a line (or a surface, in higher dimensions) that separates the two classes. In Figure 18.15(a), the decision boundary is a straight line. A linear decision boundary is called a **linear separator** and data that admit such a separator are called **linearly separable**. The linear separator in this case is defined by

$$x_2 = 1.7x_1 - 4.9 \quad \text{or} \quad -4.9 + 1.7x_1 - x_2 = 0 \ .$$

The explosions, which we want to classify with value 1, are to the right of this line with higher values of x_1 and lower values of x_2, so they are points for which $-4.9 + 1.7x_1 - x_2 > 0$, while earthquakes have $-4.9 + 1.7x_1 - x_2 < 0$. Using the convention of a dummy input $x_0 = 1$, we can write the classification hypothesis as

$$h_{\mathbf{w}}(\mathbf{x}) = 1 \text{ if } \mathbf{w} \cdot \mathbf{x} \geq 0 \text{ and } 0 \text{ otherwise.}$$

Alternatively, we can think of h as the result of passing the linear function $\mathbf{w} \cdot \mathbf{x}$ through a

THRESHOLD
FUNCTION

threshold function:

$$h_{\mathbf{w}}(\mathbf{x}) = Threshold(\mathbf{w} \cdot \mathbf{x}) \text{ where } Threshold(z) = 1 \text{ if } z \geq 0 \text{ and } 0 \text{ otherwise.}$$

The threshold function is shown in Figure 18.17(a).

Now that the hypothesis $h_{\mathbf{w}}(\mathbf{x})$ has a well-defined mathematical form, we can think about choosing the weights $\mathbf{w}$ to minimize the loss. In Sections 18.6.1 and 18.6.2, we did this both in closed form (by setting the gradient to zero and solving for the weights) and by gradient descent in weight space. Here, we cannot do either of those things because the gradient is zero almost everywhere in weight space except at those points where $\mathbf{w} \cdot \mathbf{x} = 0$, and at those points the gradient is undefined.

There is, however, a simple weight update rule that converges to a solution—that is, a linear separator that classifies the data perfectly–provided the data are linearly separable. For a single example $(\mathbf{x}, y)$, we have

$$w_i \leftarrow w_i + \alpha \left(y - h_{\mathbf{w}}(\mathbf{x}) \right) \times x_i \tag{18.7}$$
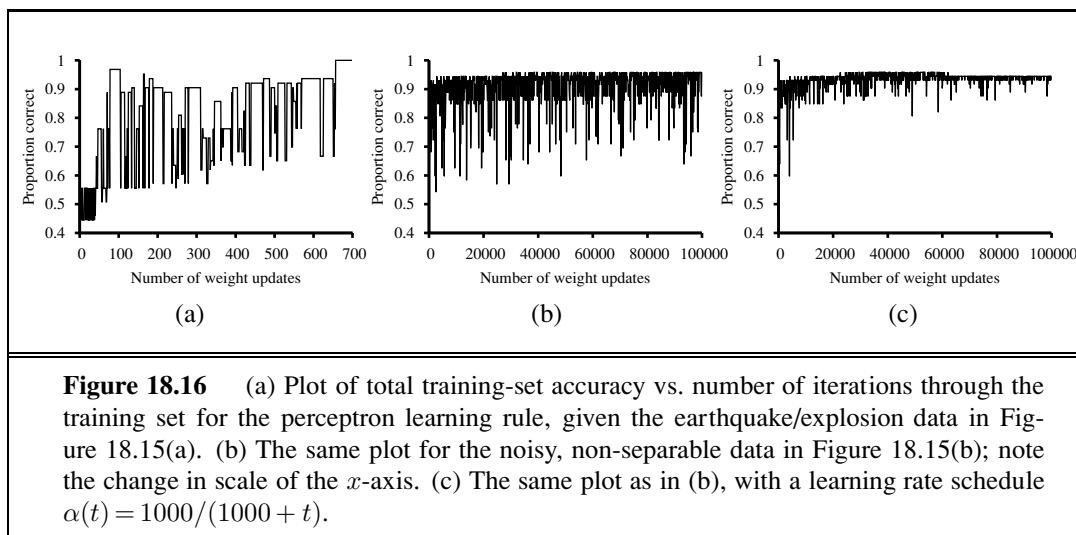
PERCEPTRON
LEARNING RULE

which is essentially identical to the Equation (18.6), the update rule for linear regression! This rule is called the **perceptron learning rule**, for reasons that will become clear in Section 18.7. Because we are considering a 0/1 classification problem, however, the behavior is somewhat different. Both the true value y and the hypothesis output $h_{\mathbf{w}}(\mathbf{x})$ are either 0 or 1, so there are three possibilities:

- If the output is correct, i.e., $y = h_{\mathbf{w}}(\mathbf{x})$, then the weights are not changed.
- If y is 1 but $h_{\mathbf{w}}(\mathbf{x})$ is 0, then w_i is *increased* when the corresponding input x_i is positive and *decreased* when x_i is negative. This makes sense, because we want to make $\mathbf{w} \cdot \mathbf{x}$ bigger so that $h_{\mathbf{w}}(\mathbf{x})$ outputs a 1.
- If y is 0 but $h_{\mathbf{w}}(\mathbf{x})$ is 1, then w_i is *decreased* when the corresponding input x_i is positive and *increased* when x_i is negative. This makes sense, because we want to make $\mathbf{w} \cdot \mathbf{x}$ smaller so that $h_{\mathbf{w}}(\mathbf{x})$ outputs a 0.

TRAINING CURVE

Typically the learning rule is applied one example at a time, choosing examples at random (as in stochastic gradient descent). Figure 18.16(a) shows a **training curve** for this learning rule applied to the earthquake/explosion data shown in Figure 18.15(a). A training curve measures the classifier performance on a fixed training set as the learning process proceeds on that same training set. The curve shows the update rule converging to a zero-error linear separator. The "convergence" process isn't exactly pretty, but it always works. This particular run takes 657 steps to converge, for a data set with 63 examples, so each example is presented roughly 10 times on average. Typically, the variation across runs is very large.

We have said that the perceptron learning rule converges to a perfect linear separator when the data points are linearly separable, but what if they are not? This situation is all too common in the real world. For example, Figure 18.15(b) adds back in the data points left out by Kebeasy *et al.* (1998) when they plotted the data shown in Figure 18.15(a). In Figure 18.16(b), we show the perceptron learning rule failing to converge even after 10,000 steps: even though it hits the minimum-error solution (three errors) many times, the algorithm keeps changing the weights. In general, the perceptron rule may not converge to a

Figure 18.16   (a) Plot of total training-set accuracy vs. number of iterations through the training set for the perceptron learning rule, given the earthquake/explosion data in Figure 18.15(a). (b) The same plot for the noisy, non-separable data in Figure 18.15(b); note the change in scale of the x-axis. (c) The same plot as in (b), with a learning rate schedule $\alpha(t) = 1000/(1000 + t)$.

stable solution for fixed learning rate α, but if α decays as $O(1/t)$ where t is the iteration number, then the rule can be shown to converge to a minimum-error solution when examples are presented in a random sequence.[7] It can also be shown that finding the minimum-error solution is NP-hard, so one expects that many presentations of the examples will be required for convergence to be achieved. Figure 18.16(b) shows the training process with a learning rate schedule $\alpha(t) = 1000/(1000 + t)$: convergence is not perfect after 100,000 iterations, but it is much better than the fixed-α case.
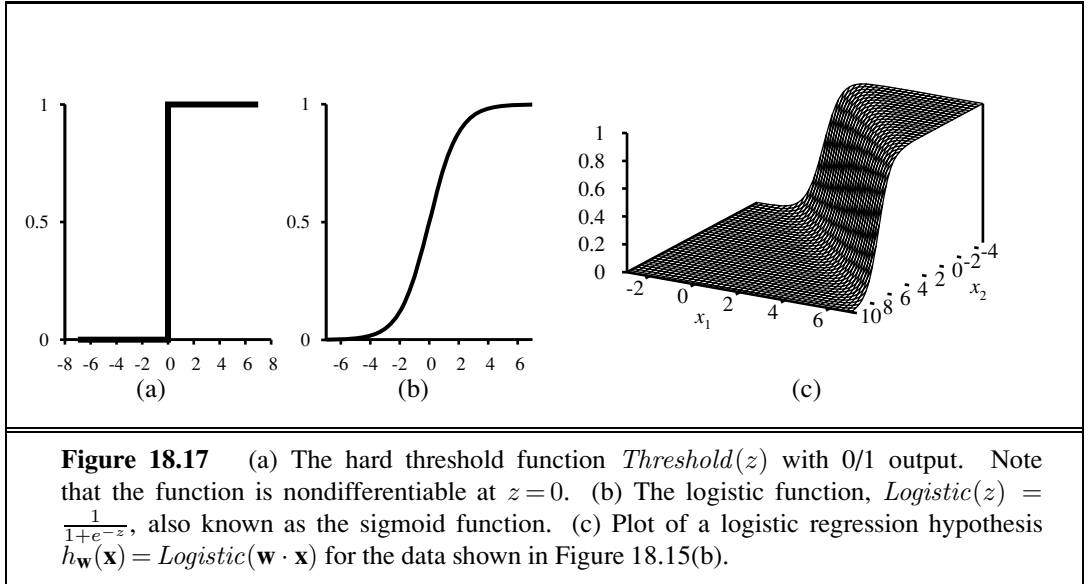
18.6.4   Linear classification with logistic regression

We have seen that passing the output of a linear function through the threshold function creates a linear classifier; yet the hard nature of the threshold causes some problems: the hypothesis $h_{\mathbf{w}}(\mathbf{x})$ is not differentiable and is in fact a discontinuous function of its inputs and its weights; this makes learning with the perceptron rule a very unpredictable adventure. Furthermore, the linear classifier always announces a completely confident prediction of 1 or 0, even for examples that are very close to the boundary; in many situations, we really need more gradated predictions.

     All of these issues can be resolved to a large extent by softening the threshold function—approximating the hard threshold with a continuous, differentiable function. In Chapter 14 (page 522), we saw two functions that look like soft thresholds: the integral of the standard normal distribution (used for the probit model) and the logistic function (used for the logit model). Although the two functions are very similar in shape, the logistic function

$$Logistic(z) = \frac{1}{1 + e^{-z}}$$

---

[7] Technically, we require that $\sum_{t=1}^{\infty} \alpha(t) = \infty$ and $\sum_{t=1}^{\infty} \alpha^2(t) < \infty$. The decay $\alpha(t) = O(1/t)$ satisfies these conditions.

Figure 18.17   (a) The hard threshold function $Threshold(z)$ with 0/1 output.   Note that the function is nondifferentiable at $z = 0$.   (b) The logistic function, $Logistic(z) = \frac{1}{1+e^{-z}}$, also known as the sigmoid function.   (c) Plot of a logistic regression hypothesis $h_{\mathbf{w}}(\mathbf{x}) = Logistic(\mathbf{w} \cdot \mathbf{x})$ for the data shown in Figure 18.15(b).

has more convenient mathematical properties. The function is shown in Figure 18.17(b). With the logistic function replacing the threshold function, we now have

$$h_{\mathbf{w}}(\mathbf{x}) = Logistic(\mathbf{w} \cdot \mathbf{x}) = \frac{1}{1+e^{-\mathbf{w}\cdot\mathbf{x}}} \ .$$

An example of such a hypothesis for the two-input earthquake/explosion problem is shown in Figure 18.17(c). Notice that the output, being a number between 0 and 1, can be interpreted as a *probability* of belonging to the class labeled 1. The hypothesis forms a soft boundary in the input space and gives a probability of 0.5 for any input at the center of the boundary region, and approaches 0 or 1 as we move away from the boundary.

LOGISTIC
REGRESSION

The process of fitting the weights of this model to minimize loss on a data set is called **logistic regression**. There is no easy closed-form solution to find the optimal value of $\mathbf{w}$ with this model, but the gradient descent computation is straightforward. Because our hypotheses no longer output just 0 or 1, we will use the L_2 loss function; also, to keep the formulas readable, we'll use g to stand for the logistic function, with g' its derivative.

For a single example $(\mathbf{x}, y)$, the derivation of the gradient is the same as for linear regression (Equation (18.5)) up to the point where the actual form of h is inserted. (For this CHAIN RULE   derivation, we will need the **chain rule**: $\partial g(f(x))/\partial x = g'(f(x))\, \partial f(x)/\partial x$.) We have

$$\begin{aligned}
\frac{\partial}{\partial w_i} Loss(\mathbf{w}) &= \frac{\partial}{\partial w_i}(y - h_{\mathbf{w}}(\mathbf{x}))^2 \\
&= 2(y - h_{\mathbf{w}}(\mathbf{x})) \times \frac{\partial}{\partial w_i}(y - h_{\mathbf{w}}(\mathbf{x})) \\
&= -2(y - h_{\mathbf{w}}(\mathbf{x})) \times g'(\mathbf{w} \cdot \mathbf{x}) \times \frac{\partial}{\partial w_i}\mathbf{w} \cdot \mathbf{x} \\
&= -2(y - h_{\mathbf{w}}(\mathbf{x})) \times g'(\mathbf{w} \cdot \mathbf{x}) \times x_i \ .
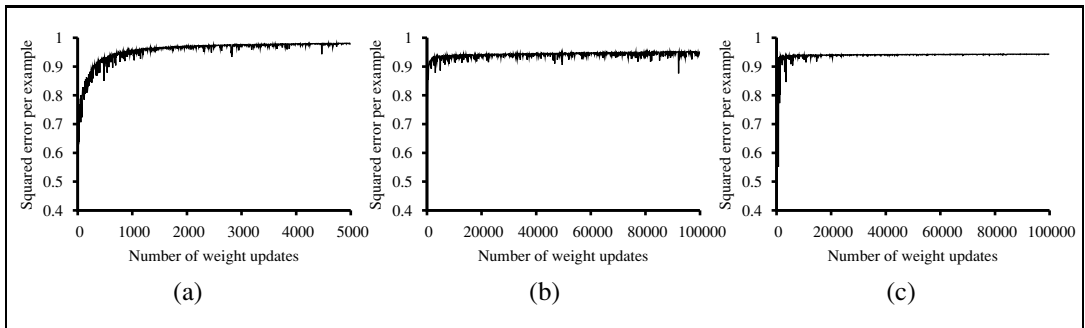\end{aligned}$$

Figure 18.18   Repeat of the experiments in Figure 18.16 using logistic regression and squared error. The plot in (a) covers 5000 iterations rather than 1000, while (b) and (c) use the same scale.

The derivative g' of the logistic function satisfies $g'(z) = g(z)(1 - g(z))$, so we have

$$g'(\mathbf{w} \cdot \mathbf{x}) = g(\mathbf{w} \cdot \mathbf{x})(1 - g(\mathbf{w} \cdot \mathbf{x})) = h_{\mathbf{w}}(\mathbf{x})(1 - h_{\mathbf{w}}(\mathbf{x}))$$

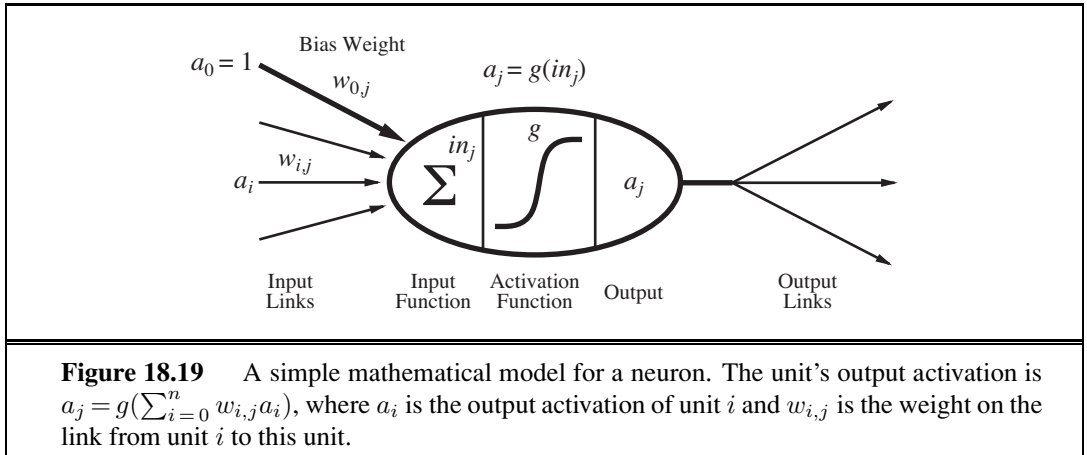so the weight update for minimizing the loss is

$$w_i \; \leftarrow \; w_i + \alpha \, (y - h_{\mathbf{w}}(\mathbf{x})) \times h_{\mathbf{w}}(\mathbf{x})(1 - h_{\mathbf{w}}(\mathbf{x})) \times x_i \; . \tag{18.8}$$

Repeating the experiments of Figure 18.16 with logistic regression instead of the linear threshold classifier, we obtain the results shown in Figure 18.18. In (a), the linearly separable case, logistic regression is somewhat slower to converge, but behaves much more predictably. In (b) and (c), where the data are noisy and nonseparable, logistic regression converges far more quickly and reliably. These advantages tend to carry over into real-world applications and logistic regression has become one of the most popular classification techniques for problems in medicine, marketing and survey analysis, credit scoring, public health, and other applications.

18.7   ARTIFICIAL NEURAL NETWORKS

We turn now to what seems to be a somewhat unrelated topic: the brain. In fact, as we will see, the technical ideas we have discussed so far in this chapter turn out to be useful in building mathematical models of the brain's activity; conversely, thinking about the brain has helped in extending the scope of the technical ideas.

Chapter 1 touched briefly on the basic findings of neuroscience—in particular, the hypothesis that mental activity consists primarily of electrochemical activity in networks of brain cells called **neurons**. (Figure 1.2 on page 11 showed a schematic diagram of a typical neuron.) Inspired by this hypothesis, some of the earliest AI work aimed to create artificial **neural networks**. (Other names for the field include **connectionism**, **parallel distributed processing**, and **neural computation**.) Figure 18.19 shows a simple mathematical model of the neuron devised by McCulloch and Pitts (1943). Roughly speaking, it "fires" when a linear combination of its inputs exceeds some (hard or soft) threshold—that is, it implements

NEURAL NETWORK

Figure 18.19   A simple mathematical model for a neuron. The unit's output activation is $a_j = g(\sum_{i=0}^{n} w_{i,j}a_i)$, where a_i is the output activation of unit i and $w_{i,j}$ is the weight on the link from unit i to this unit.

a linear classifier of the kind described in the preceding section. A neural network is just a collection of units connected together; the properties of the network are determined by its topology and the properties of the "neurons."

Since 1943, much more detailed and realistic models have been developed, both for neurons and for larger systems in the brain, leading to the modern field of **computational neuroscience**. On the other hand, researchers in AI and statistics became interested in the more abstract properties of neural networks, such as their ability to perform distributed computation, to tolerate noisy inputs, and to learn. Although we understand now that other kinds of systems—including Bayesian networks—have these properties, neural networks remain one of the most popular and effective forms of learning system and are worthy of study in their own right.

COMPUTATIONAL
NEUROSCIENCE

18.7.1   Neural network structures

UNIT
LINK
ACTIVATION
WEIGHT

Neural networks are composed of nodes or **units** (see Figure 18.19) connected by directed **links**. A link from unit i to unit j serves to propagate the **activation** a_i from i to j.[8] Each link also has a numeric **weight** $w_{i,j}$ associated with it, which determines the strength and sign of the connection. Just as in linear regression models, each unit has a dummy input $a_0 = 1$ with an associated weight $w_{0,j}$. Each unit j first computes a weighted sum of its inputs:

$$in_j = \sum_{i=0}^{n} w_{i,j}a_i \ .$$

ACTIVATION
FUNCTION

Then it applies an **activation function** g to this sum to derive the output:

$$a_j = g(in_j) = g\left(\sum_{i=0}^{n} w_{i,j}a_i\right) \ . \tag{18.9}$$

---

[8] A note on notation: for this section, we are forced to suspend our usual conventions. Input attributes are still indexed by i , so that an "external" activation a_i is given by input x_i; but index j will refer to internal units rather than examples. Throughout this section, the mathematical derivations concern a single generic example **x**, omitting the usual summations over examples to obtain results for the whole data set.

PERCEPTRON
SIGMOID
PERCEPTRON

The activation function g is typically either a hard threshold (Figure 18.17(a)), in which case the unit is called a **perceptron**, or a logistic function (Figure 18.17(b)), in which case the term **sigmoid perceptron** is sometimes used. Both of these nonlinear activation function ensure the important property that the entire network of units can represent a nonlinear function (see Exercise 18.22). As mentioned in the discussion of logistic regression (page 725), the logistic activation function has the added advantage of being differentiable.

FEED-FORWARD
NETWORK

RECURRENT
NETWORK

Having decided on the mathematical model for individual "neurons," the next task is to connect them together to form a network. There are two fundamentally distinct ways to do this. A **feed-forward network** has connections only in one direction—that is, it forms a directed acyclic graph. Every node receives input from "upstream" nodes and delivers output to "downstream" nodes; there are no loops. A feed-forward network represents a function of its current input; thus, it has no internal state other than the weights themselves. A **recurrent network**, on the other hand, feeds its outputs back into its own inputs. This means that the activation levels of the network form a dynamical system that may reach a stable state or exhibit oscillations or even chaotic behavior. Moreover, the response of the network to a given input depends on its initial state, which may depend on previous inputs. Hence, recurrent networks (unlike feed-forward networks) can support short-term memory. This makes them more interesting as models of the brain, but also more difficult to understand. This section will concentrate on feed-forward networks; some pointers for further reading on recurrent networks are given at the end of the chapter.

LAYERS

HIDDEN UNIT

Feed-forward networks are usually arranged in **layers**, such that each unit receives input only from units in the immediately preceding layer. In the next two subsections, we will look at single-layer networks, in which every unit connects directly from the network's inputs to its outputs, and multilayer networks, which have one or more layers of **hidden units** that are not connected to the outputs of the network. So far in this chapter, we have considered only learning problems with a single output variable y, but neural networks are often used in cases where multiple outputs are appropriate. For example, if we want to train a network to add two input bits, each a 0 or a 1, we will need one output for the sum bit and one for the carry bit. Also, when the learning problem involves classification into more than two classes—for example, when learning to categorize images of handwritten digits—it is common to use one output unit for each class.

18.7.2   Single-layer feed-forward neural networks (perceptrons)

PERCEPTRON
NETWORK

A network with all the inputs connected directly to the outputs is called a **single-layer neural network**, or a **perceptron network**. Figure 18.20 shows a simple two-input, two-output perceptron network. With such a network, we might hope to learn the two-bit adder function, for example. Here are all the training data we will need:

x_1	x_2	y_3 (carry)	y_4 (sum)
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

The first thing to notice is that a perceptron network with m outputs is really m separate networks, because each weight affects only one of the outputs. Thus, there will be m separate training processes. Furthermore, depending on the type of activation function used, the training processes will be either the **perceptron learning rule** (Equation (18.7) on page 724) or gradient descent rule for the **logistic regression** (Equation (18.8) on page 727).

If you try either method on the two-bit-adder data, something interesting happens. Unit 3 learns the carry function easily, but unit 4 completely fails to learn the sum function. No, unit 4 is not defective! The problem is with the sum function itself. We saw in Section 18.6 that linear classifiers (whether hard or soft) can represent linear decision boundaries in the input space. This works fine for the carry function, which is a logical AND (see Figure 18.21(a)). The sum function, however, is an XOR (exclusive OR) of the two inputs. As Figure 18.21(c) illustrates, this function is not linearly separable so the perceptron cannot learn it.

The linearly separable functions constitute just a small fraction of all Boolean functions; Exercise 18.20 asks you to quantify this fraction. The inability of perceptrons to learn even such simple functions as XOR was a significant setback to the nascent neural network
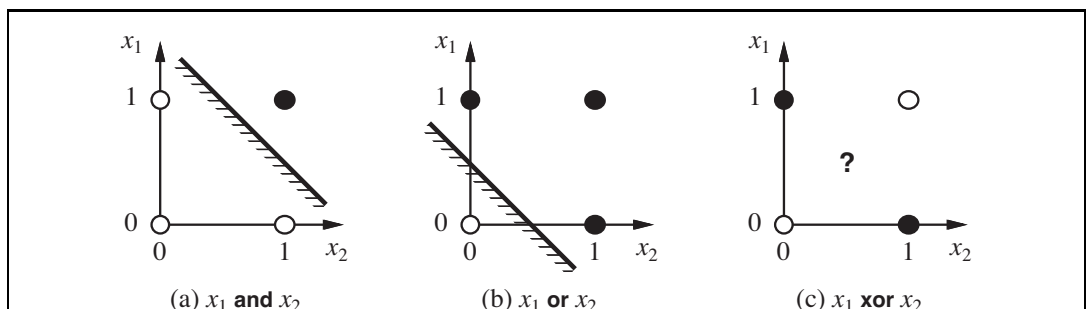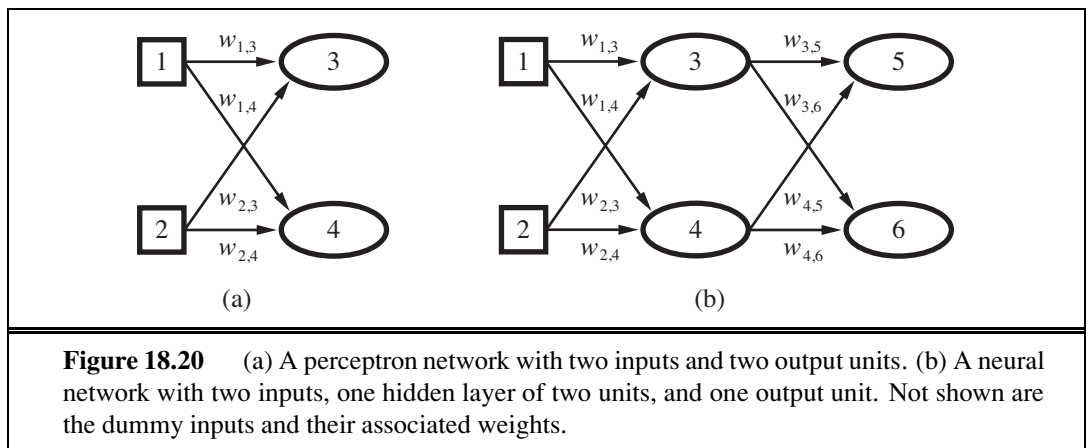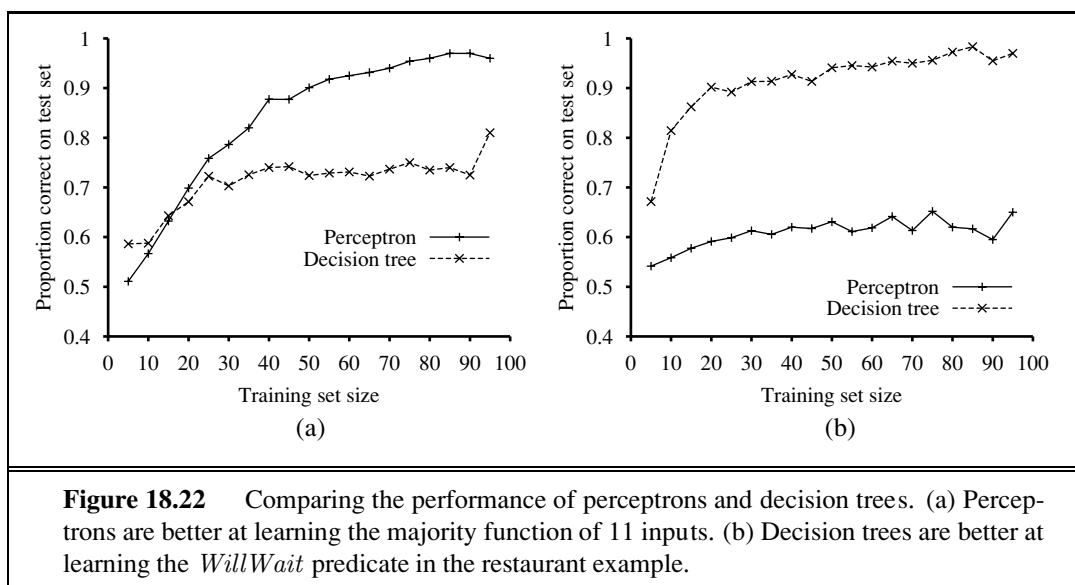

Figure 18.20   (a) A perceptron network with two inputs and two output units. (b) A neural network with two inputs, one hidden layer of two units, and one output unit. Not shown are the dummy inputs and their associated weights.

Figure 18.21   Linear separability in threshold perceptrons. Black dots indicate a point in the input space where the value of the function is 1, and white dots indicate a point where the value is 0. The perceptron returns 1 on the region on the non-shaded side of the line. In (c), no such line exists that correctly classifies the inputs.

Figure 18.22   Comparing the performance of perceptrons and decision trees. (a) Perceptrons are better at learning the majority function of 11 inputs. (b) Decision trees are better at learning the *WillWait* predicate in the restaurant example.
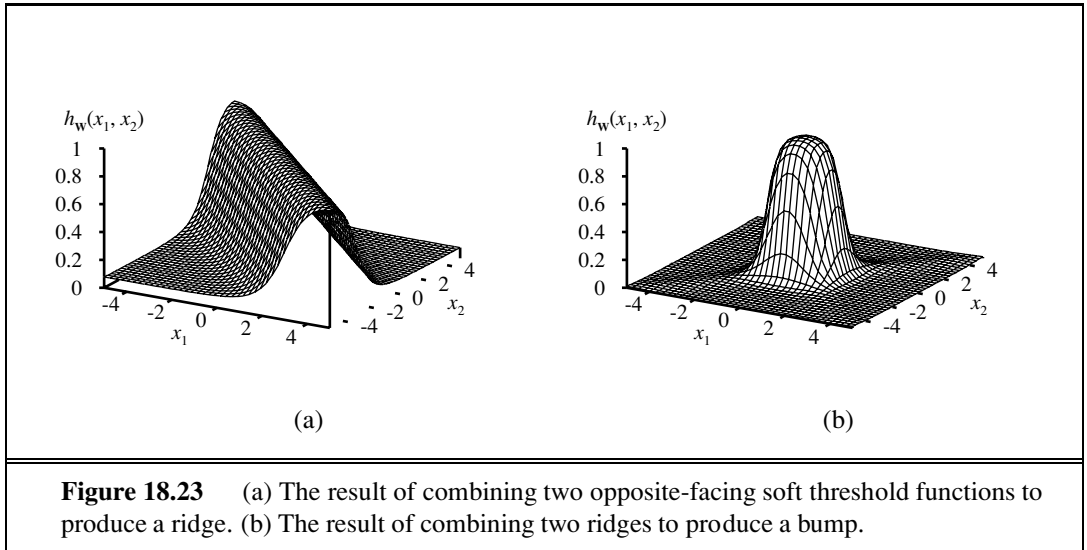
community in the 1960s. Perceptrons are far from useless, however. Section 18.6.4 noted that logistic regression (i.e., training a sigmoid perceptron) is even today a very popular and effective tool. Moreover, a perceptron can represent some quite "complex" Boolean functions very compactly. For example, the **majority function**, which outputs a 1 only if more than half of its n inputs are 1, can be represented by a perceptron with each $w_i = 1$ and with $w_0 = -n/2$. A decision tree would need exponentially many nodes to represent this function.

Figure 18.22 shows the learning curve for a perceptron on two different problems. On the left, we show the curve for learning the majority function with 11 Boolean inputs (i.e., the function outputs a 1 if 6 or more inputs are 1). As we would expect, the perceptron learns the function quite quickly, because the majority function is linearly separable. On the other hand, the decision-tree learner makes no progress, because the majority function is very hard (although not impossible) to represent as a decision tree. On the right, we have the restaurant example. The solution problem is easily represented as a decision tree, but is not linearly separable. The best plane through the data correctly classifies only 65%.

18.7.3   Multilayer feed-forward neural networks

(McCulloch and Pitts, 1943) were well aware that a single threshold unit would not solve all their problems. In fact, their paper proves that such a unit can represent the basic Boolean functions AND, OR, and NOT and then goes on to argue that any desired functionality can be obtained by connecting large numbers of units into (possibly recurrent) networks of arbitrary depth. The problem was that nobody knew how to train such networks.

This turns out to be an easy problem if we think of a network the right way: as a function $h_{\mathbf{w}}(\mathbf{x})$ parameterized by the weights $\mathbf{w}$. Consider the simple network shown in Figure 18.20(b), which has two input units, two hidden units, and two output unit. (In addition, each unit has a dummy input fixed at 1.) Given an input vector $\mathbf{x} = (x_1, x_2)$, the activations

Figure 18.23 (a) The result of combining two opposite-facing soft threshold functions to produce a ridge. (b) The result of combining two ridges to produce a bump.

of the input units are set to $(a_1, a_2) = (x_1, x_2)$. The output at unit 5 is given by

$$
\begin{aligned}
a_5 &= g(w_{0,5,} + w_{3,5}\, a_3 + w_{4,5}\, a_4) \\
&= g(w_{0,5,} + w_{3,5}\, g(w_{0,3} + w_{1,3}\, a_1 + w_{2,3}\, a_2) + w_{4,5}\, g(w_04 + w_{1,4}\, a_1 + w_{2,4}\, a_2)) \\
&= g(w_{0,5,} + w_{3,5}\, g(w_{0,3} + w_{1,3}\, x_1 + w_{2,3}\, x_2) + w_{4,5}\, g(w_04 + w_{1,4}\, x_1 + w_{2,4}\, x_2)).
\end{aligned}
$$

Thus, we have the output expressed as a function of the inputs and the weights. A similar expression holds for unit 6. As long as we can calculate the derivatives of such expressions with respect to the weights, we can use the gradient-descent loss-minimization method to train the network. Section 18.7.4 shows exactly how to do this. And because the function represented by a network can be highly nonlinear—composed, as it is, of nested nonlinear soft threshold functions—we can see neural networks as a tool for doing **nonlinear regression**.

NONLINEAR
REGRESSION

Before delving into learning rules, let us look at the ways in which networks generate complicated functions. First, remember that each unit in a sigmoid network represents a soft threshold in its input space, as shown in Figure 18.17(c) (page 726). With one hidden layer and one output layer, as in Figure 18.20(b), each output unit computes a soft-thresholded linear combination of several such functions. For example, by adding two opposite-facing soft threshold functions and thresholding the result, we can obtain a "ridge" function as shown in Figure 18.23(a). Combining two such ridges at right angles to each other (i.e., combining the outputs from four hidden units), we obtain a "bump" as shown in Figure 18.23(b).

With more hidden units, we can produce more bumps of different sizes in more places. In fact, with a single, sufficiently large hidden layer, it is possible to represent any continuous function of the inputs with arbitrary accuracy; with two layers, even discontinuous functions can be represented.[9] Unfortunately, for any *particular* network structure, it is harder to characterize exactly which functions can be represented and which ones cannot.

---

[9] The proof is complex, but the main point is that the required number of hidden units grows exponentially with the number of inputs. For example, $2^n/n$ hidden units are needed to encode all Boolean functions of n inputs.

18.7.4   Learning in multilayer networks

First, let us dispense with one minor complication arising in multilayer networks: interactions among the learning problems when the network has multiple outputs. In such cases, we should think of the network as implementing a vector function $\mathbf{h_w}$ rather than a scalar function $h_\mathbf{w}$; for example, the network in Figure 18.20(b) returns a vector $[a_5, a_6]$. Similarly, the target output will be a vector $\mathbf{y}$. Whereas a perceptron network decomposes into m separate learning problems for an m-output problem, this decomposition fails in a multilayer network. For example, both a_5 and a_6 in Figure 18.20(b) depend on all of the input-layer weights, so updates to those weights will depend on errors in both a_5 and a_6. Fortunately, this dependency is very simple in the case of any loss function that is *additive* across the components of the error vector $\mathbf{y} - \mathbf{h_w}(\mathbf{x})$. For the L_2 loss, we have, for any weight w,

$$\frac{\partial}{\partial w} Loss(\mathbf{w}) = \frac{\partial}{\partial w}|\mathbf{y} - \mathbf{h_w}(\mathbf{x})|^2 = \frac{\partial}{\partial w}\sum_k (y_k - a_k)^2 = \sum_k \frac{\partial}{\partial w}(y_k - a_k)^2 \quad (18.10)$$

where the index k ranges over nodes in the output layer. Each term in the final summation is just the gradient of the loss for the kth output, computed as if the other outputs did not exist. Hence, we can decompose an m-output learning problem into m learning problems, provided we remember to add up the gradient contributions from each of them when updating the weights.

The major complication comes from the addition of hidden layers to the network. Whereas the error $\mathbf{y} - \mathbf{h_w}$ at the output layer is clear, the error at the hidden layers seems mysterious because the training data do not say what value the hidden nodes should have. Fortunately, it turns out that we can **back-propagate** the error from the output layer to the hidden layers. The back-propagation process emerges directly from a derivation of the overall error gradient. First, we will describe the process with an intuitive justification; then, we will show the derivation.

BACK-PROPAGATION

At the output layer, the weight-update rule is identical to Equation (18.8). We have multiple output units, so let Err_k be the kth component of the error vector $\mathbf{y} - \mathbf{h_w}$. We will also find it convenient to define a modified error $\Delta_k = Err_k \times g'(in_k)$, so that the weight-update rule becomes

$$w_{j,k} \leftarrow w_{j,k} + \alpha \times a_j \times \Delta_k . \quad (18.11)$$

To update the connections between the input units and the hidden units, we need to define a quantity analogous to the error term for output nodes. Here is where we do the error back-propagation. The idea is that hidden node j is "responsible" for some fraction of the error Δ_k in each of the output nodes to which it connects. Thus, the Δ_k values are divided according to the strength of the connection between the hidden node and the output node and are propagated back to provide the Δ_j values for the hidden layer. The propagation rule for the Δ values is the following:

$$\Delta_j = g'(in_j)\sum_k w_{j,k}\Delta_k . \quad (18.12)$$

---

function BACK-PROP-LEARNING(*examples*, *network*) **returns** a neural network
  inputs: *examples*, a set of examples, each with input vector **x** and output vector **y**
        network, a multilayer network with L layers, weights $w_{i,j}$, activation function g
  local variables: Δ, a vector of errors, indexed by network node

  repeat
      for each weight $w_{i,j}$ in *network* **do**
        $w_{i,j} \leftarrow$ a small random number
      for each example $(\mathbf{x}, \mathbf{y})$ **in** *examples* **do**
        /* *Propagate the inputs forward to compute the outputs* */
        for each node i in the input layer **do**
          $a_i \leftarrow x_i$
        for $\ell = 2$ **to** L **do**
          for each node j in layer ℓ **do**
            $in_j \leftarrow \sum_i w_{i,j}\, a_i$
            $a_j \leftarrow g(in_j)$
        /* *Propagate deltas backward from output layer to input layer* */
        for each node j in the output layer **do**
          $\Delta[j] \leftarrow g'(in_j) \times (y_j - a_j)$
        for $\ell = L - 1$ **to** 1 **do**
          for each node i in layer ℓ **do**
            $\Delta[i] \leftarrow g'(in_i) \sum_j w_{i,j}\, \Delta[j]$
        /* *Update every weight in network using deltas* */
        for each weight $w_{i,j}$ in *network* **do**
          $w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta[j]$
  until some stopping criterion is satisfied
  return *network*

Figure 18.24   The back-propagation algorithm for learning in multilayer networks.

Now the weight-update rule for the weights between the inputs and the hidden layer is essentially identical to the update rule for the output layer:

$$w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta_j \;.$$

The back-propagation process can be summarized as follows:

- Compute the Δ values for the output units, using the observed error.
- Starting with output layer, repeat the following for each layer in the network, until the earliest hidden layer is reached:
    - Propagate the Δ values back to the previous layer.
    - Update the weights between the two layers.

The detailed algorithm is shown in Figure 18.24.

    For the mathematically inclined, we will now derive the back-propagation equations from first principles. The derivation is quite similar to the gradient calculation for logistic

regression (leading up to Equation (18.8) on page 727), except that we have to use the chain
rule more than once.

Following Equation (18.10), we compute just the gradient for $Loss_k = (y_k - a_k)^2$ at
the kth output. The gradient of this loss with respect to weights connecting the hidden layer
to the output layer will be zero except for weights $w_{j,k}$ that connect to the kth output unit.
For those weights, we have

$$\frac{\partial Loss_k}{\partial w_{j,k}} = -2(y_k - a_k)\frac{\partial a_k}{\partial w_{j,k}} = -2(y_k - a_k)\frac{\partial g(in_k)}{\partial w_{j,k}}$$

$$= -2(y_k - a_k)g'(in_k)\frac{\partial in_k}{\partial w_{j,k}} = -2(y_k - a_k)g'(in_k)\frac{\partial}{\partial w_{j,k}}\left(\sum_j w_{j,k}a_j\right)$$

$$= -2(y_k - a_k)g'(in_k)a_j = -a_j\Delta_k \; ,$$

with Δ_k defined as before. To obtain the gradient with respect to the $w_{i,j}$ weights connecting
the input layer to the hidden ¡layer, we have to expand out the activations a_j and reapply the
chain rule. We will show the derivation in gory detail because it is interesting to see how the
derivative operator propagates back through the network:

$$\frac{\partial Loss_k}{\partial w_{i,j}} = -2(y_k - a_k)\frac{\partial a_k}{\partial w_{i,j}} = -2(y_k - a_k)\frac{\partial g(in_k)}{\partial w_{i,j}}$$

$$= -2(y_k - a_k)g'(in_k)\frac{\partial in_k}{\partial w_{i,j}} = -2\Delta_k\frac{\partial}{\partial w_{i,j}}\left(\sum_j w_{j,k}a_j\right)$$

$$= -2\Delta_k w_{j,k}\frac{\partial a_j}{\partial w_{i,j}} = -2\Delta_k w_{j,k}\frac{\partial g(in_j)}{\partial w_{i,j}}$$

$$= -2\Delta_k w_{j,k}g'(in_j)\frac{\partial in_j}{\partial w_{i,j}}$$

$$= -2\Delta_k w_{j,k}g'(in_j)\frac{\partial}{\partial w_{i,j}}\left(\sum_i w_{i,j}a_i\right)$$

$$= -2\Delta_k w_{j,k}g'(in_j)a_i = -a_i\Delta_j \; ,$$

where Δ_j is defined as before. Thus, we obtain the update rules obtained earlier from intuitive
considerations. It is also clear that the process can be continued for networks with more than
one hidden layer, which justifies the general algorithm given in Figure 18.24.

Having made it through (or skipped over) all the mathematics, let's see how a single-
hidden-layer network performs on the restaurant problem. First, we need to determine the
structure of the network. We have 10 attributes describing each example, so we will need
10 input units. Should we have one hidden layer or two? How many nodes in each layer?
Should they be fully connected? There is no good theory that will tell us the answer. (See the
next section.) As always, we can use cross-validation: try several different structures and see
which one works best. It turns out that a network with one hidden layer containing four nodes
is about right for this problem. In Figure 18.25, we show two curves. The first is a training
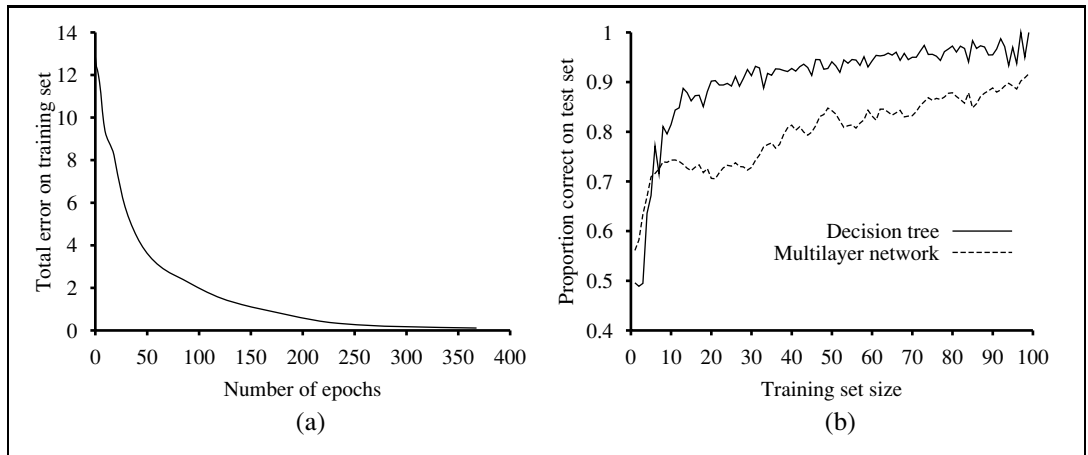curve showing the mean squared error on a given training set of 100 restaurant examples

Figure 18.25   (a) Training curve showing the gradual reduction in error as weights are modified over several epochs, for a given set of examples in the restaurant domain. (b) Comparative learning curves showing that decision-tree learning does slightly better on the restaurant problem than back-propagation in a multilayer network.

during the weight-updating process. This demonstrates that the network does indeed converge to a perfect fit to the training data. The second curve is the standard learning curve for the restaurant data. The neural network does learn well, although not quite as fast as decision-tree learning; this is perhaps not surprising, because the data were generated from a simple decision tree in the first place.

Neural networks are capable of far more complex learning tasks of course, although it must be said that a certain amount of twiddling is needed to get the network structure right and to achieve convergence to something close to the global optimum in weight space. There are literally tens of thousands of published applications of neural networks. Section 18.11.1 looks at one such application in more depth.

18.7.5   Learning neural network structures

So far, we have considered the problem of learning weights, given a fixed network structure; just as with Bayesian networks, we also need to understand how to find the best network structure. If we choose a network that is too big, it will be able to memorize all the examples by forming a large lookup table, but will not necessarily generalize well to inputs that have not been seen before.[10] In other words, like all statistical models, neural networks are subject to **overfitting** when there are too many parameters in the model. We saw this in Figure 18.1 (page 696), where the high-parameter models in (b) and (c) fit all the data, but might not generalize as well as the low-parameter models in (a) and (d).

If we stick to fully connected networks, the only choices to be made concern the number

---

[10] It has been observed that very large networks *do* generalize well *as long as the weights are kept small*. This restriction keeps the activation values in the *linear* region of the sigmoid function $g(x)$ where x is close to zero. This, in turn, means that the network behaves like a linear function (Exercise 18.22) with far fewer parameters.

of hidden layers and their sizes. The usual approach is to try several and keep the best. The **cross-validation** techniques of Chapter 18 are needed if we are to avoid **peeking** at the test set. That is, we choose the network architecture that gives the highest prediction accuracy on the validation sets.

OPTIMAL BRAIN DAMAGE

If we want to consider networks that are not fully connected, then we need to find some effective search method through the very large space of possible connection topologies. The **optimal brain damage** algorithm begins with a fully connected network and removes connections from it. After the network is trained for the first time, an information-theoretic approach identifies an optimal selection of connections that can be dropped. The network is then retrained, and if its performance has not decreased then the process is repeated. In addition to removing connections, it is also possible to remove units that are not contributing much to the result.

TILING

Several algorithms have been proposed for growing a larger network from a smaller one. One, the **tiling** algorithm, resembles decision-list learning. The idea is to start with a single unit that does its best to produce the correct output on as many of the training examples as possible. Subsequent units are added to take care of the examples that the first unit got wrong. The algorithm adds only as many units as are needed to cover all the examples.

18.8   NONPARAMETRIC MODELS

Linear regression and neural networks use the training data to estimate a fixed set of parameters $\mathbf{w}$. That defines our hypothesis $h_{\mathbf{w}}(\mathbf{x})$, and at that point we can throw away the training data, because they are all summarized by $\mathbf{w}$. A learning model that summarizes data with a set of parameters of fixed size (independent of the number of training examples) is called a
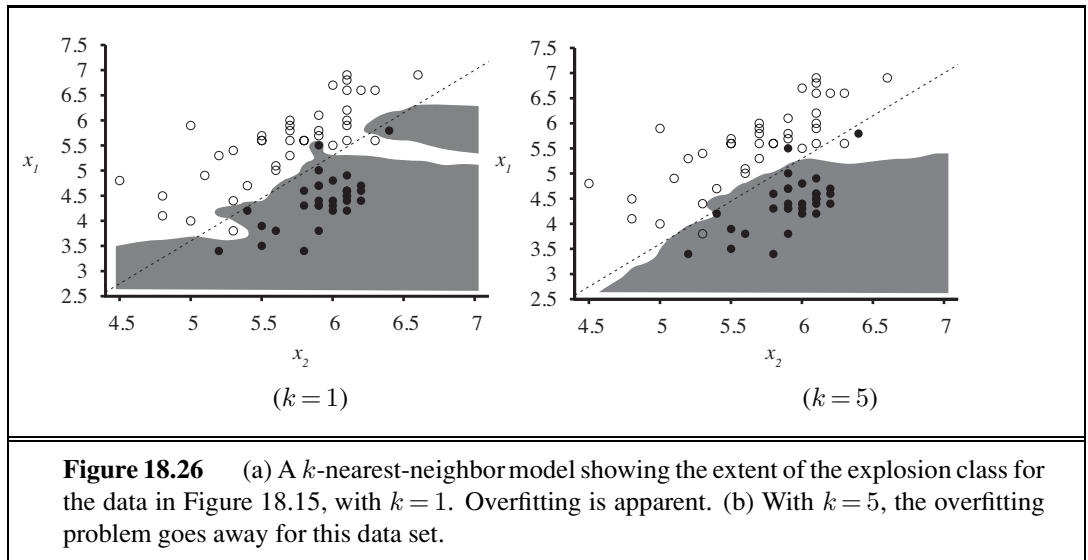
PARAMETRIC MODEL

parametric model.

No matter how much data you throw at a parametric model, it won't change its mind about how many parameters it needs. When data sets are small, it makes sense to have a strong restriction on the allowable hypotheses, to avoid overfitting. But when there are thousands or millions or billions of examples to learn from, it seems like a better idea to let the data speak for themselves rather than forcing them to speak through a tiny vector of parameters. If the data say that the correct answer is a very wiggly function, we shouldn't restrict ourselves to linear or slightly wiggly functions.

NONPARAMETRIC MODEL

A **nonparametric model** is one that cannot be characterized by a bounded set of parameters. For example, suppose that each hypothesis we generate simply retains within itself all of the training examples and uses all of them to predict the next example. Such a hypothesis family would be nonparametric because the effective number of parameters is unbounded—

INSTANCE-BASED LEARNING

it grows with the number of examples. This approach is called **instance-based learning** or

TABLE LOOKUP

memory-based learning. The simplest instance-based learning method is **table lookup**: take all the training examples, put them in a lookup table, and then when asked for $h(\mathbf{x})$, see if $\mathbf{x}$ is in the table; if it is, return the corresponding y. The problem with this method is that it does not generalize well: when $\mathbf{x}$ is not in the table all it can do is return some default value.

Figure 18.26   (a) A k-nearest-neighbor model showing the extent of the explosion class for the data in Figure 18.15, with $k = 1$. Overfitting is apparent. (b) With $k = 5$, the overfitting problem goes away for this data set.

18.8.1   Nearest neighbor models

NEAREST
NEIGHBORS

We can improve on table lookup with a slight variation: given a query $\mathbf{x}_q$, find the k examples that are *nearest* to $\mathbf{x}_q$. This is called k-**nearest neighbors** lookup. We'll use the notation $NN(k, \mathbf{x}_q)$ to denote the set of k nearest neighbors.

To do classification, first find $NN(k, \mathbf{x}_q)$, then take the plurality vote of the neighbors (which is the majority vote in the case of binary classification). To avoid ties, k is always chosen to be an odd number. To do regression, we can take the mean or median of the k neighbors, or we can solve a linear regression problem on the neighbors.

In Figure 18.26, we show the decision boundary of k-nearest-neighbors classification for $k = 1$ and 5 on the earthquake data set from Figure 18.15. Nonparametric methods are still subject to underfitting and overfitting, just like parametric methods. In this case 1-nearest neighbors is overfitting; it reacts too much to the black outlier in the upper right and the white outlier at (5.4, 3.7). The 5-nearest-neighbors decision boundary is good; higher k would underfit. As usual, cross-validation can be used to select the best value of k.

The very word "nearest" implies a distance metric. How do we measure the distance from a query point $\mathbf{x}_q$ to an example point $\mathbf{x}_j$? Typically, distances are measured with a

MINKOWSKI
DISTANCE

Minkowski distance or L^p norm, defined as

$$L^p(\mathbf{x}_j, \mathbf{x}_q) = (\sum_i |x_{j,i} - x_{q,i}|^p)^{1/p} .$$

With $p = 2$ this is Euclidean distance and with $p = 1$ it is Manhattan distance. With Boolean attribute values, the number of attributes on which the two points differ is called the **Ham-**

HAMMING DISTANCE

ming distance. Often $p = 2$ is used if the dimensions are measuring similar properties, such as the width, height and depth of parts on a conveyor belt, and Manhattan distance is used if they are dissimilar, such as age, weight, and gender of a patient. Note that if we use the raw numbers from each dimension then the total distance will be affected by a change in scale in any dimension. That is, if we change dimension i from measurements in centimeters to

miles while keeping the other dimensions the same, we'll get different nearest neighbors. To avoid this, it is common to apply **normalization** to the measurements in each dimension. One simple approach is to compute the mean μ_i and standard deviation σ_i of the values in each dimension, and rescale them so that $x_{j,i}$ becomes $(x_{j,i} - \mu_i)/\sigma_i$. A more complex metric known as the **Mahalanobis distance** takes into account the covariance between dimensions.

MAHALANOBIS
DISTANCE

In low-dimensional spaces with plenty of data, nearest neighbors works very well: we are likely to have enough nearby data points to get a good answer. But as the number of dimensions rises we encounter a problem: the nearest neighbors in high-dimensional spaces are usually not very near! Consider k-nearest-neighbors on a data set of N points uniformly distributed throughout the interior of an n-dimensional unit hypercube. We'll define the k-neighborhood of a point as the smallest hypercube that contains the k-nearest neighbors. Let ℓ be the average side length of a neighborhood. Then the volume of the neighborhood (which contains k points) is ℓ^n and the volume of the full cube (which contains N points) is 1. So, on average, $\ell^n = k/N$. Taking nth roots of both sides we get $\ell = (k/N)^{1/n}$.

To be concrete, let $k = 10$ and $N = 1,000,000$. In two dimensions ($n = 2$; a unit square), the average neighborhood has $\ell = 0.003$, a small fraction of the unit square, and in 3 dimensions ℓ is just 2% of the edge length of the unit cube. But by the time we get to 17 dimensions, ℓ is half the edge length of the unit hypercube, and in 200 dimensions it is 94%. This problem has been called the **curse of dimensionality**.

CURSE OF
DIMENSIONALITY

Another way to look at it: consider the points that fall within a thin shell making up the outer 1% of the unit hypercube. These are outliers; in general it will be hard to find a good value for them because we will be extrapolating rather than interpolating. In one dimension, these outliers are only 2% of the points on the unit line (those points where $x < .01$ or $x > .99$), but in 200 dimensions, over 98% of the points fall within this thin shell—almost all the points are outliers. You can see an example of a poor nearest-neighbors fit on outliers if you look ahead to Figure 18.28(b).

The $NN(k, \mathbf{x}_q)$ function is conceptually trivial: given a set of N examples and a query $\mathbf{x}_q$, iterate through the examples, measure the distance to $\mathbf{x}_q$ from each one, and keep the best k. If we are satisfied with an implementation that takes $O(N)$ execution time, then that is the end of the story. But instance-based methods are designed for large data sets, so we would like an algorithm with sublinear run time. Elementary analysis of algorithms tells us that exact table lookup is $O(N)$ with a sequential table, $O(\log N)$ with a binary tree, and $O(1)$ with a hash table. We will now see that binary trees and hash tables are also applicable for finding nearest neighbors.

18.8.2 Finding nearest neighbors with k-d trees

K-D TREE

A balanced binary tree over data with an arbitrary number of dimensions is called a **k-d tree**, for k-dimensional tree. (In our notation, the number of dimensions is n, so they would be n-d trees. The construction of a k-d tree is similar to the construction of a one-dimensional balanced binary tree. We start with a set of examples and at the root node we split them along the ith dimension by testing whether $x_i \leq m$. We chose the value m to be the median of the examples along the ith dimension; thus half the examples will be in the left branch of the tree
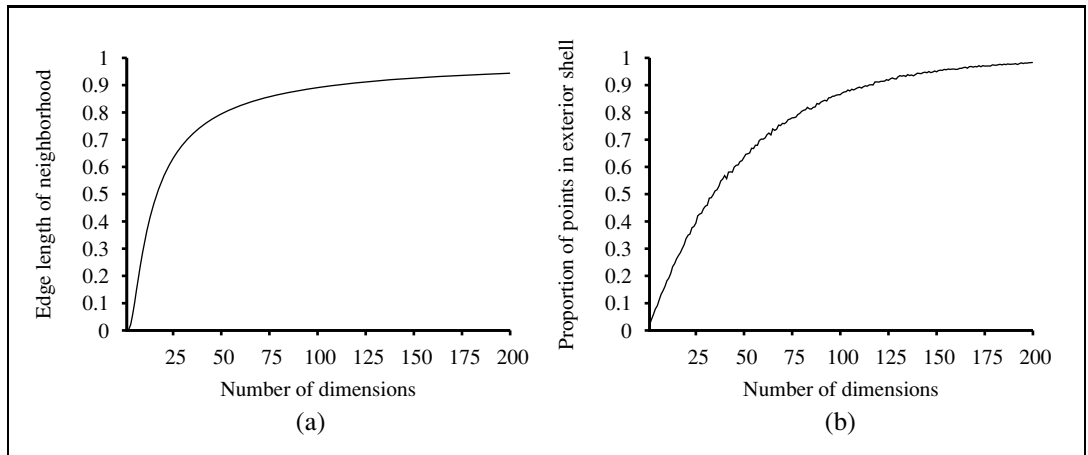
Figure 18.27 The curse of dimensionality: (a) The length of the average neighborhood for 10-nearest-neighbors in a unit hypercube with 1,000,000 points, as a function of the number of dimensions. (b) The proportion of points that fall within a thin shell consisting of the outer 1% of the hypercube, as a function of the number of dimensions. Sampled from 10,000 randomly distributed points.

and half in the right. We then recursively make a tree for the left and right sets of examples, stopping when there are fewer than two examples left. To choose a dimension to split on at each node of the tree, one can simply select dimension $i \bmod n$ at level i of the tree. (Note that we may need to split on any given dimension several times as we proceed down the tree.) Another strategy is to split on the dimension that has the widest spread of values.

Exact lookup from a k-d tree is just like lookup from a binary tree (with the slight complication that you need to pay attention to which dimension you are testing at each node). But nearest neighbor lookup is more complicated. As we go down the branches, splitting the examples in half, in some cases we can discard the other half of the examples. But not always. Sometimes the point we are querying for falls very close to the dividing boundary. The query point itself might be on the left hand side of the boundary, but one or more of the k nearest neighbors might actually be on the right-hand side. We have to test for this possibility by computing the distance of the query point to the dividing boundary, and then searching both sides if we can't find k examples on the left that are closer than this distance. Because of this problem, k-d trees are appropriate only when there are many more examples than dimensions, preferably at least 2^n examples. Thus, k-d trees work well with up to 10 dimensions with thousands of examples or up to 20 dimensions with millions of examples. If we don't have enough examples, lookup is no faster than a linear scan of the entire data set.

18.8.3 Locality-sensitive hashing

Hash tables have the potential to provide even faster lookup than binary trees. But how can we find nearest neighbors using a hash table, when hash codes rely on an *exact* match? Hash codes randomly distribute values among the bins, but we want to have near points grouped LOCALITY-SENSITIVE together in the same bin; we want a **locality-sensitive hash** (LSH).
HASH

We can't use hashes to solve $NN(k, \mathbf{x}_q)$ exactly, but with a clever use of randomized algorithms, we can find an *approximate* solution. First we define the **approximate near-neighbors** problem: given a data set of example points and a query point $\mathbf{x}_q$, find, with high probability, an example point (or points) that is near $\mathbf{x}_q$. To be more precise, we require that if there is a point $\mathbf{x}_j$ that is within a radius r of $\mathbf{x}_q$, then with high probability the algorithm will find a point $\mathbf{x}_{j'}$ that is within distance cr of q. If there is no point within radius r then the algorithm is allowed to report failure. The values of c and "high probability" are parameters of the algorithm.
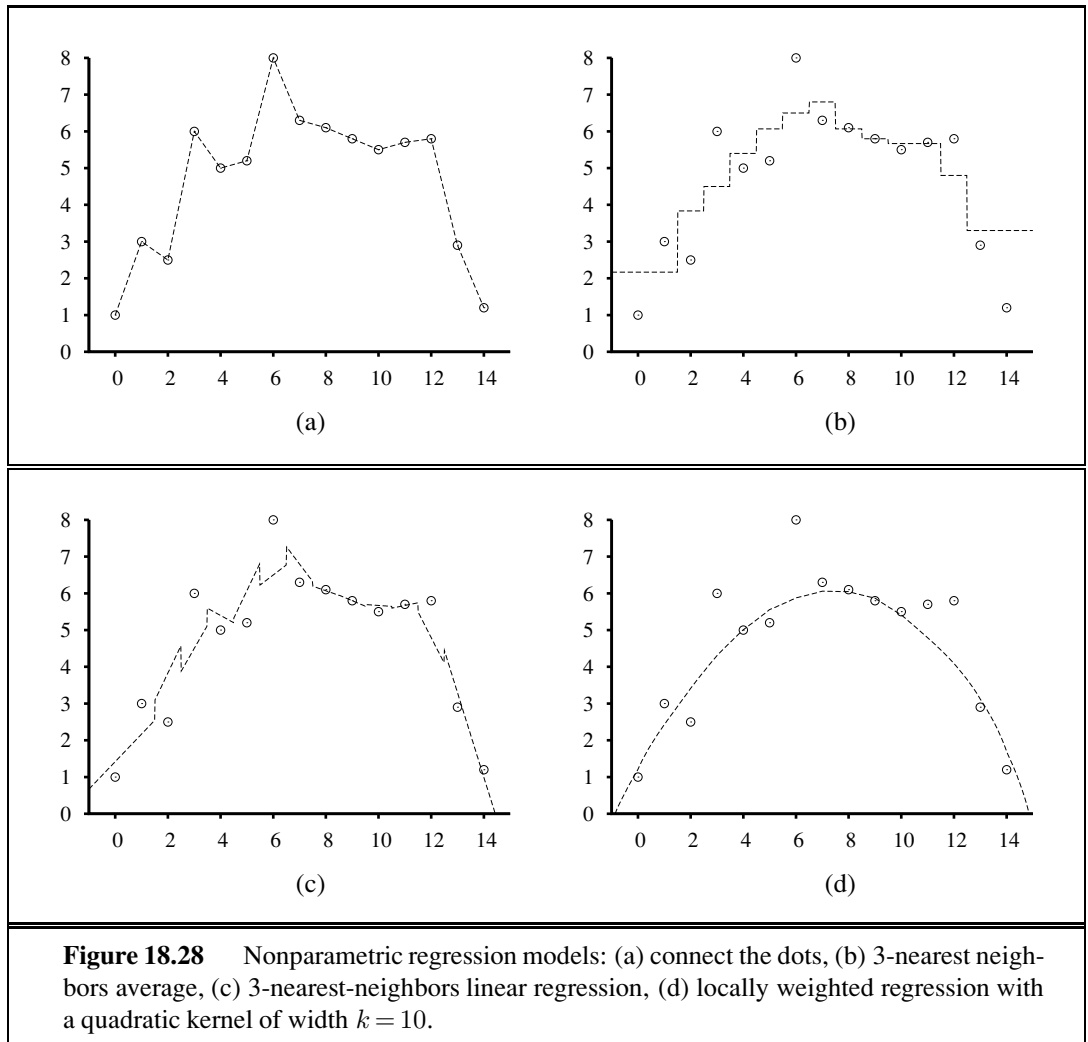
To solve approximate near neighbors, we will need a hash function $g(\mathbf{x})$ that has the property that, for any two points $\mathbf{x}_j$ and $\mathbf{x}_{j'}$, the probability that they have the same hash code is small if their distance is more than cr, and is high if their distance is less than r. For simplicity we will treat each point as a bit string. (Any features that are not Boolean can be encoded into a set of Boolean features.)

The intuition we rely on is that if two points are close together in an n-dimensional space, then they will necessarily be close when projected down onto a one-dimensional space (a line). In fact, we can discretize the line into bins—hash buckets—so that, with high probability, near points project down to exactly the same bin. Points that are far away from each other will tend to project down into different bins for most projections, but there will always be a few projections that coincidentally project far-apart points into the same bin. Thus, the bin for point $\mathbf{x}_q$ contains many (but not all) points that are near to $\mathbf{x}_q$, as well as some points that are far away.

The trick of LSH is to create *multiple* random projections and combine them. A random projection is just a random subset of the bit-string representation. We choose ℓ different random projections and create ℓ hash tables, $g_1(\mathbf{x}), \ldots, g_\ell(\mathbf{x})$. We then enter all the examples into each hash table. Then when given a query point $\mathbf{x}_q$, we fetch the set of points in bin $g_k(q)$ for each k, and union these sets together into a set of candidate points, C. Then we compute the actual distance to $\mathbf{x}_q$ for each of the points in C and return the k closest points. With high probability, each of the points that are near to $\mathbf{x}_q$ will show up in at least one of the bins, and although some far-away points will show up as well, we can ignore those. With large real-world problems, such as finding the near neighbors in a data set of 13 million Web images using 512 dimensions (Torralba *et al.*, 2008), locality-sensitive hashing needs to examine only a few thousand images out of 13 million to find nearest neighbors; a thousand-fold speedup over exhaustive or k-d tree approaches.

18.8.4   Nonparametric regression

Now we'll look at nonparametric approaches to *regression* rather than classification. Figure 18.28 shows an example of some different models. In (a), we have perhaps the simplest method of all, known informally as "connect-the-dots," and superciliously as "piecewise-linear nonparametric regression." This model creates a function $h(x)$ that, when given a query x_q, solves the ordinary linear regression problem with just two points: the training examples immediately to the left and right of x_q. When noise is low, this trivial method is actually not too bad, which is why it is a standard feature of charting software in spreadsheets.
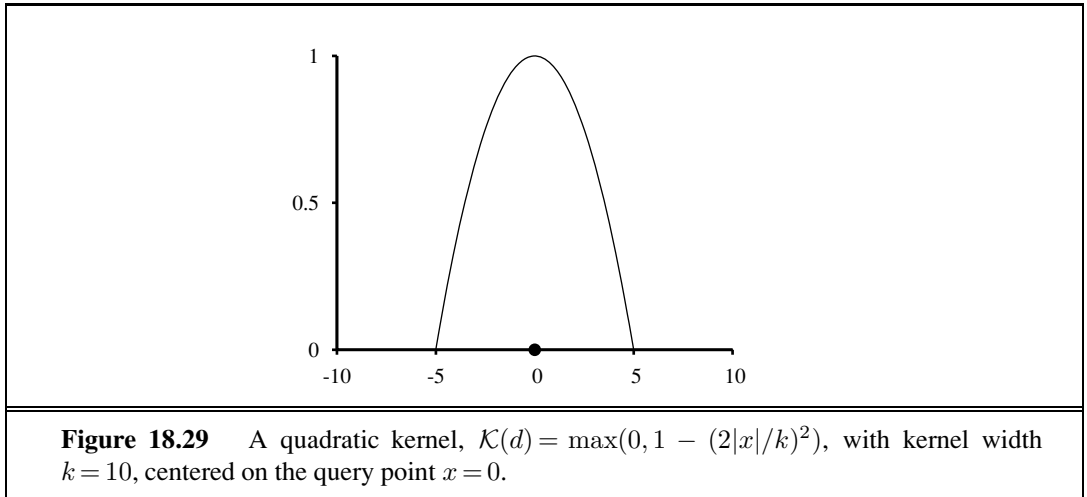
Figure 18.28   Nonparametric regression models: (a) connect the dots, (b) 3-nearest neighbors average, (c) 3-nearest-neighbors linear regression, (d) locally weighted regression with a quadratic kernel of width $k = 10$.

But when the data are noisy, the resulting function is spiky, and does not generalize well.

NEAREST-
NEIGHBORS
REGRESSION

k-**nearest-neighbors regression** (Figure 18.28(b)) improves on connect-the-dots. Instead of using just the two examples to the left and right of a query point x_q, we use the k nearest neighbors (here 3). A larger value of k tends to smooth out the magnitude of the spikes, although the resulting function has discontinuities. In (b), we have the k-nearest-neighbors average: $h(x)$ is the mean value of the k points, $\sum y_j / k$. Notice that at the outlying points, near $x = 0$ and $x = 14$, the estimates are poor because all the evidence comes from one side (the interior), and ignores the trend. In (c), we have k-nearest-neighbor linear regression, which finds the best line through the k examples. This does a better job of capturing trends at the outliers, but is still discontinuous. In both (b) and (c), we're left with the question of how to choose a good value for k. The answer, as usual, is cross-validation.

LOCALLY WEIGHTED
REGRESSION

Locally weighted regression (Figure 18.28(d)) gives us the advantages of nearest neighbors, without the discontinuities. To avoid discontinuities in $h(x)$, we need to avoid disconti-

Figure 18.29   A quadratic kernel, $\mathcal{K}(d) = \max(0, 1 - (2|x|/k)^2)$, with kernel width $k = 10$, centered on the query point $x = 0$.

nuities in the set of examples we use to estimate $h(x)$. The idea of locally weighted regression is that at each query point x_q, the examples that are close to x_q are weighted heavily, and the examples that are farther away are weighted less heavily or not at all. The decrease in weight over distance is always gradual, not sudden.

KERNEL

We decide how much to weight each example with a function known as a **kernel**. A kernel function looks like a bump; in Figure 18.29 we see the specific kernel used to generate Figure 18.28(d). We can see that the weight provided by this kernel is highest in the center and reaches zero at a distance of ± 5. Can we choose just any function for a kernel? No. First, note that we invoke a kernel function $\mathcal{K}$ with $\mathcal{K}(Distance(\mathbf{x}_j, \mathbf{x}_q))$, where $\mathbf{x}_q$ is a query point that is a given distance from $\mathbf{x}_j$, and we want to know how much to weight that distance. So $\mathcal{K}$ should be symmetric around 0 and have a maximum at 0. The area under the kernel must remain bounded as we go to $\pm\infty$. Other shapes, such as Gaussians, have been used for kernels, but the latest research suggests that the choice of shape doesn't matter much. We do have to be careful about the width of the kernel. Again, this is a parameter of the model that is best chosen by cross-validation. Just as in choosing the k for nearest neighbors, if the kernels are too wide we'll get underfitting and if they are too narrow we'll get overfitting. In Figure 18.29(d), the value of $k = 10$ gives a smooth curve that looks about right—but maybe it does not pay enough attention to the outlier at $x = 6$; a narrower kernel width would be more responsive to individual points.

Doing locally weighted regression with kernels is now straightforward. For a given query point $\mathbf{x}_q$ we solve the following weighted regression problem using gradient descent:

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_j \mathcal{K}(Distance(\mathbf{x}_q, \mathbf{x}_j)) \, (y_j - \mathbf{w} \cdot \mathbf{x}_j)^2 \, ,$$

where $Distance$ is any of the distance metrics discussed for nearest neighbors. Then the answer is $h(\mathbf{x}_q) = \mathbf{w}^* \cdot \mathbf{x}_q$.

Note that we need to solve a new regression problem for *every* query point—that's what it means to be *local*. (In ordinary linear regression, we solved the regression problem once, globally, and then used the same $h_{\mathbf{w}}$ for any query point.) Mitigating against this extra work

is the fact that each regression problem will be easier to solve, because it involves only the examples with nonzero weight—the examples whose kernels overlap the query point. When kernel widths are small, this may be just a few points.

Most nonparametric models have the advantage that it is easy to do leave-one-out cross-validation without having to recompute everything. With a k-nearest-neighbors model, for instance, when given a test example $(\mathbf{x}, y)$ we retrieve the k nearest neighbors once, compute the per-example loss $L(y, h(\mathbf{x}))$ from them, and record that as the leave-one-out result for every example that is not one of the neighbors. Then we retrieve the $k + 1$ nearest neighbors and record distinct results for leaving out each of the k neighbors. With N examples the whole process is $O(k)$, not $O(kN)$.

18.9   SUPPORT VECTOR MACHINES

SUPPORT VECTOR
MACHINE
The **support vector machine** or SVM framework is currently the most popular approach for "off-the-shelf" supervised learning: if you don't have any specialized prior knowledge about a domain, then the SVM is an excellent method to try first. There are three properties that make SVMs attractive:

1. SVMs construct a **maximum margin separator**—a decision boundary with the largest possible distance to example points. This helps them generalize well.

2. SVMs create a linear separating hyperplane, but they have the ability to embed the data into a higher-dimensional space, using the so-called **kernel trick**. Often, data that are not linearly separable in the original input space are easily separable in the higher-dimensional space. The high-dimensional linear separator is actually nonlinear in the original space. This means the hypothesis space is greatly expanded over methods that use strictly linear representations.

3. SVMs are a nonparametric method—they retain training examples and potentially need to store them all. On the other hand, in practice they often end up retaining only a small fraction of the number of examples—sometimes as few as a small constant times the number of dimensions. Thus SVMs combine the advantages of nonparametric and parametric models: they have the flexibility to represent complex functions, but they are resistant to overfitting.

You could say that SVMs are successful because of one key insight and one neat trick. We will cover each in turn. In Figure 18.30(a), we have a binary classification problem with three candidate decision boundaries, each a linear separator. Each of them is consistent with all the examples, so from the point of view of 0/1 loss, each would be equally good. Logistic regression would find some separating line; the exact location of the line depends on *all* the example points. The key insight of SVMs is that some examples are more important than others, and that paying attention to them can lead to better generalization.

Consider the lowest of the three separating lines in (a). It comes very close to 5 of the black examples. Although it classifies all the examples correctly, and thus minimizes loss, it
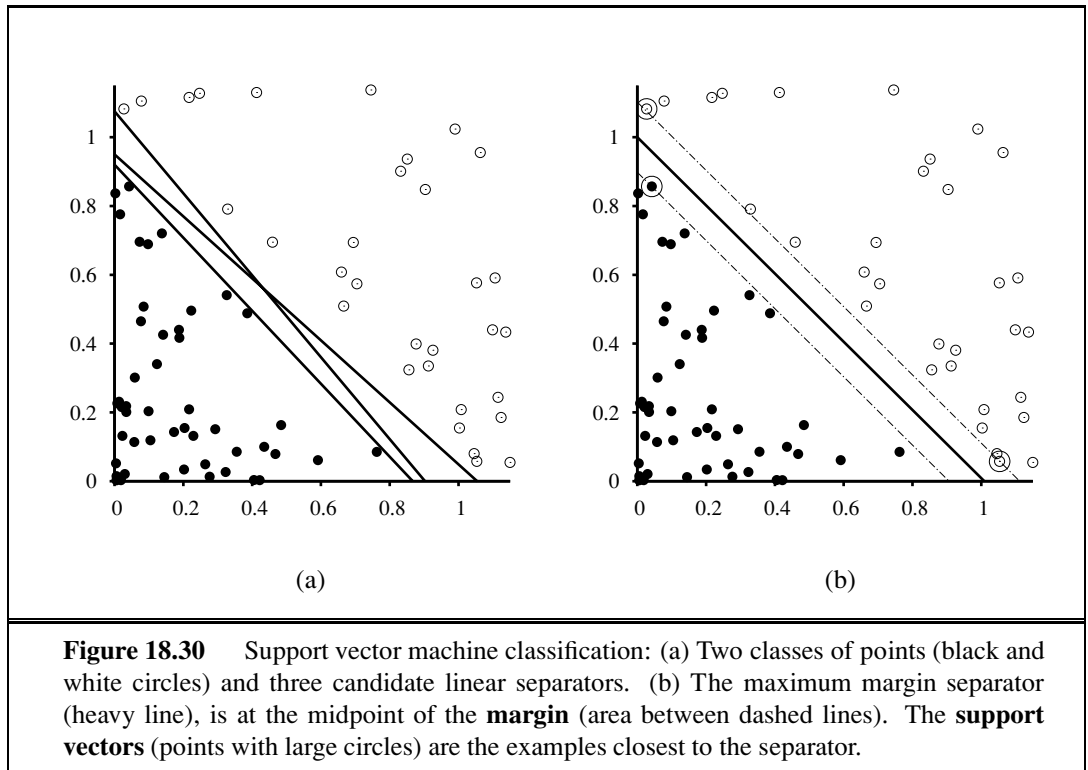
Figure 18.30   Support vector machine classification: (a) Two classes of points (black and white circles) and three candidate linear separators. (b) The maximum margin separator (heavy line), is at the midpoint of the **margin** (area between dashed lines). The **support vectors** (points with large circles) are the examples closest to the separator.

should make you nervous that so many examples are close to the line; it may be that other black examples will turn out to fall on the other side of the line.

SVMs address this issue: Instead of minimizing expected *empirical loss* on the training data, SVMs attempt to minimize expected *generalization* loss. We don't know where the as-yet-unseen points may fall, but under the probabilistic assumption that they are drawn from the same distribution as the previously seen examples, there are some arguments from computational learning theory (Section 18.5) suggesting that we minimize generalization loss by choosing the separator that is farthest away from the examples we have seen so far. We call this separator, shown in Figure 18.30(b) the **maximum margin separator**. The **margin** is the width of the area bounded by dashed lines in the figure—twice the distance from the separator to the nearest example point.

MAXIMUM MARGIN
SEPARATOR

MARGIN

Now, how do we find this separator? Before showing the equations, some notation: Traditionally SVMs use the convention that class labels are +1 and -1, instead of the +1 and 0 we have been using so far. Also, where we put the intercept into the weight vector $\mathbf{w}$ (and a corresponding dummy 1 value into $x_{j,0}$), SVMs do not do that; they keep the intercept as a separate parameter, b. With that in mind, the separator is defined as the set of points $\{\mathbf{x} : \mathbf{w} \cdot \mathbf{x} + b = 0\}$. We could search the space of $\mathbf{w}$ and b with gradient descent to find the parameters that maximize the margin while correctly classifying all the examples.

However, it turns out there is another approach to solving this problem. We won't show the details, but will just say that there is an alternative representation called the dual

representation, in which the optimal solution is found by solving

$$\underset{\alpha}{\operatorname{argmax}} \sum_j \alpha_j - \frac{1}{2} \sum_{j,k} \alpha_j \alpha_k y_j y_k (\mathbf{x}_j \cdot \mathbf{x}_k) \qquad (18.13)$$

QUADRATIC
PROGRAMMING

subject to the constraints $\alpha_j \geq 0$ and $\sum_j \alpha_j y_j = 0$. This is a **quadratic programming**
optimization problem, for which there are good software packages. Once we have found the
vector α we can get back to $\mathbf{w}$ with the equation $\mathbf{w} = \sum_j \alpha_j \mathbf{x}_j$, or we can stay in the dual
representation. There are three important properties of Equation (18.13). First, the expression
is convex; it has a single global maximum that can be found efficiently. Second, *the data enter
the expression only in the form of dot products of pairs of points.* This second property is also
true of the equation for the separator itself; once the optimal α_j have been calculated, it is

$$h(\mathbf{x}) = \operatorname{sign} \left(\sum_j \alpha_j y_j (\mathbf{x} \cdot \mathbf{x}_j) - b \right) . \qquad (18.14)$$

SUPPORT VECTOR

A final important property is that the weights α_j associated with each data point are *zero* ex-
cept for the **support vectors**—the points closest to the separator. (They are called "support"
vectors because they "hold up" the separating plane.) Because there are usually many fewer
support vectors than examples, SVMs gain some of the advantages of parametric models.

What if the examples are not linearly separable? Figure 18.31(a) shows an input space
defined by attributes $\mathbf{x} = (x_1, x_2)$, with positive examples ($y = +1$) inside a circular region
and negative examples ($y = -1$) outside. Clearly, there is no linear separator for this problem.
Now, suppose we re-express the input data—i.e., we map each input vector $\mathbf{x}$ to a new vector
of feature values, $F(\mathbf{x})$. In particular, let us use the three features

$$f_1 = x_1^2 , \qquad f_2 = x_2^2 , \qquad f_3 = \sqrt{2} x_1 x_2 . \qquad (18.15)$$

We will see shortly where these came from, but for now, just look at what happens. Fig-
ure 18.31(b) shows the data in the new, three-dimensional space defined by the three features;
the data are *linearly separable* in this space! This phenomenon is actually fairly general: if
data are mapped into a space of sufficiently high dimension, then they will almost always be
linearly separable—if you look at a set of points from enough directions, you'll find a way to
make them line up. Here, we used only three dimensions;[11] Exercise 18.16 asks you to show
that four dimensions suffice for linearly separating a circle anywhere in the plane (not just at
the origin), and five dimensions suffice to linearly separate any ellipse. In general (with some
special cases excepted) if we have N data points then they will always be separable in spaces
of $N - 1$ dimensions or more (Exercise 18.25).

Now, we would not usually expect to find a linear separator in the input space $\mathbf{x}$, but
we can find linear separators in the high-dimensional feature space $F(\mathbf{x})$ simply by replacing
$\mathbf{x}_j \cdot \mathbf{x}_k$ in Equation (18.13) with $F(\mathbf{x}_j) \cdot F(\mathbf{x}_k)$. This by itself is not remarkable—replacing $\mathbf{x}$ by
$F(\mathbf{x})$ in *any* learning algorithm has the required effect—but the dot product has some special
properties. It turns out that $F(\mathbf{x}_j) \cdot F(\mathbf{x}_k)$ can often be computed without first computing F

---

[11] The reader may notice that we could have used just f_1 and f_2, but the 3D mapping illustrates the idea better.
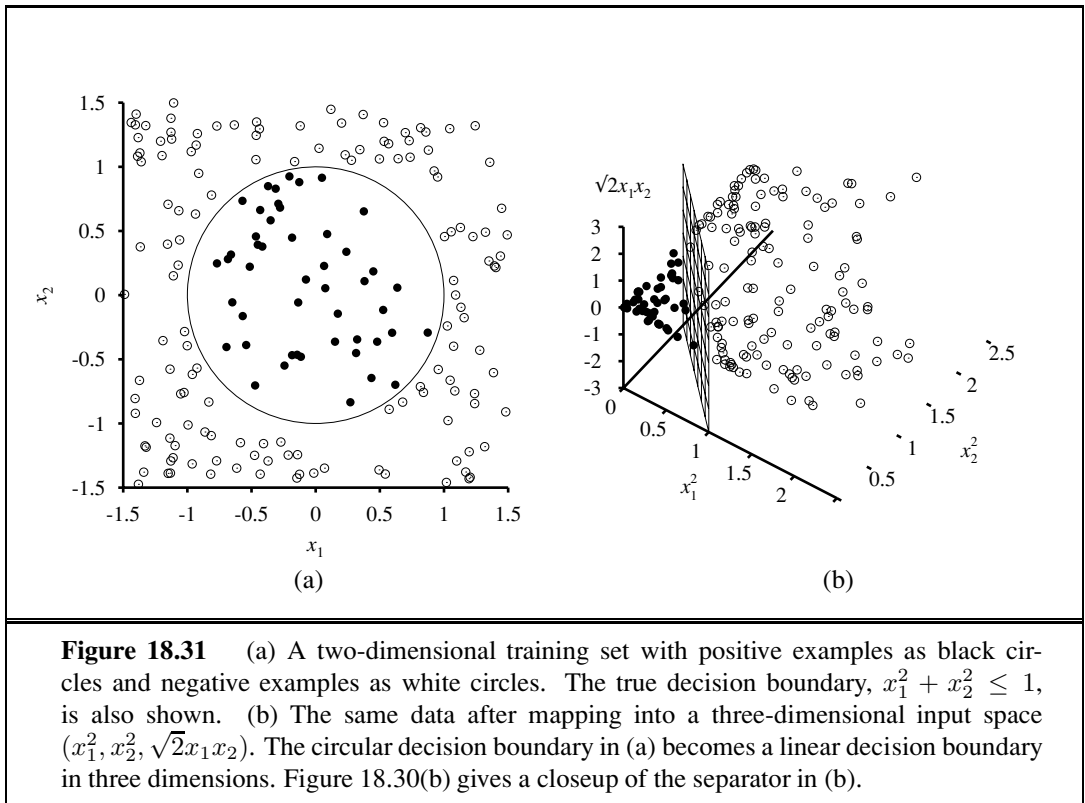
Figure 18.31   (a) A two-dimensional training set with positive examples as black circles and negative examples as white circles. The true decision boundary, $x_1^2 + x_2^2 \leq 1$, is also shown. (b) The same data after mapping into a three-dimensional input space $(x_1^2, x_2^2, \sqrt{2}x_1x_2)$. The circular decision boundary in (a) becomes a linear decision boundary in three dimensions. Figure 18.30(b) gives a closeup of the separator in (b).

for each point. In our three-dimensional feature space defined by Equation (18.15), a little bit of algebra shows that

$$F(\mathbf{x}_j) \cdot F(\mathbf{x}_k) = (\mathbf{x}_j \cdot \mathbf{x}_k)^2 \ .$$

KERNEL FUNCTION   (That's why the $\sqrt{2}$ is in f_3.) The expression $(\mathbf{x}_j \cdot \mathbf{x}_k)^2$ is called a **kernel function**,[12] and is usually written as $K(\mathbf{x}_j, \mathbf{x}_k)$. The kernel function can be applied to pairs of input data to evaluate dot products in some corresponding feature space. So, we can find linear separators in the higher-dimensional feature space $F(\mathbf{x})$ simply by replacing $\mathbf{x}_j \cdot \mathbf{x}_k$ in Equation (18.13) with a kernel function $K(\mathbf{x}_j, \mathbf{x}_k)$. Thus, we can learn in the higher-dimensional space, but we compute only kernel functions rather than the full list of features for each data point.

The next step is to see that there's nothing special about the kernel $K(\mathbf{x}_j, \mathbf{x}_k) = (\mathbf{x}_j \cdot \mathbf{x}_k)^2$. It corresponds to a particular higher-dimensional feature space, but other kernel functions correspond to other feature spaces. A venerable result in mathematics, **Mercer's theo-**
MERCER'S THEOREM   **rem** (1909), tells us that any "reasonable"[13] kernel function corresponds to *some* feature space. These feature spaces can be very large, even for innocuous-looking kernels. For ex-
POLYNOMIAL KERNEL   ample, the **polynomial kernel**, $K(\mathbf{x}_j, \mathbf{x}_k) = (1 + \mathbf{x}_j \cdot \mathbf{x}_k)^d$, corresponds to a feature space whose dimension is exponential in d.

---

[12] This usage of "kernel function" is slightly different from the kernels in locally weighted regression. Some SVM kernels are distance metrics, but not all are.
[13] Here, "reasonable" means that the matrix $\mathbf{K}_{jk} = K(\mathbf{x}_j, \mathbf{x}_k)$ is positive definite.

KERNEL TRICK

            This then is the clever **kernel trick**: Plugging these kernels into Equation (18.13),
*optimal linear separators can be found efficiently in feature spaces with billions of (or, in
some cases, infinitely many) dimensions.* The resulting linear separators, when mapped back
to the original input space, can correspond to arbitrarily wiggly, nonlinear decision bound-
aries between the positive and negative examples.

            In the case of inherently noisy data, we may not want a linear separator in some high-
dimensional space. Rather, we'd like a decision surface in a lower-dimensional space that
does not cleanly separate the classes, but reflects the reality of the noisy data. That is pos-

SOFT MARGIN

sible with the **soft margin** classifier, which allows examples to fall on the wrong side of the
decision boundary, but assigns them a penalty proportional to the distance required to move
them back on the correct side.

            The kernel method can be applied not only with learning algorithms that find optimal
linear separators, but also with any other algorithm that can be reformulated to work only
with dot products of pairs of data points, as in Equations 18.13 and 18.14. Once this is

KERNELIZATION

done, the dot product is replaced by a kernel function and we have a **kernelized** version
of the algorithm. This can be done easily for k-nearest-neighbors and perceptron learning
(Section 18.7.2), among others.

18.10   ENSEMBLE LEARNING

ENSEMBLE
LEARNING

So far we have looked at learning methods in which a single hypothesis, chosen from a
hypothesis space, is used to make predictions. The idea of **ensemble learning** methods is
to select a collection, or **ensemble**, of hypotheses from the hypothesis space and combine
their predictions. For example, during cross-validation we might generate twenty different
decision trees, and have them vote on the best classification for a new example.

            The motivation for ensemble learning is simple. Consider an ensemble of $K = 5$ hy-
potheses and suppose that we combine their predictions using simple majority voting. For the
ensemble to misclassify a new example, *at least three of the five hypotheses have to misclas-
sify it*. The hope is that this is much less likely than a misclassification by a single hypothesis.
Suppose we assume that each hypothesis h_k in the ensemble has an error of p—that is, the
probability that a randomly chosen example is misclassified by h_k is p. Furthermore, suppose
we assume that the errors made by each hypothesis are *independent*. In that case, if p is small,
then the probability of a large number of misclassifications occurring is minuscule. For ex-
ample, a simple calculation (Exercise 18.18) shows that using an ensemble of five hypotheses
reduces an error rate of 1 in 10 down to an error rate of less than 1 in 100. Now, obviously
the assumption of independence is unreasonable, because hypotheses are likely to be misled
in the same way by any misleading aspects of the training data. But if the hypotheses are at
least a little bit different, thereby reducing the correlation between their errors, then ensemble
learning can be very useful.

            Another way to think about the ensemble idea is as a generic way of enlarging the
hypothesis space. That is, think of the ensemble itself as a hypothesis and the new hypothesis
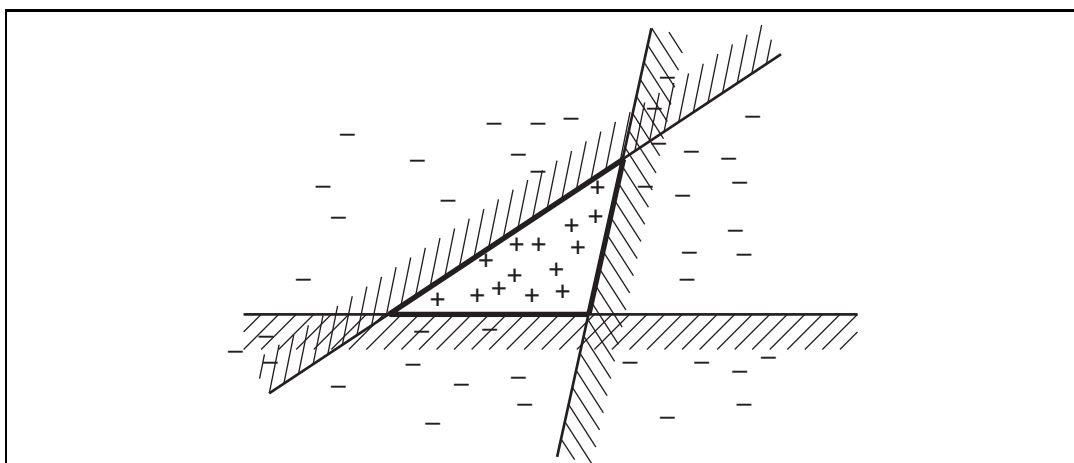
Figure 18.32   Illustration of the increased expressive power obtained by ensemble learn-
ing. We take three linear threshold hypotheses, each of which classifies positively on the
unshaded side, and classify as positive any example classified positively by all three. The
resulting triangular region is a hypothesis not expressible in the original hypothesis space.

space as the set of all possible ensembles constructable from hypotheses in the original space.
Figure 18.32 shows how this can result in a more expressive hypothesis space. If the original
hypothesis space allows for a simple and efficient learning algorithm, then the ensemble
method provides a way to learn a much more expressive class of hypotheses without incurring
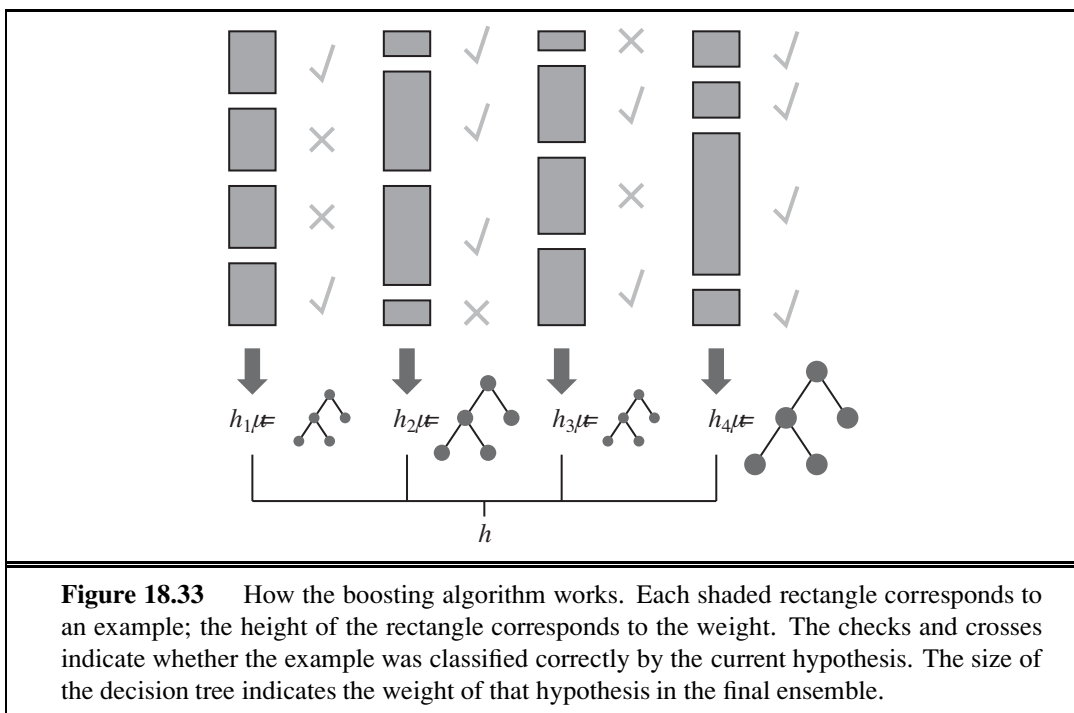much additional computational or algorithmic complexity.

BOOSTING

WEIGHTED TRAINING
SET

       The most widely used ensemble method is called **boosting**. To understand how it works,
we need first to explain the idea of a **weighted training set**. In such a training set, each
example has an associated weight $w_j \geq 0$. The higher the weight of an example, the higher
is the importance attached to it during the learning of a hypothesis. It is straightforward to
modify the learning algorithms we have seen so far to operate with weighted training sets.[14]

       Boosting starts with $w_j = 1$ for all the examples (i.e., a normal training set). From this
set, it generates the first hypothesis, h_1. This hypothesis will classify some of the training ex-
amples correctly and some incorrectly. We would like the next hypothesis to do better on the
misclassified examples, so we increase their weights while decreasing the weights of the cor-
rectly classified examples. From this new weighted training set, we generate hypothesis h_2.
The process continues in this way until we have generated K hypotheses, where K is an input
to the boosting algorithm. The final ensemble hypothesis is a weighted-majority combination
of all the K hypotheses, each weighted according to how well it performed on the training set.
Figure 18.33 shows how the algorithm works conceptually. There are many variants of the ba-
sic boosting idea, with different ways of adjusting the weights and combining the hypotheses.
One specific algorithm, called ADABOOST, is shown in Figure 18.34. ADABOOST has a very

WEAK LEARNING

important property: if the input learning algorithm L is a **weak learning** algorithm—which

---

[14] For learning algorithms in which this is not possible, one can instead create a **replicated training set** where
the jth example appears w_j times, using randomization to handle fractional weights.

Figure 18.33 How the boosting algorithm works. Each shaded rectangle corresponds to an example; the height of the rectangle corresponds to the weight. The checks and crosses indicate whether the example was classified correctly by the current hypothesis. The size of the decision tree indicates the weight of that hypothesis in the final ensemble.

means that L always returns a hypothesis with accuracy on the training set that is slightly better than random guessing (i.e., $50\%+\epsilon$ for Boolean classification)—then ADABOOST will return a hypothesis that *classifies the training data perfectly* for large enough K. Thus, the algorithm *boosts* the accuracy of the original learning algorithm on the training data. This result holds no matter how inexpressive the original hypothesis space and no matter how complex the function being learned.

DECISION STUMP

Let us see how well boosting does on the restaurant data. We will choose as our original hypothesis space the class of **decision stumps**, which are decision trees with just one test, at the root. The lower curve in Figure 18.35(a) shows that unboosted decision stumps are not very effective for this data set, reaching a prediction performance of only 81% on 100 training examples. When boosting is applied (with $K=5$), the performance is better, reaching 93% after 100 examples.

An interesting thing happens as the ensemble size K increases. Figure 18.35(b) shows the training set performance (on 100 examples) as a function of K. Notice that the error reaches zero when K is 20; that is, a weighted-majority combination of 20 decision stumps suffices to fit the 100 examples exactly. As more stumps are added to the ensemble, the error remains at zero. The graph also shows that *the test set performance continues to increase long after the training set error has reached zero.* At $K = 20$, the test performance is 0.95 (or 0.05 error), and the performance increases to 0.98 as late as $K = 137$, before gradually dropping to 0.95.

This finding, which is quite robust across data sets and hypothesis spaces, came as quite a surprise when it was first noticed. Ockham's razor tells us not to make hypotheses more

```
function ADABOOST(examples, L, K) returns a weighted-majority hypothesis
   inputs: examples, set of N labeled examples (x_1, y_1), ..., (x_N, y_N)
           L, a learning algorithm
           K, the number of hypotheses in the ensemble
   local variables: w, a vector of N example weights, initially 1/N
                    h, a vector of K hypotheses
                    z, a vector of K hypothesis weights

   for k = 1 to K do
       h[k] ← L(examples, w)
       error ← 0
       for j = 1 to N do
           if h[k](x_j) ≠ y_j then error ← error + w[j]
       for j = 1 to N do
           if h[k](x_j) = y_j then w[j] ← w[j] · error/(1 − error)
       w ← NORMALIZE(w)
       z[k] ← log (1 − error)/error
   return WEIGHTED-MAJORITY(h, z)
```

Figure 18.34   The ADABOOST variant of the boosting method for ensemble learning. The algorithm generates hypotheses by successively reweighting the training examples. The function WEIGHTED-MAJORITY generates a hypothesis that returns the output value with the highest vote from the hypotheses in **h**, with votes weighted by **z**.
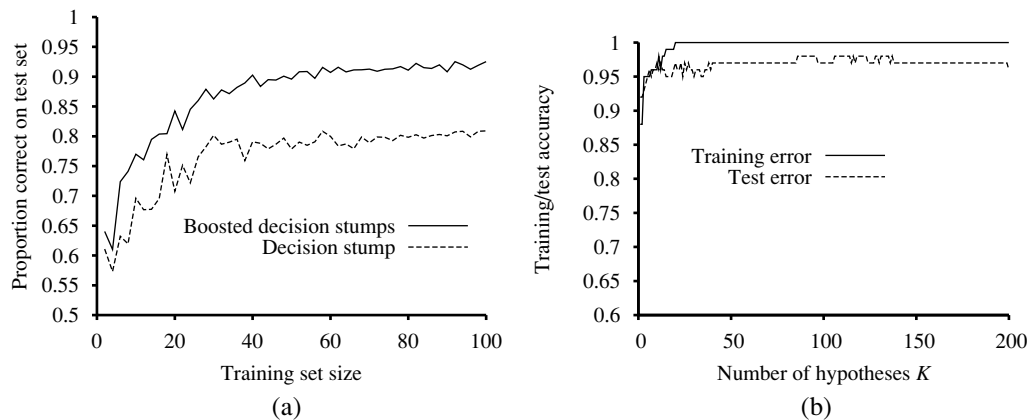

Figure 18.35   (a) Graph showing the performance of boosted decision stumps with $K = 5$ versus unboosted decision stumps on the restaurant data. (b) The proportion correct on the training set and the test set as a function of K, the number of hypotheses in the ensemble. Notice that the test set accuracy improves slightly even after the training accuracy reaches 1, i.e., after the ensemble fits the data exactly.

complex than necessary, but the graph tells us that the predictions *improve* as the ensemble hypothesis gets more complex! Various explanations have been proposed for this. One view is that boosting approximates **Bayesian learning** (see Chapter 20), which can be shown to be an optimal learning algorithm, and the approximation improves as more hypotheses are added. Another possible explanation is that the addition of further hypotheses enables the ensemble to be *more definite* in its distinction between positive and negative examples, which helps it when it comes to classifying new examples.

18.10.1   Online Learning

So far, everything we have done in this chapter has relied on the assumption that the data are i.i.d. (independent and identically distributed). On the one hand, that is a sensible assumption: if the future bears no resemblance to the past, then how can we predict anything? On the other hand, it is too strong an assumption: it is rare that our inputs have captured all the information that would make the future truly independent of the past.

ONLINE LEARNING

In this section we examine what to do when the data are not i.i.d.; when they can change over time. In this case, it matters *when* we make a prediction, so we will adopt the perspective called **online learning**: an agent receives an input x_j from nature, predicts the corresponding y_j, and then is told the correct answer. Then the process repeats with x_{j+1}, and so on. One might think this task is hopeless—if nature is adversarial, all the predictions may be wrong. It turns out that there are some guarantees we can make.

RANDOMIZED
WEIGHTED
MAJORITY
ALGORITHM

Let us consider the situation where our input consists of predictions from a panel of experts. For example, each day a set of K pundits predicts whether the stock market will go up or down, and our task is to pool those predictions and make our own. One way to do this is to keep track of how well each expert performs, and choose to believe them in proportion to their past performance. This is called the **randomized weighted majority algorithm**. We can described it more formally:

1. Initialize a set of weights $\{w_1, \ldots, w_K\}$ all to 1.
2. Receive the predictions $\{\hat{y}_1, \ldots, \hat{y}_K\}$ from the experts.
3. Randomly choose an expert k^*, in proportion to its weight: $P(k) = w_k/(\sum_{k'} w_{k'})$.
4. Predict $\hat{y}_{k^*}$.
5. Receive the correct answer y.
6. For each expert k such that $\hat{y}_k \neq y$, update $w_k \leftarrow \beta w_k$

Here β is a number, $0 < \beta < 1$, that tells how much to penalize an expert for each mistake.

REGRET

We measure the success of this algorithm in terms of **regret**, which is defined as the number of additional mistakes we make compared to the expert who, in hindsight, had the best prediction record. Let M^* be the number of mistakes made by the best expert. Then the number of mistakes, M, made by the random weighted majority algorithm, is bounded by[15]

$$M < \frac{M^* \ln(1/\beta) + \ln K}{1 - \beta} .$$

---

[15] See (Blum, 1996) for the proof.

This bound holds for *any* sequence of examples, even ones chosen by adversaries trying to do their worst. To be specific, when there are $K = 10$ experts, if we choose $\beta = 1/2$ then our number of mistakes is bounded by $1.39M^* + 4.6$, and if $\beta = 3/4$ by $1.15M^* + 9.2$. In general, if β is close to 1 then we are responsive to change over the long run; if the best expert changes, we will pick up on it before too long. However, we pay a penalty at the beginning, when we start with all experts trusted equally; we may accept the advice of the bad experts for too long. When β is closer to 0, these two factors are reversed. Note that we can choose β to get asymptotically close to M^* in the long run; this is called **no-regret learning** (because the average amount of regret per trial tends to 0 as the number of trials increases).

NO-REGRET
LEARNING

    Online learning is helpful when the data may be changing rapidly over time. It is also useful for applications that involve a large collection of data that is constantly growing, even if changes are gradual. For example, with a database of millions of Web images, you wouldn't want to train, say, a linear regression model on all the data, and then retrain from scratch every time a new image is added. It would be more practical to have an online algorithm that allows images to be added incrementally. For most learning algorithms based on minimizing loss, there is an online version based on minimizing regret. It is a bonus that many of these online algorithms come with guaranteed bounds on regret.
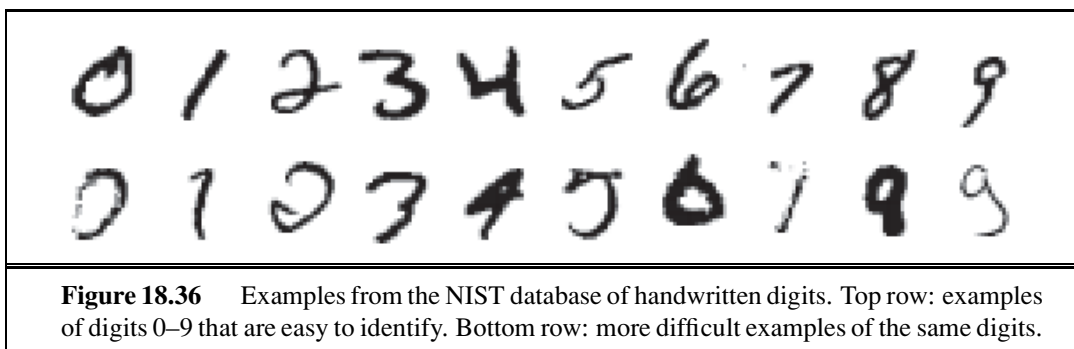
    To some observers, it is surprising that there are such tight bounds on how well we can do compared to a panel of experts. To others, the really surprising thing is that when panels of human experts congregate—predicting stock market prices, sports outcomes, or political contests—the viewing public is so willing to listen to them pontificate and so unwilling to quantify their error rates.

18.11   PRACTICAL MACHINE LEARNING

We have introduced a wide range of machine learning techniques, each illustrated with simple learning tasks. In this section, we consider two aspects of practical machine learning. The first involves finding algorithms capable of learning to recognize handwritten digits and squeezing every last drop of predictive performance out of them. The second involves anything but—pointing out that obtaining, cleaning, and representing the data can be at least as important as algorithm engineering.

18.11.1   Case study: Handwritten digit recognition

Recognizing handwritten digits is an important problem with many applications, including automated sorting of mail by postal code, automated reading of checks and tax returns, and data entry for hand-held computers. It is an area where rapid progress has been made, in part because of better learning algorithms and in part because of the availability of better training sets. The United States National Institute of Science and Technology (**NIST**) has archived a database of 60,000 labeled digits, each $20 \times 20 = 400$ pixels with 8-bit grayscale values. It has become one of the standard benchmark problems for comparing new learning algorithms. Some example digits are shown in Figure 18.36.

Figure 18.36   Examples from the NIST database of handwritten digits. Top row: examples of digits 0–9 that are easy to identify. Bottom row: more difficult examples of the same digits.

Many different learning approaches have been tried. One of the first, and probably the simplest, is the **3-nearest-neighbor** classifier, which also has the advantage of requiring no training time. As a memory-based algorithm, however, it must store all 60,000 images, and its run time performance is slow. It achieved a test error rate of 2.4%.

A **single-hidden-layer neural network** was designed for this problem with 400 input units (one per pixel) and 10 output units (one per class). Using cross-validation, it was found that roughly 300 hidden units gave the best performance. With full interconnections between layers, there were a total of 123,300 weights. This network achieved a 1.6% error rate.

A series of **specialized neural networks** called LeNet were devised to take advantage of the structure of the problem—that the input consists of pixels in a two–dimensional array, and that small changes in the position or slant of an image are unimportant. Each network had an input layer of 32×32 units, onto which the 20×20 pixels were centered so that each input unit is presented with a local neighborhood of pixels. This was followed by three layers of hidden units. Each layer consisted of several planes of $n \times n$ arrays, where n is smaller than the previous layer so that the network is down-sampling the input, and where the weights of every unit in a plane are constrained to be identical, so that the plane is acting as a feature detector: it can pick out a feature such as a long vertical line or a short semi-circular arc. The output layer had 10 units. Many versions of this architecture were tried; a representative one had hidden layers with 768, 192, and 30 units, respectively. The training set was augmented by applying affine transformations to the actual inputs: shifting, slightly rotating, and scaling the images. (Of course, the transformations have to be small, or else a 6 will be transformed into a 9!) The best error rate achieved by LeNet was 0.9%.

A **boosted neural network** combined three copies of the LeNet architecture, with the second one trained on a mix of patterns that the first one got 50% wrong, and the third one trained on patterns for which the first two disagreed. During testing, the three nets voted with the majority ruling. The test error rate was 0.7%.

A **support vector machine** (see Section 18.9) with 25,000 support vectors achieved an error rate of 1.1%. This is remarkable because the SVM technique, like the simple nearest-neighbor approach, required almost no thought or iterated experimentation on the part of the developer, yet it still came close to the performance of LeNet, which had had years of development. Indeed, the support vector machine makes no use of the structure of the problem, and would perform just as well if the pixels were presented in a permuted order.

A **virtual support vector machine** starts with a regular SVM and then improves it with a technique that is designed to take advantage of the structure of the problem. Instead of allowing products of all pixel pairs, this approach concentrates on kernels formed from pairs of nearby pixels. It also augments the training set with transformations of the examples, just as LeNet did. A virtual SVM achieved the best error rate recorded to date, 0.56%.

Shape matching is a technique from computer vision used to align corresponding parts of two different images of objects (Belongie *et al.*, 2002). The idea is to pick out a set of points in each of the two images, and then compute, for each point in the first image, which point in the second image it corresponds to. From this alignment, we then compute a transformation between the images. The transformation gives us a measure of the distance between the images. This distance measure is better motivated than just counting the number of differing pixels, and it turns out that a 3–nearest neighbor algorithm using this distance measure performs very well. Training on only 20,000 of the 60,000 digits, and using 100 sample points per image extracted from a Canny edge detector, a shape matching classifier achieved 0.63% test error.

Humans are estimated to have an error rate of about 0.2% on this problem. This figure is somewhat suspect because humans have not been tested as extensively as have machine learning algorithms. On a similar data set of digits from the United States Postal Service, human errors were at 2.5%.
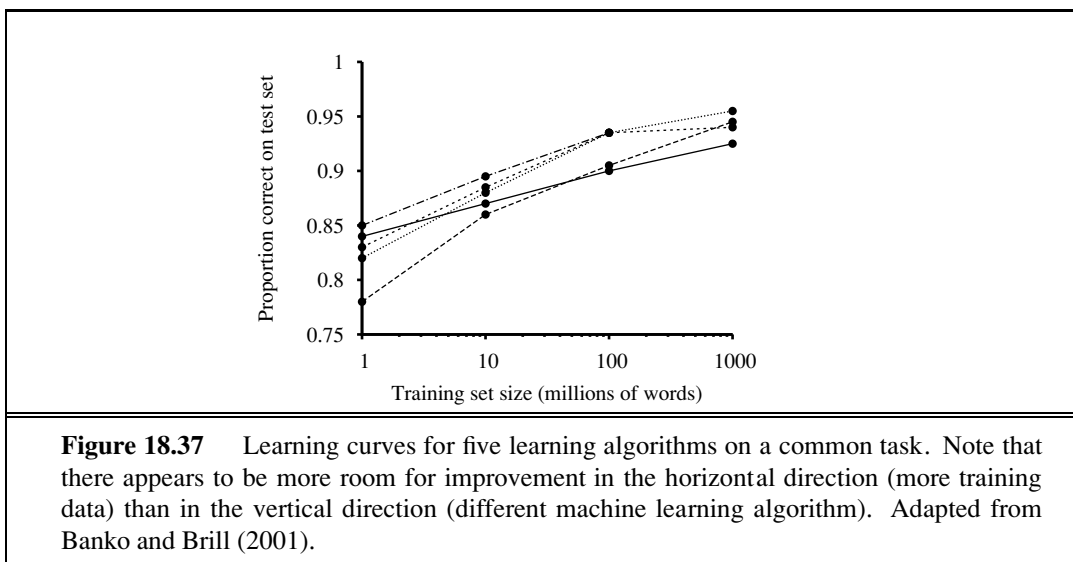
The following figure summarizes the error rates, run time performance, memory requirements, and amount of training time for the seven algorithms we have discussed. It also adds another measure, the percentage of digits that must be rejected to achieve 0.5% error. For example, if the SVM is allowed to reject 1.8% of the inputs—that is, pass them on for someone else to make the final judgment—then its error rate on the remaining 98.2% of the inputs is reduced from 1.1% to 0.5%.

The following table summarizes the error rate and some of the other characteristics of the seven techniques we have discussed.

	3 NN	300 Hidden	LeNet	Boosted LeNet	SVM	Virtual SVM	Shape Match
Error rate (pct.)	2.4	1.6	0.9	0.7	1.1	0.56	0.63
Run time (millisec/digit)	1000	10	30	50	2000	200	
Memory requirements (Mbyte)	12	.49	.012	.21	11		
Training time (days)	0	7	14	30	10		
% rejected to reach 0.5% error	8.1	3.2	1.8	0.5	1.8		

18.11.2   Case study: Word senses and house prices

In a textbook we need to deal with simple, toy data to get the ideas across: a small data set, usually in two dimensions. But in practical applications of machine learning, the data set is usually large, multidimensional, and messy. The data are not handed to the analyst in a prepackaged set of $(\mathbf{x}, y)$ values; rather the analyst needs to go out and acquire the right data. There is a task to be accomplished, and most of the engineering problem is deciding what data are necessary to accomplish the task; a smaller part is choosing and implementing an

Figure 18.37   Learning curves for five learning algorithms on a common task. Note that
there appears to be more room for improvement in the horizontal direction (more training
data) than in the vertical direction (different machine learning algorithm). Adapted from
Banko and Brill (2001).

appropriate machine learning method to process the data. Figure 18.37 shows a typical real-
world example, comparing five learning algorithms on the task of word-sense classification
(given a sentence such as "The bank folded," classify the word "bank" as "money-bank" or
"river-bank"). The point is that machine learning researchers have focused mainly on the
vertical direction: Can I invent a new learning algorithm that performs better than previously
published algorithms on a standard training set of 1 million words? But the graph shows
there is more room for improvement in the horizontal direction: instead of inventing a new
algorithm, all I need to do is gather 10 million words of training data; even the *worst* algorithm
at 10 million words is performing better than the *best* algorithm at 1 million. As we gather
even more data, the curves continue to rise, dwarfing the differences between algorithms.

Consider another problem: the task of estimating the true value of houses that are for
sale. In Figure 18.13 we showed a toy version of this problem, doing linear regression of
house size to asking price. You probably noticed many limitations of this model. First, it is
measuring the wrong thing: we want to estimate the selling price of a house, not the asking
price. To solve this task we'll need data on actual sales. But that doesn't mean we should
throw away the data about asking price—we can use it as one of the input features. Besides
the size of the house, we'll need more information: the number of rooms, bedrooms and
bathrooms; whether the kitchen and bathrooms have been recently remodeled; the age of
the house; we'll also need information about the lot, and the neighborhood. But how do
we define neighborhood? By zip code? What if part of one zip code is on the "wrong"
side of the highway or train tracks, and the other part is desirable? What about the school
district? Should the *name* of the school district be a feature, or the *average test scores*? In
addition to deciding what features to include, we will have to deal with missing data; different
areas have different customs on what data are reported, and individual cases will always be
missing some data. If the data you want are not available, perhaps you can set up a social
networking site to encourage people to share and correct data. In the end, this process of

deciding what features to use, and how to use them, is just as important as choosing between linear regression, decision trees, or some other form of learning.

That said, one *does* have to pick a method (or methods) for a problem. There is no guaranteed way to pick the best method, but there are some rough guidelines. Decision trees are good when there are a lot of discrete features and you believe that many of them may be irrelevant. Nonparametric methods are good when you have a lot of data and no prior knowledge, and when you don't want to worry too much about choosing just the right features (as long as there are fewer than 20 or so). However, nonparametric methods usually give you a function h that is more expensive to run. Support vector machines are often considered the best method to try first, provided the data set is not too large.

18.12   SUMMARY

This chapter has concentrated on inductive learning of functions from examples. The main points were as follows:

- Learning takes many forms, depending on the nature of the agent, the component to be improved, and the available feedback.
- If the available feedback provides the correct answer for example inputs, then the learning problem is called **supervised learning**. The task is to learn a function $y = h(x)$. Learning a discrete-valued function is called **classification**; learning a continuous function is called **regression**.
- Inductive learning involves finding a hypothesis that agrees well with the examples. **Ockham's razor** suggests choosing the simplest consistent hypothesis. The difficulty of this task depends on the chosen representation.
- **Decision trees** can represent all Boolean functions. The **information-gain** heuristic provides an efficient method for finding a simple, consistent decision tree.
- The performance of a learning algorithm is measured by the **learning curve**, which shows the prediction accuracy on the **test set** as a function of the **training-set** size.
- When there are multiple models to choose from, **cross-validation** can be used to select a model that will generalize well.
- Sometimes not all errors are equal. A **loss function** tells us how bad each error is; the goal is then to minimize loss over a validation set.
- **Computational learning theory** analyzes the sample complexity and computational complexity of inductive learning. There is a tradeoff between the expressiveness of the hypothesis language and the ease of learning.
- **Linear regression** is a widely used model. The optimal parameters of a linear regression model can be found by gradient descent search, or computed exactly.
- A linear classifier with a hard threshold—also known as a **perceptron**—can be trained by a simple weight update rule to fit data that are **linearly separable**. In other cases, the rule fails to converge.

- **Logistic regression** replaces the perceptron's hard threshold with a soft threshold defined by a logistic function. Gradient descent works well even for noisy data that are not linearly separable.

- **Neural networks** represent complex nonlinear functions with a network of linear-threshold units. termMultilayer feed-forward neural networks can represent any function, given enough units. The **back-propagation** algorithm implements a gradient descent in parameter space to minimize the output error.

- **Nonparametric models** use all the data to make each prediction, rather than trying to summarize the data first with a few parameters. Examples include **nearest neighbors** and **locally weighted regression**.

- **Support vector machines** find linear separators with **maximum margin** to improve the generalization performance of the classifier. **Kernel methods** implicitly transform the input data into a high-dimensional space where a linear separator may exist, even if the original data are non-separable.

- Ensemble methods such as **boosting** often perform better than individual methods. In **online learning** we can aggregate the opinions of experts to come arbitrarily close to the best expert's performance, even when the distribution of the data is constantly shifting.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Chapter 1 outlined the history of philosophical investigations into inductive learning. William of Ockham[16] (1280–1349), the most influential philosopher of his century and a major contributor to medieval epistemology, logic, and metaphysics, is credited with a statement called "Ockham's Razor"—in Latin, *Entia non sunt multiplicanda praeter necessitatem*, and in English, "Entities are not to be multiplied beyond necessity." Unfortunately, this laudable piece of advice is nowhere to be found in his writings in precisely these words (although he did say "Pluralitas non est ponenda sine necessitate," or "plurality shouldn't be posited without necessity"). A similar sentiment was expressed by Aristotle in 350 B.C. in *Physics* book I, chapter VI: "For the more limited, if adequate, is always preferable."

The first notable use of decision trees was in EPAM, the "Elementary Perceiver And Memorizer" (Feigenbaum, 1961), which was a simulation of human concept learning. ID3 (Quinlan, 1979) added the crucial idea of choosing the attribute with maximum entropy; it is the basis for the decision tree algorithm in this chapter. Information theory was developed by Claude Shannon to aid in the study of communication (Shannon and Weaver, 1949). (Shannon also contributed one of the earliest examples of machine learning, a mechanical mouse named Theseus that learned to navigate through a maze by trial and error.) The χ^2 method of tree pruning was described by Quinlan (1986). C4.5, an industrial-strength decision tree package, can be found in Quinlan (1993). An independent tradition of decision tree learning exists in the statistical literature. *Classification and Regression Trees* (Breiman *et al.*, 1984), known as the "CART book," is the principal reference.

---

[16] The name is often misspelled as "Occam," perhaps from the French rendering, "Guillaume d'Occam."

Cross-validation was first introduced by Larson (1931), and in a form close to what we show by Stone (1974) and Golub *et al.* (1979). The regularization procedure is due to Tikhonov (1963). Guyon and Elisseeff (2003) introduce a journal issue devoted to the problem of feature selection. Banko and Brill (2001) and Halevy *et al.* (2009) discuss the advantages of using large amounts of data. It was Robert Mercer, a speech researcher who said in 1985 "There is no data like more data." (Lyman and Varian, 2003) estimate that about 5 exabytes (5×10^{18} bytes) of data was produced in 2002, and that the rate of production is doubling every 3 years.

Theoretical analysis of learning algorithms began with the work of Gold (1967) on **identification in the limit**. This approach was motivated in part by models of scientific discovery from the philosophy of science (Popper, 1962), but has been applied mainly to the problem of learning grammars from example sentences (Osherson *et al.*, 1986).

Whereas the identification-in-the-limit approach concentrates on eventual convergence, the study of **Kolmogorov complexity** or **algorithmic complexity**, developed independently by Solomonoff (1964, 2009) and Kolmogorov (1965), attempts to provide a formal definition for the notion of simplicity used in Ockham's razor. To escape the problem that simplicity depends on the way in which information is represented, it is proposed that simplicity be measured by the length of the shortest program for a universal Turing machine that correctly reproduces the observed data. Although there are many possible universal Turing machines, and hence many possible "shortest" programs, these programs differ in length by at most a constant that is independent of the amount of data. This beautiful insight, which essentially shows that *any* initial representation bias will eventually be overcome by the data itself, is marred only by the undecidability of computing the length of the shortest program. Approximate measures such as the **minimum description length**, or MDL (Rissanen, 1984, 2007) can be used instead and have produced excellent results in practice. The text by Li and Vitanyi (1993) is the best source for Kolmogorov complexity.

KOLMOGOROV
COMPLEXITY

MINIMUM
DESCRIPTION
LENGTH

The theory of PAC-learning was inaugurated by Leslie Valiant (1984). His work stressed the importance of computational and sample complexity. With Michael Kearns (1990), Valiant showed that several concept classes cannot be PAC-learned tractably, even though sufficient information is available in the examples. Some positive results were obtained for classes such as decision lists (Rivest, 1987).

An independent tradition of sample-complexity analysis has existed in statistics, beginning with the work on **uniform convergence theory** (Vapnik and Chervonenkis, 1971). The so-called **VC dimension** provides a measure roughly analogous to, but more general than, the $\ln |\mathcal{H}|$ measure obtained from PAC analysis. The VC dimension can be applied to continuous function classes, to which standard PAC analysis does not apply. PAC-learning theory and VC theory were first connected by the "four Germans" (none of whom actually is German): Blumer, Ehrenfeucht, Haussler, and Warmuth (1989).

UNIFORM
CONVERGENCE
THEORY
VC DIMENSION

Linear regression with squared error loss goes back to Legendre (1805) and Gauss (1809), who were both working on predicting orbits around the sun. The modern use of multivariate regression for machine learning is covered in texts such as Bishop (2007). Ng (2004) analyzed the differences between L_1 and L_2 regularization.

The term **logistic function** comes from Pierre-François Verhulst (1804–1849), a statistician who used the curve to model population growth with limited resources, a more realistic model than the unconstrained geometric growth proposed by Thomas Malthus. Verhulst called it the *courbe logistique*, because of its relation to the logarithmic curve. The term **regression** is due to Francis Galton, nineteenth century statistician, cousin of Charles Darwin, and initiator of the fields of meteorology, fingerprint analysis, and statistical correlation, who used it in the sense of regression to the mean. The term **curse of dimensionality** comes from Richard Bellman (1961).

Logistic regression can be solved with gradient descent, or with the Newton-Raphson method (Newton, 1671; Raphson, 1690). A variant of the Newton method called L-BFGS is sometimes used for large-dimensional problems; the L stands for "limited memory," meaning that it avoids creating the full matrices all at once, and instead creates parts of them on the fly. BFGS are authors' initials (Byrd *et al.*, 1995).

Nearest-neighbors models date back at least to Fix and Hodges (1951) and have been a standard tool in statistics and pattern recognition ever since. Within AI, they were popularized by Stanfill and Waltz (1986), who investigated methods for adapting the distance metric to the data. Hastie and Tibshirani (1996) developed a way to localize the metric to each point in the space, depending on the distribution of data around that point. Gionis *et al.* (1999) introduced locality-sensitive hashing, which has revolutionized the retrieval of similar objects in high-dimensional spaces, particularly in computer vision. Andoni and Indyk (2006) provide a recent survey of LSH and related methods.

The ideas behind kernel machines come from Aizerman *et al.* (1964) (who also introduced the kernel trick), but the full development of the theory is due to Vapnik and his colleagues (Boser *et al.*, 1992). SVMs were made practical with the introduction of the soft-margin classifier for handling noisy data in a paper that won the 2008 ACM Theory and Practice Award (Cortes and Vapnik, 1995), and of the Sequential Minimal Optimization (SMO) algorithm for efficiently solving SVM problems using quadratic programming (Platt, 1999). SVMs have proven to be very popular and effective for tasks such as text categorization (Joachims, 2001), computational genomics (Cristianini and Hahn, 2007), and natural language processing, such as the handwritten digit recognition of DeCoste and Schölkopf (2002). As part of this process, many new kernels have been designed that work with strings, trees, and other nonnumerical data types. A related technique that also uses the kernel trick to implicitly represent an exponential feature space is the voted perceptron (Freund and Schapire, 1999; Collins and Duffy, 2002). Textbooks on SVMs include Cristianini and Shawe-Taylor (2000) and Schölkopf and Smola (2002). A friendlier exposition appears in the *AI Magazine* article by Cristianini and Schölkopf (2002). Bengio and LeCun (2007) show some of the limitations of SVMs and other local, nonparametric methods for learning functions that have a global structure but do not have local smoothness.

Ensemble learning is an increasingly popular technique for improving the performance of learning algorithms. **Bagging** (Breiman, 1996), the first effective method, combines hypotheses learned from multiple **bootstrap** data sets, each generated by subsampling the original data set. The **boosting** method described in this chapter originated with theoretical work by Schapire (1990). The ADABOOST algorithm was developed by Freund and Schapire

BAGGING

(1996) and analyzed theoretically by Schapire (2003). Friedman *et al.* (2000) explain boosting from a statistician's viewpoint. Online learning is covered in a survey by Blum (1996) and a book by Cesa-Bianchi and Lugosi (2006). Dredze *et al.* (2008) introduce the idea of confidence-weighted online learning for classification: in addition to keeping a weight for each parameter, they also maintain a measure of confidence, so that a new example can have a large effect on features that were rarely seen before (and thus had low confidence) and a small effect on common features that have already been well-estimated.

The literature on neural networks is rather too large (approximately 150,000 papers to date) to cover in detail. Cowan and Sharp (1988b, 1988a) survey the early history, beginning with the work of McCulloch and Pitts (1943). (As mentioned in Chapter 1, John McCarthy has pointed to the work of Nicolas Rashevsky (1936, 1938) as the earliest mathematical model of neural learning.) Norbert Wiener, a pioneer of cybernetics and control theory (Wiener, 1948), worked with McCulloch and Pitts and influenced a number of young researchers including Marvin Minsky, who may have been the first to develop a working neural network in hardware in 1951 (see Minsky and Papert, 1988, pp. ix–x). Turing (1948) wrote a research report titled *Intelligent Machinery* that begins with the sentence "I propose to investigate the question as to whether it is possible for machinery to show intelligent behaviour" and goes on to describe a recurrent neural network architecture he called "B-type unorganized machines" and an approach to training them. Unfortunately, the report went unpublished until 1969, and was all but ignored until recently.

Frank Rosenblatt (1957) invented the modern "perceptron" and proved the perceptron convergence theorem (1960), although it had been foreshadowed by purely mathematical work outside the context of neural networks (Agmon, 1954; Motzkin and Schoenberg, 1954). Some early work was also done on multilayer networks, including **Gamba perceptrons** (Gamba *et al.*, 1961) and **madalines** (Widrow, 1962). *Learning Machines* (Nilsson, 1965) covers much of this early work and more. The subsequent demise of early perceptron research efforts was hastened (or, the authors later claimed, merely explained) by the book *Perceptrons* (Minsky and Papert, 1969), which lamented the field's lack of mathematical rigor. The book pointed out that single-layer perceptrons could represent only linearly separable concepts and noted the lack of effective learning algorithms for multilayer networks.

The papers in (Hinton and Anderson, 1981), based on a conference in San Diego in 1979, can be regarded as marking a renaissance of connectionism. The two-volume "PDP" (Parallel Distributed Processing) anthology (Rumelhart *et al.*, 1986a) and a short article in *Nature* (Rumelhart *et al.*, 1986b) attracted a great deal of attention—indeed, the number of papers on "neural networks" multiplied by a factor of 200 between 1980–84 and 1990–94. The analysis of neural networks using the physical theory of magnetic spin glasses (Amit *et al.*, 1985) tightened the links between statistical mechanics and neural network theory—providing not only useful mathematical insights but also *respectability*. The back-propagation technique had been invented quite early (Bryson and Ho, 1969) but it was rediscovered several times (Werbos, 1974; Parker, 1985).

The probabilistic interpretation of neural networks has several sources, including Baum and Wilczek (1988) and Bridle (1990). The role of the sigmoid function is discussed by Jordan (1995). Bayesian parameter learning for neural networks was proposed by MacKay

(1992) and is explored further by Neal (1996). The capacity of neural networks to represent functions was investigated by Cybenko (1988, 1989), who showed that two hidden layers are enough to represent any function and a single layer is enough to represent any *continuous* function. The "optimal brain damage" method for removing useless connections is by LeCun et al. (1989), and Sietsma and Dow (1988) show how to remove useless units. The tiling algorithm for growing larger structures is due to Mézard and Nadal (1989). LeCun *et al.* (1995) survey a number of algorithms for handwritten digit recognition. Improved error rates since then were reported by Belongie *et al.* (2002) for shape matching and DeCoste and Schölkopf (2002) for virtual support vectors. At the time of writing, the best test error rate reported is 0.39% by Ranzato *et al.* (2007) using a convolutional neural network.

The complexity of neural network learning has been investigated by researchers in computational learning theory. Early computational results were obtained by Judd (1990), who showed that the general problem of finding a set of weights consistent with a set of examples is NP-complete, even under very restrictive assumptions. Some of the first sample complexity results were obtained by Baum and Haussler (1989), who showed that the number of examples required for effective learning grows as roughly $W \log W$, where W is the number of weights.[17] Since then, a much more sophisticated theory has been developed (Anthony and Bartlett, 1999), including the important result that the representational capacity of a network depends on the *size* of the weights as well as on their number, a result that should not be surprising in the light of our discussion of regularization.

RADIAL BASIS
FUNCTION

The most popular kind of neural network that we did not cover is the **radial basis function**, or RBF, network. A radial basis function combines a weighted collection of kernels (usually Gaussians, of course) to do function approximation. RBF networks can be trained in two phases: first, an unsupervised clustering approach is used to train the parameters of the Gaussians—the means and variances—are trained, as in Section 20.3.1. In the second phase, the relative weights of the Gaussians are determined. This is a system of linear equations, which we know how to solve directly. Thus, both phases of RBF training have a nice benefit: the first phase is unsupervised, and thus does not require labeled training data, and the second phase, although supervised, is efficient. See Bishop (1995) for more details.

HOPFIELD NETWORK

Recurrent networks, in which units are linked in cycles, were mentioned in the chapter but not explored in depth. **Hopfield networks** (Hopfield, 1982) are probably the best-understood class of recurrent networks. They use *bidirectional* connections with *symmetric* weights (i.e., $w_{i,j} = w_{j,i}$), all of the units are both input and output units, the activation function g is the sign function, and the activation levels can only be ± 1. A Hopfield network

ASSOCIATIVE
MEMORY

functions as an **associative memory**: after the network trains on a set of examples, a new stimulus will cause it to settle into an activation pattern corresponding to the example in the training set that *most closely resembles* the new stimulus. For example, if the training set consists of a set of photographs, and the new stimulus is a small piece of one of the photographs, then the network activation levels will reproduce the photograph from which the piece was taken. Notice that the original photographs are not stored separately in the network; each

---

[17] This approximately confirmed "Uncle Bernie's rule." The rule was named after Bernie Widrow, who recommended using roughly ten times as many examples as weights.

weight is a partial encoding of all the photographs. One of the most interesting theoretical results is that Hopfield networks can reliably store up to $0.138N$ training examples, where N is the number of units in the network.

BOLTZMANN
MACHINE

Boltzmann machines (Hinton and Sejnowski, 1983, 1986) also use symmetric weights, but include hidden units. In addition, they use a *stochastic* activation function, such that the probability of the output being 1 is some function of the total weighted input. Boltzmann machines therefore undergo state transitions that resemble a simulated annealing search (see Chapter 4) for the configuration that best approximates the training set. It turns out that Boltzmann machines are very closely related to a special case of Bayesian networks evaluated with a stochastic simulation algorithm. (See Section 14.5.)

For neural nets, Bishop (1995), Ripley (1996), and Haykin (2008) are the leading texts. The field of computational neuroscience is covered by Dayan and Abbott (2001).

The approach taken in this chapter was influenced by the excellent course notes of David Cohn, Tom Mitchell, Andrew Moore, and Andrew Ng. There are several top-notch textbooks in Machine Learning (Mitchell, 1997; Bishop, 2007) and in the closely allied and overlapping fields of pattern recognition (Ripley, 1996; Duda *et al.*, 2001), statistics (Wasserman, 2004; Hastie *et al.*, 2001), data mining (Hand *et al.*, 2001; Witten and Frank, 2005), computational learning theory (Kearns and Vazirani, 1994; Vapnik, 1998) and information theory (Shannon and Weaver, 1949; MacKay, 2002; Cover and Thomas, 2006). Other books concentrate on implementations (Segaran, 2007; Marsland, 2009) and comparisons of algorithms (Michie *et al.*, 1994). Current research in machine learning is published in the annual proceedings of the International Conference on Machine Learning (ICML) and the conference on Neural Information Processing Systems (NIPS), in *Machine Learning* and the *Journal of Machine Learning Research*, and in mainstream AI journals.

EXERCISES

18.1   Consider the problem faced by an infant learning to speak and understand a language. Explain how this process fits into the general learning model. Describe the percepts and actions of the infant, and the types of learning the infant must do. Describe the subfunctions the infant is trying to learn in terms of inputs and outputs, and available example data.

18.2   Repeat Exercise 18.1 for the case of learning to play tennis (or some other sport with which you are familiar). Is this supervised learning or reinforcement learning?

18.3   Suppose we generate a training set from a decision tree and then apply decision-tree learning to that training set. Is it the case that the learning algorithm will eventually return the correct tree as the training-set size goes to infinity? Why or why not?

18.4   In the recursive construction of decision trees, it sometimes happens that a mixed set of positive and negative examples remains at a leaf node, even after all the attributes have been used. Suppose that we have p positive examples and n negative examples.

a. Show that the solution used by DECISION-TREE-LEARNING, which picks the majority classification, minimizes the absolute error over the set of examples at the leaf.

b. Show that the **class probability** $p/(p + n)$ minimizes the sum of squared errors.

18.5   Suppose that an attribute splits the set of examples E into subsets E_k and that each subset has p_k positive examples and n_k negative examples. Show that the attribute has strictly positive information gain unless the ratio $p_k/(p_k + n_k)$ is the same for all k.

18.6   Consider the following data set comprised of three binary input attributes (A_1, A_2, and A_3) and one binary output:

| **Example**	A_1	A_2	A_3	Output y
$\mathbf{x}_1$	1	0	0	0
$\mathbf{x}_2$	1	0	1	0
$\mathbf{x}_3$	0	1	0	0
$\mathbf{x}_4$	1	1	1	1
$\mathbf{x}_5$	1	1	0	1

Use the algorithm in Figure 18.5 (page 702) to learn a decision tree for these data. Show the computations made to determine the attribute to split at each node.

18.7   A decision *graph* is a generalization of a decision tree that allows nodes (i.e., attributes used for splits) to have multiple parents, rather than just a single parent. The resulting graph must still be acyclic. Now, consider the XOR function of *three* binary input attributes, which produces the value 1 if and only if an odd number of the three input attributes has value 1.

a. Draw a minimal-sized decision *tree* for the three-input XOR function.
b. Draw a minimal-sized decision *graph* for the three-input XOR function.

18.8   This exercise considers χ^2 pruning of decision trees (Section 18.3.5).

a. Create a data set with two input attributes, such that the information gain at the root of the tree for both attributes is zero, but there is a decision tree of depth 2 that is consistent with all the data. What would χ^2 pruning do on this data set if applied bottom up? If applied top down?
b. Modify DECISION-TREE-LEARNING to include χ^2-pruning. You might wish to consult Quinlan (1986) or Kearns and Mansour (1998) for details.
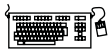
18.9   The standard DECISION-TREE-LEARNING algorithm described in the chapter does not handle cases in which some examples have missing attribute values.

a. First, we need to find a way to classify such examples, given a decision tree that includes tests on the attributes for which values can be missing. Suppose that an example **x** has a missing value for attribute A and that the decision tree tests for A at a node that **x** reaches. One way to handle this case is to pretend that the example has *all* possible values for the attribute, but to weight each value according to its frequency among all of the examples that reach that node in the decision tree. The classification algorithm should follow all branches at any node for which a value is missing and should multiply

the weights along each path. Write a modified classification algorithm for decision trees that has this behavior.

b. Now modify the information-gain calculation so that in any given collection of examples C at a given node in the tree during the construction process, the examples with missing values for any of the remaining attributes are given "as-if" values according to the frequencies of those values in the set C.

18.10 In Section 18.3.6, we noted that attributes with many different possible values can cause problems with the gain measure. Such attributes tend to split the examples into numerous small classes or even singleton classes, thereby appearing to be highly relevant according to the gain measure. The **gain-ratio** criterion selects attributes according to the ratio between their gain and their intrinsic information content—that is, the amount of information contained in the answer to the question, "What is the value of this attribute?" The gain-ratio criterion therefore tries to measure how efficiently an attribute provides information on the correct classification of an example. Write a mathematical expression for the information content of an attribute, and implement the gain ratio criterion in DECISION-TREE-LEARNING.

18.11 Suppose you are running a learning experiment on a new algorithm for Boolean classification. You have a data set consisting of 100 positive and 100 negative examples. You plan to use leave-one-out cross-validation and compare your algorithm to a baseline function, a simple majority classifier. (A majority classifier is given a set of training data and then always outputs the class that is in the majority in the training set, regardless of the input.) You expect the majority classifier to score about 50% on leave-one-out cross-validation, but to your surprise, it scores zero every time. Can you explain why?

18.12 Construct a *decision list* to classify the data below. Select tests to be as small as possible (in terms of attributes), breaking ties among tests with the same number of attributes by selecting the one that classifies the greatest number of examples correctly. If multiple tests have the same number of attributes and classify the same number of examples, then break the tie using attributes with lower index numbers (e.g., select A_1 over A_2).

Example	A_1	A_2	A_3	A_4	y
$\mathbf{x}_1$	1	0	0	0	1
$\mathbf{x}_2$	1	0	1	1	1
$\mathbf{x}_3$	0	1	0	0	1
$\mathbf{x}_4$	0	1	1	0	0
$\mathbf{x}_5$	1	1	0	1	1
$\mathbf{x}_6$	0	1	0	1	0
$\mathbf{x}_7$	0	0	1	1	1
$\mathbf{x}_8$	0	0	1	0	0

18.13 Prove that a decision list can represent the same function as a decision tree while using at most as many rules as there are leaves in the decision tree for that function. Give an example of a function represented by a decision list using strictly fewer rules than the number of leaves in a minimal-sized decision tree for that same function.

18.14   This exercise concerns the expressiveness of decision lists (Section 18.5).

  a. Show that decision lists can represent any Boolean function, if the size of the tests is not limited.

  b. Show that if the tests can contain at most k literals each, then decision lists can represent any function that can be represented by a decision tree of depth k.

18.15   Suppose a 7-nearest-neighbors regression search returns $\{7, 6, 8, 4, 7, 11, 100\}$ as the 7 nearest y values for a given x value. What is the value of $\hat{y}$ that minimizes the L_1 loss function on this data? There is a common name in statistics for this value as a function of the y values; what is it? Answer the same two questions for the L_2 loss function.

18.16   Figure 18.31 showed how a circle at the origin can be linearly separated by mapping from the features (x_1, x_2) to the two dimensions (x_1^2, x_2^2). But what if the circle is not located at the origin? What if it is an ellipse, not a circle? The general equation for a circle (and hence the decision boundary) is $(x_1 - a)^2 + (x_2 - b)^2 - r^2 = 0$, and the general equation for an ellipse is $c(x_1 - a)^2 + d(x_2 - b)^2 - 1 = 0$.

  a. Expand out the equation for the circle and show what the weights w_i would be for the decision boundary in the four-dimensional feature space (x_1, x_2, x_1^2, x_2^2). Explain why this means that any circle is linearly separable in this space.

  b. Do the same for ellipses in the five-dimensional feature space $(x_1, x_2, x_1^2, x_2^2, x_1 x_2)$.

18.17   Construct a support vector machine that computes the XOR function. Use values of +1 and –1 (instead of 1 and 0) for both inputs and outputs, so that an example looks like $([-1, 1], 1)$ or $([-1, -1], -1)$. Map the input $[x_1, x_2]$ into a space consisting of x_1 and $x_1 x_2$. Draw the four input points in this space, and the maximal margin separator. What is the margin? Now draw the separating line back in the original Euclidean input space.

18.18   Consider an ensemble learning algorithm that uses simple majority voting among K learned hypotheses. Suppose that each hypothesis has error ϵ and that the errors made by each hypothesis are independent of the others'. Calculate a formula for the error of the ensemble algorithm in terms of K and ϵ, and evaluate it for the cases where $K = 5$, 10, and 20 and $\epsilon = 0.1$, 0.2, and 0.4. If the independence assumption is removed, is it possible for the ensemble error to be *worse* than ϵ?

18.19   Construct by hand a neural network that computes the XOR function of two inputs. Make sure to specify what sort of units you are using.

18.20   Recall from Chapter 18 that there are 2^{2^n} distinct Boolean functions of n inputs. How many of these are representable by a threshold perceptron?

18.21   Section 18.6.4 (page 725) noted that the output of the logistic function could be interpreted as a *probability* p assigned by the model to the proposition that $f(\mathbf{x}) = 1$; the probability that $f(\mathbf{x}) = 0$ is therefore $1 - p$. Write down the probability p as a function of $\mathbf{x}$ and calculate the derivative of $\log p$ with respect to each weight w_i. Repeat the process for $\log(1-p)$. These calculations give a learning rule for minimizing the negative-log-likelihood

loss function for a probabilistic hypothesis. Comment on any resemblance to other learning rules in the chapter.

18.22 Suppose you had a neural network with linear activation functions. That is, for each unit the output is some constant c times the weighted sum of the inputs.

  a. Assume that the network has one hidden layer. For a given assignment to the weights **w**, write down equations for the value of the units in the output layer as a function of **w** and the input layer **x**, without any explicit mention of the output of the hidden layer. Show that there is a network with no hidden units that computes the same function.

  b. Repeat the calculation in part (a), but this time do it for a network with any number of hidden layers.

  c. Suppose a network with one hidden layer and linear activation functions has n input and output nodes and h hidden nodes. What effect does the transformation in part (a) to a network with no hidden layers have on the total number of weights? Discuss in particular the case $h \ll n$.

18.23 Suppose that a training set contains only a single example, repeated 100 times. In 80 of the 100 cases, the single output value is 1; in the other 20, it is 0. What will a back-propagation network predict for this example, assuming that it has been trained and reaches a global optimum? (*Hint:* to find the global optimum, differentiate the error function and set it to zero.)

18.24 The neural network whose learning performance is measured in Figure 18.25 has four hidden nodes. This number was chosen somewhat arbitrarily. Use a cross-validation method to find the best number of hidden nodes.

18.25 Consider the problem of separating N data points into positive and negative examples using a linear separator. Clearly, this can always be done for $N = 2$ points on a line of dimension $d = 1$, regardless of how the points are labeled or where they are located (unless the points are in the same place).

  a. Show that it can always be done for $N = 3$ points on a plane of dimension $d = 2$, unless they are collinear.

  b. Show that it cannot always be done for $N = 4$ points on a plane of dimension $d = 2$.

  c. Show that it can always be done for $N = 4$ points in a space of dimension $d = 3$, unless they are coplanar.

  d. Show that it cannot always be done for $N = 5$ points in a space of dimension $d = 3$.

  e. The ambitious student may wish to prove that N points in general position (but not $N + 1$) are linearly separable in a space of dimension $N - 1$.

19 KNOWLEDGE IN LEARNING

In which we examine the problem of learning when you know something already.

In all of the approaches to learning described in the previous chapter, the idea is to construct a function that has the input–output behavior observed in the data. In each case, the learning methods can be understood as searching a hypothesis space to find a suitable function, starting from only a very basic assumption about the form of the function, such as "second-degree polynomial" or "decision tree" and perhaps a preference for simpler hypotheses. Doing this amounts to saying that before you can learn something new, you must first forget (almost) everything you know. In this chapter, we study learning methods that can take advantage PRIOR KNOWLEDGE of **prior knowledge** about the world. In most cases, the prior knowledge is represented as general first-order logical theories; thus for the first time we bring together the work on knowledge representation and learning.

19.1 A LOGICAL FORMULATION OF LEARNING

Chapter 18 defined pure inductive learning as a process of finding a hypothesis that agrees with the observed examples. Here, we specialize this definition to the case where the hypothesis is represented by a set of logical sentences. Example descriptions and classifications will also be logical sentences, and a new example can be classified by inferring a classification sentence from the hypothesis and the example description. This approach allows for incremental construction of hypotheses, one sentence at a time. It also allows for prior knowledge, because sentences that are already known can assist in the classification of new examples. The logical formulation of learning may seem like a lot of extra work at first, but it turns out to clarify many of the issues in learning. It enables us to go well beyond the simple learning methods of Chapter 18 by using the full power of logical inference in the service of learning.

19.1.1 Examples and hypotheses

Recall from Chapter 18 the restaurant learning problem: learning a rule for deciding whether to wait for a table. Examples were described by **attributes** such as *Alternate*, *Bar*, *Fri/Sat*,

and so on. In a logical setting, an example is described by a logical sentence; the attributes become unary predicates. Let us generically call the ith example X_i. For instance, the first example from Figure 18.3 (page 700) is described by the sentences

$$Alternate(X_1) \wedge \neg Bar(X_1) \wedge \neg Fri/Sat(X_1) \wedge Hungry(X_1) \wedge \ldots$$

We will use the notation $D_i(X_i)$ to refer to the description of X_i, where D_i can be any logical expression taking a single argument. The classification of the example is given by a literal using the goal predicate, in this case

$$WillWait(X_1) \qquad \text{or} \qquad \neg WillWait(X_1) \, .$$

The complete training set can thus be expressed as the conjunction of all the example descriptions and goal literals.

The aim of inductive learning in general is to find a hypothesis that classifies the examples well and generalizes well to new examples. Here we are concerned with hypotheses expressed in logic; each hypothesis h_j will have the form

$$\forall x \ \ Goal(x) \ \Leftrightarrow \ C_j(x) \, ,$$

where $C_j(x)$ is a candidate definition—some expression involving the attribute predicates. For example, a decision tree can be interpreted as a logical expression of this form. Thus, the tree in Figure 18.6 (page 702) expresses the following logical definition (which we will call h_r for future reference):

$$
\begin{aligned}
\forall r \ \ WillWait(r) \ \Leftrightarrow \ & Patrons(r, Some) \\
\vee \ & Patrons(r, Full) \wedge Hungry(r) \wedge Type(r, French) \\
\vee \ & Patrons(r, Full) \wedge Hungry(r) \wedge Type(r, Thai) \qquad (19.1) \\
& \wedge Fri/Sat(r) \\
\vee \ & Patrons(r, Full) \wedge Hungry(r) \wedge Type(r, Burger) \, .
\end{aligned}
$$

EXTENSION

Each hypothesis predicts that a certain set of examples—namely, those that satisfy its candidate definition—will be examples of the goal predicate. This set is called the **extension** of the predicate. Two hypotheses with different extensions are therefore logically inconsistent with each other, because they disagree on their predictions for at least one example. If they have the same extension, they are logically equivalent.

The hypothesis space $\mathcal{H}$ is the set of all hypotheses $\{h_1, \ldots, h_n\}$ that the learning algorithm is designed to entertain. For example, the DECISION-TREE-LEARNING algorithm can entertain any decision tree hypothesis defined in terms of the attributes provided; its hypothesis space therefore consists of all these decision trees. Presumably, the learning algorithm believes that one of the hypotheses is correct; that is, it believes the sentence

$$h_1 \vee h_2 \vee h_3 \vee \ldots \vee h_n \, . \tag{19.2}$$

As the examples arrive, hypotheses that are not **consistent** with the examples can be ruled out. Let us examine this notion of consistency more carefully. Obviously, if hypothesis h_j is consistent with the entire training set, it has to be consistent with each example in the training set. What would it mean for it to be inconsistent with an example? There are two possible ways that this can happen:

FALSE NEGATIVE

- An example can be a **false negative** for the hypothesis, if the hypothesis says it should be negative but in fact it is positive. For instance, the new example X_{13} described by
$Patrons(X_{13}, Full) \land \neg Hungry(X_{13}) \land \ldots \land WillWait(X_{13})$
would be a false negative for the hypothesis h_r given earlier. From h_r and the example description, we can deduce both $WillWait(X_{13})$, which is what the example says, and $\neg WillWait(X_{13})$, which is what the hypothesis predicts. The hypothesis and the example are therefore logically inconsistent.

FALSE POSITIVE

- An example can be a **false positive** for the hypothesis, if the hypothesis says it should be positive but in fact it is negative.[1]

If an example is a false positive or false negative for a hypothesis, then the example and the hypothesis are logically inconsistent with each other. Assuming that the example is a correct observation of fact, then the hypothesis can be ruled out. Logically, this is exactly analogous to the resolution rule of inference (see Chapter 9), where the disjunction of hypotheses corresponds to a clause and the example corresponds to a literal that resolves against one of the literals in the clause. An ordinary logical inference system therefore could, in principle, learn from the example by eliminating one or more hypotheses. Suppose, for example, that the example is denoted by the sentence I_1, and the hypothesis space is $h_1 \lor h_2 \lor h_3 \lor h_4$. Then if I_1 is inconsistent with h_2 and h_3, the logical inference system can deduce the new hypothesis space $h_1 \lor h_4$.

We therefore can characterize inductive learning in a logical setting as a process of gradually eliminating hypotheses that are inconsistent with the examples, narrowing down the possibilities. Because the hypothesis space is usually vast (or even infinite in the case of first-order logic), we do not recommend trying to build a learning system using resolution-based theorem proving and a complete enumeration of the hypothesis space. Instead, we will describe two approaches that find logically consistent hypotheses with much less effort.
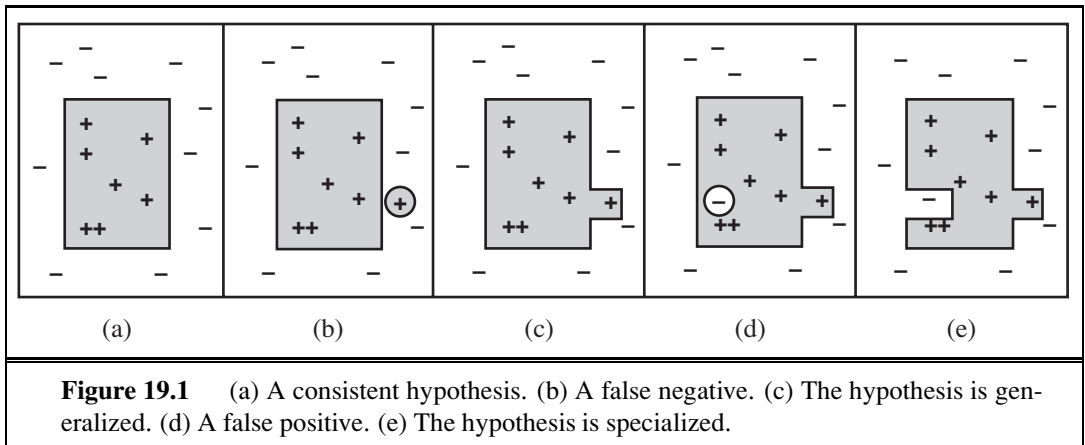
19.1.2 Current-best-hypothesis search

CURRENT-BEST-
HYPOTHESIS

The idea behind **current-best-hypothesis** search is to maintain a single hypothesis, and to adjust it as new examples arrive in order to maintain consistency. The basic algorithm was described by John Stuart Mill (1843), and may well have appeared even earlier.

Suppose we have some hypothesis such as h_r, of which we have grown quite fond. As long as each new example is consistent, we need do nothing. Then along comes a false negative example, X_{13}. What do we do? Figure 19.1(a) shows h_r schematically as a region: everything inside the rectangle is part of the extension of h_r. The examples that have actually been seen so far are shown as "+" or "−", and we see that h_r correctly categorizes all the examples as positive or negative examples of $WillWait$. In Figure 19.1(b), a new example (circled) is a false negative: the hypothesis says it should be negative but it is actually positive.

GENERALIZATION

The extension of the hypothesis must be increased to include it. This is called **generalization**; one possible generalization is shown in Figure 19.1(c). Then in Figure 19.1(d), we see a false positive: the hypothesis says the new example (circled) should be positive, but it actually is

---

[1] The terms "false positive" and "false negative" are used in medicine to describe erroneous results from lab tests. A result is a false positive if it indicates that the patient has the disease when in fact no disease is present.

Figure 19.1   (a) A consistent hypothesis. (b) A false negative. (c) The hypothesis is generalized. (d) A false positive. (e) The hypothesis is specialized.

---

function CURRENT-BEST-LEARNING(*examples*, *h*) **returns** a hypothesis or fail

   if *examples* is empty **then**
     return *h*
   e ← FIRST(*examples*)
   if *e* is consistent with *h* **then**
     return CURRENT-BEST-LEARNING(REST(*examples*), *h*)
   else if *e* is a false positive for *h* **then**
     for each *h'* **in** specializations of *h* consistent with *examples* seen so far **do**
       h'' ← CURRENT-BEST-LEARNING(REST(*examples*), *h'*)
       if $h'' \neq fail$ **then return** *h''*
   else if *e* is a false negative for *h* **then**
     for each *h'* **in** generalizations of *h* consistent with *examples* seen so far **do**
       h'' ← CURRENT-BEST-LEARNING(REST(*examples*), *h'*)
       if $h'' \neq fail$ **then return** *h''*
   return *fail*

Figure 19.2   The current-best-hypothesis learning algorithm. It searches for a consistent hypothesis that fits all the examples and backtracks when no consistent specialization/generalization can be found. To start the algorithm, any hypothesis can be passed in; it will be specialized or gneralized as needed.

negative. The extension of the hypothesis must be decreased to exclude the example. This is
SPECIALIZATION   called **specialization**; in Figure 19.1(e) we see one possible specialization of the hypothesis.
The "more general than" and "more specific than" relations between hypotheses provide the
logical structure on the hypothesis space that makes efficient search possible.

     We can now specify the CURRENT-BEST-LEARNING algorithm, shown in Figure 19.2.
Notice that each time we consider generalizing or specializing the hypothesis, we must check
for consistency with the other examples, because an arbitrary increase/decrease in the extension might include/exclude previously seen negative/positive examples.

We have defined generalization and specialization as operations that change the *extension* of a hypothesis. Now we need to determine exactly how they can be implemented as syntactic operations that change the candidate definition associated with the hypothesis, so that a program can carry them out. This is done by first noting that generalization and specialization are also *logical* relationships between hypotheses. If hypothesis h_1, with definition C_1, is a generalization of hypothesis h_2 with definition C_2, then we must have

$$\forall x \ C_2(x) \Rightarrow C_1(x) .$$

DROPPING
CONDITIONS

Therefore in order to construct a generalization of h_2, we simply need to find a definition C_1 that is logically implied by C_2. This is easily done. For example, if $C_2(x)$ is $Alternate(x) \wedge Patrons(x, Some)$, then one possible generalization is given by $C_1(x) \equiv Patrons(x, Some)$. This is called **dropping conditions**. Intuitively, it generates a weaker definition and therefore allows a larger set of positive examples. There are a number of other generalization operations, depending on the language being operated on. Similarly, we can specialize a hypothesis by adding extra conditions to its candidate definition or by removing disjuncts from a disjunctive definition. Let us see how this works on the restaurant example, using the data in Figure 18.3.

- The first example, X_1, is positive. The attribute $Alternate(X_1)$ is true, so let the initial hypothesis be

$$h_1 : \ \forall x \ WillWait(x) \Leftrightarrow Alternate(x) .$$

- The second example, X_2, is negative. h_1 predicts it to be positive, so it is a false positive. Therefore, we need to specialize h_1. This can be done by adding an extra condition that will rule out X_2, while continuing to classify X_1 as positive. One possibility is

$$h_2 : \ \forall x \ WillWait(x) \Leftrightarrow Alternate(x) \wedge Patrons(x, Some) .$$

- The third example, X_3, is positive. h_2 predicts it to be negative, so it is a false negative. Therefore, we need to generalize h_2. We drop the $Alternate$ condition, yielding

$$h_3 : \ \forall x \ WillWait(x) \Leftrightarrow Patrons(x, Some) .$$

- The fourth example, X_4, is positive. h_3 predicts it to be negative, so it is a false negative. We therefore need to generalize h_3. We cannot drop the $Patrons$ condition, because that would yield an all-inclusive hypothesis that would be inconsistent with X_2. One possibility is to add a disjunct:

$$h_4 : \ \forall x \ WillWait(x) \Leftrightarrow Patrons(x, Some)$$
$$\vee \left(Patrons(x, Full) \wedge Fri/Sat(x) \right) .$$

Already, the hypothesis is starting to look reasonable. Obviously, there are other possibilities consistent with the first four examples; here are two of them:

$$h_4' : \ \forall x \ WillWait(x) \Leftrightarrow \neg WaitEstimate(x, 30\text{-}60) .$$

$$h_4'' : \ \forall x \ WillWait(x) \Leftrightarrow Patrons(x, Some)$$
$$\vee \left(Patrons(x, Full) \wedge WaitEstimate(x, 10\text{-}30) \right) .$$

The CURRENT-BEST-LEARNING algorithm is described nondeterministically, because at any point, there may be several possible specializations or generalizations that can be applied. The

---

function VERSION-SPACE-LEARNING(*examples*) **returns** a version space
   local variables: V, the version space: the set of all hypotheses

   $V \leftarrow$ the set of all hypotheses
   for each example e in *examples* **do**
      if V is not empty **then** $V \leftarrow$ VERSION-SPACE-UPDATE(V, e)
   return V

---

function VERSION-SPACE-UPDATE(V, e) **returns** an updated version space

   $V \leftarrow \{h \in V :\ h \text{ is consistent with } e\}$

Figure 19.3   The version space learning algorithm. It finds a subset of V that is consistent with all the *examples*.

---

choices that are made will not necessarily lead to the simplest hypothesis, and may lead to an unrecoverable situation where no simple modification of the hypothesis is consistent with all of the data. In such cases, the program must backtrack to a previous choice point.

The CURRENT-BEST-LEARNING algorithm and its variants have been used in many machine learning systems, starting with Patrick Winston's (1970) "arch-learning" program. With a large number of examples and a large space, however, some difficulties arise:

1. Checking all the previous examples over again for each modification is very expensive.
2. The search process may involve a great deal of backtracking. As we saw in Chapter 18, hypothesis space can be a doubly exponentially large place.
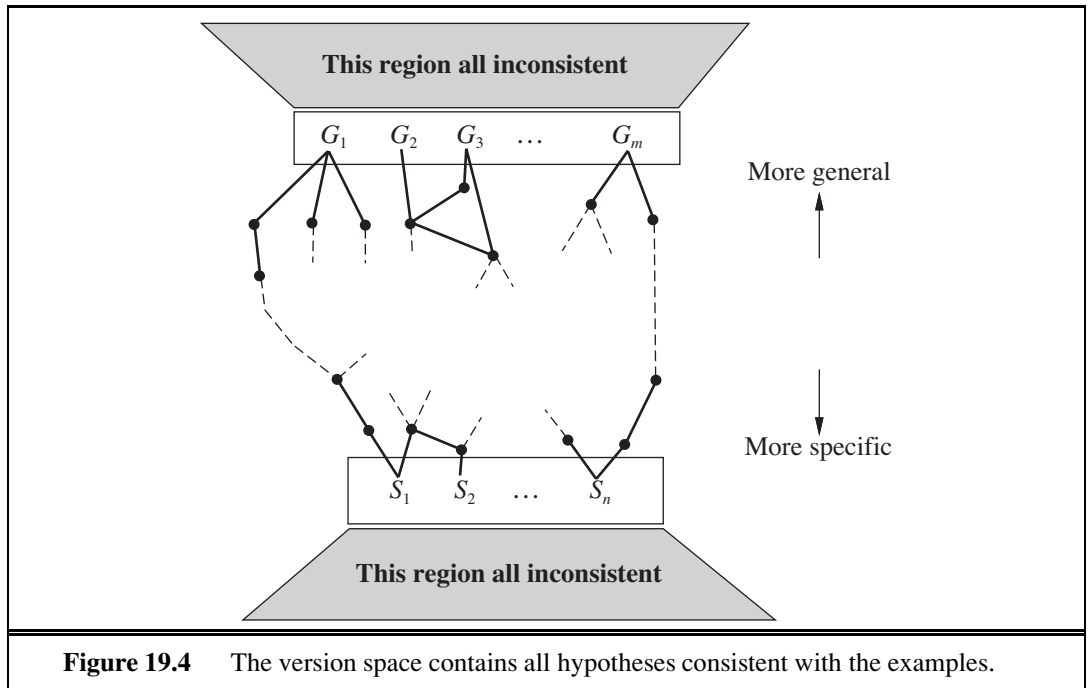
19.1.3   Least-commitment search

Backtracking arises because the current-best-hypothesis approach has to *choose* a particular hypothesis as its best guess even though it does not have enough data yet to be sure of the choice. What we can do instead is to keep around all and only those hypotheses that are consistent with all the data so far. Each new example will either have no effect or will get rid of some of the hypotheses. Recall that the original hypothesis space can be viewed as a disjunctive sentence

$$h_1 \vee h_2 \vee h_3 \ldots \vee h_n \ .$$

As various hypotheses are found to be inconsistent with the examples, this disjunction shrinks, retaining only those hypotheses not ruled out. Assuming that the original hypothesis space does in fact contain the right answer, the reduced disjunction must still contain the right answer because only incorrect hypotheses have been removed. The set of hypotheses remaining is called the **version space**, and the learning algorithm (sketched in Figure 19.3) is called the version space learning algorithm (also the **candidate elimination** algorithm).

VERSION SPACE

CANDIDATE
ELIMINATION

One important property of this approach is that it is *incremental*: one never has to go back and reexamine the old examples. All remaining hypotheses are guaranteed to be consistent with them already. But there is an obvious problem. We already said that the

This region all inconsistent

G_1 G_2 G_3 ... G_m

More general

More specific

S_1 S_2 ... S_n

This region all inconsistent

| **Figure 19.4** | The version space contains all hypotheses consistent with the examples. |

hypothesis space is enormous, so how can we possibly write down this enormous disjunction?

The following simple analogy is very helpful. How do you represent all the real numbers between 1 and 2? After all, there are an infinite number of them! The answer is to use an interval representation that just specifies the boundaries of the set: [1,2]. It works because we have an *ordering* on the real numbers.

We also have an ordering on the hypothesis space, namely, generalization/specialization. This is a partial ordering, which means that each boundary will not be a point but rather a set of hypotheses called a **boundary set**. The great thing is that we can represent the entire version space using just two boundary sets: a most general boundary (the **G-set**) and a most specific boundary (the **S-set**). *Everything in between is guaranteed to be consistent with the examples*. Before we prove this, let us recap:

- The current version space is the set of hypotheses consistent with all the examples so far. It is represented by the S-set and G-set, each of which is a set of hypotheses.
- Every member of the S-set is consistent with all observations so far, and there are no consistent hypotheses that are more specific.
- Every member of the G-set is consistent with all observations so far, and there are no consistent hypotheses that are more general.

We want the initial version space (before any examples have been seen) to represent all possible hypotheses. We do this by setting the G-set to contain *True* (the hypothesis that contains everything), and the S-set to contain *False* (the hypothesis whose extension is empty).

Figure 19.4 shows the general structure of the boundary-set representation of the version space. To show that the representation is sufficient, we need the following two properties:
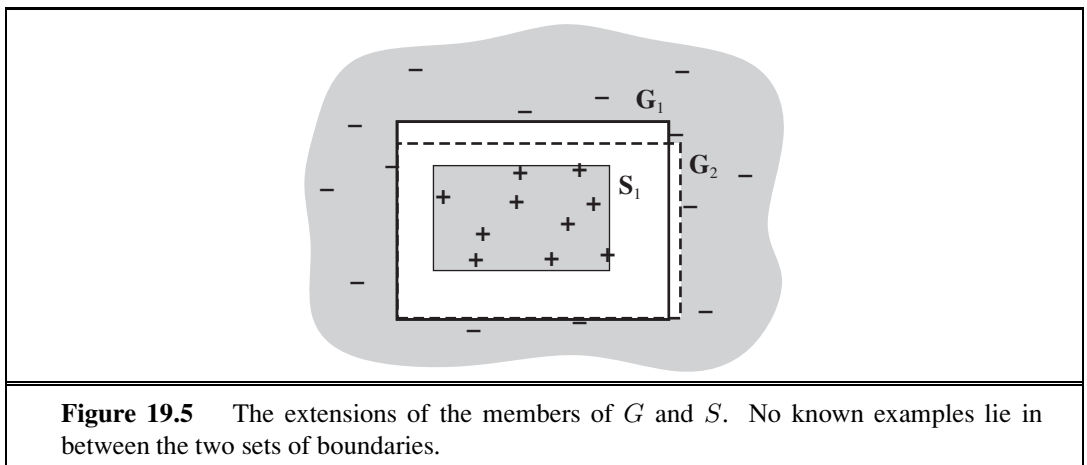
BOUNDARY SET
G-SET
S-SET

1. Every consistent hypothesis (other than those in the boundary sets) is more specific than some member of the G-set, and more general than some member of the S-set. (That is, there are no "stragglers" left outside.) This follows directly from the definitions of S and G. If there were a straggler h, then it would have to be no more specific than any member of G, in which case it belongs in G; or no more general than any member of S, in which case it belongs in S.

2. Every hypothesis more specific than some member of the G-set and more general than some member of the S-set is a consistent hypothesis. (That is, there are no "holes" between the boundaries.) Any h between S and G must reject all the negative examples rejected by each member of G (because it is more specific), and must accept all the positive examples accepted by any member of S (because it is more general). Thus, h must agree with all the examples, and therefore cannot be inconsistent. Figure 19.5 shows the situation: there are no known examples outside S but inside G, so any hypothesis in the gap must be consistent.

We have therefore shown that *if* S and G are maintained according to their definitions, then they provide a satisfactory representation of the version space. The only remaining problem is how to *update* S and G for a new example (the job of the VERSION-SPACE-UPDATE function). This may appear rather complicated at first, but from the definitions and with the help of Figure 19.4, it is not too hard to reconstruct the algorithm.

Figure 19.5   The extensions of the members of G and S. No known examples lie in between the two sets of boundaries.

We need to worry about the members S_i and G_i of the S- and G-sets. For each one, the new example may be a false positive or a false negative.

1. False positive for S_i: This means S_i is too general, but there are no consistent specializations of S_i (by definition), so we throw it out of the S-set.

2. False negative for S_i: This means S_i is too specific, so we replace it by all its immediate generalizations, provided they are more specific than some member of G.

3. False positive for G_i: This means G_i is too general, so we replace it by all its immediate specializations, provided they are more general than some member of S.

4. False negative for G_i: This means G_i is too specific, but there are no consistent generalizations of G_i (by definition) so we throw it out of the G-set.

We continue these operations for each new example until one of three things happens:

1. We have exactly one hypothesis left in the version space, in which case we return it as the unique hypothesis.

2. The version space *collapses*—either S or G becomes empty, indicating that there are no consistent hypotheses for the training set. This is the same case as the failure of the simple version of the decision tree algorithm.

3. We run out of examples and have several hypotheses remaining in the version space. This means the version space represents a disjunction of hypotheses. For any new example, if all the disjuncts agree, then we can return their classification of the example. If they disagree, one possibility is to take the majority vote.
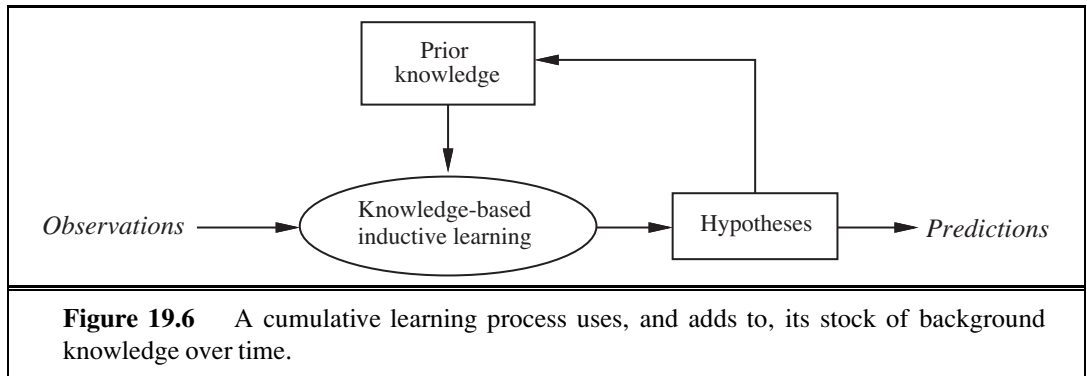
We leave as an exercise the application of the VERSION-SPACE-LEARNING algorithm to the restaurant data.

There are two principal drawbacks to the version-space approach:

- If the domain contains noise or insufficient attributes for exact classification, the version space will always collapse.

- If we allow unlimited disjunction in the hypothesis space, the S-set will always contain a single most-specific hypothesis, namely, the disjunction of the descriptions of the positive examples seen to date. Similarly, the G-set will contain just the negation of the disjunction of the descriptions of the negative examples.

- For some hypothesis spaces, the number of elements in the S-set or G-set may grow exponentially in the number of attributes, even though efficient learning algorithms exist for those hypothesis spaces.

GENERALIZATION
HIERARCHY

To date, no completely successful solution has been found for the problem of noise. The problem of disjunction can be addressed by allowing only limited forms of disjunction or by including a **generalization hierarchy** of more general predicates. For example, instead of using the disjunction $WaitEstimate(x, \textit{30-60}) \vee WaitEstimate(x, >60)$, we might use the single literal $LongWait(x)$. The set of generalization and specialization operations can be easily extended to handle this.

The pure version space algorithm was first applied in the Meta-DENDRAL system, which was designed to learn rules for predicting how molecules would break into pieces in a mass spectrometer (Buchanan and Mitchell, 1978). Meta-DENDRAL was able to generate rules that were sufficiently novel to warrant publication in a journal of analytical chemistry— the first real scientific knowledge generated by a computer program. It was also used in the elegant LEX system (Mitchell *et al.*, 1983), which was able to learn to solve symbolic integration problems by studying its own successes and failures. Although version space methods are probably not practical in most real-world learning problems, mainly because of noise, they provide a good deal of insight into the logical structure of hypothesis space.

Figure 19.6   A cumulative learning process uses, and adds to, its stock of background knowledge over time.

19.2 KNOWLEDGE IN LEARNING

The preceding section described the simplest setting for inductive learning. To understand the role of prior knowledge, we need to talk about the logical relationships among hypotheses, example descriptions, and classifications. Let $Descriptions$ denote the conjunction of all the example descriptions in the training set, and let $Classifications$ denote the conjunction of all the example classifications. Then a $Hypothesis$ that "explains the observations" must satisfy the following property (recall that $\models$ means "logically entails"):

$$Hypothesis \wedge Descriptions \models Classifications . \tag{19.3}$$

ENTAILMENT
CONSTRAINT

We call this kind of relationship an **entailment constraint**, in which $Hypothesis$ is the "unknown." Pure inductive learning means solving this constraint, where $Hypothesis$ is drawn from some predefined hypothesis space. For example, if we consider a decision tree as a logical formula (see Equation (19.1) on page 769), then a decision tree that is consistent with all the examples will satisfy Equation (19.3). If we place *no* restrictions on the logical form of the hypothesis, of course, then $Hypothesis = Classifications$ also satisfies the constraint. Ockham's razor tells us to prefer *small*, consistent hypotheses, so we try to do better than simply memorizing the examples.

This simple knowledge-free picture of inductive learning persisted until the early 1980s. The modern approach is to design agents that *already know something* and are trying to learn some more. This may not sound like a terrifically deep insight, but it makes quite a difference to the way we design agents. It might also have some relevance to our theories about how science itself works. The general idea is shown schematically in Figure 19.6.

An autonomous learning agent that uses background knowledge must somehow obtain the background knowledge in the first place, in order for it to be used in the new learning episodes. This method must itself be a learning process. The agent's life history will therefore be characterized by *cumulative*, or *incremental*, development. Presumably, the agent could start out with nothing, performing inductions *in vacuo* like a good little pure induction program. But once it has eaten from the Tree of Knowledge, it can no longer pursue such naive speculations and should use its background knowledge to learn more and more effectively. The question is then how to actually do this.

19.2.1   Some simple examples

Let us consider some commonsense examples of learning with background knowledge. Many apparently rational cases of inferential behavior in the face of observations clearly do not follow the simple principles of pure induction.

- Sometimes one leaps to general conclusions after only one observation. Gary Larson once drew a cartoon in which a bespectacled caveman, Zog, is roasting his lizard on the end of a pointed stick. He is watched by an amazed crowd of his less intellectual contemporaries, who have been using their bare hands to hold their victuals over the fire. This enlightening experience is enough to convince the watchers of a general principle of painless cooking.

- Or consider the case of the traveler to Brazil meeting her first Brazilian. On hearing him speak Portuguese, she immediately concludes that Brazilians speak Portuguese, yet on discovering that his name is Fernando, she does not conclude that all Brazilians are called Fernando. Similar examples appear in science. For example, when a freshman physics student measures the density and conductance of a sample of copper at a particular temperature, she is quite confident in generalizing those values to all pieces of copper. Yet when she measures its mass, she does not even consider the hypothesis that all pieces of copper have that mass. On the other hand, it would be quite reasonable to make such a generalization over all pennies.

- Finally, consider the case of a pharmacologically ignorant but diagnostically sophisticated medical student observing a consulting session between a patient and an expert internist. After a series of questions and answers, the expert tells the patient to take a course of a particular antibiotic. The medical student infers the general rule that that particular antibiotic is effective for a particular type of infection.

These are all cases in which *the use of background knowledge allows much faster learning than one might expect from a pure induction program.*

19.2.2   Some general schemes

In each of the preceding examples, one can appeal to prior knowledge to try to justify the generalizations chosen. We will now look at what kinds of entailment constraints are operating in each case. The constraints will involve the *Background* knowledge, in addition to the *Hypothesis* and the observed *Descriptions* and *Classifications*.

In the case of lizard toasting, the cavemen generalize by *explaining* the success of the pointed stick: it supports the lizard while keeping the hand away from the fire. From this explanation, they can infer a general rule: that any long, rigid, sharp object can be used to toast small, soft-bodied edibles. This kind of generalization process has been called **explanation-based learning**, or **EBL**. Notice that the general rule *follows logically* from the background knowledge possessed by the cavemen. Hence, the entailment constraints satisfied by EBL are the following:

EXPLANATION-BASED LEARNING

$$Hypothesis \land Descriptions \models Classifications$$
$$Background \models Hypothesis \; .$$

Because EBL uses Equation (19.3), it was initially thought to be a way to learn from examples. But because it requires that the background knowledge be sufficient to explain the *Hypothesis*, which in turn explains the observations, *the agent does not actually learn anything factually new from the example.* The agent *could have* derived the example from what it already knew, although that might have required an unreasonable amount of computation. EBL is now viewed as a method for converting first-principles theories into useful, special-purpose knowledge. We describe algorithms for EBL in Section 19.3.

The situation of our traveler in Brazil is quite different, for she cannot necessarily explain why Fernando speaks the way he does, unless she knows her papal bulls. Moreover, the same generalization would be forthcoming from a traveler entirely ignorant of colonial history. The relevant prior knowledge in this case is that, within any given country, most people tend to speak the same language; on the other hand, Fernando is not assumed to be the name of all Brazilians because this kind of regularity does not hold for names. Similarly, the freshman physics student also would be hard put to explain the particular values that she discovers for the conductance and density of copper. She does know, however, that the material of which an object is composed and its temperature together determine its conductance. In each case, the prior knowledge *Background* concerns the **relevance** of a set of features to the goal predicate. This knowledge, *together with the observations*, allows the agent to infer a new, general rule that explains the observations:

$$Hypothesis \wedge Descriptions \models Classifications \ ,$$
$$Background \wedge Descriptions \wedge Classifications \models Hypothesis \ .$$
$$(19.4)$$

RELEVANCE

RELEVANCE-BASED LEARNING

We call this kind of generalization **relevance-based learning**, or **RBL** (although the name is not standard). Notice that whereas RBL does make use of the content of the observations, it does not produce hypotheses that go beyond the logical content of the background knowledge and the observations. It is a *deductive* form of learning and cannot by itself account for the creation of new knowledge starting from scratch.

In the case of the medical student watching the expert, we assume that the student's prior knowledge is sufficient to infer the patient's disease D from the symptoms. This is not, however, enough to explain the fact that the doctor prescribes a particular medicine M. The student needs to propose another rule, namely, that M generally is effective against D. Given this rule and the student's prior knowledge, the student can now explain why the expert prescribes M in this particular case. We can generalize this example to come up with the entailment constraint

$$Background \wedge Hypothesis \wedge Descriptions \models Classifications \ .$$
$$(19.5)$$

That is, *the background knowledge and the new hypothesis combine to explain the examples.* As with pure inductive learning, the learning algorithm should propose hypotheses that are as simple as possible, consistent with this constraint. Algorithms that satisfy constraint (19.5) are called **knowledge-based inductive learning**, or **KBIL**, algorithms.

KNOWLEDGE-BASED INDUCTIVE LEARNING

INDUCTIVE LOGIC PROGRAMMING

KBIL algorithms, which are described in detail in Section 19.5, have been studied mainly in the field of **inductive logic programming**, or **ILP**. In ILP systems, prior knowledge plays two key roles in reducing the complexity of learning:

1. Because any hypothesis generated must be consistent with the prior knowledge as well as with the new observations, the effective hypothesis space size is reduced to include only those theories that are consistent with what is already known.

2. For any given set of observations, the size of the hypothesis required to construct an explanation for the observations can be much reduced, because the prior knowledge will be available to help out the new rules in explaining the observations. The smaller the hypothesis, the easier it is to find.

In addition to allowing the use of prior knowledge in induction, ILP systems can formulate hypotheses in general first-order logic, rather than in the restricted attribute-based language of Chapter 18. This means that they can learn in environments that cannot be understood by simpler systems.

19.3   EXPLANATION-BASED LEARNING

Explanation-based learning is a method for extracting general rules from individual observations. As an example, consider the problem of differentiating and simplifying algebraic expressions (Exercise 9.17). If we differentiate an expression such as X^2 with respect to X, we obtain $2X$. (We use a capital letter for the arithmetic unknown X, to distinguish it from the logical variable x.) In a logical reasoning system, the goal might be expressed as ASK($Derivative(X^2, X) = d$, KB), with solution $d = 2X$.

Anyone who knows differential calculus can see this solution "by inspection" as a result of practice in solving such problems. A student encountering such problems for the first time, or a program with no experience, will have a much more difficult job. Application of the standard rules of differentiation eventually yields the expression $1 \times (2 \times (X^{(2-1)}))$, and eventually this simplifies to $2X$. In the authors' logic programming implementation, this takes 136 proof steps, of which 99 are on dead-end branches in the proof. After such an experience, we would like the program to solve the same problem much more quickly the next time it arises.

MEMOIZATION   The technique of **memoization** has long been used in computer science to speed up programs by saving the results of computation. The basic idea of memo functions is to accumulate a database of input–output pairs; when the function is called, it first checks the database to see whether it can avoid solving the problem from scratch. Explanation-based learning takes this a good deal further, by creating *general* rules that cover an entire class of cases. In the case of differentiation, memoization would remember that the derivative of X^2 with respect to X is $2X$, but would leave the agent to calculate the derivative of Z^2 with respect to Z from scratch. We would like to be able to extract the general rule that for any arithmetic unknown u, the derivative of u^2 with respect to u is $2u$. (An even more general rule for u^n can also be produced, but the current example suffices to make the point.) In logical terms, this is expressed by the rule

$$ArithmeticUnknown(u) \;\Rightarrow\; Derivative(u^2, u) = 2u \,.$$

If the knowledge base contains such a rule, then any new case that is an instance of this rule can be solved immediately.

This is, of course, merely a trivial example of a very general phenomenon. Once something is understood, it can be generalized and reused in other circumstances. It becomes an "obvious" step and can then be used as a building block in solving problems still more complex. Alfred North Whitehead (1911), co-author with Bertrand Russell of *Principia Mathematica*, wrote *"Civilization advances by extending the number of important operations that we can do without thinking about them,"* perhaps himself applying EBL to his understanding of events such as Zog's discovery. If you have understood the basic idea of the differentiation example, then your brain is already busily trying to extract the general principles of explanation-based learning from it. Notice that you hadn't *already* invented EBL before you saw the example. Like the cavemen watching Zog, you (and we) needed an example before we could generate the basic principles. This is because *explaining why* something is a good idea is much easier than coming up with the idea in the first place.

19.3.1   Extracting general rules from examples

The basic idea behind EBL is first to construct an explanation of the observation using prior knowledge, and then to establish a definition of the class of cases for which the same explanation structure can be used. This definition provides the basis for a rule covering all of the cases in the class. The "explanation" can be a logical proof, but more generally it can be any reasoning or problem-solving process whose steps are well defined. The key is to be able to identify the necessary conditions for those same steps to apply to another case.

We will use for our reasoning system the simple backward-chaining theorem prover described in Chapter 9. The proof tree for $Derivative(X^2, X) = 2X$ is too large to use as an example, so we will use a simpler problem to illustrate the generalization method. Suppose our problem is to simplify $1 \times (0 + X)$. The knowledge base includes the following rules:

$$Rewrite(u, v) \land Simplify(v, w) \;\Rightarrow\; Simplify(u, w) \;.$$
$$Primitive(u) \;\Rightarrow\; Simplify(u, u) \;.$$
$$ArithmeticUnknown(u) \;\Rightarrow\; Primitive(u) \;.$$
$$Number(u) \;\Rightarrow\; Primitive(u) \;.$$
$$Rewrite(1 \times u, u) \;.$$
$$Rewrite(0 + u, u) \;.$$
$$\vdots$$

The proof that the answer is X is shown in the top half of Figure 19.7. The EBL method actually constructs two proof trees simultaneously. The second proof tree uses a *variabilized* goal in which the constants from the original goal are replaced by variables. As the original proof proceeds, the variabilized proof proceeds in step, using *exactly the same rule applications*. This could cause some of the variables to become instantiated. For example, in order to use the rule $Rewrite(1 \times u, u)$, the variable x in the subgoal $Rewrite(x \times (y + z), v)$ must be bound to 1. Similarly, y must be bound to 0 in the subgoal $Rewrite(y + z, v')$ in order to use the rule $Rewrite(0 + u, u)$. Once we have the generalized proof tree, we take the leaves
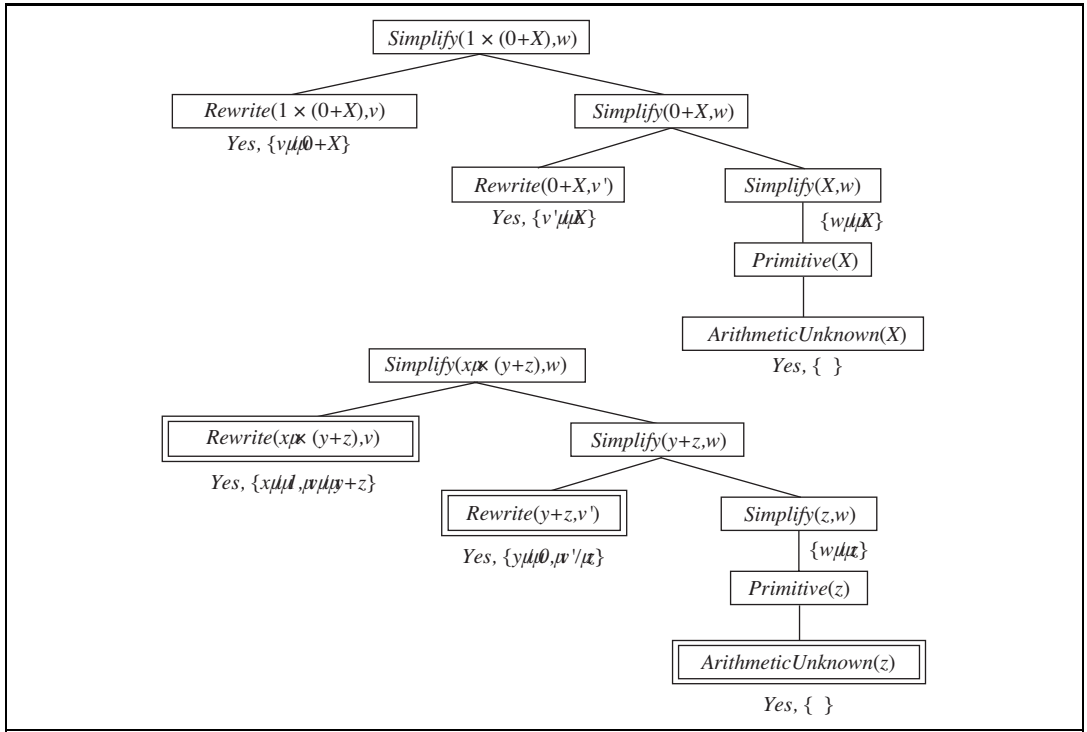
$Simplify(1 \times (0+X),w)$

$Rewrite(1 \times (0+X),v)$
$Yes, \{v/\mu/0+X\}$

$Simplify(0+X,w)$

$Rewrite(0+X,v')$
$Yes, \{v'/\mu/X\}$

$Simplify(X,w)$
$\{w/\mu/X\}$

$Primitive(X)$

$ArithmeticUnknown(X)$
$Yes, \{ \ \}$

$Simplify(x/\mu \times (y+z),w)$

$Rewrite(x/\mu \times (y+z),v)$
$Yes, \{x/\mu/l ,v/\mu/y+z\}$

$Simplify(y+z,w)$

$Rewrite(y+z,v')$
$Yes, \{y/\mu/0 ,v'/\mu z\}$

$Simplify(z,w)$
$\{w/\mu z\}$

$Primitive(z)$

$ArithmeticUnknown(z)$
$Yes, \{ \ \}$

Figure 19.7   Proof trees for the simplification problem. The first tree shows the proof for
the original problem instance, from which we can derive

$$ArithmeticUnknown(z) \ \Rightarrow \ Simplify(1 \times (0+z), z) \ .$$

The second tree shows the proof for a problem instance with all constants replaced by vari-
ables, from which we can derive a variety of other rules.

(with the necessary bindings) and form a general rule for the goal predicate:

$$Rewrite(1 \times (0+z), 0+z) \land Rewrite(0+z, z) \land ArithmeticUnknown(z)$$
$$\Rightarrow \ Simplify(1 \times (0+z), z) \ .$$

Notice that the first two conditions on the left-hand side are true *regardless of the value of z*.
We can therefore drop them from the rule, yielding

$$ArithmeticUnknown(z) \ \Rightarrow \ Simplify(1 \times (0+z), z) \ .$$

In general, conditions can be dropped from the final rule if they impose no constraints on the
variables on the right-hand side of the rule, because the resulting rule will still be true and
will be more efficient. Notice that we cannot drop the condition $ArithmeticUnknown(z)$,
because not all possible values of z are arithmetic unknowns. Values other than arithmetic
unknowns might require different forms of simplification: for example, if z were 2×3, then
the correct simplification of $1 \times (0 + (2 \times 3))$ would be 6 and not 2×3.

    To recap, the basic EBL process works as follows:

1. Given an example, construct a proof that the goal predicate applies to the example using
   the available background knowledge.

2. In parallel, construct a generalized proof tree for the variabilized goal using the same inference steps as in the original proof.

3. Construct a new rule whose left-hand side consists of the leaves of the proof tree and whose right-hand side is the variabilized goal (after applying the necessary bindings from the generalized proof).

4. Drop any conditions from the left-hand side that are true regardless of the values of the variables in the goal.

19.3.2   Improving efficiency

The generalized proof tree in Figure 19.7 actually yields more than one generalized rule. For example, if we terminate, or **prune**, the growth of the right-hand branch in the proof tree when it reaches the $Primitive$ step, we get the rule

$$Primitive(z) \ \Rightarrow \ Simplify(1 \times (0 + z), z) \ .$$

This rule is as valid as, but *more general* than, the rule using $ArithmeticUnknown$, because it covers cases where z is a number. We can extract a still more general rule by pruning after the step $Simplify(y + z, w)$, yielding the rule

$$Simplify(y + z, w) \ \Rightarrow \ Simplify(1 \times (y + z), w) \ .$$

In general, a rule can be extracted from *any partial subtree* of the generalized proof tree. Now we have a problem: which of these rules do we choose?

The choice of which rule to generate comes down to the question of efficiency. There are three factors involved in the analysis of efficiency gains from EBL:

1. Adding large numbers of rules can slow down the reasoning process, because the inference mechanism must still check those rules even in cases where they do not yield a solution. In other words, it increases the **branching factor** in the search space.

2. To compensate for the slowdown in reasoning, the derived rules must offer significant increases in speed for the cases that they do cover. These increases come about mainly because the derived rules avoid dead ends that would otherwise be taken, but also because they shorten the proof itself.

3. Derived rules should be as general as possible, so that they apply to the largest possible set of cases.

OPERATIONALITY

A common approach to ensuring that derived rules are efficient is to insist on the **operationality** of each subgoal in the rule. A subgoal is operational if it is "easy" to solve. For example, the subgoal $Primitive(z)$ is easy to solve, requiring at most two steps, whereas the subgoal $Simplify(y + z, w)$ could lead to an arbitrary amount of inference, depending on the values of y and z. If a test for operationality is carried out at each step in the construction of the generalized proof, then we can prune the rest of a branch as soon as an operational subgoal is found, keeping just the operational subgoal as a conjunct of the new rule.

Unfortunately, there is usually a tradeoff between operationality and generality. More specific subgoals are generally easier to solve but cover fewer cases. Also, operationality is a matter of degree: one or two steps is definitely operational, but what about 10 or 100?

Finally, the cost of solving a given subgoal depends on what other rules are available in the knowledge base. It can go up or down as more rules are added. Thus, EBL systems really face a very complex optimization problem in trying to maximize the efficiency of a given initial knowledge base. It is sometimes possible to derive a mathematical model of the effect on overall efficiency of adding a given rule and to use this model to select the best rule to add. The analysis can become very complicated, however, especially when recursive rules are involved. One promising approach is to address the problem of efficiency empirically, simply by adding several rules and seeing which ones are useful and actually speed things up.

Empirical analysis of efficiency is actually at the heart of EBL. What we have been calling loosely the "efficiency of a given knowledge base" is actually the average-case complexity on a distribution of problems. *By generalizing from past example problems, EBL makes the knowledge base more efficient for the kind of problems that it is reasonable to expect.* This works as long as the distribution of past examples is roughly the same as for future examples—the same assumption used for PAC-learning in Section 18.5. If the EBL system is carefully engineered, it is possible to obtain significant speedups. For example, a very large Prolog-based natural language system designed for speech-to-speech translation between Swedish and English was able to achieve real-time performance only by the application of EBL to the parsing process (Samuelsson and Rayner, 1991).

19.4   LEARNING USING RELEVANCE INFORMATION

Our traveler in Brazil seems to be able to make a confident generalization concerning the language spoken by other Brazilians. The inference is sanctioned by her background knowledge, namely, that people in a given country (usually) speak the same language. We can express this in first-order logic as follows:[2]

$$Nationality(x, n) \land Nationality(y, n) \land Language(x, l) \Rightarrow Language(y, l) \text{ . (19.6)}$$

(Literal translation: "If x and y have the same nationality n and x speaks language l, then y also speaks it.") It is not difficult to show that, from this sentence and the observation that

$$Nationality(Fernando, Brazil) \land Language(Fernando, Portuguese) \text{ ,}$$

the following conclusion is entailed (see Exercise 19.1):

$$Nationality(x, Brazil) \Rightarrow Language(x, Portuguese) \text{ .}$$

Sentences such as (19.6) express a strict form of relevance: given nationality, language is fully determined. (Put another way: language is a function of nationality.) These sentences **FUNCTIONAL DEPENDENCY** are called **functional dependencies** or **determinations**. They occur so commonly in certain **DETERMINATION** kinds of applications (e.g., defining database designs) that a special syntax is used to write them. We adopt the notation of Davies (1985):

$$Nationality(x, n) \succ Language(x, l) \text{ .}$$

---

[2]   We assume for the sake of simplicity that a person speaks only one language. Clearly, the rule would have to be amended for countries such as Switzerland and India.

As usual, this is simply a syntactic sugaring, but it makes it clear that the determination is really a relationship between the predicates: nationality determines language. The relevant properties determining conductance and density can be expressed similarly:

$$Material(x, m) \land Temperature(x, t) \succ Conductance(x, \rho) \; ;$$
$$Material(x, m) \land Temperature(x, t) \succ Density(x, d) \; .$$

The corresponding generalizations follow logically from the determinations and observations.

19.4.1   Determining the hypothesis space

Although the determinations sanction general conclusions concerning all Brazilians, or all pieces of copper at a given temperature, they cannot, of course, yield a general predictive theory for *all* nationalities, or for *all* temperatures and materials, from a single example. Their main effect can be seen as limiting the space of hypotheses that the learning agent need consider. In predicting conductance, for example, one need consider only material and temperature and can ignore mass, ownership, day of the week, the current president, and so on. Hypotheses can certainly include terms that are in turn determined by material and temperature, such as molecular structure, thermal energy, or free-electron density. *Determinations specify a sufficient basis vocabulary from which to construct hypotheses concerning the target predicate.* This statement can be proven by showing that a given determination is logically equivalent to a statement that the correct definition of the target predicate is one of the set of all definitions expressible using the predicates on the left-hand side of the determination.

Intuitively, it is clear that a reduction in the hypothesis space size should make it easier to learn the target predicate. Using the basic results of computational learning theory (Section 18.5), we can quantify the possible gains. First, recall that for Boolean functions, $\log(|\mathcal{H}|)$ examples are required to converge to a reasonable hypothesis, where $|\mathcal{H}|$ is the size of the hypothesis space. If the learner has n Boolean features with which to construct hypotheses, then, in the absence of further restrictions, $|\mathcal{H}| = O(2^{2^n})$, so the number of examples is $O(2^n)$. If the determination contains d predicates in the left-hand side, the learner will require only $O(2^d)$ examples, a reduction of $O(2^{n-d})$.

19.4.2   Learning and using relevance information

As we stated in the introduction to this chapter, prior knowledge is useful in learning; but it too has to be learned. In order to provide a complete story of relevance-based learning, we must therefore provide a learning algorithm for determinations. The learning algorithm we now present is based on a straightforward attempt to find the simplest determination consistent with the observations. A determination $P \succ Q$ says that if any examples match on P, then they must also match on Q. A determination is therefore consistent with a set of examples if every pair that matches on the predicates on the left-hand side also matches on the goal predicate. For example, suppose we have the following examples of conductance measurements on material samples:

---

function MINIMAL-CONSISTENT-DET(E, A) **returns** a set of attributes
   inputs: E, a set of examples
          A, a set of attributes, of size n

   for $i = 0$ **to** n **do**
      for each subset A_i of A of size i **do**
         if CONSISTENT-DET?(A_i, E) **then return** A_i

---

function CONSISTENT-DET?(A, E) **returns** a truth value
   inputs: A, a set of attributes
          E, a set of examples
   local variables: H, a hash table

   for each example e **in** E **do**
      if some example in H has the same values as e for the attributes A
        but a different classification **then return** $false$
      store the class of e in H, indexed by the values for attributes A of the example e
   return $true$

Figure 19.8   An algorithm for finding a minimal consistent determination.

Sample	Mass	Temperature	Material	Size	Conductance
S1	12	26	Copper	3	0.59
S1	12	100	Copper	3	0.57
S2	24	26	Copper	6	0.59
S3	12	26	Lead	2	0.05
S3	12	100	Lead	2	0.04
S4	24	26	Lead	4	0.05

The minimal consistent determination is $Material \wedge Temperature \succ Conductance$. There is a nonminimal but consistent determination, namely, $Mass \wedge Size \wedge Temperature \succ Conductance$. This is consistent with the examples because mass and size determine density and, in our data set, we do not have two different materials with the same density. As usual, we would need a larger sample set in order to eliminate a nearly correct hypothesis.

     There are several possible algorithms for finding minimal consistent determinations. The most obvious approach is to conduct a search through the space of determinations, checking all determinations with one predicate, two predicates, and so on, until a consistent determination is found. We will assume a simple attribute-based representation, like that used for decision tree learning in Chapter 18. A determination d will be represented by the set of attributes on the left-hand side, because the target predicate is assumed to be fixed. The basic algorithm is outlined in Figure 19.8.

     The time complexity of this algorithm depends on the size of the smallest consistent determination. Suppose this determination has p attributes out of the n total attributes. Then the algorithm will not find it until searching the subsets of A of size p. There are $\binom{n}{p} = O(n^p)$
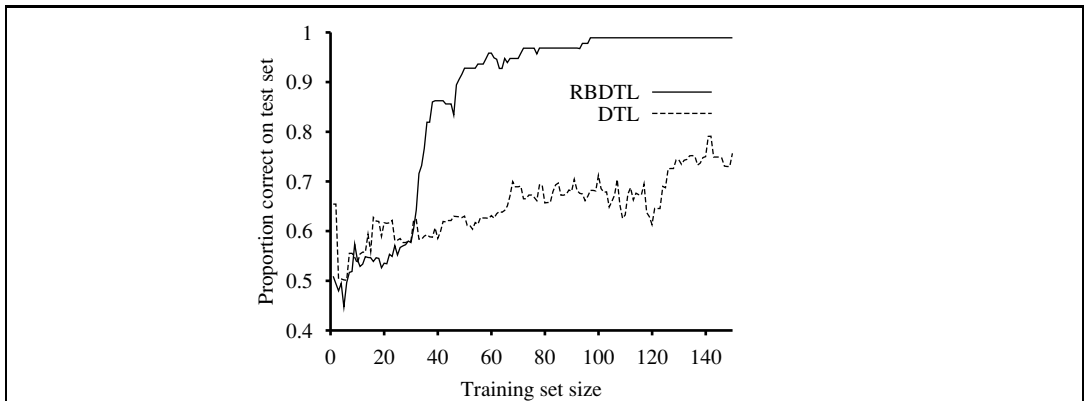
Figure 19.9   A performance comparison between DECISION-TREE-LEARNING and RBDTL on randomly generated data for a target function that depends on only 5 of 16 attributes.

such subsets; hence the algorithm is exponential in the size of the minimal determination. It turns out that the problem is NP-complete, so we cannot expect to do better in the general case. In most domains, however, there will be sufficient local structure (see Chapter 14 for a definition of locally structured domains) that p will be small.

Given an algorithm for learning determinations, a learning agent has a way to construct a minimal hypothesis within which to learn the target predicate. For example, we can combine MINIMAL-CONSISTENT-DET with the DECISION-TREE-LEARNING algorithm. This yields a relevance-based decision-tree learning algorithm RBDTL that first identifies a minimal set of relevant attributes and then passes this set to the decision tree algorithm for learning. Unlike DECISION-TREE-LEARNING, RBDTL simultaneously learns and uses relevance information in order to minimize its hypothesis space. We expect that RBDTL will learn faster than DECISION-TREE-LEARNING, and this is in fact the case. Figure 19.9 shows the learning performance for the two algorithms on randomly generated data for a function that depends on only 5 of 16 attributes. Obviously, in cases where all the available attributes are relevant, RBDTL will show no advantage.

DECLARATIVE BIAS   This section has only scratched the surface of the field of **declarative bias**, which aims to understand how prior knowledge can be used to identify the appropriate hypothesis space within which to search for the correct target definition. There are many unanswered questions:

- How can the algorithms be extended to handle noise?
- Can we handle continuous-valued variables?
- How can other kinds of prior knowledge be used, besides determinations?
- How can the algorithms be generalized to cover any first-order theory, rather than just an attribute-based representation?

Some of these questions are addressed in the next section.

19.5   INDUCTIVE LOGIC PROGRAMMING

Inductive logic programming (ILP) combines inductive methods with the power of first-order representations, concentrating in particular on the representation of hypotheses as logic programs.[3] It has gained popularity for three reasons. First, ILP offers a rigorous approach to the general knowledge-based inductive learning problem. Second, it offers complete algorithms for inducing general, first-order theories from examples, which can therefore learn successfully in domains where attribute-based algorithms are hard to apply. An example is in learning how protein structures fold (Figure 19.10). The three-dimensional configuration of a protein molecule cannot be represented reasonably by a set of attributes, because the configuration inherently refers to *relationships* between objects, not to attributes of a single object. First-order logic is an appropriate language for describing the relationships. Third, inductive logic programming produces hypotheses that are (relatively) easy for humans to read. For example, the English translation in Figure 19.10 can be scrutinized and criticized by working biologists. This means that inductive logic programming systems can participate in the scientific cycle of experimentation, hypothesis generation, debate, and refutation. Such participation would not be possible for systems that generate "black-box" classifiers, such as neural networks.

19.5.1   An example

Recall from Equation (19.5) that the general knowledge-based induction problem is to "solve" the entailment constraint

$$Background \land Hypothesis \land Descriptions \models Classifications$$

for the unknown *Hypothesis*, given the *Background* knowledge and examples described by *Descriptions* and *Classifications*. To illustrate this, we will use the problem of learning family relationships from examples. The descriptions will consist of an extended family tree, described in terms of *Mother*, *Father*, and *Married* relations and *Male* and *Female* properties. As an example, we will use the family tree from Exercise 8.14, shown here in Figure 19.11. The corresponding descriptions are as follows:

$Father(Philip, Charles)$ $Father(Philip, Anne)$ $\ldots$
$Mother(Mum, Margaret)$ $Mother(Mum, Elizabeth)$ $\ldots$
$Married(Diana, Charles)$ $Married(Elizabeth, Philip)$ $\ldots$
$Male(Philip)$ $Male(Charles)$ $\ldots$
$Female(Beatrice)$ $Female(Margaret)$ $\ldots$

The sentences in *Classifications* depend on the target concept being learned. We might want to learn *Grandparent*, *BrotherInLaw*, or *Ancestor*, for example. For *Grandparent*, the

---

[3]   It might be appropriate at this point for the reader to refer to Chapter 7 for some of the underlying concepts, including Horn clauses, conjunctive normal form, unification, and resolution.

complete set of *Classifications* contains $20 \times 20 = 400$ conjuncts of the form

$Grandparent(Mum, Charles)$ $Grandparent(Elizabeth, Beatrice)$ ...
$\neg Grandparent(Mum, Harry)$ $\neg Grandparent(Spencer, Peter)$ ...

We could of course learn from a subset of this complete set.

The object of an inductive learning program is to come up with a set of sentences for the *Hypothesis* such that the entailment constraint is satisfied. Suppose, for the moment, that the agent has no background knowledge: *Background* is empty. Then one possible solution

H:5[111-113]

H:1[19-37]

H:3[71-84]

H:4[61-64]

H:6[79-88]

H:1[8-17]

H:2[26-33]

H:5[66-70]

H:4[93-108]

E:2[96-98]
E:1[57-59]

H:2[41-64]

H:7[99-106]

H:3[40-50]

2mhr - Four-helical up-and-down bundle

1omd - EF-Hand

(a)   (b)
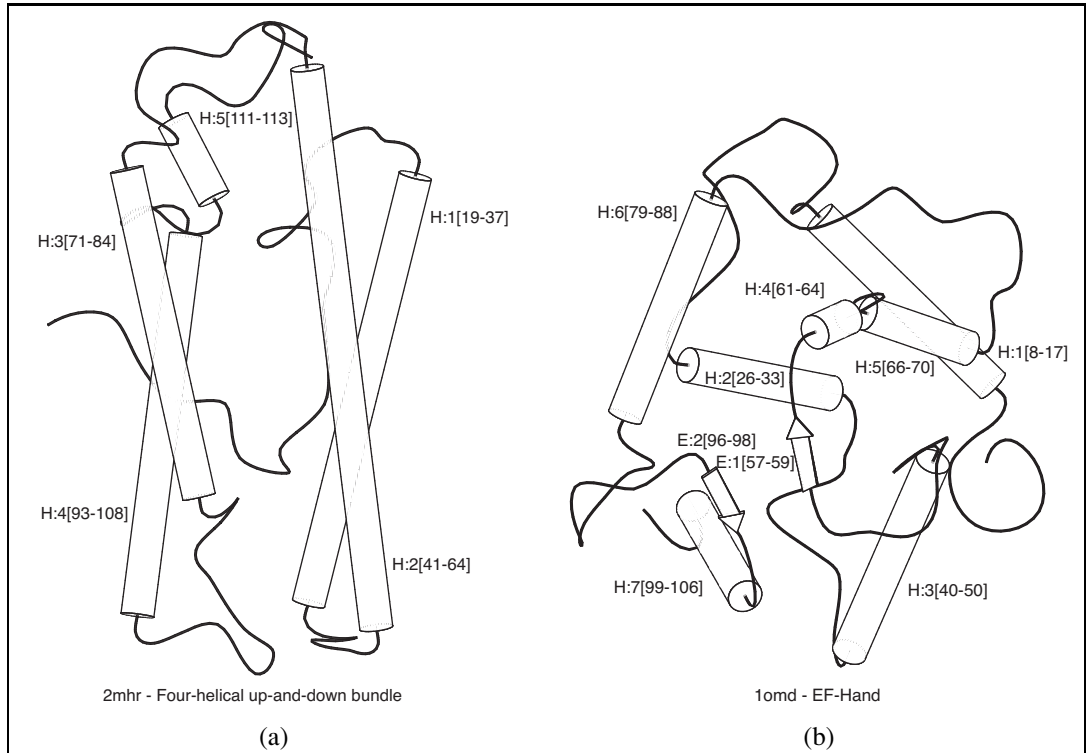
Figure 19.10   (a) and (b) show positive and negative examples, respectively, of the "four-helical up-and-down bundle" concept in the domain of protein folding.   Each example structure is coded into a logical expression of about 100 conjuncts such as $TotalLength(D2mhr, 118) \wedge NumberHelices(D2mhr, 6) \wedge \ldots$. From these descriptions and from classifications such as $Fold(\text{FOUR-HELICAL-UP-AND-DOWN-BUNDLE}, D2mhr)$, the ILP system PROGOL (Muggleton, 1995) learned the following rule:

$Fold(\text{FOUR-HELICAL-UP-AND-DOWN-BUNDLE}, p) \Leftarrow$
$\quad Helix(p, h_1) \wedge Length(h_1, \text{HIGH}) \wedge Position(p, h_1, n)$
$\quad \wedge (1 \leq n \leq 3) \wedge Adjacent(p, h_1, h_2) \wedge Helix(p, h_2)$.

This kind of rule could not be learned, or even represented, by an attribute-based mechanism such as we saw in previous chapters. The rule can be translated into English as " Protein p has fold class "Four-helical up-and-down-bundle" if it contains a long helix h_1 at a secondary structure position between 1 and 3 and h_1 is next to a second helix."

for *Hypothesis* is the following:

$$Grandparent(x, y) \quad \Leftrightarrow \quad [\exists z \;\; Mother(x, z) \wedge Mother(z, y)]$$
$$\vee \quad [\exists z \;\; Mother(x, z) \wedge Father(z, y)]$$
$$\vee \quad [\exists z \;\; Father(x, z) \wedge Mother(z, y)]$$
$$\vee \quad [\exists z \;\; Father(x, z) \wedge Father(z, y)] \;.$$

Notice that an attribute-based learning algorithm, such as DECISION-TREE-LEARNING, will get nowhere in solving this problem. In order to express *Grandparent* as an attribute (i.e., a unary predicate), we would need to make *pairs* of people into objects:

$$Grandparent(\langle Mum, Charles \rangle) \ldots$$

Then we get stuck in trying to represent the example descriptions. The only possible attributes are horrible things such as

$$FirstElementIsMotherOfElizabeth(\langle Mum, Charles \rangle) \;.$$

The definition of *Grandparent* in terms of these attributes simply becomes a large disjunction of specific cases that does not generalize to new examples at all. *Attribute-based learning algorithms are incapable of learning relational predicates.* Thus, one of the principal advantages of ILP algorithms is their applicability to a much wider range of problems, including relational problems.

The reader will certainly have noticed that a little bit of background knowledge would help in the representation of the *Grandparent* definition. For example, if *Background* included the sentence

$$Parent(x, y) \Leftrightarrow [Mother(x, y) \vee Father(x, y)] \;,$$

then the definition of *Grandparent* would be reduced to

$$Grandparent(x, y) \Leftrightarrow [\exists z \;\; Parent(x, z) \wedge Parent(z, y)] \;.$$

This shows how background knowledge can dramatically reduce the size of hypotheses required to explain the observations.

It is also possible for ILP algorithms to *create* new predicates in order to facilitate the expression of explanatory hypotheses. Given the example data shown earlier, it is entirely reasonable for the ILP program to propose an additional predicate, which we would call

Figure 19.11   A typical family tree.

CONSTRUCTIVE
INDUCTION
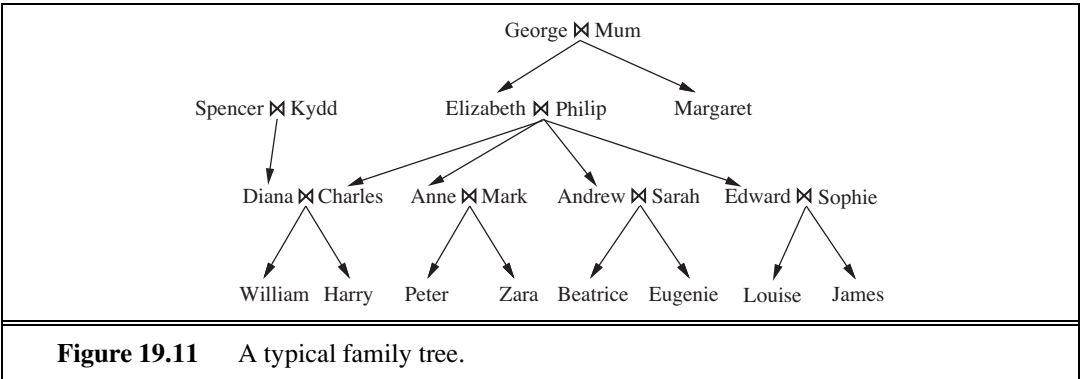
"*Parent*," in order to simplify the definitions of the target predicates. Algorithms that can generate new predicates are called **constructive induction** algorithms. Clearly, constructive induction is a necessary part of the picture of cumulative learning. It has been one of the hardest problems in machine learning, but some ILP techniques provide effective mechanisms for achieving it.

In the rest of this chapter, we will study the two principal approaches to ILP. The first uses a generalization of decision tree methods, and the second uses techniques based on inverting a resolution proof.

19.5.2   Top-down inductive learning methods

The first approach to ILP works by starting with a very general rule and gradually specializing it so that it fits the data. This is essentially what happens in decision-tree learning, where a decision tree is gradually grown until it is consistent with the observations. To do ILP we use first-order literals instead of attributes, and the hypothesis is a set of clauses instead of a decision tree. This section describes FOIL (Quinlan, 1990), one of the first ILP programs.

Suppose we are trying to learn a definition of the $Grandfather(x, y)$ predicate, using the same family data as before. As with decision-tree learning, we can divide the examples into positive and negative examples. Positive examples are

$\langle George, Anne \rangle$, $\langle Philip, Peter \rangle$, $\langle Spencer, Harry \rangle$, ...

and negative examples are

$\langle George, Elizabeth \rangle$, $\langle Harry, Zara \rangle$, $\langle Charles, Philip \rangle$, ...

Notice that each example is a *pair* of objects, because $Grandfather$ is a binary predicate. In all, there are 12 positive examples in the family tree and 388 negative examples (all the other pairs of people).

FOIL constructs a set of clauses, each with $Grandfather(x, y)$ as the head. The clauses must classify the 12 positive examples as instances of the $Grandfather(x, y)$ relationship, while ruling out the 388 negative examples. The clauses are Horn clauses, with the extension that negated literals are allowed in the body of a clause and are interpreted using negation as failure, as in Prolog. The initial clause has an empty body:

$$\Rightarrow Grandfather(x, y) .$$

This clause classifies every example as positive, so it needs to be specialized. We do this by adding literals one at a time to the left-hand side. Here are three potential additions:

$$Father(x, y) \Rightarrow Grandfather(x, y) .$$
$$Parent(x, z) \Rightarrow Grandfather(x, y) .$$
$$Father(x, z) \Rightarrow Grandfather(x, y) .$$

(Notice that we are assuming that a clause defining $Parent$ is already part of the background knowledge.) The first of these three clauses incorrectly classifies all of the 12 positive examples as negative and can thus be ignored. The second and third agree with all of the positive examples, but the second is incorrect on a larger fraction of the negative examples—twice as many, because it allows mothers as well as fathers. Hence, we prefer the third clause.

Now we need to specialize this clause further, to rule out the cases in which x is the father of some z, but z is not a parent of y. Adding the single literal $Parent(z, y)$ gives

$$Father(x, z) \wedge Parent(z, y) \ \Rightarrow \ Grandfather(x, y) \ ,$$

which correctly classifies all the examples. FOIL will find and choose this literal, thereby solving the learning task. In general, the solution is a set of Horn clauses, each of which implies the target predicate. For example, if we didn't have the *Parent* predicate in our vocabulary, then the solution might be

$$Father(x, z) \wedge Father(z, y) \ \Rightarrow \ Grandfather(x, y)$$
$$Father(x, z) \wedge Mother(z, y) \ \Rightarrow \ Grandfather(x, y) \ .$$

Note that each of these clauses covers some of the positive examples, that together they cover all the positive examples, and that NEW-CLAUSE is designed in such a way that no clause will incorrectly cover a negative example. In general FOIL will have to search through many unsuccessful clauses before finding a correct solution.

This example is a very simple illustration of how FOIL operates. A sketch of the complete algorithm is shown in Figure 19.12. Essentially, the algorithm repeatedly constructs a clause, literal by literal, until it agrees with some subset of the positive examples and none of the negative examples. Then the positive examples covered by the clause are removed from the training set, and the process continues until no positive examples remain. The two main subroutines to be explained are NEW-LITERALS, which constructs all possible new literals to add to the clause, and CHOOSE-LITERAL, which selects a literal to add.

NEW-LITERALS takes a clause and constructs all possible "useful" literals that could be added to the clause. Let us use as an example the clause

$$Father(x, z) \ \Rightarrow \ Grandfather(x, y) \ .$$

There are three kinds of literals that can be added:

1. *Literals using predicates*: the literal can be negated or unnegated, any existing predicate (including the goal predicate) can be used, and the arguments must all be variables. Any variable can be used for any argument of the predicate, with one restriction: each literal must include *at least one* variable from an earlier literal or from the head of the clause. Literals such as $Mother(z, u)$, $Married(z, z)$, $\neg Male(y)$, and $Grandfather(v, x)$ are allowed, whereas $Married(u, v)$ is not. Notice that the use of the predicate from the head of the clause allows FOIL to learn *recursive* definitions.

2. *Equality and inequality literals*: these relate variables already appearing in the clause. For example, we might add $z \neq x$. These literals can also include user-specified constants. For learning arithmetic we might use 0 and 1, and for learning list functions we might use the empty list $[\,]$.

3. *Arithmetic comparisons*: when dealing with functions of continuous variables, literals such as $x > y$ and $y \leq z$ can be added. As in decision-tree learning, a constant threshold value can be chosen to maximize the discriminatory power of the test.

The resulting branching factor in this search space is very large (see Exercise 19.6), but FOIL can also use type information to reduce it. For example, if the domain included numbers as

---

function FOIL(*examples*, *target*) **returns** a set of Horn clauses
   inputs: *examples*, set of examples
          target, a literal for the goal predicate
   local variables: *clauses*, set of clauses, initially empty

   while *examples* contains positive examples **do**
      clause ← NEW-CLAUSE(*examples*, *target*)
      remove positive examples covered by *clause* from *examples*
      add *clause* to *clauses*
   return *clauses*

---

function NEW-CLAUSE(*examples*, *target*) **returns** a Horn clause
   local variables: *clause*, a clause with *target* as head and an empty body
                  l, a literal to be added to the clause
                  extended_examples, a set of examples with values for new variables

   extended_examples ← *examples*
   while *extended_examples* contains negative examples **do**
      l ← CHOOSE-LITERAL(NEW-LITERALS(*clause*), *extended_examples*)
      append *l* to the body of *clause*
      extended_examples ← set of examples created by applying EXTEND-EXAMPLE
         to each example in *extended_examples*
   return *clause*

---

function EXTEND-EXAMPLE(*example*, *literal*) **returns** a set of examples
   if *example* satisfies *literal*
      then return the set of examples created by extending *example* with
        each possible constant value for each new variable in *literal*
   else return the empty set

Figure 19.12   Sketch of the FOIL algorithm for learning sets of first-order Horn clauses from examples. NEW-LITERALS and CHOOSE-LITERAL are explained in the text.

well as people, type restrictions would prevent NEW-LITERALS from generating literals such as $Parent(x, n)$, where x is a person and n is a number.

CHOOSE-LITERAL uses a heuristic somewhat similar to information gain (see page 704) to decide which literal to add. The exact details are not important here, and a number of different variations have been tried. One interesting additional feature of FOIL is the use of Ockham's razor to eliminate some hypotheses. If a clause becomes longer (according to some metric) than the total length of the positive examples that the clause explains, that clause is not considered as a potential hypothesis. This technique provides a way to avoid overcomplex clauses that fit noise in the data.

FOIL and its relatives have been used to learn a wide variety of definitions. One of the most impressive demonstrations (Quinlan and Cameron-Jones, 1993) involved solving a long sequence of exercises on list-processing functions from Bratko's (1986) Prolog textbook. In

each case, the program was able to learn a correct definition of the function from a small set of examples, using the previously learned functions as background knowledge.

19.5.3   Inductive learning with inverse deduction

The second major approach to ILP involves inverting the normal deductive proof process. **Inverse resolution** is based on the observation that if the example $Classifications$ follow from $Background \land Hypothesis \land Descriptions$, then one must be able to prove this fact by resolution (because resolution is complete). If we can "run the proof backward," then we can find a $Hypothesis$ such that the proof goes through. The key, then, is to find a way to invert the resolution process.

   We will show a backward proof process for inverse resolution that consists of individual backward steps. Recall that an ordinary resolution step takes two clauses C_1 and C_2 and resolves them to produce the **resolvent** C. An inverse resolution step takes a resolvent C and produces two clauses C_1 and C_2, such that C is the result of resolving C_1 and C_2. Alternatively, it may take a resolvent C and clause C_1 and produce a clause C_2 such that C is the result of resolving C_1 and C_2.

   The early steps in an inverse resolution process are shown in Figure 19.13, where we focus on the positive example $Grandparent(George, Anne)$. The process begins at the end of the proof (shown at the bottom of the figure). We take the resolvent C to be empty clause (i.e. a contradiction) and C_2 to be $\neg Grandparent(George, Anne)$, which is the negation of the goal example. The first inverse step takes C and C_2 and generates the clause $Grandparent(George, Anne)$ for C_1. The next step takes this clause as C and the clause $Parent(Elizabeth, Anne)$ as C_2, and generates the clause

$$\neg Parent(Elizabeth, y) \lor Grandparent(George, y)$$

as C_1. The final step treats this clause as the resolvent. With $Parent(George, Elizabeth)$ as C_2, one possible clause C_1 is the hypothesis

$$Parent(x, z) \land Parent(z, y) \ \Rightarrow \ Grandparent(x, y) \ .$$

Now we have a resolution proof that the hypothesis, descriptions, and background knowledge entail the classification $Grandparent(George, Anne)$.

   Clearly, inverse resolution involves a search. Each inverse resolution step is nondeterministic, because for any C, there can be many or even an infinite number of clauses C_1 and C_2 that resolve to C. For example, instead of choosing $\neg Parent(Elizabeth, y) \lor Grandparent(George, y)$ for C_1 in the last step of Figure 19.13, the inverse resolution step might have chosen any of the following sentences:

$$\neg Parent(Elizabeth, Anne) \lor Grandparent(George, Anne) \ .$$
$$\neg Parent(z, Anne) \lor Grandparent(George, Anne) \ .$$
$$\neg Parent(z, y) \lor Grandparent(George, y) \ .$$
$$\vdots$$

(See Exercises 19.4 and 19.5.) Furthermore, the clauses that participate in each step can be chosen from the $Background$ knowledge, from the example $Descriptions$, from the negated

Classifications, or from hypothesized clauses that have already been generated in the inverse resolution tree. The large number of possibilities means a large branching factor (and therefore an inefficient search) without additional controls. A number of approaches to taming the search have been tried in implemented ILP systems:

1. Redundant choices can be eliminated—for example, by generating only the most specific hypotheses possible and by requiring that all the hypothesized clauses be consistent with each other, and with the observations. This last criterion would rule out the clause $\neg Parent(z, y) \vee Grandparent(George, y)$, listed before.

2. The proof strategy can be restricted. For example, we saw in Chapter 9 that **linear resolution** is a complete, restricted strategy. Linear resolution produces proof trees that have a linear branching structure—the whole tree follows one line, with only single clauses branching off that line (as in Figure 19.13).

3. The representation language can be restricted, for example by eliminating function symbols or by allowing only Horn clauses. For instance, PROGOL operates with Horn clauses using **inverse entailment**. The idea is to change the entailment constraint

$$Background \wedge Hypothesis \wedge Descriptions \models Classifications$$

to the logically equivalent form

$$Background \wedge Descriptions \wedge \neg Classifications \models \neg Hypothesis.$$

From this, one can use a process similar to the normal Prolog Horn-clause deduction, with negation-as-failure to derive *Hypothesis*. Because it is restricted to Horn clauses, this is an incomplete method, but it can be more efficient than full resolution. It is also possible to apply complete inference with inverse entailment (Inoue, 2001).

4. Inference can be done with model checking rather than theorem proving. The PROGOL system (Muggleton, 1995) uses a form of model checking to limit the search. That
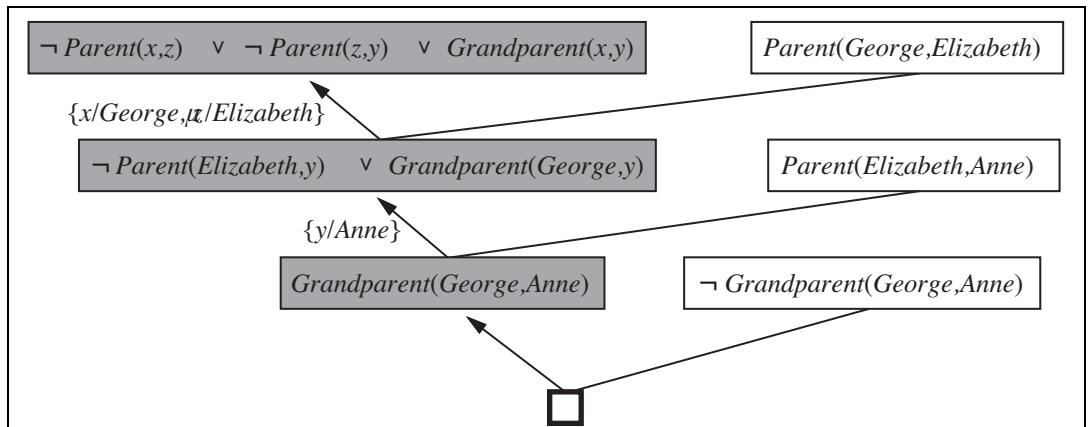
INVERSE
ENTAILMENT

Figure 19.13   Early steps in an inverse resolution process. The shaded clauses are generated by inverse resolution steps from the clause to the right and the clause below. The unshaded clauses are from the *Descriptions* and *Classifications* (including negated *Classifications*).

is, like answer set programming, it generates possible values for logical variables, and checks for consistency.

5. Inference can be done with ground propositional clauses rather than in first-order logic. The LINUS system (Lavrauc and Duzeroski, 1994) works by translating first-order theories into propositional logic, solving them with a propositional learning system, and then translating back. Working with propositional formulas can be more efficient on some problems, as we saw with SATPLAN in Chapter 10.

19.5.4   Making discoveries with inductive logic programming

An inverse resolution procedure that inverts a complete resolution strategy is, in principle, a complete algorithm for learning first-order theories. That is, if some unknown *Hypothesis* generates a set of examples, then an inverse resolution procedure can generate *Hypothesis* from the examples. This observation suggests an interesting possibility: Suppose that the available examples include a variety of trajectories of falling bodies. Would an inverse resolution program be theoretically capable of inferring the law of gravity? The answer is clearly yes, because the law of gravity allows one to explain the examples, given suitable background mathematics. Similarly, one can imagine that electromagnetism, quantum mechanics, and the theory of relativity are also within the scope of ILP programs. Of course, they are also within the scope of a monkey with a typewriter; we still need better heuristics and new ways to structure the search space.

One thing that inverse resolution systems *will* do for you is invent new predicates. This ability is often seen as somewhat magical, because computers are often thought of as "merely working with what they are given." In fact, new predicates fall directly out of the inverse resolution step. The simplest case arises in hypothesizing two new clauses C_1 and C_2, given a clause C. The resolution of C_1 and C_2 eliminates a literal that the two clauses share; hence, it is quite possible that the eliminated literal contained a predicate that does not appear in C. Thus, when working backward, one possibility is to generate a new predicate from which to reconstruct the missing literal.

Figure 19.14 shows an example in which the new predicate P is generated in the process of learning a definition for $Ancestor$. Once generated, P can be used in later inverse resolution steps. For example, a later step might hypothesize that $Mother(x, y) \Rightarrow P(x, y)$. Thus, the new predicate P has its meaning constrained by the generation of hypotheses that involve it. Another example might lead to the constraint $Father(x, y) \Rightarrow P(x, y)$. In other words, the predicate P is what we usually think of as the $Parent$ relationship. As we mentioned earlier, the invention of new predicates can significantly reduce the size of the definition of the goal predicate. Hence, by including the ability to invent new predicates, inverse resolution systems can often solve learning problems that are infeasible with other techniques.

Some of the deepest revolutions in science come from the invention of new predicates and functions—for example, Galileo's invention of acceleration or Joule's invention of thermal energy. Once these terms are available, the discovery of new laws becomes (relatively) easy. The difficult part lies in realizing that some new entity, with a specific relationship to existing entities, will allow an entire body of observations to be explained with a much

Figure 19.14   An inverse resolution step that generates a new predicate P.
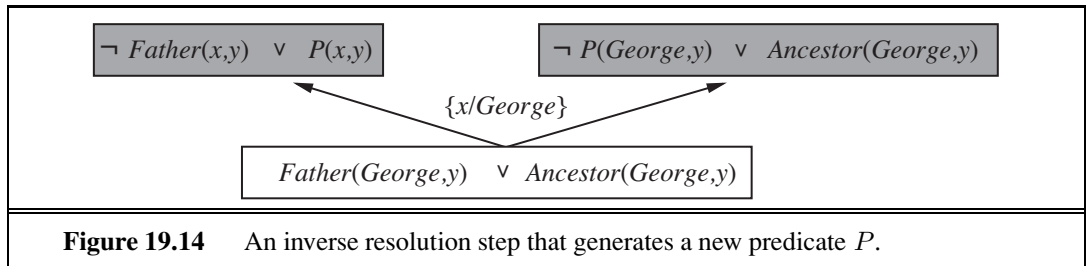
simpler and more elegant theory than previously existed.

As yet, ILP systems have not made discoveries on the level of Galileo or Joule, but their discoveries have been deemed publishable in the scientific literature. For example, in the *Journal of Molecular Biology*, Turcotte *et al.* (2001) describe the automated discovery of rules for protein folding by the ILP program PROGOL. Many of the rules discovered by PROGOL could have been derived from known principles, but most had not been previously published as part of a standard biological database. (See Figure 19.10 for an example.). In related work, Srinivasan *et al.* (1994) dealt with the problem of discovering molecular-structure-based rules for the mutagenicity of nitroaromatic compounds. These compounds are found in automobile exhaust fumes. For 80% of the compounds in a standard database, it is possible to identify four important features, and linear regression on these features outperforms ILP. For the remaining 20%, the features alone are not predictive, and ILP identifies relationships that allow it to outperform linear regression, neural nets, and decision trees. Most impressively, King *et al.* (2009) endowed a robot with the ability to perform molecular biology experiments and extended ILP techniques to include experiment design, thereby creating an autonomous scientist that actually discovered new knowledge about the functional genomics of yeast. For all these examples it appears that the ability both to represent relations and to use background knowledge contribute to ILP's high performance. The fact that the rules found by ILP can be interpreted by humans contributes to the acceptance of these techniques in biology journals rather than just computer science journals.

ILP has made contributions to other sciences besides biology. One of the most important is natural language processing, where ILP has been used to extract complex relational information from text. These results are summarized in Chapter 23.

19.6   SUMMARY

This chapter has investigated various ways in which prior knowledge can help an agent to learn from new experiences. Because much prior knowledge is expressed in terms of relational models rather than attribute-based models, we have also covered systems that allow learning of relational models. The important points are:

- The use of prior knowledge in learning leads to a picture of **cumulative learning**, in which learning agents improve their learning ability as they acquire more knowledge.
- Prior knowledge helps learning by eliminating otherwise consistent hypotheses and by

"filling in" the explanation of examples, thereby allowing for shorter hypotheses. These contributions often result in faster learning from fewer examples.

- Understanding the different logical roles played by prior knowledge, as expressed by **entailment constraints**, helps to define a variety of learning techniques.

- **Explanation-based learning** (EBL) extracts general rules from single examples by *explaining* the examples and generalizing the explanation. It provides a deductive method for turning first-principles knowledge into useful, efficient, special-purpose expertise.

- **Relevance-based learning** (RBL) uses prior knowledge in the form of determinations to identify the relevant attributes, thereby generating a reduced hypothesis space and speeding up learning. RBL also allows deductive generalizations from single examples.

- **Knowledge-based inductive learning** (KBIL) finds inductive hypotheses that explain sets of observations with the help of background knowledge.

- **Inductive logic programming** (ILP) techniques perform KBIL on knowledge that is expressed in first-order logic. ILP methods can learn relational knowledge that is not expressible in attribute-based systems.

- ILP can be done with a top-down approach of refining a very general rule or through a bottom-up approach of inverting the deductive process.

- ILP methods naturally generate new predicates with which concise new theories can be expressed and show promise as general-purpose scientific theory formation systems.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Although the use of prior knowledge in learning would seem to be a natural topic for philosophers of science, little formal work was done until quite recently. *Fact, Fiction, and Forecast*, by the philosopher Nelson Goodman (1954), refuted the earlier supposition that induction was simply a matter of seeing enough examples of some universally quantified proposition and then adopting it as a hypothesis. Consider, for example, the hypothesis "All emeralds are grue," where *grue* means "green if observed before time t, but blue if observed thereafter." At any time up to t, we might have observed millions of instances confirming the rule that emeralds are grue, and no disconfirming instances, and yet we are unwilling to adopt the rule. This can be explained only by appeal to the role of relevant prior knowledge in the induction process. Goodman proposes a variety of different kinds of prior knowledge that might be useful, including a version of determinations called **overhypotheses**. Unfortunately, Goodman's ideas were never pursued in machine learning.

The **current-best-hypothesis** approach is an old idea in philosophy (Mill, 1843). Early work in cognitive psychology also suggested that it is a natural form of concept learning in humans (Bruner *et al.*, 1957). In AI, the approach is most closely associated with the work of Patrick Winston, whose Ph.D. thesis (Winston, 1970) addressed the problem of learning descriptions of complex objects. The **version space** method (Mitchell, 1977, 1982) takes a different approach, maintaining the set of *all* consistent hypotheses and eliminating those found to be inconsistent with new examples. The approach was used in the Meta-DENDRAL

expert system for chemistry (Buchanan and Mitchell, 1978), and later in Mitchell's (1983) LEX system, which learns to solve calculus problems. A third influential thread was formed by the work of Michalski and colleagues on the AQ series of algorithms, which learned sets of logical rules (Michalski, 1969; Michalski *et al.*, 1986).

EBL had its roots in the techniques used by the STRIPS planner (Fikes *et al.*, 1972). When a plan was constructed, a generalized version of it was saved in a plan library and used in later planning as a **macro-operator**. Similar ideas appeared in Anderson's ACT* architecture, under the heading of **knowledge compilation** (Anderson, 1983), and in the SOAR architecture, as **chunking** (Laird *et al.*, 1986). **Schema acquisition** (DeJong, 1981), **analytical generalization** (Mitchell, 1982), and **constraint-based generalization** (Minton, 1984) were immediate precursors of the rapid growth of interest in EBL stimulated by the papers of Mitchell *et al.* (1986) and DeJong and Mooney (1986). Hirsh (1987) introduced the EBL algorithm described in the text, showing how it could be incorporated directly into a logic programming system. Van Harmelen and Bundy (1988) explain EBL as a variant of the **partial evaluation** method used in program analysis systems (Jones *et al.*, 1993).

Initial enthusiasm for EBL was tempered by Minton's finding (1988) that, without extensive extra work, EBL could easily slow down a program significantly. Formal probabilistic analysis of the expected payoff of EBL can be found in Greiner (1989) and Subramanian and Feldman (1990). An excellent survey of early work on EBL appears in Dietterich (1990).

ANALOGICAL
REASONING

Instead of using examples as foci for generalization, one can use them directly to solve new problems, in a process known as **analogical reasoning**. This form of reasoning ranges from a form of plausible reasoning based on degree of similarity (Gentner, 1983), through a form of deductive inference based on determinations but requiring the participation of the example (Davies and Russell, 1987), to a form of "lazy" EBL that tailors the direction of generalization of the old example to fit the needs of the new problem. This latter form of analogical reasoning is found most commonly in **case-based reasoning** (Kolodner, 1993) and **derivational analogy** (Veloso and Carbonell, 1993).

Relevance information in the form of functional dependencies was first developed in the database community, where it is used to structure large sets of attributes into manageable subsets. Functional dependencies were used for analogical reasoning by Carbonell and Collins (1973) and rediscovered and given a full logical analysis by Davies and Russell (Davies, 1985; Davies and Russell, 1987). Their role as prior knowledge in inductive learning was explored by Russell and Grosof (1987). The equivalence of determinations to a restricted-vocabulary hypothesis space was proved in Russell (1988). Learning algorithms for determinations and the improved performance obtained by RBDTL were first shown in the FOCUS algorithm, due to Almuallim and Dietterich (1991). Tadepalli (1993) describes a very ingenious algorithm for learning with determinations that shows large improvements in learning speed.

The idea that inductive learning can be performed by inverse deduction can be traced to W. S. Jevons (1874), who wrote, "The study both of Formal Logic and of the Theory of Probabilities has led me to adopt the opinion that there is no such thing as a distinct method of induction as contrasted with deduction, but that induction is simply an inverse employment of deduction." Computational investigations began with the remarkable Ph.D. thesis by

Gordon Plotkin (1971) at Edinburgh. Although Plotkin developed many of the theorems and methods that are in current use in ILP, he was discouraged by some undecidability results for certain subproblems in induction. MIS (Shapiro, 1981) reintroduced the problem of learning logic programs, but was seen mainly as a contribution to the theory of automated debugging. Work on rule induction, such as the ID3 (Quinlan, 1986) and CN2 (Clark and Niblett, 1989) systems, led to FOIL (Quinlan, 1990), which for the first time allowed practical induction of relational rules. The field of relational learning was reinvigorated by Muggleton and Buntine (1988), whose CIGOL program incorporated a slightly incomplete version of inverse resolution and was capable of generating new predicates. The inverse resolution method also appears in (Russell, 1986), with a simple algorithm given in a footnote. The next major system was GOLEM (Muggleton and Feng, 1990), which uses a covering algorithm based on Plotkin's concept of relative least general generalization. ITOU (Rouveirol and Puget, 1989) and CLINT (De Raedt, 1992) were other systems of that era. More recently, PROGOL (Muggleton, 1995) has taken a hybrid (top-down and bottom-up) approach to inverse entailment and has been applied to a number of practical problems, particularly in biology and natural language processing. Muggleton (2000) describes an extension of PROGOL to handle uncertainty in the form of stochastic logic programs.

A formal analysis of ILP methods appears in Muggleton (1991), a large collection of papers in Muggleton (1992), and a collection of techniques and applications in the book by Lavrauc and Duzeroski (1994). Page and Srinivasan (2002) give a more recent overview of the field's history and challenges for the future. Early complexity results by Haussler (1989) suggested that learning first-order sentences was intractible. However, with better understanding of the importance of syntactic restrictions on clauses, positive results have been obtained even for clauses with recursion (Duzeroski *et al.*, 1992). Learnability results for ILP are surveyed by Kietz and Duzeroski (1994) and Cohen and Page (1995).

DISCOVERY SYSTEM Although ILP now seems to be the dominant approach to constructive induction, it has not been the only approach taken. So-called **discovery systems** aim to model the process of scientific discovery of new concepts, usually by a direct search in the space of concept definitions. Doug Lenat's Automated Mathematician, or AM (Davis and Lenat, 1982), used discovery heuristics expressed as expert system rules to guide its search for concepts and conjectures in elementary number theory. Unlike most systems designed for mathematical reasoning, AM lacked a concept of proof and could only make conjectures. It rediscovered Goldbach's conjecture and the Unique Prime Factorization theorem. AM's architecture was generalized in the EURISKO system (Lenat, 1983) by adding a mechanism capable of rewriting the system's own discovery heuristics. EURISKO was applied in a number of areas other than mathematical discovery, although with less success than AM. The methodology of AM and EURISKO has been controversial (Ritchie and Hanna, 1984; Lenat and Brown, 1984).

Another class of discovery systems aims to operate with real scientific data to find new laws. The systems DALTON, GLAUBER, and STAHL (Langley *et al.*, 1987) are rule-based systems that look for quantitative relationships in experimental data from physical systems; in each case, the system has been able to recapitulate a well-known discovery from the history of science. Discovery systems based on probabilistic techniques—especially clustering algorithms that discover new categories—are discussed in Chapter 20.

EXERCISES

19.1   Show, by translating into conjunctive normal form and applying resolution, that the conclusion drawn on page 784 concerning Brazilians is sound.

19.2   For each of the following determinations, write down the logical representation and explain why the determination is true (if it is):

   a. Design and denomination determine the mass of a coin.
   b. For a given program, input determines output.
   c. Climate, food intake, exercise, and metabolism determine weight gain and loss.
   d. Baldness is determined by the baldness (or lack thereof) of one's maternal grandfather.

19.3   Would a probabilistic version of determinations be useful? Suggest a definition.

19.4   Fill in the missing values for the clauses C_1 or C_2 (or both) in the following sets of clauses, given that C is the resolvent of C_1 and C_2:

   a. $C = True \Rightarrow P(A, B), C_1 = P(x, y) \Rightarrow Q(x, y), C_2 = ??$.
   b. $C = True \Rightarrow P(A, B), C_1 = ??, C_2 = ??$.
   c. $C = P(x, y) \Rightarrow P(x, f(y)), C_1 = ??, C_2 = ??$.

If there is more than one possible solution, provide one example of each different kind.
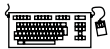
19.5   Suppose one writes a logic program that carries out a resolution inference step. That is, let $Resolve(c_1, c_2, c)$ succeed if c is the result of resolving c_1 and c_2. Normally, $Resolve$ would be used as part of a theorem prover by calling it with c_1 and c_2 instantiated to particular clauses, thereby generating the resolvent c. Now suppose instead that we call it with c instantiated and c_1 and c_2 uninstantiated. Will this succeed in generating the appropriate results of an inverse resolution step? Would you need any special modifications to the logic programming system for this to work?

19.6   Suppose that FOIL is considering adding a literal to a clause using a binary predicate P and that previous literals (including the head of the clause) contain five different variables.

   a. How many functionally different literals can be generated? Two literals are functionally identical if they differ only in the names of the *new* variables that they contain.
   b. Can you find a general formula for the number of different literals with a predicate of arity r when there are n variables previously used?
   c. Why does FOIL not allow literals that contain no previously used variables?

19.7   Using the data from the family tree in Figure 19.11, or a subset thereof, apply the FOIL algorithm to learn a definition for the $Ancestor$ predicate.