10 KNOWLEDGE REPRESENTATION

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

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

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

10.1 ONTOLOGICAL ENGINEERING

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

ONTOLOGICAL ENGINEERING

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

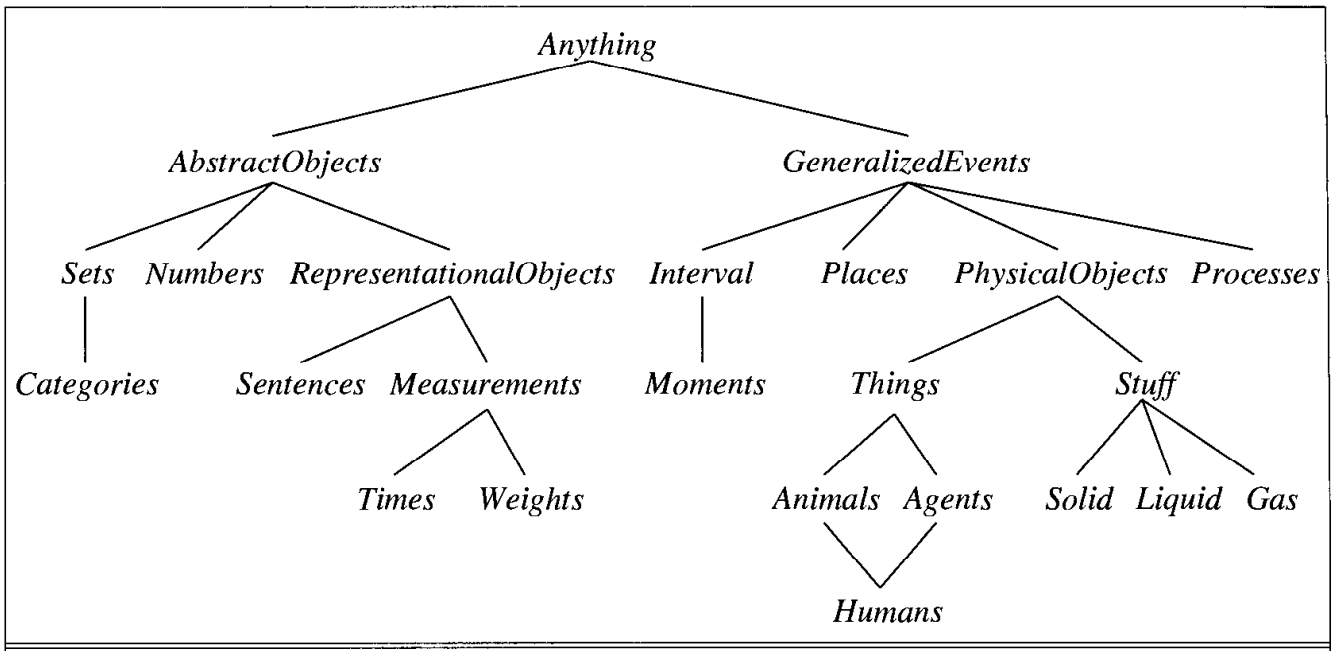


Figure 10.1 The upper ontology of the world, showing the topics to be covered later in the chapter. Each arc indicates that the lower concept is a specialization of the upper one.

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

UPPER ONTOLOGY

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

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

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

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

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

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

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

10.2 Categories and Objects

CATEGORIES

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

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

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

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

INHERITANCE

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

TAXONOMY

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

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

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

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

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

DISJOINT

EXHAUSTIVE DECOMPOSITION

PARTITION

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

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

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

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

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

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

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

Physical composition

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

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

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

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

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

COMPOSITE OBJECT

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

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

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

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

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

BUNCH

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

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

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

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

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

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

LOGICAL MINIMIZATION

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

Measurements

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

MEASURES

UNITS FUNCTION

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

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

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

NATURAL KINDS

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Substances and objects

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

INDIVIDUATION

STUFF

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

COUNT NOUNS
MASS NOUNS

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

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

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

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

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

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

INTRINSIC
EXTRINSIC

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

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

10.3 Actions, Situations, and Events

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

The ontology of situation calculus

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

SITUATION CALCULUS

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

SITUATIONS

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

FLUENTS

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

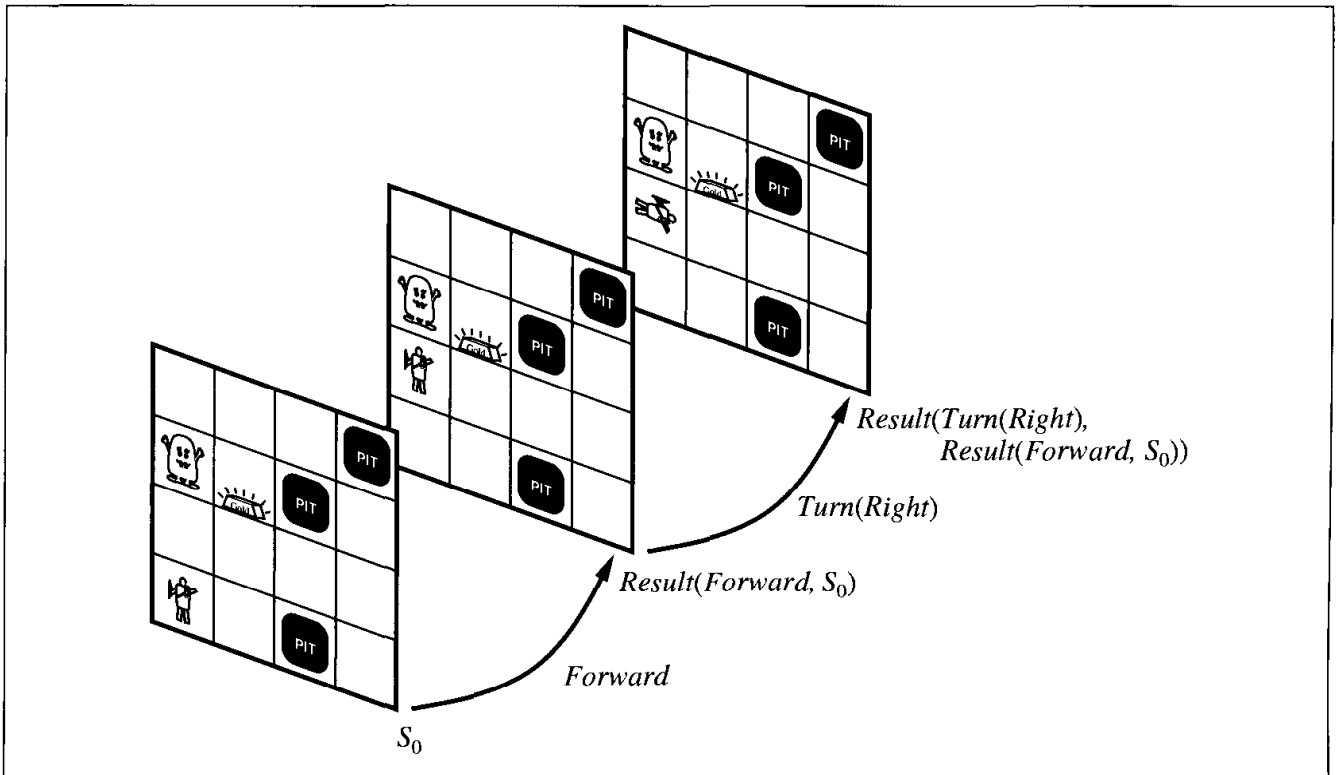


Figure 10.2 In situation calculus, each situation (except S_0) is the result of an action.

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

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

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

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

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

PROJECTION
PLANNING

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

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

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

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

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

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

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

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

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

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

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

Describing actions in situation calculus

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

POSSIBILITY AXIOM
EFFECT AXIOM

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

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

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

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

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

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

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

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

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

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

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

FRAME PROBLEM

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

FRAME AXIOM

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

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

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

REPRESENTATIONAL FRAME PROBLEM

INFERENTIAL FRAME PROBLEM

QUALIFICATION PROBLEM

Solving the representational frame problem

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

SUCCESSOR-STATE AXIOM

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

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

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

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

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

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

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

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

IMPLICIT EFFECT

RAMIFICATION PROBLEM

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

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

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

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

UNIQUE NAMES AXIOM

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

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

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

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

UNIQUE ACTION AXIOMS

Solving the inferential frame problem

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

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

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

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

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

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

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

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

Time and event calculus

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

EVENT CALCULUS

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

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

EVENT CALCULUS AXIOM:

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

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

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

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

Generalized events

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

GENERALIZED EVENT

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

SUBEVENTS

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

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

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

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

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

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

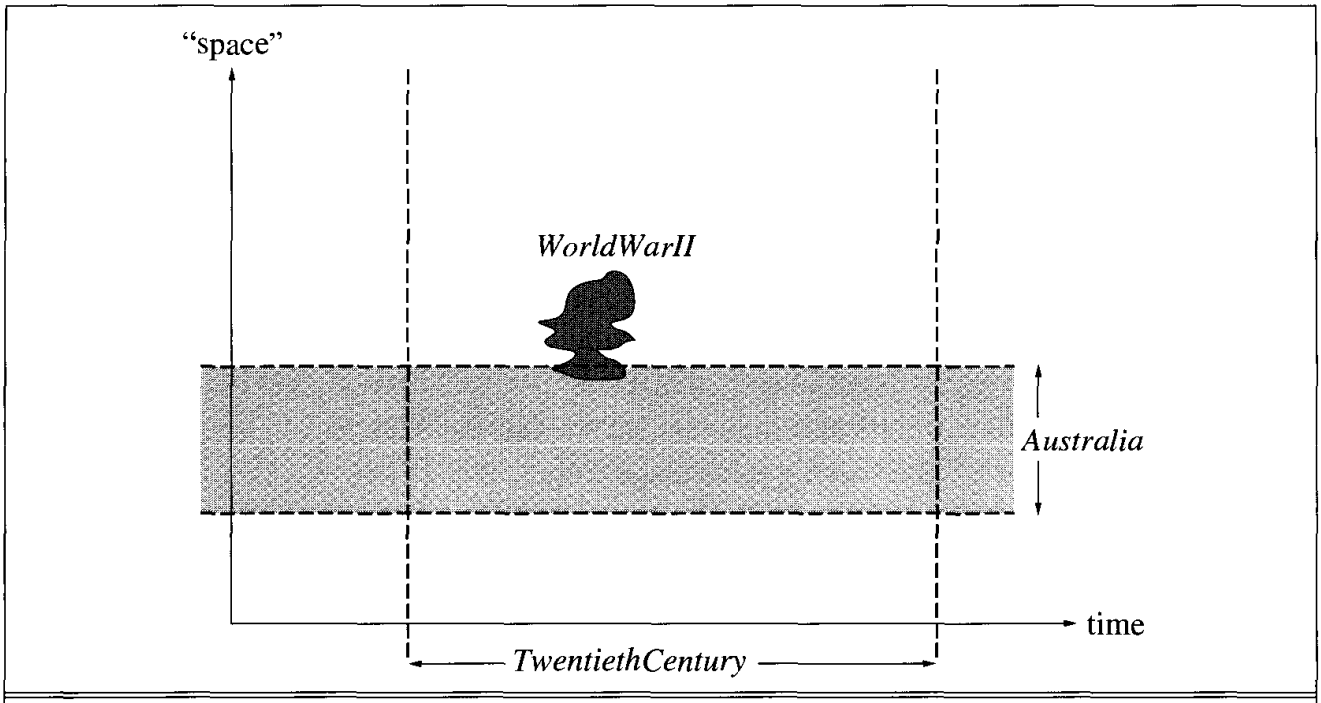


Figure 10.3 Generalized events. A universe has spatial and temporal dimensions; in this figure we show only a single spatial dimension. All events are *PartOf* the universe. An event, such as $WorldWarII$, occurs in a portion of space–time with somewhat arbitrary and time-varying borders. An $Interval$,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

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

$$In(Sydney, Australia) .$$

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

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

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

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

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

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

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

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

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

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

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

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

Thus, we have:

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

Processes

DISCRETE EVENTS

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

PROCESS
LIQUID EVENT

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

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

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

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

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

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

TEMPORAL SUBSTANCES
SPATIAL SUBSTANCES
STATES

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

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

FLUENT CALCULUS

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

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

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

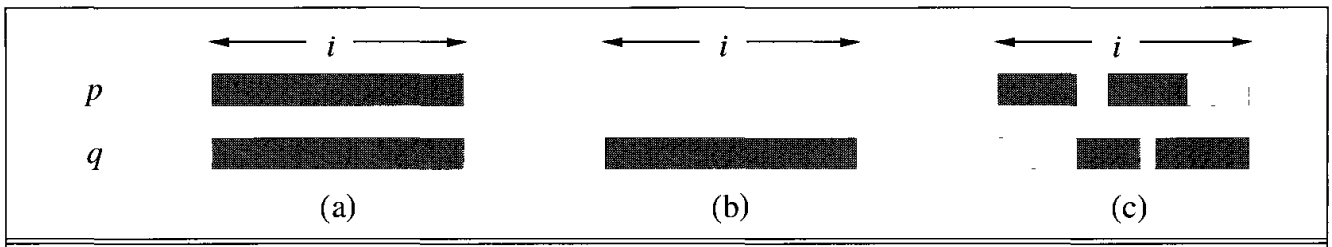


Figure 10.4 A depiction of complex events. (a) $T(Both(p.\, q).\, i)$, also denoted as $T(p \circ q, i)$. (b) $T(OneOf(p, q), i)$. (c) $T(Either(p.\, q).\, i)$.

Intervals

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

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

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

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

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

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

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

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

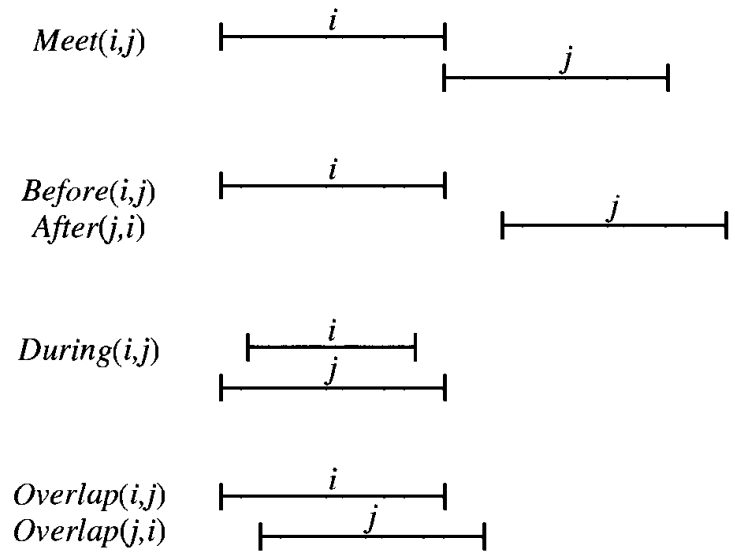


Figure 10.5 Predicates on time intervals.

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

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

Fluents and objects

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

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

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

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

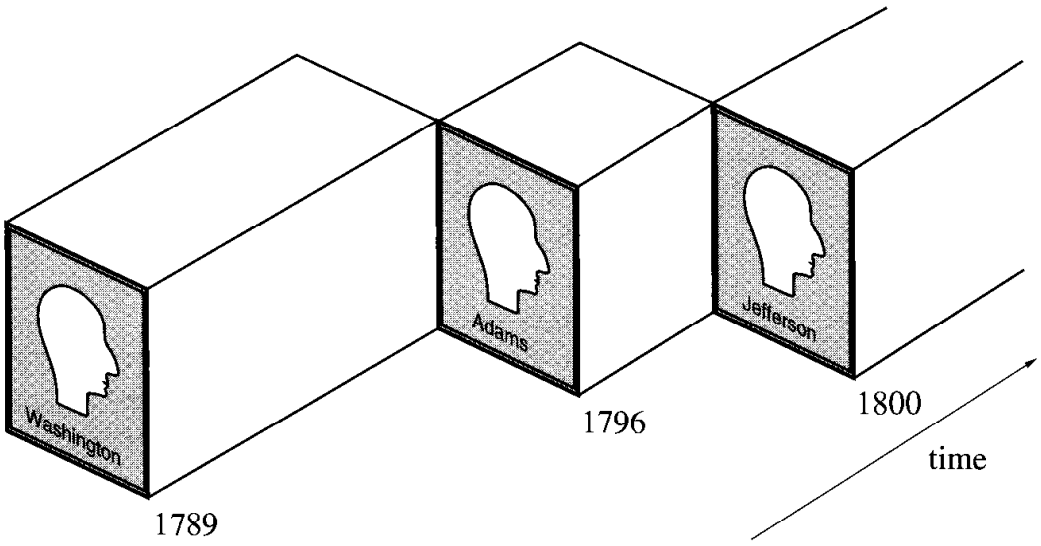


Figure 10.6 A schematic view of the object $President(USA)$ for the first 15 years of its existence.

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

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

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

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

10.4 MENTAL EVENTS AND MENTAL OBJECTS

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

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

A formal theory of beliefs

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

PROPOSITIONAL ATTITUDE

REIFICATION

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

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

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

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

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

REFERENTIAL TRANSPARENCY

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

OPAQUE

MODAL LOGIC

MODAL OPERATOR

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

SYNTACTIC THEORY

STRINGS

UNIQUE STRING AXIOM

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

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

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

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

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

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

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

UNQUOTED

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

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

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

LOGICAL OMNISCIENCE

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

Knowledge and belief

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

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

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

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

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

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

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

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

Knowledge, time, and action

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

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

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

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

KNOWLEDGE PRECONDITIONS
KNOWLEDGE EFFECTS

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

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

RUNTIME VARIABLES

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

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

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

10.5 The Internet Shopping World

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

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

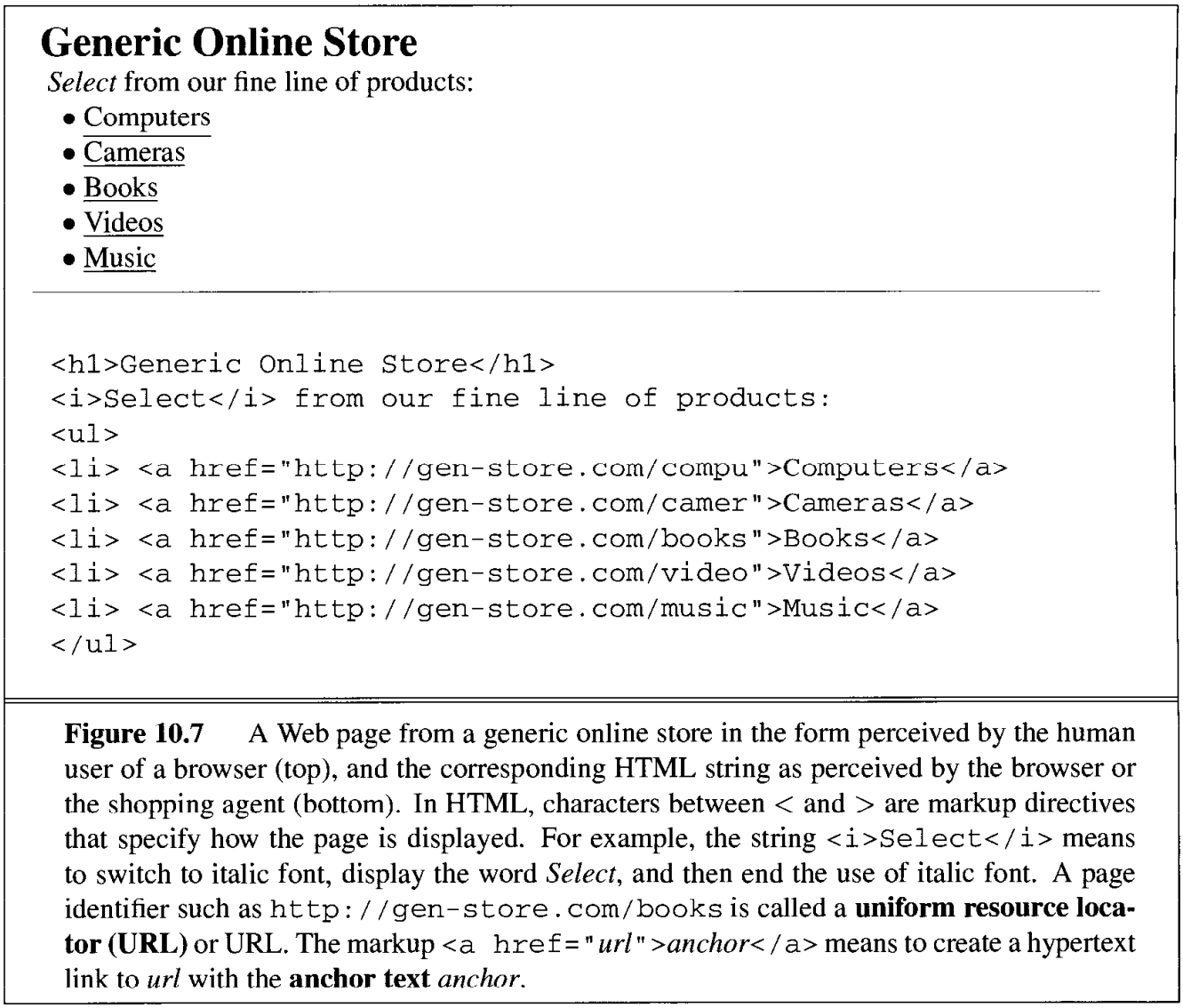


Figure 10.7 A Web page from a generic online store in the form perceived by the human user of a browser (top), and the corresponding HTML string as perceived by the browser or the shopping agent (bottom). In HTML, characters between < and > are markup directives that specify how the page is displayed.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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The logical definition of $RelevantCategoryName$ is as follows:

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

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

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

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

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

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

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

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

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

PROCEDURAL ATTACHMENT

Comparing offers

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

WRAPPER

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

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

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

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

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

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

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

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

10.6 Reasoning Systems for Categories

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

Semantic networks

EXISTENTIAL GRAPHS

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

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

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

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

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

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

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

MULTIPLE INHERITANCE

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

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

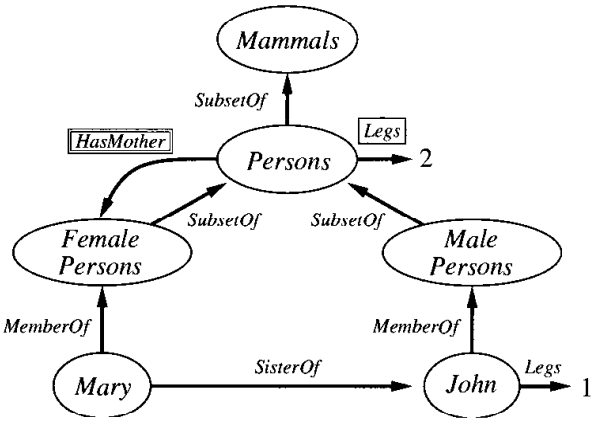


Figure 10.9 A semantic network with four objects (John, Mary, 1, and 2) and four categories. Relations are denoted by labeled links.

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

INVERSE LINK

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

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

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

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

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

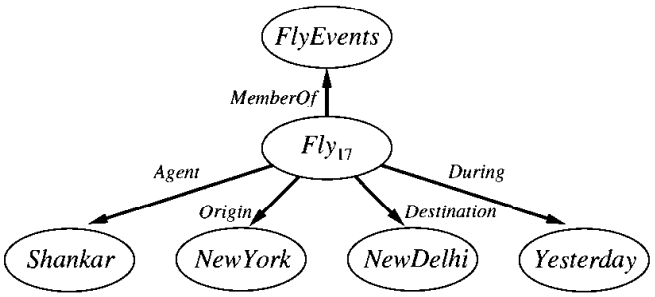


Figure 10.10 A fragment of a semantic network showing the representation of the logical assertion $Fly(Shankar, NewYork, NewDelhi, Yesterday)$.

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

DEFAULT VALUES

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

OVERRIDING

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

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

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

Description logics

DESCRIPTION LOGICS

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

SUBSUMPTION

CLASSIFICATION

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

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

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

The equivalent in first-order logic would be

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

Notice that the description logic effectively allows direct logical operations on predicates, rather than having to first create sentences to be joined by connectives. Any description in CLASSIC can be written in first-order logic, but some descriptions are more straightforward in CLASSIC. For example, to describe the set of men with at least three sons who are all unemployed and married to doctors and at most two daughters who are all professors in physics or math departments, we would use

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

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

Perhaps the most important aspect of description logics is their emphasis on tractability of inference. A problem instance is solved by describing it and then asking if it is subsumed by one of several possible solution categories. In standard first-order logic systems, predicting the solution time is often impossible. It is frequently left to the user to engineer the representation to detour around sets of sentences that seem to be causing the system to take several weeks to solve a problem. The thrust in description logics, on the other hand, is to ensure that subsumption-testing can be solved in time polynomial in the size of the descriptions.[10]

This sounds wonderful in principle, until one realizes that it can only have one of two consequences: either hard problems cannot be stated at all, or they require exponentially large descriptions! However, the tractability results do shed light on what sorts of constructs cause problems and thus help the user to understand how different representations behave.

[9] Notice that the language does *not* allow one to simply state that one concept, or category, is a subset of another. This is a deliberate policy: subsumption between categories must be derivable from some aspects of the descriptions of the categories. If not, then something is missing from the descriptions.

[10] CLASSIC provides efficient subsumption testing in practice, but the worst-case runtime is exponential.

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

Figure 10.11 The syntax of descriptions in a subset of the CLASSIC language.

For example, description logics usually lack *negation* and *disjunction*. Each forces first-order logical systems to go through a potentially exponential case analysis in order to ensure completeness. For the same reason, they are excluded from Prolog. CLASSIC allows only a limited form of disjunction in the *Fills* and *OneOf* constructs, which permit disjunction over explicitly enumerated individuals but not over descriptions. With disjunctive descriptions, nested definitions can lead easily to an exponential number of alternative routes by which one category can subsume another.

10.7 REASONING WITH DEFAULT INFORMATION

In the preceding section, we saw a simple example of an assertion with default status: people have two legs. This default can be overridden by more specific information, such as that Long John Silver has one leg. We saw that the inheritance mechanism in semantic networks implements the overriding of defaults in a simple and natural way. In this section, we study defaults more generally, with a view toward understanding the *semantics* of defaults rather than just providing a procedural mechanism.

Open and closed worlds

Suppose you were looking at a bulletin board in a university computer science department and saw a notice saying, "The following courses will be offered: CS 101, CS 102, CS 106, EE 101." Now, how many courses will be offered? If you answered "Four," you would be in agreement with a typical database system. Given a relational database with the equivalent of the four assertions

$$\mathit{Course}(\mathit{CS}, 101),\ \mathit{Course}(\mathit{CS}, 102),\ \mathit{Course}(\mathit{CS}, 106),\ \mathit{Course}(\mathit{EE}, 101), \quad (10.2)$$

the SQL query `count * from Course` returns 4. On the other hand, a first-order logical system would answer "Somewhere between one and infinity," not "four." The reason is that

the *Course* assertions do not deny the possibility that other unmentioned courses are also offered, nor do they say that the courses mentioned are different from each other.

This example shows that database systems and human communication conventions differ from first-order logic in at least two ways. First, databases (and people) assume that the information provided is *complete*, so that ground atomic sentences not asserted to be true are assumed to be false. This is called the **closed-world assumption**, or CWA. Second, we usually assume that distinct names refer to distinct objects. This is the **unique names assumption**, or UNA, which we introduced first in the context of action names in Section 10.3.

CLOSED-WORLD ASSUMPTION

First-order logic does not assume these conventions, and thus needs to be more explicit. To say that *only* the four distinct courses are offered, we would write:

$$Course(d,n) \Leftrightarrow [d,n]=[CS,101] \vee [d,n]=[CS,102] \vee [d,n]=[CS,106] \vee [d,n]=[EE,101] \ . \quad (10.3)$$

COMPLETION

Equation 10.3 is called the **completion**[11] of 10.2. In general, the completion will contain a definition—an if-and-only-if sentence—for each predicate, and each definition will contain a disjunct for each definite clause having that predicate in its head.[12] In general, the completion is constructed as follows:

1. Gather up all the clauses with the same predicate name (P) and the same arity (n).
2. Translate each clause to **Clark Normal Form**: replace

   CLARK NORMAL FORM

   $$P(t_1,\ldots,t_n) \leftarrow Body \ ,$$

   where t_i are terms, with

   $$P(v_1,\ldots,v_n) \leftarrow \exists w_1 \ldots w_m \ [v_1,\ldots,v_n]=[t_1,\ldots,t_n] \wedge Body \ ,$$

   where v_i are newly invented variables and w_i are the variables that appear in the original clause. Use the same set of v_i for every clause. This gives us a set of clauses

   $$P(v_1,\ldots,v_n) \leftarrow B_1$$
   $$\vdots$$
   $$P(v_1,\ldots,v_n) \leftarrow B_k \ .$$
3. Combine these together into one big disjunctive clause:

   $$P(v_1,\ldots,v_n) \leftarrow B_1 \vee \ldots \vee B_k \ .$$
4. Form the completion by replacing the $\leftarrow$ with an equivalence:

   $$P(v_1,\ldots,v_n) \Leftrightarrow B_1 \vee \ldots \vee B_k \ .$$

Figure 10.12 shows an example of the Clark completion for a knowledge base with both ground facts and rules. To add in the unique names assumption, we simply construct the Clark completion for the equality relation, where the only known facts are that $CS=CS$, $101=101$, and so on. This is left as an exercise.

The closed-world assumption allows us to find a **minimal model** of a relation. That is, we can find the model of the relation *Course* with the fewest elements. In Equation (10.2)

[11] Sometimes called "Clark Completion" after the inventor, Keith Clark.

[12] Notice that this is also the form of the successor-state axioms given in Section 10.3.

Horn Clauses	**Clark Completion**
$Course(CS, 101)$	$Course(d, n) \Leftrightarrow [d, n] = [CS, 101]$
$Course(CS, 102)$	$\vee\ [d, n] = [CS, 102]$
$Course(CS, 106)$	$\vee\ [d, n] = [CS, 106]$
$Course(EE, 101)$	$\vee\ [d, n] = [EE, 101]$
$Course(EE, i) \leftarrow Integer(i)$	$\vee\ \exists i\ [d, n] = [EE, i] \wedge Integer(i)$
$\wedge\ 101 \leq i \wedge i \leq 130$	$\wedge\ 101 \leq i \wedge i \leq 130$
$Course(CS, m + 100) \Leftarrow$	$\vee\ \exists m\ [d, n] = [CS, m + 100]$
$Course(CS, m) \wedge 100 \leq m$	$\wedge\ Course(CS, m) \wedge 100 \leq m$
$\wedge\ m < 200$	$\wedge\ m < 200$

Figure 10.12 The Clark Completion of a set of Horn clauses. The original Horn program (left) lists four courses explicitly and also asserts that there is a math class for every integer from 101 to 130, and that for every CS class in the 100 (undergraduate) series, there is a corresponding class in the 200 (graduate) series. The Clark completion (right) says that there are no other classes. With the completion and the unique names assumption (and the obvious definition of the *Integer* predicate), we get the desired conclusion that there are exactly 36 courses: 30 math courses and 6 CS courses.

the minimal model of *Course* has four elements; any less and we'd have a contradiction. For Horn knowledge bases, there is always a *unique* minimal model. Notice that, with the unique names assumption, this applies to the equality relation too: each term is equal only to itself. Paradoxically, this means that minimal models are maximal in the sense of having as many objects as possible.

It is possible to take a Horn program, generate the Clark completion, and hand that to a theorem prover to do inference. But it is usually more efficient to use a special-purpose inference mechanism such as Prolog, which has the closed world and unique names assumptions built into the inference mechanism.

Those who make the closed-world assumption must be careful about what kind of reasoning they will be doing. For example, in a census database it would be reasonable to make the CWA when reasoning about the current population of cities, but it would be wrong to conclude that no baby will ever be born in the future just because the database contains no entries with future birthdates. The CWA makes the database **complete**, in the sense that every atomic query is answered either positively or negatively; when we are genuinely ignorant of facts (such as future births) we cannot use the CWA. A more sophisticated knowledge representation system might allow the user to specify rules for when to apply the CWA.

Negation as failure and stable model semantics

We saw in Chapters 7 and 9 than Horn-form knowledge bases have desirable computational properties. In many applications, however, the requirement that every literal in the body of a clause be positive is rather inconvenient. We would like to say "You can go outside if it's not raining," without having to concoct predicates such as *NotRaining*. In this section, we explore the addition of a form of explicit negation to Horn clauses using the idea using of

NEGATION AS FAILURE

negation as failure. The idea is that a negative literal, *not* P, can be "proved" true just in case the proof of P fails. This is a form of default reasoning closely related to the closed world assumption: we assume something is false if it cannot be proved true. We use "*not*" to distinguish negation as failure from the logical "$\neg$" operator.

Prolog allows the *not* operator in the body of a clause. For example, consider the following Prolog program:

$$\begin{array}{l} IDEdrive \leftarrow Drive \wedge not\ SCSIdrive\ . \\ SCSIdrive \leftarrow Drive \wedge not\ IDEdrive\ . \\ SCSIcontroller \leftarrow SCSIdrive\ . \\ Drive\ . \end{array} \tag{10.4}$$

The first rule says that if we have a hard drive on a computer and it is not SCSI, then it must be IDE. The second says if it is not IDE it must be SCSI. The third says that having a SCSI drive implies having a SCSI controller, and the fourth says that we do indeed have a drive. This program has *two* minimal models:

$$\begin{array}{l} M_1 = \{Drive, IDEdrive\}\ , \\ M_2 = \{Drive, SCSIdrive, SCSIcontroller\}\ . \end{array}$$

Minimal models do not capture the intended semantics of programs with negation as failure. Consider the program

$$P \leftarrow not\ Q. \tag{10.5}$$

This has two minimal models, $\{P\}$ and $\{Q\}$. From an FOL point of view this makes sense, since $P \Leftarrow \neg Q$ is equivalent to $P \vee Q$. But from a Prolog point of view it is worrisome: Q never appears on the left hand side of an arrow, so how can it be a consequence?

STABLE MODEL
JUSTIFICATION

An alternative is the idea of a **stable model**, which is a minimal model where every atom in the model has a **justification**: a rule where the head is the atom and where every literal in the body is satisfied. Technically, we say that M is a stable model of a program H if M is the unique minimal model of the **reduct** of H with respect to M. The reduct of a program H is defined by first deleting from H any rule that has a literal *not* A in the body, where A is in the model, and then deleting any negative literals in the remaining rules. Since the reduct of H is now a list of Horn clauses, it must have a unique minimal model.

REDUCT

The reduct of $P \leftarrow not\ Q$ with respect to $\{P\}$ is P, which has minimal model $\{P\}$. Therefore $\{P\}$ is a stable model. The reduct with respect to $\{Q\}$ is the empty program, which has minimal model $\{\}$. Therefore $\{Q\}$ is not a stable model because Q has no justification in Equation (10.5). As another example, the reduct of 10.4 with respect to M_1 is as follows:

$$\begin{array}{l} IDEdrive \leftarrow Drive\ . \\ SCSIcontroller \leftarrow SCSIdrive\ . \\ Drive\ . \end{array}$$

ANSWER SET PROGRAMMING
ANSWER SETS

This has minimal model M_1, so M_1 is a stable model. **Answer set programming** is a kind of logic programming with negation as failure that works by translating the logic program into ground form and then searching for stable models (also known as **answer sets**) using propositional model checking techniques. Thus answer set programming is a descendant both of Prolog and of the fast propositional satisfiability provers such as WALKSAT. Indeed,

answer set programming has been successfully applied to problems in planning just as the propositional satisfiability provers have. The advantage of answer set planning over other planners is the degree of flexibility: the planning operators and constraints can be expressed as logic programs and are not bound to the restricted format of a particular planning formalism. The disadvantage of answer set planning is the same as for other propositional techniques: if there are very many objects in the universe, then there can be an exponential slow-down.

Circumscription and default logic

We have seen two examples where apparently natural reasoning processes violate the **monotonicity** property of logic that was proved in Chapter 7.[13] In the first example, a property inherited by all members of a category in a semantic network could be overridden by more specific information for a subcategory. In the second example, negated literals derived from a closed-world assumption could be overridden by the addition of positive literals.

Simple introspection suggests that these failures of monotonicity are widespread in commonsense reasoning. It seems that humans often "jump to conclusions." For example, when one sees a car parked on the street, one is normally willing to believe that it has four wheels even though only three are visible. (If you feel that the existence of the fourth wheel is dubious, consider also the question as to whether the three visible wheels are real or merely cardboard facsimiles.) Now, probability theory can certainly provide a conclusion that the fourth wheel exists with high probability, yet, for most people, the possibility of the car's not having four wheels *does not arise unless some new evidence presents itself.* Thus, it seems that the four-wheel conclusion is reached *by default*, in the absence of any reason to doubt it. If new evidence arrives—for example, if one sees the owner carrying a wheel and notices that the car is jacked up—then the conclusion can be retracted. This kind of reasoning is said to exhibit **nonmonotonicity**, because the set of beliefs does not grow monotonically over time as new evidence arrives. **Nonmonotonic logics** have been devised with modified notions of truth and entailment in order to capture such behavior. We will look at two such logics that have been studied extensively: circumscription and default logic.

NONMONOTONICITY

NONMONOTONIC LOGICS

CIRCUMSCRIPTION

Circumscription can be seen as a more powerful and precise version of the closed-world assumption. The idea is to specify particular predicates that are assumed to be "as false as possible"—that is, false for every object except those for which they are known to be true. For example, suppose we want to assert the default rule that birds fly. We would introduce a predicate, say $Abnormal_1(x)$, and write

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

If we say that $Abnormal_1$ is to be **circumscribed**, a circumscriptive reasoner is entitled to assume $\neg Abnormal_1(x)$ unless $Abnormal_1(x)$ is known to be true. This allows the conclusion $Flies(Tweety)$ to be drawn from the premise $Bird(Tweety)$, but the conclusion no longer holds if $Abnormal_1(Tweety)$ is asserted.

MODEL PREFERENCE

Circumscription can be viewed as an example of a **model preference** logic. In such logics, a sentence is entailed (with default status) if it is true in all *preferred* models of the KB,

[13] Recall that monotonicity requires all entailed sentences to remain entailed after new sentences are added to the KB. That is, if $KB \models \alpha$ then $KB \wedge \beta \models \alpha$.

as opposed to the requirement of truth in *all* models in classical logic. For circumscription, one model is preferred to another if it has fewer abnormal objects.[14] Let us see how this idea works in the context of multiple inheritance in semantic networks. The standard example for which multiple inheritance is problematic is called the "Nixon diamond." It arises from the observation that Richard Nixon was both a Quaker (and hence by default a pacifist) and a Republican (and hence by default not a pacifist). We can write this as follows:

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

If we circumscribe $Abnormal_2$ and $Abnormal_3$, there are two preferred models: one in which $Abnormal_2(Nixon)$ and $Pacifist(Nixon)$ hold and one in which $Abnormal_3(Nixon)$ and $\neg Pacifist(Nixon)$ hold. Thus, the circumscriptive reasoner remains properly agnostic as to whether Nixon is a pacifist. If we wish, in addition, to assert that religious beliefs take precedence over political beliefs, we can use a formalism called **prioritized circumscription** to give preference to models where $Abnormal_3$ is minimized.

PRIORITIZED CIRCUMSCRIPTION

DEFAULT LOGIC

DEFAULT RULES

Default logic is a formalism in which **default rules** can be written to generate contingent, nonmonotonic conclusions. A default rule looks like this:

$$Bird(x) : Flies(x)/Flies(x) .$$

This rule means that if $Bird(x)$ is true, and if $Flies(x)$ is consistent with the knowledge base, then $Flies(x)$ may be concluded by default. In general, a default rule has the form

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

where P is called the prerequisite, C is the conclusion, and J_i are the justifications—if any one of them can be proven false, then the conclusion cannot be drawn. Any variable that appears in J_i or C must also appear in P. The Nixon-diamond example can be represented in default logic with one fact and two default rules:

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

EXTENSION

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

Since 1980, when nonmonotonic logics were first proposed, a great deal of progress has been made in understanding their mathematical properties. Beginning in the late 1990s,

[14] For the closed-world assumption, one model is preferred to another if it has fewer true atoms—that is, preferred models are **minimal** models. There is a natural connection between the CWA and definite clause KBs, because the fixed point reached by forward chaining on such KBs is the unique minimal model. (See page 219.)

practical systems based on logic programming have shown promise as knowledge representation tools. There are still unresolved questions, however. For example, if "Cars have four wheels" is false, what does it mean to have it in one's knowledge base? What is a good set of default rules to have? If we cannot decide, for each rule separately, whether it belongs in our knowledge base, then we have a serious problem of nonmodularity. Finally, how can beliefs that have default status be used to make decisions? This is probably the hardest issue for default reasoning. Decisions often involve tradeoffs, and one therefore needs to compare the *strengths* of belief in the outcomes of different actions. In cases where the same kinds of decisions are being made repeatedly, it is possible to interpret default rules as "threshold probability" statements. For example, the default rule "My brakes are always OK" really means "The probability that my brakes are OK, given no other information, is sufficiently high that the optimal decision is for me to drive without checking them." When the decision context changes—for example, when one is driving a heavily laden truck down a steep mountain road—the default rule suddenly becomes inappropriate, even though there is no new evidence to suggest that the brakes are faulty. These considerations have led some researchers to consider how to embed default reasoning in probability theory.

10.8 TRUTH MAINTENANCE SYSTEMS

The previous section argued that many of the inferences drawn by a knowledge representation system will have only default status, rather than being absolutely certain. Inevitably, some of these inferred facts will turn out to be wrong and will have to be retracted in the face of new information. This process is called **belief revision**.[15] Suppose that a knowledge base KB contains a sentence P—perhaps a default conclusion recorded by a forward-chaining algorithm, or perhaps just an incorrect assertion—and we want to execute TELL(KB, $\neg P$). To avoid creating a contradiction, we must first execute RETRACT(KB, P). This sounds easy enough. Problems arise, however, if any *additional* sentences were inferred from P and asserted in the KB. For example, the implication $P \Rightarrow Q$ might have been used to add Q. The obvious "solution"—retracting all sentences inferred from P—fails because such sentences may have other justifications besides P. For example, if R and $R \Rightarrow Q$ are also in the KB, then Q does not have to be removed after all. **Truth maintenance systems**, or TMSs, are designed to handle exactly these kinds of complications.

BELIEF REVISION

TRUTH MAINTENANCE SYSTEM

One very simple approach to truth maintenance is to keep track of the order in which sentences are told to the knowledge base by numbering them from P_1 to P_n. When the call RETRACT(KB, P_i) is made, the system reverts to the state just before P_i was added, thereby removing both P_i and any inferences that were derived from P_i. The sentences P_{i+1} through P_n can then be added again. This is simple, and it guarantees that the knowledge base will be consistent, but retracting P_i requires retracting and reasserting $n - i$ sentences as well as

[15] Belief revision is often contrasted with **belief update**, which occurs when a knowledge base is revised to reflect a change in the world rather than new information about a fixed world. Belief update combines belief revision with reasoning about time and change; it is also related to the process of **filtering** described in Chapter 15.

undoing and redoing all the inferences drawn from those sentences. For systems to which many facts are being added—such as large commercial databases—this is impractical.

JTMS

A more efficient approach is the justification-based truth maintenance system, or **JTMS**.

JUSTIFICATION

In a JTMS, each sentence in the knowledge base is annotated with a **justification** consisting of the set of sentences from which it was inferred. For example, if the knowledge base already contains $P \Rightarrow Q$, then TELL(P) will cause Q to be added with the justification $\{P,\ P \Rightarrow Q\}$. In general, a sentence can have any number of justifications. Justifications are used to make retraction efficient. Given the call RETRACT(P), the JTMS will delete exactly those sentences for which P is a member of every justification. So, if a sentence Q had the single justification $\{P,\ P \Rightarrow Q\}$ it would be removed, if it had the additional justification $\{P,\ P \vee R \Rightarrow Q\}$ it would still be removed, but if it also had the justification $\{R,\ P \vee R \Rightarrow Q\}$, then it would be spared. In this way, the time required for retraction of P depends only on the number of sentences derived from P rather than on the number of other sentences added since P entered the knowledge base.

The JTMS assumes that sentences that are considered once will probably be considered again, so rather than deleting a sentence from the knowledge base entirely when it loses all justifications, we merely mark the sentence as being *out* of the knowledge base. If a subsequent assertion restores one of the justifications, then we mark the sentence as being back *in*. In this way, the JTMS retains all of the inference chains that it uses and need not rederive sentences when a justification becomes valid again.

In addition to handling the retraction of incorrect information, TMSs can be used to speed up the analysis of multiple hypothetical situations. Suppose, for example, that the Romanian Olympic Committee is choosing sites for the swimming, athletics, and equestrian events at the 2048 Games to be held in Romania. For example, let the first hypothesis be $Site(Swimming, Pitesti)$, $Site(Athletics, Bucharest)$, and $Site(Equestrian, Arad)$. A great deal of reasoning must then be done to work out the logistical consequences and hence the desirability of this selection. If we want to consider $Site(Athletics, Sibiu)$ instead, the TMS avoids the need to start again from scratch. Instead, we simply retract $Site(Athletics, Bucharest)$ and assert $Site(Athletics, Sibiu)$ and the TMS takes care of the necessary revisions. Inference chains generated from the choice of Bucharest can be reused with Sibiu, provided that the conclusions are the same.

ATMS

An assumption-based truth maintenance system, or **ATMS**, is designed to make this type of context-switching between hypothetical worlds particularly efficient. In a JTMS, the maintenance of justifications allows you to move quickly from one state to another by making a few retractions and assertions, but at any time only one state is represented. An ATMS represents *all* the states that have ever been considered at the same time. Whereas a JTMS simply labels each sentence as being *in* or *out*, an ATMS keeps track, for each sentence, of which assumptions would cause the sentence to be true. In other words, each sentence has a label that consists of a set of assumption sets. The sentence holds just in those cases where all the assumptions in one of the assumption sets hold.

EXPLANATIONS

Truth maintenance systems also provide a mechanism for generating **explanations**. Technically, an explanation of a sentence P is a set of sentences E such that E entails P. If the sentences in E are already known to be true, then E simply provides a sufficient ba-

ASSUMPTIONS

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

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

10.9 SUMMARY

This has been the most detailed chapter of the book so far. By delving into the details of how one represents a variety of knowledge, we hope we have given the reader a sense of how real knowledge bases are constructed. The major points are as follows:

- Large-scale knowledge representation requires a general-purpose ontology to organize and tie together the various specific domains of knowledge.
- A general-purpose ontology needs to cover a wide variety of knowledge and should be capable, in principle, of handling any domain.
- We presented an **upper ontology** based on categories and the event calculus. We covered structured objects, time and space, change, processes, substances, and beliefs.
- Actions, events, and time can be represented either in situation calculus or in more expressive representations such as event calculus and fluent calculus. Such representations enable an agent to construct plans by logical inference.
- The mental states of agents can be represented by strings that denote beliefs.
- We presented a detailed analysis of the Internet shopping domain, exercising the general ontology and showing how the domain knowledge can be used by a shopping agent.
- Special-purpose representation systems, such as **semantic networks** and **description logics**, have been devised to help in organizing a hierarchy of categories. **Inheritance** is an important form of inference, allowing the properties of objects to be deduced from their membership in categories.
- The **closed-world assumption**, as implemented in logic programs, provides a simple way to avoid having to specify lots of negative information. It is best interpreted as a **default** that can be overridden by additional information.

- **Nonmonotonic logics**, such as **circumscription** and **default logic**, are intended to capture default reasoning in general. **Answer set programming** speeds up nonmonotonic inference, much as WALKSAT speeds up propositional inference.
- **Truth maintenance systems** handle knowledge updates and revisions efficiently.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

There are plausible claims (Briggs, 1985) that formal knowledge representation research began with classical Indian theorizing about the grammar of Shastric Sanskrit, which dates back to the first millennium B.C. In the West, the use of definitions of terms in ancient Greek mathematics can be regarded as the earliest instance. Indeed, the development of technical terminology in any field can be regarded as a form of knowledge representation.

Early discussions of representation in AI tended to focus on "*problem* representation" rather than "*knowledge* representation." (See, for example, Amarel's (1968) discussion of the Missionaries and Cannibals problem.) In the 1970s, AI emphasized the development of "expert systems" (also called "knowledge-based systems") that could, if given the appropriate domain knowledge, match or exceed the performance of human experts on narrowly defined tasks. For example, the first expert system, DENDRAL (Feigenbaum *et al.*, 1971; Lindsay *et al.*, 1980), interpreted the output of a mass spectrometer (a type of instrument used to analyze the structure of organic chemical compounds) as accurately as expert chemists. Although the success of DENDRAL was instrumental in convincing the AI research community of the importance of knowledge representation, the representational formalisms used in DENDRAL are highly specific to the domain of chemistry. Over time, researchers became interested in standardized knowledge representation formalisms and ontologies that could streamline the process of creating new expert systems. In so doing, they ventured into territory previously explored by philosophers of science and of language. The discipline imposed in AI by the need for one's theories to "work" has led to more rapid and deeper progress than was the case when these problems were the exclusive domain of philosophy (although it has at times also led to the repeated reinvention of the wheel).

The creation of comprehensive taxonomies or classifications dates back to ancient times. Aristotle (384–322 B.C.) strongly emphasized classification and categorization schemes. His *Organon*, a collection of works on logic assembled by his students after his death, included a treatise called *Categories* in which he attempted to construct what we would now call an upper ontology. He also introduced the notions of **genus** and **species** for lower-level classification, although not with their modern, specifically biological meaning. Our present system of biological classification, including the use of "binomial nomenclature" (classification via genus and species in the technical sense), was invented by the Swedish biologist Carolus Linnaeus, or Carl von Linne (1707–1778). The problems associated with natural kinds and inexact category boundaries have been addressed by Wittgenstein (1953), Quine (1953), Lakoff (1987), and Schwartz (1977), among others.

Interest in larger-scale ontologies is increasing. The CYC project (Lenat, 1995; Lenat and Guha, 1990) has released a 6,000-concept upper ontology with 60,000 facts, and licenses

a much larger global ontology. The IEEE has established subcommittee P1600.1, the Standard Upper Ontology Working Group, and the Open Mind Initiative has enlisted over 7,000 Internet users to enter more than 400,000 facts about commonsense concepts. On the Web, standards such as RDF, XML, and the Semantic Web (Berners-Lee *et al.*, 2001) are emerging, but are not yet widely used. The conferences on *Formal Ontology in Information Systems* (FOIS) contain many interesting papers on both general and domain-specific ontologies.

The taxonomy used in this chapter was developed by the authors and is based in part on their experience in the CYC project and in part on work by Hwang and Schubert (1993) and Davis (1990). An inspirational discussion of the general project of commonsense knowledge representation appears in Hayes's (1978, 1985b) "The Naive Physics Manifesto."

The representation of time, change, actions, and events has been studied extensively in philosophy and theoretical computer science as well as in AI. The oldest approach is **temporal logic**, which is a specialized logic in which each model describes a complete trajectory through time (usually either linear or branching), rather than just a static relational structure. The logic includes **modal operators** that are applied to formulas; $\Box p$ means "p will be true at all times in the future," and $\Diamond p$ means "p will be true at some time in the future." The study of temporal logic was initiated by Aristotle and the Megarian and Stoic schools in ancient Greece. In modern times, Findlay (1941) was the first to suggest a formal calculus for reasoning about time, but the work of Arthur Prior (1967) is considered the most influential. Textbooks on temporal logic include those by Rescher and Urquhart (1971) and van Benthem (1983).

Theoretical computer scientists have long been interested in formalizing the properties of programs, viewed as sequences of computational actions. Burstall (1974) introduced the idea of using modal operators to reason about computer programs. Soon thereafter, Vaughan Pratt (1976) designed **dynamic logic**, in which modal operators indicate the effects of programs or other actions (see also Harel, 1984). For instance, in dynamic logic, if α is the name of a program, then "$[\alpha]p$" means "p would be true in all world states resulting from executing program α in the current world state", and "$\langle\alpha\rangle p$" means "p would be true in at least one world state resulting from executing program α in the current world state." Dynamic logic was applied to the actual analysis of programs by Fischer and Ladner (1977). Pnueli (1977) introduced the idea of using classical temporal logic to reason about programs.

Whereas temporal logic puts time directly into the model theory of the language, representations of time in AI have tended to incorporate axioms about times and events explicitly in the knowledge base, giving time no special status in the logic. This approach can allow for greater clarity and flexibility in some cases. Also, temporal knowledge expressed in first-order logic can be more easily integrated with other knowledge that has been accumulated in that notation.

The earliest treatment of time and action in AI was John McCarthy's (1963) situation calculus. The first AI system to make substantial use of general-purpose reasoning about actions in first-order logic was QA3 (Green, 1969b). Kowalski (1979b) developed the idea of reifying propositions within situation calculus.

The **frame problem** was first recognized by McCarthy and Hayes (1969). Many researchers considered the problem insoluble within first-order logic, and it spurred a great

deal of research into nonmonotonic logics. Philosophers from Dreyfus (1972) to Crockett (1994) have cited the frame problem as one symptom of the inevitable failure of the entire AI enterprise. The partial solution of the representational frame problem using successor-state axioms is due to Ray Reiter (1991); a solution of the inferential frame problem can be traced to work by Holldobler and Schneeberger (1990) on what became known as fluent calculus (Thielscher, 1999). The discussion in this chapter is based partly on the analyses by Lin and Reiter (1997) and Thielscher (1999). Books by Shanahan (1997) and Reiter (2001b) give complete, modern treatments of reasoning about action in situation calculus.

The partial resolution of the frame problem has rekindled interest in the declarative approach to reasoning about actions, which had been eclipsed by special-purpose planning systems since the early 1970s. (See Chapter 11.) Under the banner of **cognitive robotics**, much progress has been made on logical representations of action and time. The GOLOG language uses the full expressive power of logic programming to describe actions and plans (Levesque *et al.*, 1997a) and has been extended to handle concurrent actions (Giacomo *et al.*, 2000), stochastic environments (Boutilier *et al.*, 2000), and sensing (Reiter, 2001a).

COGNITIVE ROBOTICS

The event calculus was introduced by Kowalski and Sergot (1986) to handle continuous time, and there have been several variations (Sadri and Kowalski, 1995). Shanahan (1999) presents a good short overview. James Allen introduced time intervals for the same reason (Allen, 1983, 1984), arguing that intervals were much more natural than situations for reasoning about extended and concurrent events. Peter Ladkin (1986a, 1986b) introduced "concave" time intervals (intervals with gaps; essentially, unions of ordinary "convex" time intervals) and applied the techniques of mathematical abstract algebra to time representation. Allen (1991) systematically investigates the wide variety of techniques available for time representation. Shoham (1987) describes the reification of events and sets forth a novel scheme of his own for the purpose. There are significant commonalities between the event-based ontology given in this chapter and an analysis of events due to the philosopher Donald Davidson (1980). The **histories** in Pat Hayes's (1985a) ontology of liquids also have much the same flavor.

The question of the ontological status of substances has a long history. Plato proposed that substances were abstract entities entirely distinct from physical objects; he would say $MadeOf(Butter_3, Butter)$ rather than $Butter_3 \in Butter$. This leads to a substance hierarchy in which, for example, $UnsaltedButter$ is a more specific substance than $Butter$. The position adopted in this chapter, in which substances are categories of objects, was championed by Richard Montague (1973). It has also been adopted in the CYC project. Copeland (1993) mounts a serious, but not invincible, attack. The alternative approach mentioned in the chapter, in which butter is one object consisting of all buttery objects in the universe, was proposed originally by the Polish logician Leśniewski (1916). His **mereology** (the name is derived from the Greek word for "part") used the part–whole relation as a substitute for mathematical set theory, with the aim of eliminating abstract entities such as sets. A more readable exposition of these ideas is given by Leonard and Goodman (1940), and Goodman's *The Structure of Appearance* (1977) applies the ideas to various problems in knowledge representation. While some aspects of the mereological approach are awkward—for example, the need for a separate inheritance mechanism based on part–whole relations—the approach gained the

MEREOLOGY

support of Quine (1960). Harry Bunt (1985) has provided an extensive analysis of its use in knowledge representation.

MODAL LOGIC

Mental objects and states have been the subject of intensive study in philosophy and AI. **Modal logic** is the classical method for reasoning about knowledge in philosophy. Modal logic augments first-order logic with modal operators, such as B (believes) and K (knows), that take *sentences* rather than terms as arguments. The proof theory for modal logic restricts substitution within modal contexts, thereby achieving referential opacity. The modal logic of knowledge was invented by Jaakko Hintikka (1962). Saul Kripke (1963) defined the semantics of the modal logic of knowledge in terms of **possible worlds**. Roughly speaking, a world is possible for an agent if it is consistent with everything the agent knows. From this, one can derive rules of inference involving the K operator. Robert C. Moore relates the modal logic of knowledge to a style of reasoning about knowledge that refers directly to possible worlds in first-order logic (Moore, 1980, 1985). Modal logic can be an intimidatingly arcane field, but it has found significant applications in reasoning about information in distributed computer systems. The book *Reasoning about Knowledge* by Fagin *et al.* (1995) provides a thorough introduction to the modal approach. The biennial conference on *Theoretical Aspects of Reasoning About Knowledge* (TARK) covers applications of the theory of knowledge in AI, economics, and distributed systems.

The syntactic theory of mental objects was first studied in depth by Kaplan and Montague (1960), who showed that it led to paradoxes if not handled carefully. Because it has a natural model in terms of beliefs as physical configurations of a computer or a brain, it has been popular in AI in recent years. Konolige (1982) and Haas (1986) used it to describe inference engines of limited power, and Morgenstern (1987) showed how it could be used to describe knowledge preconditions in planning. The methods for planning observation actions in Chapter 12 are based on the syntactic theory. Ernie Davis (1990) gives an excellent comparison of the syntactic and modal theories of knowledge.

The Greek philosopher Porphyry (c. 234–305 A.D.), commenting on Aristotle's *Categories*, drew what might qualify as the first semantic network. Charles S. Peirce (1909) developed existential graphs as the first semantic network formalism using modern logic. Ross Quillian (1961), driven by an interest in human memory and language processing, initiated work on semantic networks within AI. An influential paper by Marvin Minsky (1975) presented a version of semantic networks called **frames**; a frame was a representation of an object or category, with attributes and relations to other objects or categories. Although the paper served to initiate interest in the field of knowledge representation *per se*, it was criticized as a recycling of earlier ideas developed in object-oriented programming, such as inheritance and the use of default values (Dahl *et al.*, 1970; Birtwistle *et al.*, 1973). It is not clear to what extent the latter papers on object-oriented programming were influenced in turn by early AI work on semantic networks.

The question of semantics arose quite acutely with respect to Quillian's semantic networks (and those of others who followed his approach), with their ubiquitous and very vague "IS-A links," as well as other early knowledge representation formalisms such as that of MERLIN (Moore and Newell, 1973) with its mysterious "flat" and "cover" operations. Woods' (1975) famous article "What's In a Link?" drew the attention of AI researchers to the

need for precise semantics in knowledge representation formalisms. Brachman (1979) elaborated on this point and proposed solutions. Patrick Hayes's (1979) "The Logic of Frames" cut even deeper, claiming that "Most of 'frames' is just a new syntax for parts of first-order logic." Drew McDermott's (1978b) "Tarskian Semantics, or, No Notation without Denotation!" argued that the model-theoretic approach to semantics used in first-order logic should be applied to all knowledge representation formalisms. This remains a controversial idea; notably, McDermott himself has reversed his position in "A Critique of Pure Reason" (McDermott, 1987). NETL (Fahlman, 1979) was a sophisticated semantic network system whose IS-A links (called "virtual copy," or VC, links) were based more on the notion of "inheritance" characteristic of frame systems or of object-oriented programming languages than on the subset relation and were much more precisely defined than Quillian's links from the pre-Woods era. NETL is particularly intriguing because it was intended to be implemented in parallel hardware to overcome the difficulty of retrieving information from large semantic networks. David Touretzky (1986) subjects inheritance to rigorous mathematical analysis. Selman and Levesque (1993) discuss the complexity of inheritance with exceptions, showing that in most formulations it is NP-complete.

The development of description logics is the most recent stage in a long line of research aimed at finding useful subsets of first-order logic for which inference is computationally tractable. Hector Levesque and Ron Brachman (1987) showed that certain logical constructs—notably, certain uses of disjunction and negation—were primarily responsible for the intractability of logical inference. Building on the KL-ONE system (Schmolze and Lipkis, 1983), a number of systems have been developed whose designs incorporate the results of theoretical complexity analysis, most notably KRYPTON (Brachman *et al.*, 1983) and Classic (Borgida *et al.*, 1989). The result has been a marked increase in the speed of inference and a much better understanding of the interaction between complexity and expressiveness in reasoning systems. Calvanese *et al.* (1999) summarize the state of the art. Against this trend, Doyle and Patil (1991) have argued that restricting the expressiveness of a language either makes it impossible to solve certain problems or encourages the user to circumvent the language restrictions through nonlogical means.

The three main formalisms for dealing with nonmonotonic inference—circumscription (McCarthy, 1980), default logic (Reiter, 1980), and modal nonmonotonic logic (McDermott and Doyle, 1980)—were all introduced in one special issue of the AI Journal. Answer set programming can be seen as an extension of negation as failure or as a refinement of circumscription; the underlying theory of stable model semantics was introduced by Gelfond and Lifschitz (1988) and the leading answer set programming systems are DLV (Eiter *et al.*, 1998) and SMODELS (Niemelä *et al.*, 2000). The disk drive example comes from the SMODELS user manual (Syrjänen, 2000). Lifschitz (2001) discusses the use of answer set programming for planning. Brewka *et al.* (1997) give a good overview of the various approaches to nonmonotonic logic. Clark (1978) covers the negation-as-failure approach to logic programming and Clark completion. Van Emden and Kowalski (1976) show that every Prolog program without negation has a unique minimal model. Recent years have seen renewed interest in applications of nonmonotonic logics to large-scale knowledge representation systems. The BENINQ systems for handling insurance benefits inquiries was perhaps the first commercially success-

ful application of a nonmonotonic inheritance system (Morgenstern, 1998). Lifschitz (2001) discusses the application of answer set programming to planning. A variety of nonmonotonic reasoning systems based on logic programming are documented in the proceedings of the conferences on *Logic Programming and Nonmonotonic Reasoning* (LPNMR).

The study of truth maintenance systems began with the TMS (Doyle, 1979) and RUP (McAllester, 1980) systems, both of which were essentially JTMSs. The ATMS approach was described in a series of papers by Johan de Kleer (1986a, 1986b, 1986c). *Building Problem Solvers* (Forbus and de Kleer, 1993) explains in depth how TMSs can be used in AI applications. Nayak and Williams (1997) show how an efficient TMS makes it feasible to plan the operations of a NASA spacecraft in real time.

For obvious reasons, this chapter does not cover *every* area of knowledge representation in depth. The three principal topics omitted are the following:

QUALITATIVE PHYSICS

◇ **Qualitative physics**: Qualitative physics is a subfield of knowledge representation concerned specifically with constructing a logical, nonnumeric theory of physical objects and processes. The term was coined by Johan de Kleer (1975), although the enterprise could be said to have started in Fahlman's (1974) BUILD, a sophisticated planner for constructing complex towers of blocks. Fahlman discovered in the process of designing it that most of the effort (80%, by his estimate) went into modeling the physics of the blocks world to calculate the stability of various subassemblies of blocks, rather than into planning per se. He sketches a hypothetical naive-physics-like process to explain why young children can solve BUILD-like problems without access to the high-speed floating-point arithmetic used in BUILD's physical modeling. Hayes (1985a) uses "histories"—four-dimensional slices of space-time similar to Davidson's events—to construct a fairly complex naive physics of liquids. Hayes was the first to prove that a bath with the plug in will eventually overflow if the tap keeps running and that a person who falls into a lake will get wet all over. De Kleer and Brown (1985) and Ken Forbus (1985) attempted to construct something like a general-purpose theory of the physical world, based on qualitative abstractions of physical equations. In recent years, qualitative physics has developed to the point where it is possible to analyze an impressive variety of complex physical systems (Sacks and Joskowicz, 1993; Yip, 1991). Qualitative techniques have been used to construct novel designs for clocks, windscreen wipers, and six-legged walkers (Subramanian, 1993; Subramanian and Wang, 1994). The collection *Readings in Qualitative Reasoning about Physical Systems* (Weld and de Kleer, 1990) provides a good introduction to the field.

SPATIAL REASONING

◇ **Spatial reasoning**: The reasoning necessary to navigate in the wumpus world and shopping world is trivial in comparison to the rich spatial structure of the real world. The earliest serious attempt to capture commonsense reasoning about space appears in the work of Ernest Davis (1986, 1990). The region connection calculus of Cohn *et al.* (1997) supports a form of qualitative spatial reasoning and has led to new kinds of geographical information system. As with qualitative physics, an agent can go a long way, so to speak, without resorting to a full metric representation. When such a representation is necessary, techniques developed in robotics (Chapter 25) can be used.

PSYCHOLOGICAL REASONING

◇ **Psychological reasoning**: Psychological reasoning involves the development of a working *psychology* for artificial agents to use in reasoning about themselves and other agents. This is often based on so-called folk psychology, the theory that humans in general are believed to use in reasoning about themselves and other humans. When AI researchers provide their artificial agents with psychological theories for reasoning about other agents, the theories are frequently based on the researchers' description of the logical agents' own design. Psychological reasoning is currently most useful within the context of natural language understanding, where divining the speaker's intentions is of paramount importance.

The proceedings of the international conferences on *Principles of Knowledge Representation and Reasoning* provide the most up-to-date sources for work in this area. *Readings in Knowledge Representation* (Brachman and Levesque, 1985) and *Formal Theories of the Commonsense World* (Hobbs and Moore, 1985) are excellent anthologies on knowledge representation; the former focuses more on historically important papers in representation languages and formalisms, the latter on the accumulation of the knowledge itself. Davis (1990), Stefik (1995), and Sowa (1999) provide textbook introductions to knowledge representation.

EXERCISES

10.1 Write sentences to define the effects of the *Shoot* action in the wumpus world. Describe its effects on the wumpus and remember that shooting uses the agent's arrow.

10.2 Within situation calculus, write an axiom to associate time 0 with the situation S_0 and another axiom to associate the time t with any situation that is derived from S_0 by a sequence of t actions.

10.3 In this exercise, we will consider the problem of planning a route for a robot to take from one city to another. The basic action taken by the robot is $Go(x, y)$, which takes it from city x to city y if there is a direct route between those cities. $DirectRoute(x, y)$ is true if and only if there is a direct route from x to y; you can assume that all such facts are already in the KB. (See the map on page 63.) The robot begins in Arad and must reach Bucharest.

a. Write a suitable logical description of the initial situation of the robot.

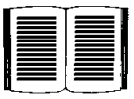
b. Write a suitable logical query whose solutions will provide possible paths to the goal.

c. Write a sentence describing the Go action.

d. Now suppose that following the direct route between two cities consumes an amount of fuel equal to the distance between the cities. The robot starts with fuel at full capacity. Augment your representation to include these considerations. Your action description should be such that the query you specified earlier will still result in feasible plans.

e. Describe the initial situation, and write a new rule or rules describing the Go action.

f. Now suppose some of the vertices are also gas stations, at which the robot can fill its tank. Extend your representation and write all the rules needed to describe gas stations, including the *Fillup* action.



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

10.5 Represent the following seven sentences using and extending the representations developed in the chapter:

a. Water is a liquid between 0 and 100 degrees.

b. Water boils at 100 degrees.

c. The water in John's water bottle is frozen.

d. Perrier is a kind of water.

e. John has Perrier in his water bottle.

f. All liquids have a freezing point.

g. A liter of water weighs more than a liter of alcohol.

Now repeat the exercise using a representation based on the mereological approach, in which, for example, $Water$ is an object containing as parts all the water in the world.

10.6 Write definitions for the following:

a. $ExhaustivePartDecomposition$

b. $PartPartition$

c. $PartwiseDisjoint$

These should be analogous to the definitions for $ExhaustiveDecomposition$, $Partition$, and $Disjoint$. Is it the case that $PartPartition(s, BunchOf(s))$? If so, prove it; if not, give a counterexample and define sufficient conditions under which it does hold.

10.7 Write a set of sentences that allows one to calculate the price of an individual tomato (or other object), given the price per pound. Extend the theory to allow the price of a bag of tomatoes to be calculated.

10.8 An alternative scheme for representing measures involves applying the units function to an abstract length object. In such a scheme, one would write $Inches(Length(L_1)) = 1.5$. How does this scheme compare with the one in the chapter? Issues include conversion axioms, names for abstract quantities (such as "50 dollars"), and comparisons of abstract measures in different units (50 inches is more than 50 centimeters).

10.9 Construct a representation for exchange rates between currencies that allows fluctuations on a daily basis.

10.10 This exercise concerns the relationships between event categories and the time intervals in which they occur.

a. Define the predicate $T(c, i)$ in terms of $During$ and $\in$.

b. Explain precisely why we do not need two different notations to describe conjunctive event categories.

c. Give a formal definition for $T(OneOf(p, q), i)$ and $T(Either(p, q), i)$.

d. Explain why it makes sense to have two forms of negation of events, analogous to the two forms of disjunction. Call them Not and $Never$ and give them formal definitions.

10.11 Define the predicate *Fixed*, where $Fixed(Location(x))$ means that the location of object x is fixed over time.

10.12 Define the predicates *Before*, *After*, *During*, and *Overlap*, using the predicate *Meet* and the functions *Start* and *End*, but not the function *Time* or the predicate $<$.

10.13 Section 10.5 used the predicates *Link* and *LinkText* to describe connections between web pages. Using the *InTag* and *GetPage* predicates, among others, write definitions for *Link* and *LinkText*.

10.14 One part of the shopping process that was not covered in this chapter is checking for compatibility between items. For example, if a customer orders a computer, is it matched with the right peripherals? If a digital camera is ordered, does it have the right memory card and batteries? Write a knowledge base that will decide whether a set of items is compatible and that can be used to suggest replacements or additional items if they are not compatible. Make sure that the knowledge base works with at least one line of products, and is easily extensible to other lines.

10.15 Add rules to extend the definition of the predicate $Name(s, c)$ so that a string such as "laptop computer" matches against the appropriate category names from a variety of stores. Try to make your definition general. Test it by looking at ten online stores, and at the category names they give for three different categories. For example, for the category of laptops, we found the names "Notebooks," "Laptops," "Notebook Computers," "Notebook," "Laptops and Notebooks," and "Notebook PCs." Some of these can be covered by explicit *Name* facts, while others could be covered by rules for handling plurals, conjunctions, etc.

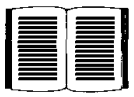
10.16 A complete solution to the problem of inexact matches to the buyer's description in shopping is very difficult and requires a full array of natural language processing and information retrieval techniques. (See Chapters 22 and 23.) One small step is to allow the user to specify minimum and maximum values for various attributes. We will insist that the buyer use the following grammar for product descriptions:

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

Here, *Category* names a product category, *Attribute* is some feature such as "CPU" or "price," and *Value* is the target value for the attribute. So the query "computer with at least a 2.5-GHz CPU for under $1000" must be re-expressed as "computer with CPU $>$ 2.5 GHz and price $<$ $1000." Implement a shopping agent that accepts descriptions in this language.

10.17 Our description of Internet shopping omitted the all-important step of actually *buying* the product. Provide a formal logical description of buying, using event calculus. That is, define the sequence of events that occurs when a buyer submits a credit card purchase and then eventually gets billed and receives the product.

10.18 Describe the event of trading something for something else. Describe buying as a kind of trading in which one of the objects traded is a sum of money.



10.19 The two preceding exercises assume a fairly primitive notion of ownership. For example, the buyer starts by *owning* the dollar bills. This picture begins to break down when, for example, one's money is in the bank, because there is no longer any specific collection of dollar bills that one owns. The picture is complicated still further by borrowing, leasing, renting, and bailment. Investigate the various commonsense and legal concepts of ownership, and propose a scheme by which they can be represented formally.

10.20 You are to create a system for advising computer science undergraduates on what courses to take over an extended period in order to satisfy the program requirements. (Use whatever requirements are appropriate for your institution.) First, decide on a vocabulary for representing all the information, and then represent it; then use an appropriate query to the system, that will return a legal program of study as a solution. You should allow for some tailoring to individual students, in that your system should ask what courses or equivalents the student has already taken, and not generate programs that repeat those courses.

Suggest ways in which your system could be improved—for example to take into account knowledge about student preferences, the workload, good and bad instructors, and so on. For each kind of knowledge, explain how it could be expressed logically. Could your system easily incorporate this information to find the *best* program of study for a student?

10.21 Figure 10.1 shows the top levels of a hierarchy for everything. Extend it to include as many real categories as possible. A good way to do this is to cover all the things in your everyday life. This includes objects and events. Start with waking up, and proceed in an orderly fashion noting everything that you see, touch, do, and think about. For example, a random sampling produces music, news, milk, walking, driving, gas, Soda Hall, carpet, talking, Professor Fateman, chicken curry, tongue, $7, sun, the daily newspaper, and so on.

You should produce both a single hierarchy chart (on a large sheet of paper) and a listing of objects and categories with the relations satisfied by members of each category. Every object should be in a category, and every category should be in the hierarchy.

10.22 (Adapted from an example by Doug Lenat.) Your mission is to capture, in logical form, enough knowledge to answer a series of questions about the following simple sentence:

> Yesterday John went to the North Berkeley Safeway supermarket and bought two pounds of tomatoes and a pound of ground beef.

Start by trying to represent the content of the sentence as a series of assertions. You should write sentences that have straightforward logical structure (e.g., statements that objects have certain properties, that objects are related in certain ways, that all objects satisfying one property satisfy another). The following might help you get started:

- Which classes, objects, and relations would you need? What are their parents, siblings and so on? (You will need events and temporal ordering, among other things.)
- Where would they fit in a more general hierarchy?
- What are the constraints and interrelationships among them?
- How detailed must you be about each of the various concepts?

The knowledge base you construct must be capable of answering a list of questions that we will give shortly. Some of the questions deal with the material stated explicitly in the story,

but most of them require one to have other background knowledge—to read between the lines. You'll have to deal with what kind of things are at a supermarket, what is involved with purchasing the things one selects, what will purchases be used for, and so on. Try to make your representation as general as possible. To give a trivial example: don't say "People buy food from Safeway," because that won't help you with those who shop at another supermarket. Don't say "Joe made spaghetti with the tomatoes and ground beef," because that won't help you with anything else at all. Also, don't turn the questions into answers; for example, question (c) asks "Did John buy any meat?"—not "Did John buy a pound of ground beef?"

Sketch the chains of reasoning that would answer the questions. In the process of doing so, you will no doubt need to create additional concepts, make additional assertions, and so on. If possible, use a logical reasoning system to demonstrate the sufficiency of your knowledge base. Many of the things you write might be only approximately correct in reality, but don't worry too much; the idea is to extract the common sense that lets you answer these questions at all. A truly complete answer to this question is *extremely* difficult, probably beyond the state of the art of current knowledge representation. But you should be able to put together a consistent set of axioms for the limited questions posed here.

a. Is John a child or an adult? [Adult]

b. Does John now have at least two tomatoes? [Yes]

c. Did John buy any meat? [Yes]

d. If Mary was buying tomatoes at the same time as John, did he see her? [Yes]

e. Are the tomatoes made in the supermarket? [No]

f. What is John going to do with the tomatoes? [Eat them]

g. Does Safeway sell deodorant? [Yes]

h. Did John bring any money to the supermarket? [Yes]

i. Does John have less money after going to the supermarket? [Yes]

10.23 Make the necessary additions or changes to your knowledge base from the previous exercise so that the questions that follow can be answered. Show that they can indeed be answered by the KB, and include in your report a discussion of the fixes, explaining why they were needed, whether they were minor or major, and so on.

a. Are there other people in Safeway while John is there? [Yes—staff!]

b. Is John a vegetarian? [No]

c. Who owns the deodorant in Safeway? [Safeway Corporation]

d. Did John have an ounce of ground beef? [Yes]

e. Does the Shell station next door have any gas? [Yes]

f. Do the tomatoes fit in John's car trunk? [Yes]

10.24 Recall that inheritance information in semantic networks can be captured logically by suitable implication sentences. In this exercise, we will consider the efficiency of using such sentences for inheritance.

a. Consider the information content in a used-car catalog such as Kelly's Blue Book—for example, that 1973 Dodge Vans are worth $575. Suppose all this information (for 11,000 models) is encoded as logical rules, as suggested in the chapter. Write down three such rules, including that for 1973 Dodge Vans. How would you use the rules to find the value of a *particular* car (e.g., JB, which is a 1973 Dodge Van), given a backward-chaining theorem prover such as Prolog?

b. Compare the time efficiency of the backward-chaining method for solving this problem with the inheritance method used in semantic nets.

c. Explain how forward chaining allows a logic-based system to solve the same problem efficiently, assuming that the KB contains only the 11,000 rules about prices.

d. Describe a situation in which neither forward nor backward chaining on the rules will allow the price query for an individual car to be handled efficiently.

e. Can you suggest a solution enabling this type of query to be solved efficiently in all cases in logic systems? [*Hint:* Remember that two cars of the same category have the same price.]

10.25 One might suppose that the syntactic distinction between unboxed links and singly boxed links in semantic networks is unnecessary, because singly boxed links are always attached to categories; an inheritance algorithm could simply assume that an unboxed link attached to a category is intended to apply to all members of that category. Show that this argument is fallacious, giving examples of errors that would arise.

13 UNCERTAINTY

In which we see what an agent should do when not all is crystal clear.

13.1 ACTING UNDER UNCERTAINTY

UNCERTAINTY

The logical agents described in Parts III and IV make the epistemological commitment that propositions are true, false, or unknown. When an agent knows enough facts about its environment, the logical approach enables it to derive plans that are guaranteed to work. This is a good thing. Unfortunately, *agents almost never have access to the whole truth about their environment.* Agents must, therefore, act under **uncertainty**. For example, an agent in the wumpus world of Chapter 7 has sensors that report only local information; most of the world is not immediately observable. A wumpus agent often will find itself unable to discover which of two squares contains a pit. If those squares are *en route* to the gold, then the agent might have to take a chance and enter one of the two squares.

The real world is far more complex than the wumpus world. For a logical agent, it might be impossible to construct a complete and correct description of how its actions will work. Suppose, for example, that the agent wants to drive someone to the airport to catch a flight and is considering a plan, A_{90}, that involves leaving home 90 minutes before the flight departs and driving at a reasonable speed. Even though the airport is only about 15 miles away, the agent will not be able to conclude with certainty that "Plan A_{90} will get us to the airport in time." Instead, it reaches the weaker conclusion "Plan A_{90} will get us to the airport in time, as long as my car doesn't break down or run out of gas, and I don't get into an accident, and there are no accidents on the bridge, and the plane doesn't leave early, and . . . ." None of these conditions can be deduced, so the plan's success cannot be inferred. This is an example of the **qualification problem** mentioned in Chapter 10.

If a logical agent cannot conclude that any particular course of action achieves its goal, then it will be unable to act. Conditional planning can overcome uncertainty to some extent, but only if the agent's sensing actions can obtain the required information and only if there are not too many different contingencies. Another possible solution would be to endow the agent with a simple but incorrect theory of the world that *does* enable it to derive a plan;

presumably, such plans will work *most* of the time, but problems arise when events contradict the agent's theory. Moreover, handling the tradeoff between the accuracy and usefulness of the agent's theory seems itself to require reasoning about uncertainty. In sum, no purely logical agent will be able to conclude that plan A_{90} is the right thing to do.

Nonetheless, let us suppose that A_{90} *is* in fact the right thing to do. What do we mean by saying this? As we discussed in Chapter 2, we mean that out of all the plans that could be executed, A_{90} is expected to maximize the agent's performance measure, given the information it has about the environment. The performance measure includes getting to the airport in time for the flight, avoiding a long, unproductive wait at the airport, and avoiding speeding tickets along the way. The information the agent has cannot guarantee any of these outcomes for A_{90}, but it can provide some degree of belief that they will be achieved. Other plans, such as A_{120}, might increase the agent's belief that it will get to the airport on time, but also increase the likelihood of a long wait. *The right thing to do—the* ***rational decision****—therefore depends on both the relative importance of various goals and the likelihood that, and degree to which, they will be achieved.* The remainder of this section hones these ideas, in preparation for the development of the general theories of uncertain reasoning and rational decisions that we present in this and subsequent chapters.

Handling uncertain knowledge

In this section, we look more closely at the nature of uncertain knowledge. We will use a simple diagnosis example to illustrate the concepts involved. Diagnosis—whether for medicine, automobile repair, or whatever—is a task that almost always involves uncertainty. Let us try to write rules for dental diagnosis using first-order logic, so that we can see how the logical approach breaks down. Consider the following rule:

$$\forall p \;\; Symptom(p, Toothache) \;\Rightarrow\; Disease(p, Cavity) \;.$$

The problem is that this rule is wrong. Not all patients with toothaches have cavities; some of them have gum disease, an abscess, or one of several other problems:

$$\forall p \;\; Symptom(p, Toothache) \;\Rightarrow\; Disease(p, Cavity) \vee Disease(p, GumDisease) \vee Disease(p, Abscess) \ldots$$

Unfortunately, in order to make the rule true, we have to add an almost unlimited list of possible causes. We could try turning the rule into a causal rule:

$$\forall p \;\; Disease(p, Cavity) \;\Rightarrow\; Symptom(p, Toothache) \;.$$

But this rule is not right either; not all cavities cause pain. The only way to fix the rule is to make it logically exhaustive: to augment the left-hand side with all the qualifications required for a cavity to cause a toothache. Even then, for the purposes of diagnosis, one must also take into account the possibility that the patient might have a toothache and a cavity that are unconnected.

Trying to use first-order logic to cope with a domain like medical diagnosis thus fails for three main reasons:

LAZINESS

- ◇ **Laziness**: It is too much work to list the complete set of antecedents or consequents needed to ensure an exceptionless rule and too hard to use such rules.

THEORETICAL IGNORANCE

◇ **Theoretical ignorance**: Medical science has no complete theory for the domain.

PRACTICAL IGNORANCE

◇ **Practical ignorance**: Even if we know all the rules, we might be uncertain about a particular patient because not all the necessary tests have been or can be run.

The connection between toothaches and cavities is just not a logical consequence in either direction. This is typical of the medical domain, as well as most other judgmental domains: law, business, design, automobile repair, gardening, dating, and so on. The agent's knowledge can at best provide only a **degree of belief** in the relevant sentences. Our main tool for dealing with degrees of belief will be **probability theory**, which assigns to each sentence a numerical degree of belief between 0 and 1. (Some alternative methods for uncertain reasoning are covered in Section 14.7.)

DEGREE OF BELIEF

PROBABILITY THEORY

Probability provides a way of **summarizing** *the uncertainty that comes from our laziness and ignorance.* We might not know for sure what afflicts a particular patient, but we believe that there is, say, an 80% chance—that is, a probability of 0.8—that the patient has a cavity if he or she has a toothache. That is, we expect that out of all the situations that are indistinguishable from the current situation as far as the agent's knowledge goes, the patient will have a cavity in 80% of them. This belief could be derived from statistical data—80% of the toothache patients seen so far have had cavities—or from some general rules, or from a combination of evidence sources. The 80% summarizes those cases in which all the factors needed for a cavity to cause a toothache are present and other cases in which the patient has both toothache and cavity but the two are unconnected. The missing 20% summarizes all the other possible causes of toothache that we are too lazy or ignorant to confirm or deny.

Assigning a probability of 0 to a given sentence corresponds to an unequivocal belief that the sentence is false, while assigning a probability of 1 corresponds to an unequivocal belief that the sentence is true. Probabilities between 0 and 1 correspond to intermediate degrees of belief in the truth of the sentence. The sentence itself is *in fact* either true or false. It is important to note that a degree of belief is different from a degree of truth. A probability of 0.8 does not mean "80% true" but rather an 80% degree of belief—that is, a fairly strong expectation. Thus, probability theory makes the same ontological commitment as logic—namely, that facts either do or do not hold in the world. Degree of truth, as opposed to degree of belief, is the subject of **fuzzy logic**, which is covered in Section 14.7.

In logic, a sentence such as "The patient has a cavity" is true or false depending on the interpretation and the world; it is true just when the fact it refers to is the case. In probability theory, a sentence such as "The probability that the patient has a cavity is 0.8" is about the agent's beliefs, not directly about the world. These beliefs depend on the percepts that the agent has received to date. These percepts constitute the **evidence** on which probability assertions are based. For example, suppose that the agent has drawn a card from a shuffled pack. Before looking at the card, the agent might assign a probability of $1/52$ to its being the ace of spades. After looking at the card, an appropriate probability for the same proposition would be 0 or 1. Thus, an assignment of probability to a proposition is analogous to saying whether a given logical sentence (or its negation) is entailed by the knowledge base, rather than whether or not it is true. Just as entailment status can change when more sentences are

EVIDENCE

added to the knowledge base, probabilities can change when more evidence is acquired.[1]

All probability statements must therefore indicate the evidence with respect to which the probability is being assessed. As the agent receives new percepts, its probability assessments are updated to reflect the new evidence. Before the evidence is obtained, we talk about **prior** or **unconditional** probability; after the evidence is obtained, we talk about **posterior** or **conditional** probability. In most cases, an agent will have some evidence from its percepts and will be interested in computing the posterior probabilities of the outcomes it cares about.

Uncertainty and rational decisions

The presence of uncertainty radically changes the way an agent makes decisions. A logical agent typically has a goal and executes any plan that is guaranteed to achieve it. An action can be selected or rejected on the basis of whether it achieves the goal, regardless of what other actions might achieve. When uncertainty enters the picture, this is no longer the case. Consider again the A_{90} plan for getting to the airport. Suppose it has a 95% chance of succeeding. Does this mean it is a rational choice? Not necessarily: There might be other plans, such as A_{120}, with higher probabilities of success. If it is vital not to miss the flight, then it is worth risking the longer wait at the airport. What about A_{1440}, a plan that involves leaving home 24 hours in advance? In most circumstances, this is not a good choice, because, although it almost guarantees getting there on time, it involves an intolerable wait.

PREFERENCES
OUTCOMES

To make such choices, an agent must first have **preferences** between the different possible **outcomes** of the various plans. A particular outcome is a completely specified state, including such factors as whether the agent arrives on time and the length of the wait at the airport. We will be using **utility theory** to represent and reason with preferences. (The term **utility** is used here in the sense of "the quality of being useful," not in the sense of the electric company or water works.) Utility theory says that every state has a degree of usefulness, or utility, to an agent and that the agent will prefer states with higher utility.

UTILITY THEORY

The utility of a state is relative to the agent whose preferences the utility function is supposed to represent. For example, the payoff functions for games in Chapter 6 are utility functions. The utility of a state in which White has won a game of chess is obviously high for the agent playing White, but low for the agent playing Black. Or again, some players (including the authors) might be happy with a draw against the world champion, whereas other players (including the former world champion) might not. There is no accounting for taste or preferences: you might think that an agent who prefers jalapeño bubble-gum ice cream to chocolate chocolate chip is odd or even misguided, but you could not say the agent is irrational. A utility function can even account for altruistic behavior, simply by including the welfare of others as one of the factors contributing to the agent's own utility.

Preferences, as expressed by utilities, are combined with probabilities in the general theory of rational decisions called **decision theory**:

DECISION THEORY

Decision theory = probability theory + utility theory .

[1] This is quite different from a sentence's becoming true or false as the world changes. Handling a changing world via probabilities requires the same kinds of mechanisms—situations, intervals, and events—that we used in Chapter 10 for logical representations. These mechanisms are discussed in Chapter 15.

The fundamental idea of decision theory is that *an agent is rational if and only if it chooses the action that yields the highest expected utility, averaged over all the possible outcomes of the action.* This is called the principle of **Maximum Expected Utility** (MEU). We saw this principle in action in Chapter 6 when we touched briefly on optimal decisions in backgammon. We will see that it is in fact a completely general principle.

Design for a decision-theoretic agent

Figure 13.1 sketches the structure of an agent that uses decision theory to select actions. The agent is identical, at an abstract level, to the logical agent described in Chapter 7. The primary difference is that the decision-theoretic agent's knowledge of the current state is uncertain;

BELIEF STATE

the agent's **belief state** is a representation of the probabilities of all possible actual states of the world. As time passes, the agent accumulates more evidence and its belief state changes. Given the belief state, the agent can make probabilistic predictions of action outcomes and hence select the action with highest expected utility. This chapter and the next concentrate on the task of representing and computing with probabilistic information in general. Chapter 15 deals with methods for the specific tasks of representing and updating the belief state and predicting the environment. Chapter 16 covers utility theory in more depth, and Chapter 17 develops algorithms for making complex decisions.

```
function DT-AGENT(percept) returns an action
  static: belief_state, probabilistic beliefs about the current state of the world
          action, the agent's action

  update belief_state based on action and percept
  calculate outcome probabilities for actions,
      given action descriptions and current belief_state
  select action with highest expected utility
      given probabilities of outcomes and utility information
  return action
```

Figure 13.1 A decision-theoretic agent that selects rational actions. The steps will be fleshed out in the next five chapters.

13.2 Basic Probability Notation

Now that we have set up the general framework for a rational agent, we will need a formal language for representing and reasoning with uncertain knowledge. Any notation for describing degrees of belief must be able to deal with two main issues: the nature of the sentences to which degrees of belief are assigned and the dependence of the degree of belief on the agent's experience. The version of probability theory we present uses an extension of propositional

logic for its sentences. The dependence on experience is reflected in the syntactic distinction between prior probability statements, which apply before any evidence is obtained, and conditional probability statements, which include the evidence explicitly.

Propositions

Degrees of belief are always applied to **propositions**—assertions that such-and-such is the case. So far we have seen two formal languages—propositional logic and first-order logic—for stating propositions. Probability theory typically uses a language that is slightly more expressive than propositional logic. This section describes that language. (Section 14.6 discusses ways to ascribe degrees of belief to assertions in first-order logic.)

RANDOM VARIABLE

The basic element of the language is the **random variable**, which can be thought of as referring to a "part" of the world whose "status" is initially unknown. For example, *Cavity* might refer to whether my lower left wisdom tooth has a cavity. Random variables play a role similar to that of CSP variables in constraint satisfaction problems and that of proposition symbols in propositional logic. We will always capitalize the names of random variables. (However, we still use lowercase, single-letter names to represent an unknown random variable, for example: $P(a) = 1 - P(\neg a)$.)

DOMAIN

Each random variable has a **domain** of values that it can take on. For example, the domain of *Cavity* might be $\langle true, false \rangle$.[2] (We will use lowercase for the names of values.) The simplest kind of proposition asserts that a random variable has a particular value drawn from its domain. For example, $Cavity = true$ might represent the proposition that I do in fact have a cavity in my lower left wisdom tooth.

As with CSP variables, random variables are typically divided into three kinds, depending on the type of the domain:

BOOLEAN RANDOM VARIABLES

- ♢ **Boolean random variables**, such as *Cavity*, have the domain $\langle true, false \rangle$. We will often abbreviate a proposition such as $Cavity = true$ simply by the lowercase name $cavity$. Similarly, $Cavity = false$ would be abbreviated by $\neg cavity$.

DISCRETE RANDOM VARIABLES

- ♢ **Discrete random variables**, which include Boolean random variables as a special case, take on values from a *countable* domain. For example, the domain of *Weather* might be $\langle sunny, rainy, cloudy, snow \rangle$. The values in the domain must be mutually exclusive and exhaustive. Where no confusion arises, we will use, for example, $snow$ as an abbreviation for $Weather = snow$.

CONTINUOUS RANDOM VARIABLES

- ♢ **Continuous random variables** take on values from the real numbers. The domain can be either the entire real line or some subset such as the interval [0,1]. For example, the proposition $X = 4.02$ asserts that the random variable X has the exact value 4.02. Propositions concerning continuous random variables can also be inequalities, such as $X \leq 4.02$

With some exceptions, we will be concentrating on the discrete case.

Elementary propositions, such as $Cavity = true$ and $Toothache = false$, can be combined to form complex propositions using all the standard logical connectives. For example,

[2] One might expect the domain to be written as a set: $\{true, false\}$. We write it as a tuple because it will be convenient later to impose an ordering on the values.

$Cavity = true \wedge Toothache = false$ is a proposition to which one may ascribe a degree of (dis)belief. As explained in the previous paragraph, this proposition may also be written as $cavity \wedge \neg toothache$.

Atomic events

ATOMIC EVENT

The notion of an **atomic event** is useful in understanding the foundations of probability theory. An atomic event is a *complete* specification of the state of the world about which the agent is uncertain. It can be thought of as an assignment of particular values to all the variables of which the world is composed. For example, if my world consists of only the Boolean variables $Cavity$ and $Toothache$, then there are just four distinct atomic events; the proposition $Cavity = false \wedge Toothache = true$ is one such event.[3]

Atomic events have some important properties:

- They are *mutually exclusive*—at most one can actually be the case. For example, $cavity \wedge toothache$ and $cavity \wedge \neg toothache$ cannot both be the case.
- The set of all possible atomic events is *exhaustive*—at least one must be the case. That is, the disjunction of all atomic events is logically equivalent to $true$.
- Any particular atomic event entails the truth or falsehood of every proposition, whether simple or complex. This can be seen by using the standard semantics for logical connectives (Chapter 7). For example, the atomic event $cavity \wedge \neg toothache$ entails the truth of $cavity$ and the falsehood of $cavity \Rightarrow toothache$.
- Any proposition is logically equivalent to the disjunction of all atomic events that entail the truth of the proposition. For example, the proposition $cavity$ is equivalent to disjunction of the atomic events $cavity \wedge toothache$ and $cavity \wedge \neg toothache$.

Exercise 13.4 asks you to prove some of these properties.

Prior probability

UNCONDITIONAL
PRIOR PROBABILITY

The **unconditional** or **prior probability** associated with a proposition a is the degree of belief accorded to it *in the absence of any other information*; it is written as $P(a)$. For example, if the prior probability that I have a cavity is 0.1, then we would write

$$P(Cavity = true) = 0.1 \quad \text{or} \quad P(cavity) = 0.1\,.$$

It is important to remember that $P(a)$ can be used only when there is no other information. As soon as some new information is known, we must reason with the *conditional* probability of a given that new information. Conditional probabilities are covered in the next section.

Sometimes, we will want to talk about the probabilities of all the possible values of a random variable. In that case, we will use an expression such as $\mathbf{P}(Weather)$, which denotes a *vector* of values for the probabilities of each individual state of the weather. Thus, instead

[3] Many standard formulations of probability theory take atomic events, also known as **sample points**, as primitive and define a random variable as a function taking an atomic event as input and returning a value from the appropriate domain. Such an approach is perhaps more general, but also less intuitive.

of writing the four equations

$$P(Weather = sunny) = 0.7$$
$$P(Weather = rain) = 0.2$$
$$P(Weather = cloudy) = 0.08$$
$$P(Weather = snow) = 0.02\ .$$

we may simply write

$$\mathbf{P}(Weather) = \langle 0.7, 0.2, 0.08, 0.02 \rangle\ .$$

PROBABILITY DISTRIBUTION

This statement defines a prior **probability distribution** for the random variable *Weather*.

We will also use expressions such as $\mathbf{P}(Weather, Cavity)$ to denote the probabilities of all combinations of the values of a set of random variables.[4] In that case, $\mathbf{P}(Weather, Cavity)$ can be represented by a 4×2 table of probabilities. This is called the **joint probability distribution** of *Weather* and *Cavity*.

JOINT PROBABILITY DISTRIBUTION

Sometimes it will be useful to think about the complete set of random variables used to describe the world. A joint probability distribution that covers this complete set is called the **full joint probability distribution**. For example, if the world consists of just the variables *Cavity*, *Toothache*, and *Weather*, then the full joint distribution is given by

FULL JOINT PROBABILITY DISTRIBUTION

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

This joint distribution can be represented as a $2 \times 2 \times 4$ table with 16 entries. A full joint distribution specifies the probability of every atomic event and is therefore a complete specification of one's uncertainty about the world in question. We will see in Section 13.4 that any probabilistic query can be answered from the full joint distribution.

For continuous variables, it is not possible to write out the entire distribution as a table, because there are infinitely many values. Instead, one usually defines the probability that a random variable takes on some value x as a parameterized function of x. For example, let the random variable X denote tomorrow's maximum temperature in Berkeley. Then the sentence

$$P(X = x) = U[18, 26](x)$$

expresses the belief that X is distributed uniformly between 18 and 26 degrees Celsius. (Several useful continuous distributions are defined in Appendix A.) Probability distributions for continuous variables are called **probability density functions**. Density functions differ in meaning from discrete distributions. For example, using the temperature distribution given earlier, we find that $P(X = 20.5) = U[18, 26](20.5) = 0.125/C$. This does *not* mean that there's a 12.5% chance that the maximum temperature will be *exactly* 20.5 degrees tomorrow; the probability that this will happen is of course zero. The technical meaning is that the probability that the temperature is in a small region around 20.5 degrees is equal, in the limit, to 0.125 divided by the width of the region in degrees Celsius:

PROBABILITY DENSITY FUNCTIONS

$$\lim_{dx \to 0} P(20.5 \leq X \leq 20.5 + dx)/dx = 0.125/C\ .$$

[4] The general notational rule is that the distribution covers all values of the variables that are capitalized. Thus, the expression $\mathbf{P}(Weather, cavity)$ is a four-element vector of probabilities for the conjunction of each weather type with $Cavity = true$.

Some authors use different symbols for discrete distributions and density functions; we use P in both cases, since confusion seldom arises and the equations are usually identical. Note that probabilities are unitless numbers, whereas density functions are measured with a unit, in this case reciprocal degrees.

Conditional probability

Once the agent has obtained some evidence concerning the previously unknown random variables making up the domain, prior probabilities are no longer applicable. Instead, we use **conditional** or **posterior** probabilities. The notation used is $P(a|b)$, where a and b are any propositions.[5] This is read as "the probability of a, given that *all we know* is b." For example,

CONDITIONAL PROBABILITY
POSTERIOR PROBABILITY

$$P(cavity|toothache) = 0.8$$

indicates that if a patient is observed to have a toothache and no other information is yet available, then the probability of the patient's having a cavity will be 0.8. A prior probability, such as $P(cavity)$, can be thought of as a special case of the conditional probability $P(cavity|\ )$, where the probability is conditioned on no evidence.

Conditional probabilities can be defined in terms of unconditional probabilities. The defining equation is

$$P(a|b) = \frac{P(a \wedge b)}{P(b)} \tag{13.1}$$

which holds whenever $P(b) > 0$. This equation can also be written as

$$P(a \wedge b) = P(a|b)P(b)$$

PRODUCT RULE

which is called the **product rule**. The product rule is perhaps easier to remember: it comes from the fact that, for a and b to be true, we need b to be true, and we also need a to be true given b. We can also have it the other way around:

$$P(a \wedge b) = P(b|a)P(a)\ .$$

In some cases, it is easier to reason in terms of prior probabilities of conjunctions, but for the most part, we will use conditional probabilities as our vehicle for probabilistic inference.

We can also use the **P** notation for conditional distributions. $\mathbf{P}(X|Y)$ gives the values of $P(X = x_i|Y = y_j)$ for each possible i, j. As an example of how this makes our notation more concise, consider applying the product rule to each case where the propositions a and b assert particular values of X and Y respectively. We obtain the following equations:

$$\begin{aligned}
P(X = x_1 \wedge Y = y_1) &= P(X = x_1|Y = y_1)P(Y = y_1)\ . \\
P(X = x_1 \wedge Y = y_2) &= P(X = x_1|Y = y_2)P(Y = y_2)\ . \\
&\vdots
\end{aligned}$$

We can combine all these into the single equation

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

Remember that this denotes a set of equations relating the corresponding individual entries in the tables, *not* a matrix multiplication of the tables.

[5] The "|" operator has the lowest possible precedence, so $P(a \wedge b|c \vee d)$ means $P((a \wedge b)|(c \vee d))$.

It is tempting, but wrong, to view conditional probabilities as if they were logical implications with uncertainty added. For example, the sentence $P(a|b) = 0.8$ *cannot* be interpreted to mean "whenever b holds, conclude that $P(a)$ is 0.8." Such an interpretation would be wrong on two counts: first, $P(a)$ always denotes the prior probability of a, not the posterior probability given some evidence; second, the statement $P(a|b) = 0.8$ is immediately relevant just when b is the *only* available evidence. When additional information c is available, the degree of belief in a is $P(a|b \wedge c)$, which might have little relation to $P(a|b)$. For example, c might tell us directly whether a is true or false. If we examine a patient who complains of toothache, and discover a cavity, then we have additional evidence $cavity$, and we conclude (trivially) that $P(cavity|toothache \wedge cavity) = 1.0$.

13.3 THE AXIOMS OF PROBABILITY

So far, we have defined a syntax for propositions and for prior and conditional probability statements about those propositions. Now we must provide some sort of semantics for probability statements. We begin with the basic axioms that serve to define the probability scale and its endpoints:

1. All probabilities are between 0 and 1. For any proposition a,

$$0 \leq P(a) \leq 1 .$$

2. Necessarily true (i.e., valid) propositions have probability 1, and necessarily false (i.e., unsatisfiable) propositions have probability 0.

$$P(true) = 1 \qquad P(false) = 0 .$$

Next, we need an axiom that connects the probabilities of logically related propositions. The simplest way to do this is to define the probability of a disjunction as follows:

3. The probability of a disjunction is given by

$$P(a \vee b) = P(a) + P(b) - P(a \wedge b) .$$

This rule is easily remembered by noting that the cases where a holds, together with the cases where b holds, certainly cover all the cases where $a \vee b$ holds; but summing the two sets of cases counts their intersection twice, so we need to subtract $P(a \wedge b)$.

KOLMOGOROV'S AXIOMS

These three axioms are often called **Kolmogorov's axioms** in honor of the Russian mathematician Andrei Kolmogorov, who showed how to build up the rest of probability theory from this simple foundation. Notice that the axioms deal only with prior probabilities rather than conditional probabilities; this is because we have already defined the latter in terms of the former via Equation (13.1).

WHERE DO PROBABILITIES COME FROM?

There has been endless debate over the source and status of probability numbers. The **frequentist** position is that the numbers can come only from *experiments*: if we test 100 people and find that 10 of them have a cavity, then we can say that the probability of a cavity is approximately 0.1. In this view, the assertion "the probability of a cavity is 0.1" means that 0.1 is the fraction that would be observed in the limit of infinitely many samples. From any finite sample, we can estimate the true fraction and also calculate how accurate our estimate is likely to be.

The **objectivist** view is that probabilities are real aspects of the universe—propensities of objects to behave in certain ways—rather than being just descriptions of an observer's degree of belief. For example, that a fair coin comes up heads with probability 0.5 is a propensity of the coin itself. In this view, frequentist measurements are attempts to observe these propensities. Most physicists agree that quantum phenomena are objectively probabilistic, but uncertainty at the macroscopic scale—e.g., in coin tossing—usually arises from ignorance of initial conditions and does not seem consistent with the propensity view.

The **subjectivist** view describes probabilities as a way of characterizing an agent's beliefs, rather than as having any external physical significance. This allows the doctor or analyst to make the numbers up—to say, "In my opinion, I expect the probability of a cavity to be about 0.1." Several more reliable techniques, such as the betting systems described on page 474, have also been developed for eliciting probability assessments from humans.

In the end, even a strict frequentist position involves subjective analysis, so the difference probably has little practical importance. The **reference class** problem illustrates the intrusion of subjectivity. Suppose that a frequentist doctor wants to know the chances that a patient has a particular disease. The doctor wants to consider other patients who are similar in important ways—age, symptoms, perhaps sex—and see what proportion of them had the disease. But if the doctor considered everything that is known about the patient—weight to the nearest gram, hair color, mother's maiden name, etc.—the result would be that there are no other patients who are exactly the same and thus no reference class from which to collect experimental data. This has been a vexing problem in the philosophy of science.

Laplace's **principle of indifference** (1816) states that propositions that are syntactically "symmetric" with respect to the evidence should be accorded equal probability. Various refinements have been proposed, culminating in the attempt by Carnap and others to develop a rigorous **inductive logic**, capable of computing the correct probability for any proposition from any collection of observations. Currently, it is believed that no unique inductive logic exists; rather, any such logic rests on a subjective prior probability distribution whose effect is diminished as more observations are collected.

Using the axioms of probability

We can derive a variety of useful facts from the basic axioms. For example, the familiar rule for negation follows by substituting $\neg a$ for b in axiom 3, giving us:

$$\begin{aligned} P(a \vee \neg a) &= P(a) + P(\neg a) - P(a \wedge \neg a) && \text{(by axiom 3 with } b = \neg a) \\ P(\mathit{true}) &- P(a) + P(\neg a) - P(\mathit{false}) && \text{(by logical equivalence)} \\ 1 &= P(a) + P(\neg a) && \text{(by axiom 2)} \\ P(\neg a) &= 1 - P(a) && \text{(by algebra).} \end{aligned}$$

The third line of this derivation is itself a useful fact and can be extended from the Boolean case to the general discrete case. Let the discrete variable D have the domain $\langle d_1, \ldots, d_n \rangle$. Then it is easy to show (Exercise 13.2) that

$$\sum_{i=1}^{n} P(D = d_i) = 1 \ .$$

That is, any probability distribution on a single variable must sum to 1.[6] It is also true that any *joint* probability distribution on any *set* of variables must sum to 1: this can be seen simply by creating a single megavariable whose domain is the cross product of the domains of the the original variables.

Recall that any proposition a is equivalent to the disjunction of all the atomic events in which a holds; call this set of events $\mathbf{e}(a)$. Recall also that atomic events are mutually exclusive, so the probability of any conjunction of atomic events is zero, by axiom 2. Hence, from axiom 3, we can derive the following simple relationship: *The probability of a proposition is equal to the sum of the probabilities of the atomic events in which it holds;* that is,

$$P(a) = \sum_{e_i \in \mathbf{e}(a)} P(e_i) \ . \tag{13.2}$$

This equation provides a simple method for computing the probability of any proposition, given a full joint distribution that specifies the probabilities of all atomic events. (See Section 13.4.) In subsequent sections we will derive additional rules for manipulating probabilities. First, however, we will examine the foundation for the axioms themselves.

Why the axioms of probability are reasonable

The axioms of probability can be seen as restricting the set of probabilistic beliefs that an agent can hold. This is somewhat analogous to the logical case, where a logical agent cannot simultaneously believe A, B, and $\neg(A \wedge B)$, for example. There is, however, an additional complication. In the logical case, the semantic definition of conjunction means that at least one of the three beliefs just mentioned *must be false in the world*, so it is unreasonable for an agent to believe all three. With probabilities, on the other hand, statements refer not to the world directly, but to the agent's own state of knowledge. Why, then, can an agent not hold the following set of beliefs, which clearly violates axiom 3?

$$\begin{aligned} P(a) &= 0.4 & \qquad P(a \wedge b) &= 0.0 \\ P(b) &= 0.3 & \qquad P(a \vee b) &= 0.8 \end{aligned} \tag{13.3}$$

[6] For continuous variables, the summation is replaced by an integral: $\int_{-\infty}^{\infty} P(X = x)\, dx = 1$.

This kind of question has been the subject of decades of intense debate between those who advocate the use of probabilities as the only legitimate form for degrees of belief and those who advocate alternative approaches. Here, we give one argument for the axioms of probability, first stated in 1931 by Bruno de Finetti.

The key to de Finetti's argument is the connection between degree of belief and actions. The idea is that if an agent has some degree of belief in a proposition a, then the agent should be able to state odds at which it is indifferent to a bet for or against a. Think of it as a game between two agents: Agent 1 states "my degree of belief in event a is 0.4." Agent 2 is then free to choose whether to bet for or against a, at stakes that are consistent with the stated degree of belief. That is, Agent 2 could choose to bet that a will occur, betting \$4 against Agent 1's \$6. Or Agent 2 could bet \$6 against \$4 that A will not occur.[7] If an agent's degrees of belief do not accurately reflect the world, then you would expect that it would tend to lose money over the long run to an opposing agent whose beliefs more accurately reflect the state of the world.

But de Finetti proved something much stronger: *If Agent 1 expresses a set of degrees of belief that violate the axioms of probability theory then there is a combination of bets by Agent 2 that* **guarantees** *that Agent 1 will lose money* **every** *time.* So if you accept the idea that an agent should be willing to "put its money where its probabilities are," then you should accept that it is irrational to have beliefs that violate the axioms of probability.

One might think that this betting game is rather contrived. For example, what if one refuses to bet? Does that end the argument? The answer is that the betting game is an abstract model for the decision-making situation in which every agent is *unavoidably* involved at every moment. Every action (including inaction) is a kind of bet, and every outcome can be seen as a payoff of the bet. Refusing to bet is like refusing to allow time to pass.

We will not provide the proof of de Finetti's theorem, but we will show an example. Suppose that Agent 1 has the set of degrees of belief from Equation (13.3). Figure 13.2 shows that if Agent 2 chooses to bet \$4 on a, \$3 on b, and \$2 on $\neg(a \vee b)$, then Agent 1 always loses money, regardless of the outcomes for a and b.

Agent 1		Agent 2		Outcome for Agent 1			
Proposition	Belief	Bet	Stakes	$a \wedge b$	$a \wedge \neg b$	$\neg a \wedge b$	$\neg a \wedge \neg b$
a	0.4	a	4 to 6	–6	–6	4	4
b	0.3	b	3 to 7	–7	3	–7	3
$a \vee b$	0.8	$\neg(a \vee b)$	2 to 8	2	2	2	–8
				–11	–1	–1	–1

Figure 13.2 Because Agent 1 has inconsistent beliefs, Agent 2 is able to devise a set of bets that guarantees a loss for Agent 1, no matter what the outcome of a and b.

[7] One might argue that the agent's preferences for different bank balances are such that the possibility of losing \$1 is not counterbalanced by an equal possibility of winning \$1. One possible response is to make the bet amounts small enough to avoid this problem. Savage's analysis (1954) circumvents the issue altogether.

Other strong philosophical arguments have been put forward for the use of probabilities, most notably those of Cox (1946) and Carnap (1950). The world being the way it is, however, practical demonstrations sometimes speak louder than proofs. The success of reasoning systems based on probability theory has been much more effective in making converts. We now look at how the axioms can be deployed to make inferences.

13.4 INFERENCE USING FULL JOINT DISTRIBUTIONS

PROBABILISTIC INFERENCE

In this section we will describe a simple method for **probabilistic inference**—that is, the computation from observed evidence of posterior probabilities for query propositions. We will use the full joint distribution as the "knowledge base" from which answers to all questions may be derived. Along the way we will also introduce several useful techniques for manipulating equations involving probabilities.

We begin with a very simple example: a domain consisting of just the three Boolean variables *Toothache*, *Cavity*, and *Catch* (the dentist's nasty steel probe catches in my tooth). The full joint distribution is a $2 \times 2 \times 2$ table as shown in Figure 13.3.

	toothache		$\neg$*toothache*	
	catch	$\neg$*catch*	*catch*	$\neg$*catch*
cavity	0.108	0.012	0.072	0.008
$\neg$*cavity*	0.016	0.064	0.144	0.576

Figure 13.3 A full joint distribution for the *Toothache*, *Cavity*, *Catch* world.

Notice that the probabilities in the joint distribution sum to 1, as required by the axioms of probability. Notice also that Equation (13.2) gives us a direct way to calculate the probability of any proposition, simple or complex: We simply identify those atomic events in which the proposition is true and add up their probabilities. For example, there are six atomic events in which $cavity \vee toothache$ holds:

$$P(cavity \vee toothache) = 0.108 + 0.012 + 0.072 + 0.008 + 0.016 + 0.064 = 0.28 .$$

One particularly common task is to extract the distribution over some subset of variables or a single variable. For example, adding the entries in the first row gives the unconditional or

MARGINAL PROBABILITY

marginal probability[8] of *cavity*:

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

MARGINALIZATION

This process is called **marginalization**, or **summing out**—because the variables other than *Cavity* are summed out. We can write the following general marginalization rule for any sets of variables $\mathbf{Y}$ and $\mathbf{Z}$:

$$\mathbf{P}(\mathbf{Y}) = \sum_{\mathbf{z}} \mathbf{P}(\mathbf{Y}, \mathbf{z}) . \tag{13.4}$$

[8] So called because of a common practice among actuaries of writing the sums of observed frequencies in the margins of insurance tables.

That is, a distribution over $\mathbf{Y}$ can be obtained by summing out all the other variables from any joint distribution containing $\mathbf{Y}$. A variant of this rule involves conditional probabilities instead of joint probabilities, using the product rule:

$$\mathbf{P}(\mathbf{Y}) = \sum_{\mathbf{z}} \mathbf{P}(\mathbf{Y}|\mathbf{z})P(\mathbf{z}) \ . \tag{13.5}$$

CONDITIONING

This rule is called **conditioning**. Marginalization and conditioning will turn out to be useful rules for all kinds of derivations involving probability expressions.

In most cases, we will be interested in computing *conditional* probabilities of some variables, given evidence about others. Conditional probabilities can be found by first using Equation (13.1) to obtain an expression in terms of unconditional probabilities and then evaluating the expression from the full joint distribution. For example, we can compute the probability of a cavity, given evidence of a toothache, as follows:

$$\begin{aligned} P(cavity|toothache) &= \frac{P(cavity \wedge toothache)}{P(toothache)} \\ &= \frac{0.108 + 0.012}{0.108 + 0.012 + 0.016 + 0.064} = 0.6 \ . \end{aligned}$$

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

$$\begin{aligned} P(\neg cavity|toothache) &= \frac{P(\neg cavity \wedge toothache)}{P(toothache)} \\ &= \frac{0.016 + 0.064}{0.108 + 0.012 + 0.016 + 0.064} = 0.4 \ . \end{aligned}$$

Notice that in these two calculations the term $1/P(toothache)$ remains constant, no matter

NORMALIZATION

which value of *Cavity* we calculate. In fact, it can be viewed as a **normalization** constant for the distribution $\mathbf{P}(Cavity|toothache)$, ensuring that it adds up to 1. Throughout the chapters dealing with probability, we will use α to denote such constants. With this notation, we can write the two preceding equations in one:

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

Normalization will turn out to be a useful shortcut in many probability calculations.

From the example, we can extract a general inference procedure. We will stick to the case in which the query involves a single variable. We will need some notation: let X be the query variable (*Cavity* in the example), let $\mathbf{E}$ be the set of evidence variables (just *Toothache* in the example), let $\mathbf{e}$ be the observed values for them, and let $\mathbf{Y}$ be the remaining unobserved variables (just *Catch* in the example). The query is $\mathbf{P}(X|\mathbf{e})$ and can be evaluated as

$$\mathbf{P}(X|\mathbf{e}) = \alpha\mathbf{P}(X, \mathbf{e}) = \alpha \sum_{\mathbf{y}} \mathbf{P}(X, \mathbf{e}, \mathbf{y}) \ , \tag{13.6}$$

where the summation is over all possible $\mathbf{y}$s (i.e., all possible combinations of values of the unobserved variables $\mathbf{Y}$). Notice that together the variables X, $\mathbf{E}$, and $\mathbf{Y}$ constitute the complete set of variables for the domain, so $\mathbf{P}(X, \mathbf{e}, \mathbf{y})$ is simply a subset of probabilities from the full joint distribution. The algorithm is shown in Figure 13.4. It loops over the values

```
function ENUMERATE-JOINT-ASK(X, e, P) returns a distribution over X
  inputs: X, the query variable
          e, observed values for variables E
          P, a joint distribution on variables {X} ∪ E ∪ Y    /* Y = hidden variables */

  Q(X) ← a distribution over X, initially empty
  for each value x_i of X do
      Q(x_i) ← ENUMERATE-JOINT(x_i, e, Y, [ ], P)
  return NORMALIZE(Q(X))

function ENUMERATE-JOINT(x, e, vars, values, P) returns a real number
  if EMPTY?(vars) then return P(x, e, values)
  Y ← FIRST(vars)
  return Σ_y ENUMERATE-JOINT(x, e, REST(vars), [y|values], P)
```

Figure 13.4 An algorithm for probabilistic inference by enumeration of the entries in a full joint distribution.

of X and the values of Y to enumerate all possible atomic events with **e** fixed, adds up their probabilities from the joint table, and normalizes the results.

Given the full joint distribution to work with, ENUMERATE-JOINT-ASK is a complete algorithm for answering probabilistic queries for discrete variables. It does not scale well, however: For a domain described by n Boolean variables, it requires an input table of size $O(2^n)$ and takes $O(2^n)$ time to process the table. In a realistic problem, there might be hundreds or thousands of random variables to consider, not just three. It quickly becomes completely impractical to define the vast numbers of probabilities required—the experience needed in order to estimate each of the table entries separately simply cannot exist.

For these reasons, the full joint distribution in tabular form is not a practical tool for building reasoning systems (although the historical notes at the end of the chapter includes one real-world application of this method). Instead, it should be viewed as the theoretical foundation on which more effective approaches may be built. The remainder of this chapter introduces some of the basic ideas required in preparation for the development of realistic systems in Chapter 14.

13.5 INDEPENDENCE

Let us expand the full joint distribution in Figure 13.3 by adding a fourth variable, *Weather*. The full joint distribution then becomes $\mathbf{P}(Toothache, Catch, Cavity, Weather)$, which has 32 entries (because *Weather* has four values). It contains four "editions" of the table shown in Figure 13.3, one for each kind of weather. It seems natural to ask what relationship these editions have to each other and to the original three-variable table. For example, how are $P(toothache, catch, cavity, Weather = cloudy)$ and $P(toothache, catch, cavity)$ related?

One way to answer this question is to use the product rule:

$$P(toothache, catch, cavity, Weather = cloudy)$$
$$= P(Weather = cloudy | toothache, catch, cavity) P(toothache, catch, cavity) .$$

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

$$P(Weather = cloudy | toothache, catch, cavity) = P(Weather = cloudy) . \quad (13.7)$$

From this, we can deduce

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

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

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

Thus, the 32-element table for four variables can be constructed from one 8-element table and one four-element table. This decomposition is illustrated schematically in Figure 13.5(a).

INDEPENDENCE

The property we used in writing Equation (13.7) is called **independence** (also **marginal independence** and **absolute independence**). In particular, the weather is independent of one's dental problems. Independence between propositions a and b can be written as

$$P(a|b) = P(a) \quad \text{or} \quad P(b|a) = P(b) \quad \text{or} \quad P(a \wedge b) = P(a)P(b) . \quad (13.8)$$

All these forms are equivalent (Exercise 13.7). Independence between variables X and Y can be written as follows (again, these are all equivalent):

$$\mathbf{P}(X|Y) = \mathbf{P}(X) \quad \text{or} \quad \mathbf{P}(Y|X) = \mathbf{P}(Y) \quad \text{or} \quad \mathbf{P}(X, Y) = \mathbf{P}(X)\mathbf{P}(Y) .$$

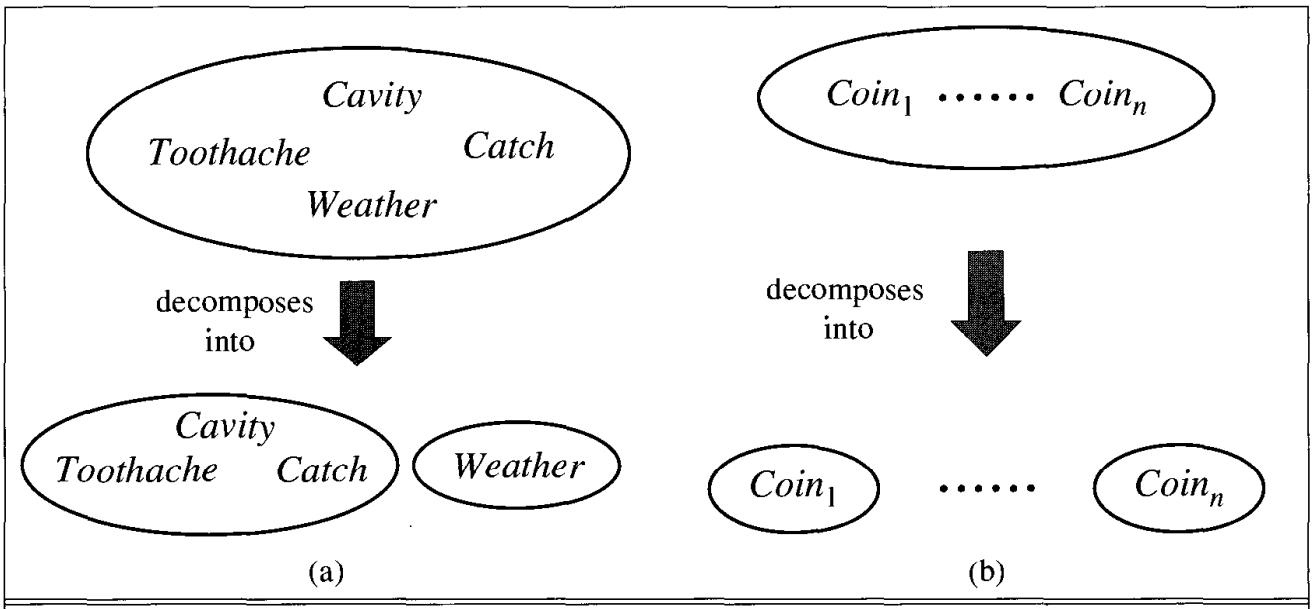


Figure 13.5 Two examples of factoring a large joint distribution into smaller distributions, using absolute independence. (a) Weather and dental problems are independent. (b) Coin flips are independent.

Independence assertions are usually based on knowledge of the domain. As we have seen, they can dramatically reduce the amount of information necessary to specify the full joint distribution. If the complete set of variables can be divided into independent subsets, then the full joint can be *factored* into separate joint distributions on those subsets. For example, the joint distribution on the outcome of n independent coin flips, $\mathbf{P}(C_1, \ldots, C_n)$, can be represented as the product of n single-variable distributions $\mathbf{P}(C_i)$. In a more practical vein, the independence of dentistry and meteorology is a good thing, because otherwise the practice of dentistry might require intimate knowledge of meteorology and *vice versa*.

When they are available, then, independence assertions can help in reducing the size of the domain representation and the complexity of the inference problem. Unfortunately, clean separation of entire sets of variables by independence is quite rare. Whenever a connection, however indirect, exists between two variables, independence will fail to hold. Moreover, even independent subsets can be quite large—for example, dentistry might involve dozens of diseases and hundreds of symptoms, all of which are interrelated. To handle such problems, we will need more subtle methods than the straightforward concept of independence.

13.6 Bayes' Rule and Its Use

On page 470, we defined the **product rule** and pointed out that it can be written in two forms because of the commutativity of conjunction:

$$P(a \wedge b) = P(a|b)P(b)$$
$$P(a \wedge b) = P(b|a)P(a) .$$

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

$$P(b|a) = \frac{P(a|b)P(b)}{P(a)} . \tag{13.9}$$

BAYES' RULE

This equation is known as **Bayes' rule** (also Bayes' law or Bayes' theorem).[9] This simple equation underlies all modern AI systems for probabilistic inference. The more general case of multivalued variables can be written in the **P** notation as

$$\mathbf{P}(Y|X) = \frac{\mathbf{P}(X|Y)\mathbf{P}(Y)}{\mathbf{P}(X)} .$$

where again this is to be taken as representing a set of equations, each dealing with specific values of the variables. We will also have occasion to use a more general version conditionalized on some background evidence **e**:

$$\mathbf{P}(Y|X, \mathbf{e}) = \frac{\mathbf{P}(X|Y, \mathbf{e})\mathbf{P}(Y|\mathbf{e})}{\mathbf{P}(X|\mathbf{e})} . \tag{13.10}$$

[9] According to rule 1 on page 1 of Strunk and White's *The Elements of Style*, it should be Bayes's rather than Bayes'. The latter is, however, more commonly used.

Applying Bayes' rule: The simple case

On the surface, Bayes' rule does not seem very useful. It requires three terms—a conditional probability and two unconditional probabilities—just to compute one conditional probability.

Bayes' rule is useful in practice because there are many cases where we do have good probability estimates for these three numbers and need to compute the fourth. In a task such as medical diagnosis, we often have conditional probabilities on causal relationships and want to derive a diagnosis. A doctor knows that the disease meningitis causes the patient to have a stiff neck, say, 50% of the time. The doctor also knows some unconditional facts: the prior probability that a patient has meningitis is 1/50,000, and the prior probability that any patient has a stiff neck is 1/20. Letting s be the proposition that the patient has a stiff neck and m be the proposition that the patient has meningitis, we have

$$\begin{aligned} P(s|m) &= 0.5 \\ P(m) &= 1/50000 \\ P(s) &= 1/20 \\ P(m|s) &= \frac{P(s|m)P(m)}{P(s)} = \frac{0.5 \times 1/50000}{1/20} = 0.0002 . \end{aligned}$$

That is, we expect only 1 in 5000 patients with a stiff neck to have meningitis. Notice that, even though a stiff neck is quite strongly indicated by meningitis (with probability 0.5), the probability of meningitis in the patient remains small. This is because the prior probability on stiff necks is much higher than that on meningitis.

Section 13.4 illustrated a process by which one can avoid assessing the probability of the evidence (here, $P(s)$) by instead computing a posterior probability for each value of the query variable (here, m and $\neg m$) and then normalizing the results. The same process can be applied when using Bayes' rule. We have

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

Thus, in order to use this approach we need to estimate $P(s|\neg m)$ instead of $P(s)$. There is no free lunch—sometimes this is easier, sometimes it is harder. The general form of Bayes' rule with normalization is

$$\mathbf{P}(Y|X) = \alpha \mathbf{P}(X|Y)\mathbf{P}(Y) , \tag{13.11}$$

where α is the normalization constant needed to make the entries in $\mathbf{P}(Y|X)$ sum to 1.

One obvious question to ask about Bayes' rule is why one might have available the conditional probability in one direction, but not the other. In the meningitis domain, perhaps the doctor knows that a stiff neck implies meningitis in 1 out of 5000 cases; that is, the doctor has quantitative information in the **diagnostic** direction from symptoms to causes. Such a doctor has no need to use Bayes' rule. Unfortunately, *diagnostic knowledge is often more fragile than causal knowledge.* If there is a sudden epidemic of meningitis, the unconditional probability of meningitis, $P(m)$, will go up. The doctor who derived the diagnostic probability $P(m|s)$ directly from statistical observation of patients before the epidemic will have no idea how to update the value, but the doctor who computes $P(m|s)$ from the other three values will see that $P(m|s)$ should go up proportionately with $P(m)$. Most importantly, the

causal information $P(s|m)$ is *unaffected* by the epidemic, because it simply reflects the way meningitis works. The use of this kind of direct causal or model-based knowledge provides the crucial robustness needed to make probabilistic systems feasible in the real world.

Using Bayes' rule: Combining evidence

We have seen that Bayes' rule can be useful for answering probabilistic queries conditioned on one piece of evidence—for example, the stiff neck. In particular, we have argued that probabilistic information is often available in the form $P(\mathit{effect}|\mathit{cause})$. What happens when we have two or more pieces of evidence? For example, what can a dentist conclude if her nasty steel probe catches in the aching tooth of a patient? If we know the full joint distribution (Figure 13.3), one can read off the answer:

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

We know, however, that such an approach will not scale up to larger numbers of variables.

We can try using Bayes' rule to reformulate the problem:

$$\mathbf{P}(\mathit{Cavity}|\mathit{toothache} \wedge \mathit{catch}) = \alpha \mathbf{P}(\mathit{toothache} \wedge \mathit{catch}|\mathit{Cavity})\mathbf{P}(\mathit{Cavity}) . \quad (13.12)$$

For this reformulation to work, we need to know the conditional probabilities of the conjunction $\mathit{toothache} \wedge \mathit{catch}$ for each value of Cavity. That might be feasible for just two evidence variables, but again it will not scale up. If there are n possible evidence variables (X rays, diet, oral hygiene, etc.), then there are 2^n possible combinations of observed values for which we would need to know conditional probabilities. We might as well go back to using the full joint distribution. This is what first led researchers away from probability theory toward approximate methods for evidence combination that, while giving incorrect answers, require fewer numbers to give any answer at all.

Rather than taking this route, we need to find some additional assertions about the domain that will enable us to simplify the expressions. The notion of **independence** in Section 13.5 provides a clue, but needs refining. It would be nice if $\mathit{Toothache}$ and Catch were independent, but they are not: if the probe catches in the tooth, it probably has a cavity and that probably causes a toothache. These variables *are* independent, however, *given the presence or the absence of a cavity*. Each is directly caused by the cavity, but neither has a direct effect on the other: toothache depends on the state of the nerves in the tooth, whereas the probe's accuracy depends on the dentist's skill, to which the toothache is irrelevant.[10] Mathematically, this property is written as

$$\mathbf{P}(\mathit{toothache} \wedge \mathit{catch}|\mathit{Cavity}) = \mathbf{P}(\mathit{toothache}|\mathit{Cavity})\mathbf{P}(\mathit{catch}|\mathit{Cavity}) . \quad (13.13)$$

CONDITIONAL INDEPENDENCE

This equation expresses the **conditional independence** of $\mathit{toothache}$ and catch given Cavity. We can plug it into Equation (13.12) to obtain the probability of a cavity:

$$\mathbf{P}(\mathit{Cavity}|\mathit{toothache} \wedge \mathit{catch}) = \alpha \mathbf{P}(\mathit{toothache}|\mathit{Cavity})\mathbf{P}(\mathit{catch}|\mathit{Cavity})\mathbf{P}(\mathit{Cavity}).$$

Now the information requirements are the same as for inference using each piece of evidence separately: the prior probability $\mathbf{P}(\mathit{Cavity})$ for the query variable and the conditional probability of each effect, given its cause.

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

The general definition of conditional independence of two variables X and Y, given a third variable Z is

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

In the dentist domain, for example, it seems reasonable to assert conditional independence of the variables *Toothache* and *Catch*, given *Cavity*:

$$\mathbf{P}(Toothache, Catch|Cavity) = \mathbf{P}(Toothache|Cavity)\mathbf{P}(Catch|Cavity) . \quad (13.14)$$

Notice that this assertion is somewhat stronger than Equation (13.13), which asserts independence only for specific values of *Toothache* and *Catch*. As with absolute independence in Equation (13.8), the equivalent forms

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

can also be used.

Section 13.5 showed that absolute independence assertions allow a decomposition of the full joint distribution into much smaller pieces. It turns out that the same is true for conditional independence assertions. For example, given the assertion in Equation (13.14), we can derive a decomposition as follows:

$$\begin{aligned}
&\mathbf{P}(Toothache, Catch, Cavity) \\
&= \mathbf{P}(Toothache, Catch|Cavity)\mathbf{P}(Cavity) \quad \text{(product rule)} \\
&= \mathbf{P}(Toothache|Cavity)\mathbf{P}(Catch|Cavity)\mathbf{P}(Cavity) \quad \text{[using (13.14)]}.
\end{aligned}$$

In this way, the original large table is decomposed into three smaller tables. The original table has seven independent numbers ($2^3 - 1$, because the numbers must sum to 1). The smaller tables contain five independent numbers ($2 \times (2^1 - 1)$ for each conditional probability distribution and $2^1 - 1$ for the prior on *Cavity*). This might not seem to be a major triumph, but the point is that, for n symptoms that are all conditionally independent given *Cavity*, the size of the representation grows as $O(n)$ instead of $O(2^n)$. Thus, *conditional independence assertions can allow probabilistic systems to scale up; moreover, they are much more commonly available than absolute independence assertions.* Conceptually, *Cavity* **separates** *Toothache* and *Catch* because it is a direct cause of both of them. The decomposition of large probabilistic domains into weakly connected subsets via conditional independence is one of the most important developments in the recent history of AI.

SEPARATION

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

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

NAIVE BAYES

Such a probability distribution is called a **naive Bayes** model—"naive" because it is often used (as a simplifying assumption) in cases where the "effect" variables are *not* conditionally independent given the cause variable. (The naive Bayes model is sometimes called a **Bayesian classifier**, a somewhat careless usage that has prompted true Bayesians to call it the **idiot Bayes** model.) In practice, naive Bayes systems can work surprisingly well, even when the independence assumption is not true. Chapter 20 describes methods for learning naive Bayes distributions from observations.

IDIOT BAYES

13.7 THE WUMPUS WORLD REVISITED

We can combine many of the ideas in this chapter to solve probabilistic reasoning problems in the wumpus world. (See Chapter 7 for a complete description of the wumpus world.) Uncertainty arises in the wumpus world because the agent's sensors give only partial, local information about the world. For example, Figure 13.6 shows a situation in which each of the three reachable squares—[1,3], [2,2], and [3,1]—might contain a pit. Pure logical inference can conclude nothing about which square is most likely to be safe, so a logical agent might be forced to choose randomly. We will see that a probabilistic agent can do much better than the logical agent.

Our aim will be to calculate the probability that each of the three squares contains a pit. (For the purposes of this example, we will ignore the wumpus and the gold.) The relevant properties of the wumpus world are that (1) a pit causes breezes in all neighboring squares, and (2) each square other than [1,1] contains a pit with probability 0.2. The first step is to identify the set of random variables we need:

- As in the propositional logic case, we want one Boolean variable P_{ij} for each square, which is true iff square $[i, j]$ actually contains a pit.
- We also have Boolean variables B_{ij} that are true iff square $[i, j]$ is breezy; we include these variables only for the observed squares—in this case, [1,1], [1,2], and [2,1].

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

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

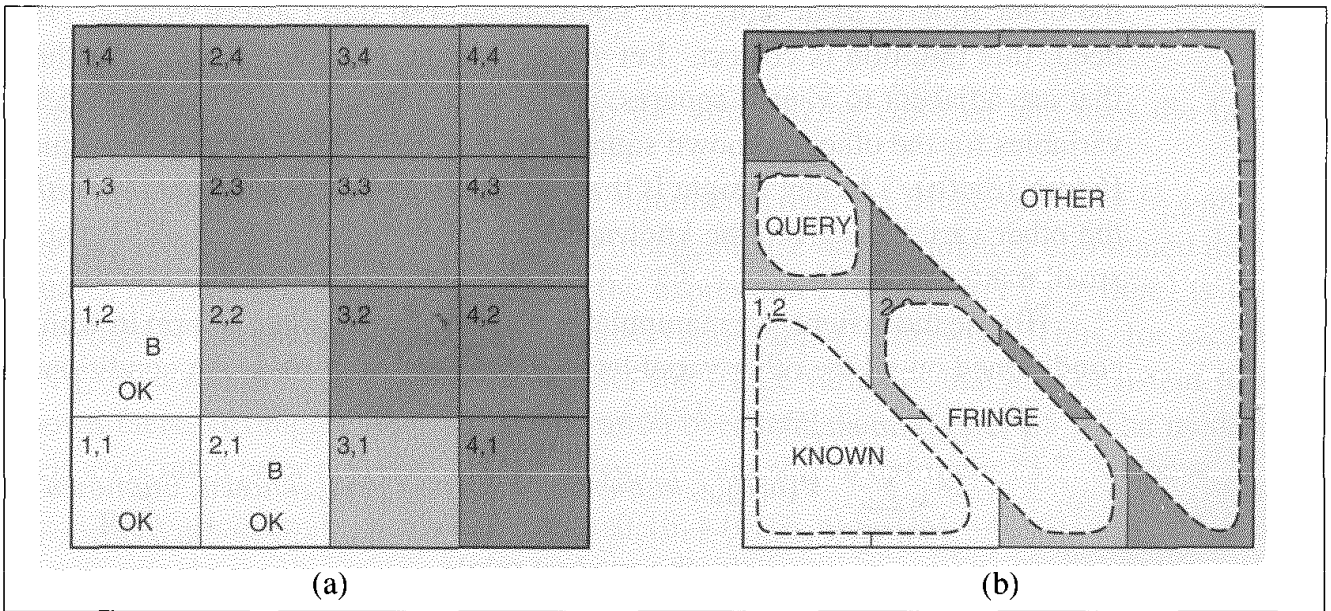


Figure 13.6 (a) After finding a breeze in both [1,2] and [2,1], the agent is stuck—there is no safe place to explore. (b) Division of the squares into *Known*, *Fringe*, and *Other*, for a query about [1,3].

This decomposition makes it very easy to see what the joint probability values should be. The first term is the conditional probability of a breeze configuration, given a pit configuration; this is 1 if the breezes are adjacent to the pits and 0 otherwise. The second term is the prior probability of a pit configuration. Each square contains a pit with probability 0.2, independently of the other squares; hence,

$$\mathbf{P}(P_{1,1},\ldots,P_{4,4}) = \prod_{i,j=1,1}^{4,4} \mathbf{P}(P_{i,j}) \; . \tag{13.15}$$

For a configuration with n pits, this is just $0.2^n \times 0.8^{16-n}$.

In the situation in Figure 13.6(a), the evidence consists of the observed breeze (or its absence) in each square that is visited, combined with the fact that each such square contains no pit. We'll abbreviate these facts as $b = \neg b_{1,1} \wedge b_{1,2} \wedge b_{2,1}$ and $known = \neg p_{1,1} \wedge \neg p_{1,2} \wedge \neg p_{2,1}$. We are interested in answering queries such as $\mathbf{P}(P_{1,3}|known, b)$: how likely is it that [1,3] contains a pit, given the observations so far?

To answer this query, we can follow the standard approach suggested by Equation (13.6) and implemented in the ENUMERATE-JOINT-ASK, namely, summing over entries from the full joint distribution. Let *Unknown* be a composite variable consisting of the $P_{i,j}$ variables for squares other than the *Known* squares and the query square [1,3]. Then, by Equation (13.6), we have

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

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

Intuition suggests that we are missing something here. Surely, one might ask, aren't the other squares irrelevant? The contents of [4,4] don't affect whether [1,3] has a pit! Indeed, this intuition is correct. Let *Fringe* be the variables (other than the query variable) that are adjacent to visited squares, in this case just [2,2] and [3,1]. Also, let *Other* be the variables for the other unknown squares; in this case, there are 10 other squares, as shown in Figure 13.6(b). The key insight is that the observed breezes are *conditionally independent* of the other variables, given the known, fringe, and query variables. The rest is, as they say, a small matter of algebra.

To use the insight, we manipulate the query formula into a form in which the breezes are conditioned on all the other variables, and then we simplify using conditional independence:

$$\begin{aligned}
&\mathbf{P}(P_{1,3}|known, b) \\
&= \alpha \sum_{unknown} \mathbf{P}(b|P_{1,3}, known, unknown)\mathbf{P}(P_{1,3}, known, unknown) \\
&\qquad\qquad \text{(by the product rule)} \\
&= \alpha \sum_{fringe} \sum_{other} \mathbf{P}(b|known, P_{1,3}, fringe, other)\mathbf{P}(P_{1,3}, known, fringe, other) \\
&= \alpha \sum_{fringe} \sum_{other} \mathbf{P}(b|known, P_{1,3}, fringe)\mathbf{P}(P_{1,3}, known, fringe, other) \; ,
\end{aligned}$$

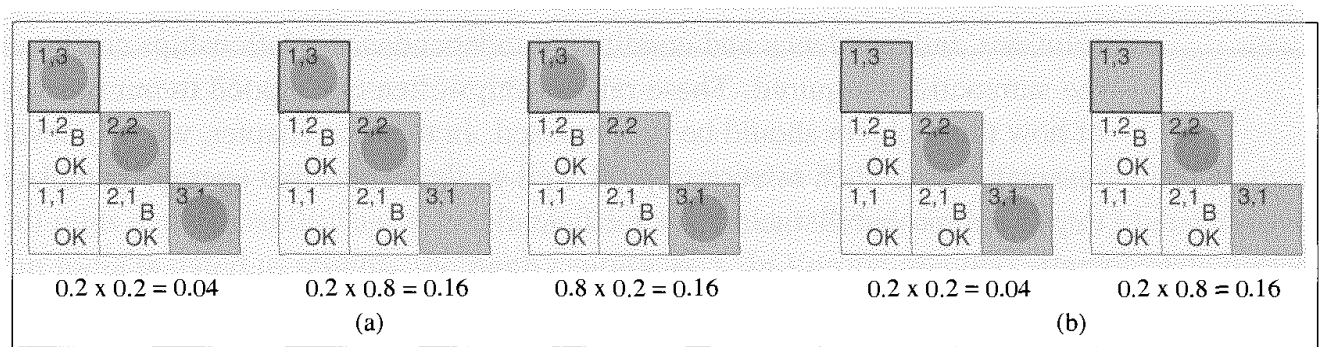


Figure 13.7 Consistent models for the fringe variables $P_{2,2}$ and $P_{3,1}$, showing $P(fringe)$ for each model: (a) three models with $P_{1,3} = true$ showing two or three pits, and (b) two models with $P_{1,3} = false$ showing one or two pits.

where the final step uses conditional independence. Now, the first term in this expression does not depend on the other variables, so we can move the summation inwards:

$$\begin{aligned}&\mathbf{P}(P_{1,3}|known, b)\\ &= \alpha \sum_{fringe} \mathbf{P}(b|known, P_{1,3}, fringe) \sum_{other} \mathbf{P}(P_{1,3}, known, fringe, other) \ .\end{aligned}$$

By independence, as in Equation (13.15), the prior term can be factored, and then the terms can be reordered:

$$\begin{aligned}&\mathbf{P}(P_{1,3}|known, b)\\ &= \alpha \sum_{fringe} \mathbf{P}(b|known, P_{1,3}, fringe) \sum_{other} \mathbf{P}(P_{1,3})P(known)P(fringe)P(other)\\ &= \alpha\, P(known)\mathbf{P}(P_{1,3}) \sum_{fringe} \mathbf{P}(b|known, P_{1,3}, fringe)P(fringe) \sum_{other} P(other)\\ &= \alpha' \mathbf{P}(P_{1,3}) \sum_{fringe} \mathbf{P}(b|known, P_{1,3}, fringe)P(fringe) \ ,\end{aligned}$$

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

Now, there are just four terms in the summation over the fringe variables $P_{2,2}$ and $P_{3,1}$. The use of independence and conditional independence has completely eliminated the other squares from consideration. Notice that the expression $\mathbf{P}(b|known, P_{1,3}, fringe)$ is 1 when the fringe is consistent with the breeze observations and 0 otherwise. Thus, for each value of $P_{1,3}$, we sum over the *logical models* for the fringe variables that are consistent with the known facts. (Compare with the enumeration over models in Figure 7.5.) The models and their associated prior probabilities—$P(fringe)$—are shown in Figure 13.7. We have

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

That is, [1,3] (and [3,1] by symmetry) contains a pit with roughly 31% probability. A similar calculation, which the reader might wish to perform, shows that [2,2] contains a pit with roughly 86% probability. The wumpus agent should definitely avoid [2,2]!

What this section has shown is that even seemingly complicated problems can be formulated precisely in probability theory and solved using simple algorithms. To get *efficient*

solutions, independence and conditional independence relationships can be used to simplify the summations required. These relationships often correspond to our natural understanding of how the problem should be decomposed. In the next chapter, we will develop formal representations for such relationships as well as algorithms that operate on those representations to perform probabilistic inference efficiently.

13.8 SUMMARY

This chapter has argued that probability is the right way to reason about uncertainty.

- Uncertainty arises because of both laziness and ignorance. It is inescapable in complex, dynamic, or inaccessible worlds.
- Uncertainty means that many of the simplifications that are possible with deductive inference are no longer valid.
- Probabilities express the agent's inability to reach a definite decision regarding the truth of a sentence. Probabilities summarize the agent's beliefs.
- Basic probability statements include **prior probabilities** and **conditional probabilities** over simple and complex propositions.
- The **full joint probability distribution** specifies the probability of each complete assignment of values to random variables. It is usually too large to create or use in its explicit form.
- The axioms of probability constrain the possible assignments of probabilities to propositions. An agent that violates the axioms will behave irrationally in some circumstances.
- When the full joint distribution is available, it can be used to answer queries simply by adding up entries for the atomic events corresponding to the query propositions.
- **Absolute independence** between subsets of random variables might allow the full joint distribution to be factored into smaller joint distributions. This could greatly reduce complexity, but seldom occurs in practice.
- **Bayes' rule** allows unknown probabilities to be computed from known conditional probabilities, usually in the causal direction. Applying Bayes' rule with many pieces of evidence will in general run into the same scaling problems as does the full joint distribution.
- **Conditional independence** brought about by direct causal relationships in the domain might allow the full joint distribution to be factored into smaller, conditional distributions. The **naive Bayes** model assumes the conditional independence of all effect variables, given a single cause variable, and grows linearly with the number of effects.
- A wumpus-world agent can calculate probabilities for unobserved aspects of the world and use them to make better decisions than a purely logical agent makes.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Although games of chance date back at least to around 300 B.C., the mathematical analysis of odds and probability appears to be much more recent. Some work done by Mahaviracarya in India is dated to roughly the ninth century A.D. In Europe, the first attempts date only to the Italian Renaissance, beginning around 1500 A.D. The first significant systematic analyses were produced by Girolamo Cardano around 1565, but they remained unpublished until 1663. By that time, the discovery by Blaise Pascal (in correspondence with Pierre Fermat in 1654) of a systematic way of calculating probabilities had for the first time established probability as a mathematical discipline. The first published textbook on probability was *De Ratiociniis in Ludo Aleae* (Huygens, 1657). Pascal also introduced conditional probability, which is covered in Huygens's textbook. The Rev. Thomas Bayes (1702–1761) introduced the rule for reasoning about conditional probabilities that was named after him. It was published posthumously (Bayes, 1763). Kolmogorov (1950, first published in German in 1933) presented probability theory in a rigorously axiomatic framework for the first time. Rényi (1970) later gave an axiomatic presentation that took conditional probability, rather than absolute probability, as primitive.

Pascal used probability in ways that required both the objective interpretation, as a property of the world based on symmetry or relative frequency, and the subjective interpretation, based on degree of belief—the former in his analyses of probabilities in games of chance, the latter in the famous "Pascal's wager" argument about the possible existence of God. However, Pascal did not clearly realize the distinction between these two interpretations. The distinction was first drawn clearly by James Bernoulli (1654–1705).

Leibniz introduced the "classical" notion of probability as a proportion of enumerated, equally probable cases, which was also used by Bernoulli, although it was brought to prominence by Laplace (1749–1827). This notion is ambiguous between the frequency interpretation and the subjective interpretation. The cases can be thought to be equally probable either because of a natural, physical symmetry between them, or simply because we do not have any knowledge that would lead us to consider one more probable than another. The use of this latter, subjective consideration to justify assigning equal probabilities is known as the *principle of indifference* (Keynes, 1921).

The debate between objectivists and subjectivists became sharper in the 20th century. Kolmogorov (1963), R. A. Fisher (1922), and Richard von Mises (1928) were advocates of the relative frequency interpretation. Karl Popper's (1959, first published in German in 1934) "propensity" interpretation traces relative frequencies to an underlying physical symmetry. Frank Ramsey (1931), Bruno de Finetti (1937), R. T. Cox (1946), Leonard Savage (1954), and Richard Jeffrey (1983) interpreted probabilities as the degrees of belief of specific individuals. Their analyses of degree of belief were closely tied to utilities and to behavior—specifically, to the willingness to place bets. Rudolf Carnap, following Leibniz and Laplace, offered a different kind of subjective interpretation of probability—not as any actual individual's degree of belief, but as the degree of belief that an idealized individual *should* have in a particular proposition a, given a particular body of evidence $\mathbf{e}$. Carnap attempted to go further

CONFIRMATION

INDUCTIVE LOGIC

than Leibniz or Laplace by making this notion of degree of **confirmation** mathematically precise, as a logical relation between a and $\mathbf{e}$. The study of this relation was intended to constitute a mathematical discipline called **inductive logic**, analogous to ordinary deductive logic (Carnap, 1948, 1950). Carnap was not able to extend his inductive logic much beyond the propositional case, and Putnam (1963) showed that some fundamental difficulties would prevent a strict extension to languages capable of expressing arithmetic.

The question of reference classes is closely tied to the attempt to find an inductive logic. The approach of choosing the "most specific" reference class of sufficient size was formally proposed by Reichenbach (1949). Various attempts have been made, notably by Henry Kyburg (1977, 1983), to formulate more sophisticated policies in order to avoid some obvious fallacies that arise with Reichenbach's rule, but such approaches remain somewhat *ad hoc*. More recent work by Bacchus, Grove, Halpern, and Koller (1992) extends Carnap's methods to first-order theories, thereby avoiding many of the difficulties associated with the straightfoward reference-class method.

Bayesian probabilistic reasoning has been used in AI since the 1960s, especially in medical diagnosis. It was used not only to make a diagnosis from available evidence, but also to select further questions and tests using the theory of information value (Section 16.6) when available evidence was inconclusive (Gorry, 1968; Gorry *et al.*, 1973). One system outperformed human experts in the diagnosis of acute abdominal illnesses (de Dombal *et al.*, 1974). These early Bayesian systems suffered from a number of problems, however. Because they lacked any theoretical model of the conditions they were diagnosing, they were vulnerable to unrepresentative data occurring in situations for which only a small sample was available (de Dombal *et al.*, 1981). Even more fundamentally, because they lacked a concise formalism (such as the one to be described in Chapter 14) for representing and using conditional independence information, they depended on the acquisition, storage, and processing of enormous tables of probabilistic data. Because of these difficulties, probabilistic methods for coping with uncertainty fell out of favor in AI from the 1970s to the mid-1980s. Developments since the late 1980s are described in the next chapter.

The naive Bayes representation for joint distributions has been studied extensively in the pattern recognition literature since the 1950s (Duda and Hart, 1973). It has also been used, often unwittingly, in text retrieval, beginning with the work of Maron (1961). The probabilistic foundations of this technique, described further in Exercise 13.18, were elucidated by Robertson and Sparck Jones (1976). Domingos and Pazzani (1997) provide an explanation for the surprising success of naive Bayesian reasoning even in domains where the independence assumptions are clearly violated.

There are many good introductory textbooks on probability theory, including those by Chung (1979) and Ross (1988). Morris DeGroot (1989) offers a combined introduction to probability and statistics from a Bayesian standpoint, as well as a more advanced text (1970). Richard Hamming's (1991) textbook gives a mathematically sophisticated introduction to probability theory from the standpoint of a propensity interpretation based on physical symmetry. Hacking (1975) and Hald (1990) cover the early history of the concept of probability. Bernstein (1996) gives an entertaining popular account of the story of risk.

EXERCISES

13.1 Show from first principles that $P(a|b \wedge a) = 1$.

13.2 Using the axioms of probability, prove that any probability distribution on a discrete random variable must sum to 1.

13.3 Would it be rational for an agent to hold the three beliefs $P(A) = 0.4$, $P(B) = 0.3$, and $P(A \vee B) = 0.5$? If so, what range of probabilities would be rational for the agent to hold for $A \wedge B$? Make up a table like the one in Figure 13.2, and show how it supports your argument about rationality. Then draw another version of the table where $P(A \vee B) = 0.7$. Explain why it is rational to have this probability, even though the table shows one case that is a loss and three that just break even. (*Hint:* what is Agent 1 committed to about the probability of each of the four cases, especially the case that is a loss?)

13.4 This question deals with the properties of atomic events, as discussed on page 468.

a. Prove that the disjunction of all possible atomic events is logically equivalent to *true*. [*Hint:* Use a proof by induction on the number of random variables.]

b. Prove that any proposition is logically equivalent to the disjunction of the atomic events that entail its truth.

13.5 Consider the domain of dealing 5-card poker hands from a standard deck of 52 cards, under the assumption that the dealer is fair.

a. How many atomic events are there in the joint probability distribution (i.e., how many 5-card hands are there)?

b. What is the probability of each atomic event?

c. What is the probability of being dealt a royal straight flush? Four of a kind?

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

a. $P(toothache)$

b. $\mathbf{P}(Cavity)$

c. $\mathbf{P}(Toothache|cavity)$

d. $\mathbf{P}(Cavity|toothache \vee catch)$.

13.7 Show that the three forms of independence in Equation (13.8) are equivalent.

13.8 After your yearly checkup, the doctor has bad news and good news. The bad news is that you tested positive for a serious disease and that the test is 99% accurate (i.e., the probability of testing positive when you do have the disease is 0.99, as is the probability of testing negative when you don't have the disease). The good news is that this is a rare disease, striking only 1 in 10,000 people of your age. Why is it good news that the disease is rare? What are the chances that you actually have the disease?

13.9 It is quite often useful to consider the effect of some specific propositions in the context of some general background evidence that remains fixed, rather than in the complete absence of information. The following questions ask you to prove more general versions of the product rule and Bayes' rule, with respect to some background evidence $\mathbf{e}$:

a. Prove the conditionalized version of the general product rule:

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

b. Prove the conditionalized version of Bayes' rule in Equation (13.10).

13.10 Show that the statement

$$\mathbf{P}(A, B|C) = \mathbf{P}(A|C)\mathbf{P}(B|C)$$

is equivalent to either of the statements

$$\mathbf{P}(A|B, C) = \mathbf{P}(A|C) \quad \text{and} \quad \mathbf{P}(B|A, C) = \mathbf{P}(B|C) .$$

13.11 Suppose you are given a bag containing n unbiased coins. You are told that $n - 1$ of these coins are normal, with heads on one side and tails on the other, whereas one coin is a fake, with heads on both sides.

a. Suppose you reach into the bag, pick out a coin uniformly at random, flip it, and get a head. What is the (conditional) probability that the coin you chose is the fake coin?

b. Suppose you continue flipping the coin for a total of k times after picking it and see k heads. Now what is the conditional probability that you picked the fake coin?

c. Suppose you wanted to decide whether the chosen coin was fake by flipping it k times. The decision procedure returns FAKE if all k flips come up heads, otherwise it returns NORMAL. What is the (unconditional) probability that this procedure makes an error?

13.12 In this exercise, you will complete the normalization calculation for the meningitis example. First, make up a suitable value for $P(S|\neg M)$, and use it to calculate unnormalized values for $P(M|S)$ and $P(\neg M|S)$ (i.e., ignoring the $P(S)$ term in the Bayes' rule expression). Now normalize these values so that they add to 1.

13.13 This exercise investigates the way in which conditional independence relationships affect the amount of information needed for probabilistic calculations.

a. Suppose we wish to calculate $P(h|e_1, e_2)$ and we have no conditional independence information. Which of the following sets of numbers are sufficient for the calculation?

(i) $\mathbf{P}(E_1, E_2), \mathbf{P}(H), \mathbf{P}(E_1|H), \mathbf{P}(E_2|H)$
(ii) $\mathbf{P}(E_1, E_2), \mathbf{P}(H), \mathbf{P}(E_1, E_2|H)$
(iii) $\mathbf{P}(H), \mathbf{P}(E_1|H), \mathbf{P}(E_2|H)$

b. Suppose we know that $\mathbf{P}(E_1|H, E_2) = \mathbf{P}(E_1|H)$ for all values of H, E_1, E_2. Now which of the three sets are sufficient?

13.14 Let X, Y, Z be Boolean random variables. Label the eight entries in the joint distribution $\mathbf{P}(X, Y, Z)$ as a through h. Express the statement that X and Y are conditionally

independent given Z as a set of equations relating a through h. How many *nonredundant* equations are there?

13.15 (Adapted from Pearl (1988).) Suppose you are a witness to a nighttime hit-and-run accident involving a taxi in Athens. All taxis in Athens are blue or green. You swear, under oath, that the taxi was blue. Extensive testing shows that, under the dim lighting conditions, discrimination between blue and green is 75% reliable. Is it possible to calculate the most likely color for the taxi? (*Hint:* distinguish carefully between the proposition that the taxi *is* blue and the proposition that it *appears* blue.)

What about now, given that 9 out of 10 Athenian taxis are green?

13.16 (Adapted from Pearl (1988).) Three prisoners, A, B, and C, are locked in their cells. It is common knowledge that one of them will be executed the next day and the others pardoned. Only the governor knows which one will be executed. Prisoner A asks the guard a favor: "Please ask the governor who will be executed, and then take a message to one of my friends B or C to let him know that he will be pardoned in the morning." The guard agrees, and comes back later and tells A that he gave the pardon message to B.

What are A's chances of being executed, given this information? (Answer this *mathematically*, not by energetic waving of hands.)

13.17 Write out a general algorithm for answering queries of the form $\mathbf{P}(Cause|\mathbf{e})$, using a naive Bayes distribution. You should assume that the evidence $\mathbf{e}$ may assign values to *any subset* of the effect variables.

13.18 Text categorization is the task of assigning a given document to one of a fixed set of categories, on the basis of the text it contains. Naive Bayes models are often used for this task. In these models, the query variable is the document category, and the "effect" variables are the presence or absence of each word in the language; the assumption is that words occur independently in documents, with frequencies determined by the document category.

a. Explain precisely how such a model can be constructed, given as "training data" a set of documents that have been assigned to categories.

b. Explain precisely how to categorize a new document.

c. Is the independence assumption reasonable? Discuss.

13.19 In our analysis of the wumpus world, we used the fact that each square contains a pit with probability 0.2, independently of the contents of the other squares. Suppose instead that exactly $N/5$ pits are scattered uniformly at random among the N squares other than [1,1]. Are the variables $P_{i,j}$ and $P_{k,l}$ still independent? What is the joint distribution $\mathbf{P}(P_{1,1}, \ldots, P_{4,4})$ now? Redo the calculation for the probabilities of pits in [1,3] and [2,2].

14 PROBABILISTIC REASONING

In which we explain how to build network models to reason under uncertainty according to the laws of probability theory.

Chapter 13 gave the syntax and semantics of probability theory. We remarked on the importance of independence and conditional independence relationships in simplifying probabilistic representations of the world. This chapter introduces a systematic way to represent such relationships explicitly in the form of **Bayesian networks**. We define the syntax and semantics of these networks and show how they can be used to capture uncertain knowledge in a natural and efficient way. We then show how probabilistic inference, although computationally intractable in the worst case, can be done efficiently in many practical situations. We also describe a variety of approximate inference algorithms that are often applicable when exact inference is infeasible. We explore ways in which probability theory can be applied to worlds with objects and relations—that is, to *first-order*, as opposed to *propositional*, representations. Finally, we survey alternative approaches to uncertain reasoning.

14.1 REPRESENTING KNOWLEDGE IN AN UNCERTAIN DOMAIN

In Chapter 13, we saw that the full joint probability distribution can answer any question about the domain, but can become intractably large as the number of variables grows. Furthermore, specifying probabilities for atomic events is rather unnatural and can be very difficult unless a large amount of data is available from which to gather statistical estimates.

We also saw that independence and conditional independence relationships among variables can greatly reduce the number of probabilities that need to be specified in order to define the full joint distribution. This section introduces a data structure called a **Bayesian network**[1] to represent the dependencies among variables and to give a concise specification of *any* full joint probability distribution.

BAYESIAN NETWORK

[1] This is the most common name, but there are many others, including **belief network**, **probabilistic network**, **causal network**, and **knowledge map**. In statistics, the term **graphical model** refers to a somewhat broader class that includes Bayesian networks. An extension of Bayesian networks called a **decision network** or **influence diagram** will be covered in Chapter 16.

A Bayesian network is a directed graph in which each *node* is annotated with quantitative probability information. The full specification is as follows:

1. A set of random variables makes up the nodes of the network. Variables may be discrete or continuous.
2. A set of directed links or arrows connects pairs of nodes. If there is an arrow from node X to node Y, X is said to be a *parent* of Y.
3. Each node X_i has a conditional probability distribution $P(X_i|Parents(X_i))$ that quantifies the effect of the parents on the node.
4. The graph has no directed cycles (and hence is a directed, acyclic graph, or DAG).

The topology of the network—the set of nodes and links—specifies the conditional independence relationships that hold in the domain, in a way that will be made precise shortly. The *intuitive* meaning of an arrow in a properly constructed network is usually that X has a *direct influence* on Y. It is usually easy for a domain expert to decide what direct influences exist in the domain—much easier, in fact, than actually specifying the probabilities themselves. Once the topology of the Bayesian network is laid out, we need only specify a conditional probability distribution for each variable, given its parents. We will see that the combination of the topology and the conditional distributions suffices to specify (implicitly) the full joint distribution for all the variables.

Recall the simple world described in Chapter 13, consisting of the variables *Toothache*, *Cavity*, *Catch*, and *Weather*. We argued that *Weather* is independent of the other variables; furthermore, we argued that *Toothache* and *Catch* are conditionally independent, given *Cavity*. These relationships are represented by the Bayesian network structure shown in Figure 14.1. Formally, the conditional independence of *Toothache* and *Catch* given *Cavity* is indicated by the *absence* of a link between *Toothache* and *Catch*. Intuitively, the network represents the fact that *Cavity* is a direct cause of *Toothache* and *Catch*, whereas no direct causal relationship exists between *Toothache* and *Catch*.

Now consider the following example, which is just a little more complex. You have a new burglar alarm installed at home. It is fairly reliable at detecting a burglary, but also responds on occasion to minor earthquakes. (This example is due to Judea Pearl, a resident of Los Angeles—hence the acute interest in earthquakes.) You also have two neighbors, John and Mary, who have promised to call you at work when they hear the alarm. John always calls

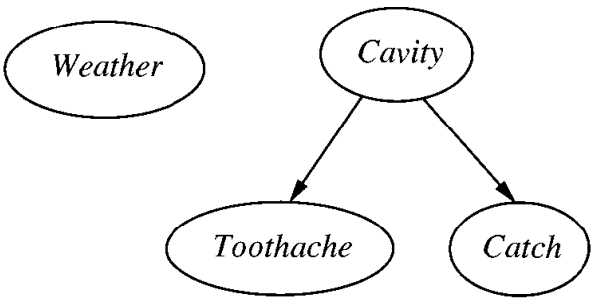


Figure 14.1 A simple Bayesian network in which *Weather* is independent of the other three variables and *Toothache* and *Catch* are conditionally independent, given *Cavity*.

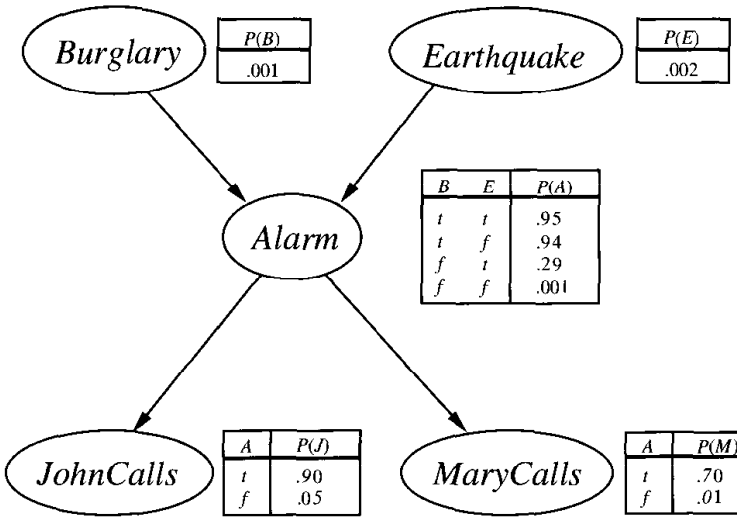


Figure 14.2 A typical Bayesian network, showing both the topology and the conditional probability tables (CPTs). In the CPTs, the letters B, E, A, J, and M stand for *Burglary*, *Earthquake*, *Alarm*, *JohnCalls*, and *MaryCalls*, respectively.

when he hears the alarm, but sometimes confuses the telephone ringing with the alarm and calls then, too. Mary, on the other hand, likes rather loud music and sometimes misses the alarm altogether. Given the evidence of who has or has not called, we would like to estimate the probability of a burglary. The Bayesian network for this domain appears in Figure 14.2.

For the moment, let us ignore the conditional distributions in the figure and concentrate on the topology of the network. In the case of the burglary network, the topology shows that burglary and earthquakes directly affect the probability of the alarm's going off, but whether John and Mary call depends only on the alarm. The network thus represents our assumptions that they do not perceive any burglaries directly, they do not notice the minor earthquakes, and they do not confer before calling.

Notice that the network does not have nodes corresponding to Mary's currently listening to loud music or to the telephone ringing and confusing John. These factors are summarized in the uncertainty associated with the links from *Alarm* to *JohnCalls* and *MaryCalls*. This shows both laziness and ignorance in operation: it would be a lot of work to find out why those factors would be more or less likely in any particular case, and we have no reasonable way to obtain the relevant information anyway. The probabilities actually summarize a *potentially infinite* set of circumstances in which the alarm might fail to go off (high humidity, power failure, dead battery, cut wires, a dead mouse stuck inside the bell, etc.) or John or Mary might fail to call and report it (out to lunch, on vacation, temporarily deaf, passing helicopter, etc.). In this way, a small agent can cope with a very large world, at least approximately. The degree of approximation can be improved if we introduce additional relevant information.

CONDITIONAL PROBABILITY TABLE

Now let us turn to the conditional distributions shown in Figure 14.2. In the figure, each distribution is shown as a **conditional probability table**, or CPT. (This form of table can be used for discrete variables; other representations, including those suitable for continuous variables, are described in Section 14.2.) Each row in a CPT contains the conditional probability of each node value for a **conditioning case**. A conditioning case is just a possible

CONDITIONING CASE

combination of values for the parent nodes—a miniature atomic event, if you like. Each row must sum to 1, because the entries represent an exhaustive set of cases for the variable. For Boolean variables, once you know that the probability of a true value is p, the probability of false must be $1 - p$, so we often omit the second number, as in Figure 14.2. In general, a table for a Boolean variable with k Boolean parents contains 2^k independently specifiable probabilities. A node with no parents has only one row, representing the prior probabilities of each possible value of the variable.

14.2 THE SEMANTICS OF BAYESIAN NETWORKS

The previous section described what a network is, but not what it means. There are two ways in which one can understand the semantics of Bayesian networks. The first is to see the network as a representation of the joint probability distribution. The second is to view it as an encoding of a collection of conditional independence statements. The two views are equivalent, but the first turns out to be helpful in understanding how to *construct* networks, whereas the second is helpful in designing inference procedures.

Representing the full joint distribution

A Bayesian network provides a complete description of the domain. Every entry in the full joint probability distribution (hereafter abbreviated as "joint") can be calculated from the information in the network. A generic entry in the joint distribution is the probability of a conjunction of particular assignments to each variable, such as $P(X_1 = x_1 \wedge \ldots \wedge X_n = x_n)$. We use the notation $P(x_1, \ldots, x_n)$ as an abbreviation for this. The value of this entry is given by the formula

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

where $parents(X_i)$ denotes the specific values of the variables in $Parents(X_i)$. Thus, each entry in the joint distribution is represented by the product of the appropriate elements of the conditional probability tables (CPTs) in the Bayesian network. The CPTs therefore provide a decomposed representation of the joint distribution.

To illustrate this, we can calculate the probability that the alarm has sounded, but neither a burglary nor an earthquake has occurred, and both John and Mary call. We use single-letter names for the variables:

$$\begin{aligned} &P(j \wedge m \wedge a \wedge \neg b \wedge \neg e) \\ &\quad = P(j|a)P(m|a)P(a|\neg b \wedge \neg e)P(\neg b)P(\neg e) \\ &\quad = 0.90 \times 0.70 \times 0.001 \times 0.999 \times 0.998 = 0.00062 \,. \end{aligned}$$

Section 13.4 explained that the full joint distribution can be used to answer any query about the domain. If a Bayesian network is a representation of the joint distribution, then it too can be used to answer any query, by summing all the relevant joint entries. Section 14.4 explains how to do this, but also describes methods that are much more efficient.

A method for constructing Bayesian networks

Equation (14.1) defines what a given Bayesian network means. It does not, however, explain how to *construct* a Bayesian network in such a way that the resulting joint distribution is a good representation of a given domain. We will now show that Equation (14.1) implies certain conditional independence relationships that can be used to guide the knowledge engineer in constructing the topology of the network. First, we rewrite the joint distribution in terms of a conditional probability, using the product rule (see Chapter 13):

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

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

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

CHAIN RULE

This identity holds true for any set of random variables and is called the **chain rule**. Comparing it with Equation (14.1), we see that the specification of the joint distribution is equivalent to the general assertion that, for every variable X_i in the network,

$$\mathbf{P}(X_i|X_{i-1}, \ldots, X_1) = \mathbf{P}(X_i|Parents(X_i)) , \tag{14.2}$$

provided that $Parents(X_i) \subseteq \{X_{i-1}, \ldots, X_1\}$. This last condition is satisfied by labeling the nodes in any order that is consistent with the partial order implicit in the graph structure.

What Equation (14.2) says is that the Bayesian network is a correct representation of the domain only if each node is conditionally independent of its predecessors in the node ordering, given its parents. Hence, in order to construct a Bayesian network with the correct structure for the domain, we need to choose parents for each node such that this property holds. Intuitively, the parents of node X_i should contain all those nodes in $X_1, \ldots, X_{i-1}$ that *directly influence* X_i. For example, suppose we have completed the network in Figure 14.2 except for the choice of parents for *MaryCalls*. *MaryCalls* is certainly influenced by whether there is a *Burglary* or an *Earthquake*, but not *directly* influenced. Intuitively, our knowledge of the domain tells us that these events influence Mary's calling behavior only through their effect on the alarm. Also, given the state of the alarm, whether John calls has no influence on Mary's calling. Formally speaking, we believe that the following conditional independence statement holds:

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

Compactness and node ordering

As well as being a complete and nonredundant representation of the domain, a Bayesian network can often be far more *compact* than the full joint distribution. This property is what makes it feasible to handle domains with many variables. The compactness of Bayesian networks is an example of a very general property of **locally structured** (also called **sparse**) systems. In a locally structured system, each subcomponent interacts directly with only a bounded number of other components, regardless of the total number of components. Local

LOCALLY STRUCTURED
SPARSE

structure is usually associated with linear rather than exponential growth in complexity. In the case of Bayesian networks, it is reasonable to suppose that in most domains each random variable is directly influenced by at most k others, for some constant k. If we assume n Boolean variables for simplicity, then the amount of information needed to specify each conditional probability table will be at most 2^k numbers, and the complete network can be specified by $n2^k$ numbers. In contrast, the joint distribution contains 2^n numbers. To make this concrete, suppose we have $n = 30$ nodes, each with five parents ($k = 5$). Then the Bayesian network requires 960 numbers, but the full joint distribution requires over a billion.

There are domains in which each variable can be influenced directly by all the others, so that the network is fully connected. Then specifying the conditional probability tables requires the same amount of information as specifying the joint distribution. In some domains, there will be slight dependencies that should strictly be included by adding a new link. But if these dependencies are very tenuous, then it may not be worth the additional complexity in the network for the small gain in accuracy. For example, one might object to our burglary network on the grounds that if there is an earthquake, then John and Mary would not call even if they heard the alarm, because they assume that the earthquake is the cause. Whether to add the link from *Earthquake* to *JohnCalls* and *MaryCalls* (and thus enlarge the tables) depends on comparing the importance of getting more accurate probabilities with the cost of specifying the extra information.

Even in a locally structured domain, constructing a locally structured Bayesian network is not a trivial problem. We require not only that each variable be directly influenced by only a few others, but also that the network topology actually reflect those direct influences with the appropriate set of parents. Because of the way that the construction procedure works, the "direct influencers" will have to be added to the network first if they are to become parents of the node they influence. Therefore, *the correct order in which to add nodes is to add the "root causes" first, then the variables they influence,* and so on, until we reach the "leaves," which have no direct causal influence on the other variables.

What happens if we happen to choose the wrong order? Let us consider the burglary example again. Suppose we decide to add the nodes in the order *MaryCalls*, *JohnCalls*, *Alarm*, *Burglary*, *Earthquake*. Then we get the somewhat more complicated network shown in Figure 14.3(a). The process goes as follows:

- Adding *MaryCalls*: No parents.
- Adding *JohnCalls*: If Mary calls, that probably means the alarm has gone off, which of course would make it more likely that John calls. Therefore, *JohnCalls* needs *MaryCalls* as a parent
- Adding *Alarm*: Clearly, if both call, it is more likely that the alarm has gone off than if just one or neither call, so we need both *MaryCalls* and *JohnCalls* as parents.
- Adding *Burglary*: If we know the alarm state, then the call from John or Mary might give us information about our phone ringing or Mary's music, but not about burglary:

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

Hence we need just *Alarm* as parent.

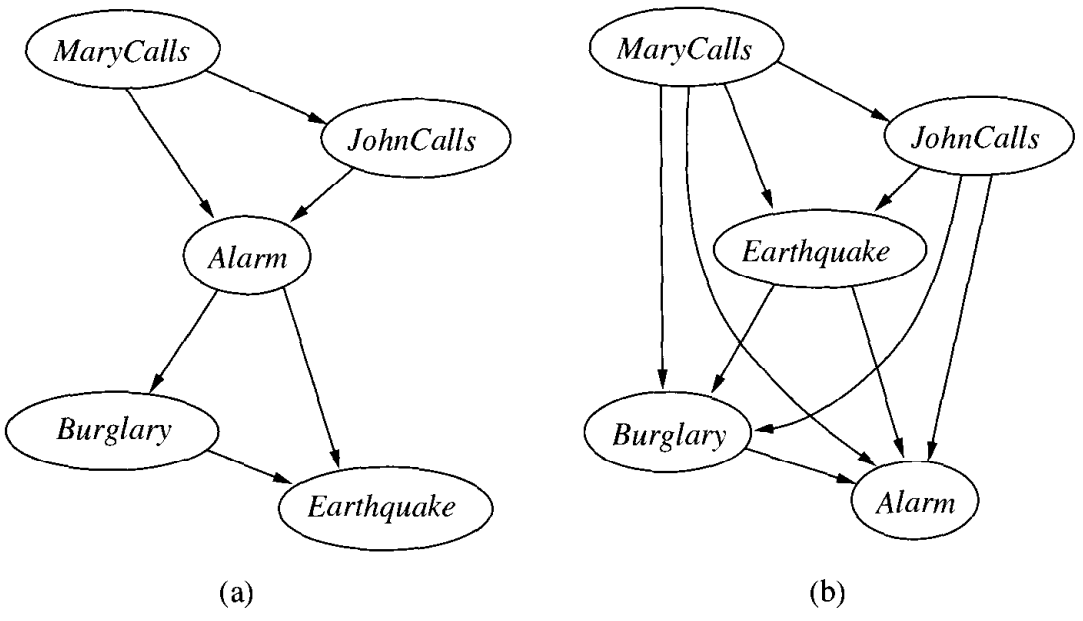


Figure 14.3 Network structure depends on order of introduction. In each network, we have introduced nodes in top-to-bottom order.

- Adding *Earthquake*: if the alarm is on, it is more likely that there has been an earthquake. (The alarm is an earthquake detector of sorts.) But if we know that there has been a burglary, then that explains the alarm, and the probability of an earthquake would be only slightly above normal. Hence, we need both *Alarm* and *Burglary* as parents.

The resulting network has two more links than the original network in Figure 14.2 and requires three more probabilities to be specified. What's worse, some of the links represent tenuous relationships that require difficult and unnatural probability judgments, such as assessing the probability of *Earthquake*, given *Burglary* and *Alarm*. This phenomenon is quite general and is related to the distinction between causal and diagnostic models introduced in Chapter 8. If we try to build a diagnostic model with links from symptoms to causes (as from *MaryCalls* to *Alarm* or *Alarm* to *Burglary*), we end up having to specify additional dependencies between otherwise independent causes (and often between separately occurring symptoms as well). *If we stick to a causal model, we end up having to specify fewer numbers, and the numbers will often be easier to come up with.* In the domain of medicine, for example, it has been shown by Tversky and Kahneman (1982) that expert physicians prefer to give probability judgments for causal rules rather than for diagnostic ones.

Figure 14.3(b) shows a very bad node ordering: *MaryCalls*, *JohnCalls*, *Earthquake*, *Burglary*, *Alarm*. This network requires 31 distinct probabilities to be specified—exactly the same as the full joint distribution. It is important to realize, however, that any of the three networks can represent *exactly the same joint distribution.* The last two versions simply fail to represent all the conditional independence relationships and hence end up specifying a lot of unnecessary numbers instead.

Conditional independence relations in Bayesian networks

We have provided a "numerical" semantics for Bayesian networks in terms of the representation of the full joint distribution, as in Equation (14.1). Using this semantics to derive a method for constructing Bayesian networks, we were led to the consequence that a node is conditionally independent of its predecessors, given its parents. It turns out that we can also go in the other direction. We can start from a "topological" semantics that specifies the conditional independence relationships encoded by the graph structure, and from these we can derive the "numerical" semantics. The topological semantics is given by either of the following specifications, which are equivalent:[2]

DESCENDANTS

1. A node is conditionally independent of its non-**descendants**, given its parents. For example, in Figure 14.2, *JohnCalls* is independent of *Burglary* and *Earthquake*, given the value of *Alarm*.

MARKOV BLANKET

2. A node is conditionally independent of all other nodes in the network, given its parents, children, and children's parents—that is, given its **Markov blanket**. For example, *Burglary* is independent of *JohnCalls* and *MaryCalls*, given *Alarm* and *Earthquake*.

These specifications are illustrated in Figure 14.4. From these conditional independence assertions and the CPTs, the full joint distribution can be reconstructed; thus, the "numerical" semantics and the "topological" semantics are equivalent.

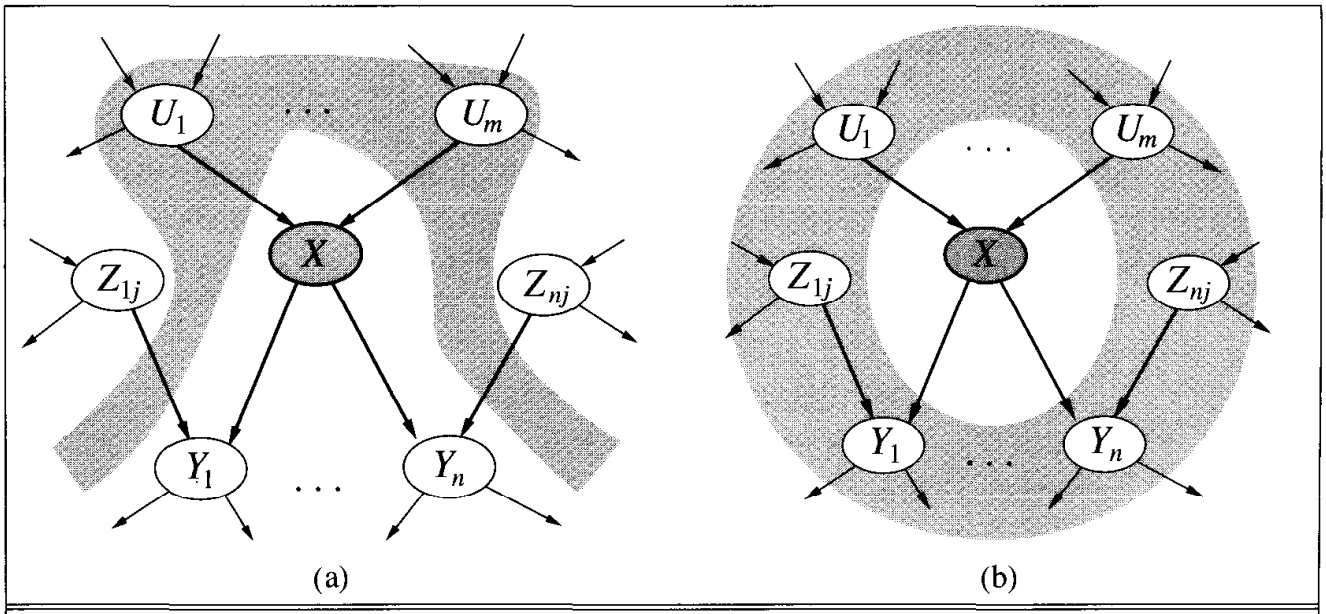


Figure 14.4 (a) A node X is conditionally independent of its non-descendants (e.g., the Z_{ij}s) given its parents (the U_is shown in the gray area). (b) A node X is conditionally independent of all other nodes in the network given its Markov blanket (the gray area).

[2] There is also a general topological criterion called **d-separation** for deciding whether a set of nodes **X** is independent of another set **Y**, given a third set **Z**. The criterion is rather complicated and is not needed for deriving the algorithms in this chapter, so we omit it. Details may be found in Russell and Norvig (1995) or Pearl (1988). Shachter (1998) gives a more intuitive method of ascertaining d-separation.

14.3 Efficient Representation of Conditional Distributions

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

CANONICAL DISTRIBUTION

DETERMINISTIC NODES

The simplest example is provided by **deterministic nodes**. A deterministic node has its value specified exactly by the values of its parents, with no uncertainty. The relationship can be a logical one: for example, the relationship between the parent nodes *Canadian*, *US*, *Mexican* and the child node *NorthAmerican* is simply that the child is the disjunction of the parents. The relationship can also be numerical: for example, if the parent nodes are the prices of a particular model of car at several dealers, and the child node is the price that a bargain hunter ends up paying, then the child node is the minimum of the parent values; or if the parent nodes are the inflows (rivers, runoff, precipitation) into a lake and the outflows (rivers, evaporation, seepage) from the lake and the child is the change in the water level of the lake, then the value of the child is the difference between the inflow parents and the outflow parents.

NOISY-OR

Uncertain relationships can often be characterized by so-called "noisy" logical relationships. The standard example is the **noisy-OR** relation, which is a generalization of the logical OR. In propositional logic, we might say that *Fever* is true if and only if *Cold*, *Flu*, or *Malaria* is true. The noisy-OR model allows for uncertainty about the ability of each parent to cause the child to be true—the causal relationship between parent and child may be *inhibited*, and so a patient could have a cold, but not exhibit a fever. The model makes two assumptions. First, it assumes that all the possible causes are listed. (This is not as strict as it seems, because we can always add a so-called **leak node** that covers "miscellaneous causes.") Second, it assumes that inhibition of each parent is independent of inhibition of any other parents: for example, whatever inhibits *Malaria* from causing a fever is independent of whatever inhibits *Flu* from causing a fever. Given these assumptions, *Fever* is *false* if and only if all its *true* parents are inhibited, and the probability of this is the product of the inhibition probabilities for each parent. Let us suppose these individual inhibition probabilities are as follows:

LEAK NODE

$$P(\neg fever|cold, \neg flu, \neg malaria) = 0.6\ ,$$
$$P(\neg fever|\neg cold, flu, \neg malaria) = 0.2\ ,$$
$$P(\neg fever|\neg cold, \neg flu, malaria) = 0.1\ .$$

Then, from this information and the noisy-OR assumptions, the entire CPT can be built. The following table shows how:

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

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

Bayesian nets with continuous variables

Many real-world problems involve continuous quantities, such as height, mass, temperature, and money; in fact, much of statistics deals with random variables whose domains are continuous. By definition, continuous variables have an infinite number of possible values, so it is impossible to specify conditional probabilities explicitly for each value. One possible way to handle continuous variables is to avoid them by using **discretization**—that is, dividing up the possible values into a fixed set of intervals. For example, temperatures could be divided into (<0°C), (0°C−100°C), and (>100°C). Discretization is sometimes an adequate solution, but often results in a considerable loss of accuracy and very large CPTs. The other solution is to define standard families of probability density functions (see Appendix A) that are specified by a finite number of **parameters**. For example, a Gaussian (or normal) distribution $N(\mu, \sigma^2)(x)$ has the mean μ and the variance σ^2 as parameters.

DISCRETIZATION

PARAMETERS

HYBRID BAYESIAN NETWORK

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

For the *Cost* variable, we need to specify $\mathbf{P}(Cost|Harvest, Subsidy)$. The discrete parent is handled by explicit enumeration—that is, specifying both $P(Cost|Harvest, subsidy)$ and $P(Cost|Harvest, \neg subsidy)$. To handle *Harvest*, we specify how the distribution over the cost c depends on the continuous value h of *Harvest*. In other words, we specify the *parameters* of the cost distribution as a function of h. The most common choice is the **linear**

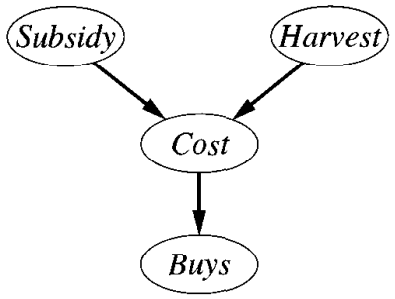


Figure 14.5 A simple network with discrete variables (*Subsidy* and *Buys*) and continuous variables (*Harvest* and *Cost*).

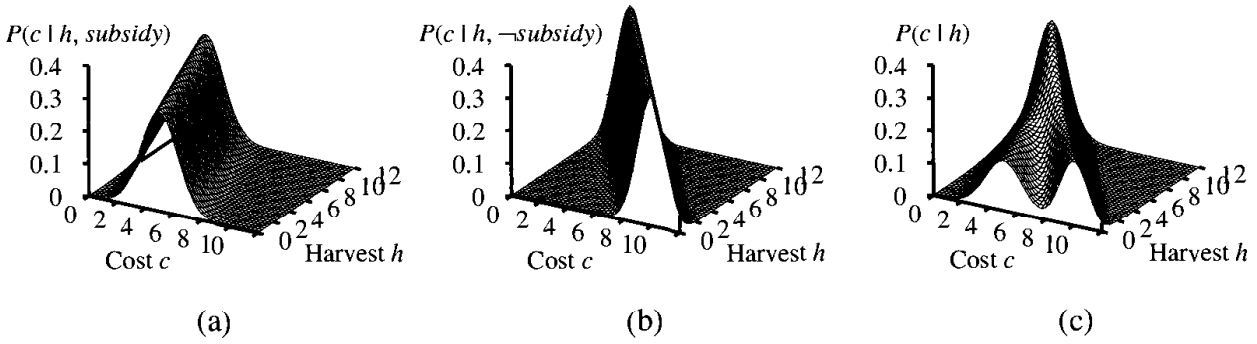


Figure 14.6 The graphs in (a) and (b) show the probability distribution over *Cost* as a function of *Harvest* size, with *Subsidy* true and false respectively. Graph (c) shows the distribution $P(Cost|Harvest)$, obtained by summing over the two subsidy cases.

LINEAR GAUSSIAN

Gaussian distribution, in which the child has a Gaussian distribution whose mean μ varies linearly with the value of the parent and whose standard deviation σ is fixed. We need two distributions, one for *subsidy* and one for $\neg subsidy$, with different parameters:

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

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

For this example, then, the conditional distribution for *Cost* is specified by naming the linear Gaussian distribution and providing the parameters a_t, b_t, σ_t, a_f, b_f, and σ_f. Figures 14.6(a) and (b) show these two relationships. Notice that in each case the slope is negative, because price decreases as supply increases. (Of course, the assumption of linearity implies that the price becomes negative at some point; the linear model is reasonable only if the harvest size is limited to a narrow range.) Figure 14.6(c) shows the distribution $P(c|h)$, averaging over the two possible values of *Subsidy* and assuming that each has prior probability 0.5. This shows that even with very simple models, quite interesting distributions can be represented.

The linear Gaussian conditional distribution has some special properties. A network containing only continuous variables with linear Gaussian distributions has a joint distribu-

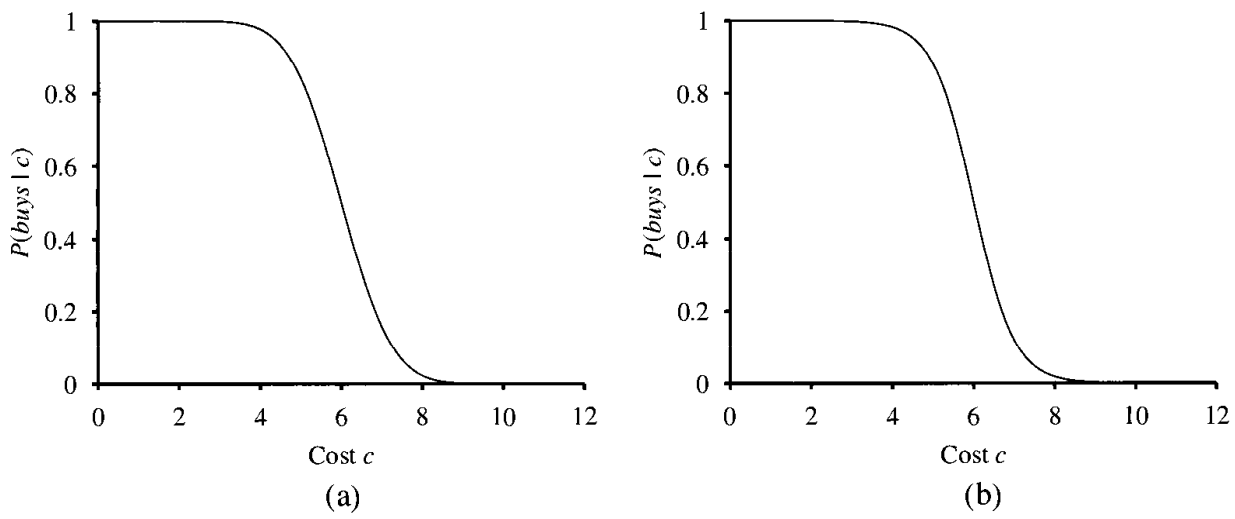


Figure 14.7 (a) A probit distribution for the probability of *Buys* given *Cost*, with $\mu = 6.0$ and $\sigma = 1.0$. (b) A logit distribution with the same parameters.

tion that is a multivariate Gaussian distribution over all the variables (Exercise 14.5).[3] (A multivariate Gaussian distribution is a surface in more than one dimension that has a peak at the mean (in n dimensions) and drops off on all sides.) When discrete variables are added (provided that no discrete variable is a child of a continuous variable), the network defines a **conditional Gaussian**, or CG, distribution: given any assignment to the discrete variables, the distribution over the continuous variables is a multivariate Gaussian.

CONDITIONAL GAUSSIAN

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

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

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

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

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

PROBIT DISTRIBUTION

This **probit distribution** is illustrated in Figure 14.7(a). The form can be justified by proposing that the underlying decision process has a hard threshold, but that the precise location of the threshold is subject to random Gaussian noise. An alternative to the probit model is the **logit distribution**, which uses the **sigmoid function** to produce a soft threshold:

LOGIT DISTRIBUTION

SIGMOID FUNCTION

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

[3] It follows that inference in linear Gaussian networks takes only $O(n^3)$ time in the worst case, regardless of the network topology. In Section 14.4, we will see that inference for networks of discrete variables is NP-hard.

This is illustrated in Figure 14.7(b). The two distributions look similar, but the logit actually has much longer "tails." The probit is often a better fit to real situations, but the logit is sometimes easier to deal with mathematically. It is used widely in neural networks (Chapter 20). Both probit and logit can be generalized to handle multiple continuous parents by taking a linear combination of the parent values. Extensions for a multivalued discrete child are explored in Exercise 14.6.

14.4 Exact Inference in Bayesian Networks

EVENT

HIDDEN VARIABLES

The basic task for any probabilistic inference system is to compute the posterior probability distribution for a set of **query variables**, given some observed **event**—that is, some assignment of values to a set of **evidence variables**. We will use the notation introduced in Chapter 13: X denotes the query variable; $\mathbf{E}$ denotes the set of evidence variables $E_1, \ldots, E_m$, and $\mathbf{e}$ is a particular observed event; $\mathbf{Y}$ will denote the nonevidence variables $Y_1, \ldots, Y_l$ (sometimes called the **hidden variables**). Thus, the complete set of variables $\mathbf{X} = \{X\} \cup \mathbf{E} \cup \mathbf{Y}$. A typical query asks for the posterior probability distribution $\mathbf{P}(X|\mathbf{e})$.[4]

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

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

In this section we will discuss exact algorithms for computing posterior probabilities and will consider the complexity of this task. It turns out that the general case is intractable, so Section 14.5 covers methods for approximate inference.

Inference by enumeration

Chapter 13 explained that any conditional probability can be computed by summing terms from the full joint distribution. More specifically, a query $\mathbf{P}(X|\mathbf{e})$ can be answered using Equation (13.6), which we repeat here for convenience:

$$\mathbf{P}(X|\mathbf{e}) = \alpha \mathbf{P}(X, \mathbf{e}) = \alpha \sum_{\mathbf{y}} \mathbf{P}(X, \mathbf{e}, \mathbf{y}) \ .$$

Now, a Bayesian network gives a complete representation of the full joint distribution. More specifically, Equation (14.1) shows that the terms $P(x, \mathbf{e}, \mathbf{y})$ in the joint distribution can be written as products of conditional probabilities from the network. Therefore, *a query can be answered using a Bayesian network by computing sums of products of conditional probabilities from the network.*

In Figure 13.4, an algorithm, ENUMERATE-JOINT-ASK, was given for inference by enumeration from the full joint distribution. The algorithm takes as input a full joint distribution $\mathbf{P}$ and looks up values therein. It is a simple matter to modify the algorithm so that it takes

[4] We will assume that the query variable is not among the evidence variables; if it is, then the posterior distribution for X simply gives probability 1 to the observed value. For simplicity, we have also assumed that the query is just a single variable. Our algorithms can be extended easily to handle a joint query over several variables.

as input a Bayesian network bn and "looks up" joint entries by multiplying the corresponding CPT entries from bn.

Consider the query $\mathbf{P}(Burglary|JohnCalls = true, MaryCalls = true)$. The hidden variables for this query are $Earthquake$ and $Alarm$. From Equation (13.6), using initial letters for the variables in order to shorten the expressions, we have[5]

$$\mathbf{P}(B|j,m) = \alpha\,\mathbf{P}(B,j,m) = \alpha \sum_e \sum_a \mathbf{P}(B,e,a,j,m) .$$

The semantics of Bayesian networks (Equation (14.1)) then gives us an expression in terms of CPT entries. For simplicity, we will do this just for $Burglary = true$:

$$P(b|j,m) = \alpha \sum_e \sum_a P(b)P(e)P(a|b,e)P(j|a)P(m|a) .$$

To compute this expression, we have to add four terms, each computed by multiplying five numbers. In the worst case, where we have to sum out almost all the variables, the complexity of the algorithm for a network with n Boolean variables is $O(n2^n)$.

An improvement can be obtained from the following simple observations: the $P(b)$ term is a constant and can be moved outside the summations over a and e, and the $P(e)$ term can be moved outside the summation over a. Hence, we have

$$P(b|j,m) = \alpha\, P(b) \sum_e P(e) \sum_a P(a|b,e)P(j|a)P(m|a) . \tag{14.3}$$

This expression can be evaluated by looping through the variables in order, multiplying CPT entries as we go. For each summation, we also need to loop over the variable's possible values. The structure of this computation is shown in Figure 14.8. Using the numbers from Figure 14.2, we obtain $P(b|j,m) = \alpha \times 0.00059224$. The corresponding computation for $\neg b$ yields $\alpha \times 0.0014919$; hence

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

That is, the chance of a burglary, given calls from both neighbors, is about 28%.

The evaluation process for the expression in Equation (14.3) is shown as an expression tree in Figure 14.8. The ENUMERATION-ASK algorithm in Figure 14.9 evaluates such trees using depth-first recursion. Thus, the space complexity of ENUMERATION-ASK is only linear in the number of variables—effectively, the algorithm sums over the full joint distribution without ever constructing it explicitly. Unfortunately, its time complexity for a network with n Boolean variables is always $O(2^n)$—better than the $O(n2^n)$ for the simple approach described earlier, but still rather grim. One thing to note about the tree in Figure 14.8 is that it makes explicit the *repeated subexpressions* that are evaluated by the algorithm. The products $P(j|a)P(m|a)$ and $P(j|\neg a)P(m|\neg a)$ are computed twice, once for each value of e. The next section describes a general method that avoids such wasted computations.

[5] An expression such as $\sum_e P(a,e)$ means to sum $P(A = a, E = e)$ for all possible values of e. There is an ambiguity in that $P(e)$ is used to mean both $P(E = true)$ and $P(E = e)$, but it should be clear from context which is intended; in particular, in the context of a sum the latter is intended.

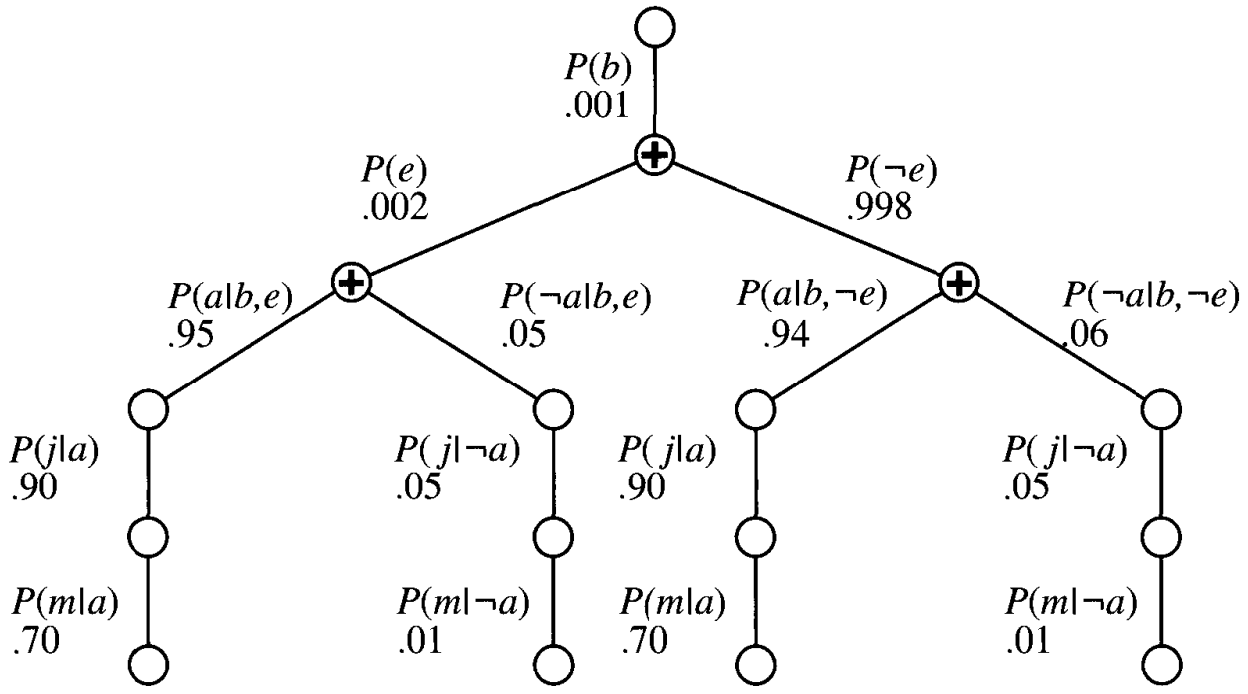


Figure 14.8 The structure of the expression shown in Equation (14.3). The evaluation proceeds top-down, multiplying values along each path and summing at the "+" nodes. Notice the repetition of the paths for j and m.

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

  $\mathbf{Q}(X) \leftarrow$ a distribution over X, initially empty
  for each value x_i of X **do**
    extend **e** with value x_i for X
    $\mathbf{Q}(x_i) \leftarrow$ ENUMERATE-ALL(VARS[bn], **e**)
  return NORMALIZE($\mathbf{Q}(X)$)

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

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

The variable elimination algorithm

The enumeration algorithm can be improved substantially by eliminating repeated calculations of the kind illustrated in Figure 14.8. The idea is simple: do the calculation once and save the results for later use. This is a form of dynamic programming. There are several versions of this approach; we present the **variable elimination** algorithm, which is the simplest. Variable elimination works by evaluating expressions such as Equation (14.3) in *right-to-left* order (that is, *bottom-up* in Figure 14.8). Intermediate results are stored, and summations over each variable are done only for those portions of the expression that depend on the variable.

VARIABLE ELIMINATION

Let us illustrate this process for the burglary network. We evaluate the expression

$$\mathbf{P}(B|j,m) = \alpha \underbrace{\mathbf{P}(B)}_{B} \sum_e \underbrace{P(e)}_{E} \sum_a \underbrace{\mathbf{P}(a|B,e)}_{A} \underbrace{P(j|a)}_{J} \underbrace{P(m|a)}_{M} .$$

Notice that we have annotated each part of the expression with the name of the associated variable; these parts are called **factors**. The steps are as follows:

FACTORS

- The factor for M, $P(m|a)$, does not require summing over M (because M's value is already fixed). We store the probability, given each value of a, in a two-element vector,
$$\mathbf{f}_M(A) = \begin{pmatrix} P(m|a) \\ P(m|\neg a) \end{pmatrix} .$$
(The $\mathbf{f}_M$ means that M was used to produce $\mathbf{f}$.)
- Similarly, we store the factor for J as the two-element vector $\mathbf{f}_J(A)$.
- The factor for A is $\mathbf{P}(a|B,e)$, which will be a $2 \times 2 \times 2$ matrix $\mathbf{f}_A(A,B,E)$.
- Now we must sum out A from the product of these three factors. This will give us a 2×2 matrix whose indices range over just B and E. We put a bar over A in the name of the matrix to indicate that A has been summed out:
$$\begin{aligned} \mathbf{f}_{\bar{A}JM}(B,E) &= \sum_a \mathbf{f}_A(a,B,E) \times \mathbf{f}_J(a) \times \mathbf{f}_M(a) \\ &= \mathbf{f}_A(a,B,E) \times \mathbf{f}_J(a) \times \mathbf{f}_M(a) \\ &+ \mathbf{f}_A(\neg a,B,E) \times \mathbf{f}_J(\neg a) \times \mathbf{f}_M(\neg a) . \end{aligned}$$
The multiplication process used here is called a **pointwise product** and will be described shortly.

POINTWISE PRODUCT

- We process E in the same way: sum out E from the product of $\mathbf{f}_E(E)$ and $\mathbf{f}_{\bar{A}JM}(B,E)$:
$$\begin{aligned} \mathbf{f}_{\bar{E}\bar{A}JM}(B) &= \mathbf{f}_E(e) \times \mathbf{f}_{\bar{A}JM}(B,e) \\ &+ \mathbf{f}_E(\neg e) \times \mathbf{f}_{\bar{A}JM}(B,\neg e) . \end{aligned}$$
- Now we can compute the answer simply by multiplying the factor for B (i.e., $\mathbf{f}_B(B) = \mathbf{P}(B)$), by the accumulated matrix $\mathbf{f}_{\bar{E}\bar{A}JM}(B)$:
$$\mathbf{P}(B|j,m) = \alpha \mathbf{f}_B(B) \times \mathbf{f}_{\bar{E}\bar{A}JM}(B) .$$

Exercise 14.7(a) asks you to check that this process yields the correct answer.

Examining this sequence of steps, we see that there are two basic computational operations required: pointwise product of a pair of factors, and summing out a variable from a product of factors.

The pointwise product is not matrix multiplication, nor is it element-by-element multiplication. The pointwise product of two factors $\mathbf{f}_1$ and $\mathbf{f}_2$ yields a new factor $\mathbf{f}$ whose variables are the *union* of the variables in $\mathbf{f}_1$ and $\mathbf{f}_2$. Suppose the two factors have variables $Y_1, \ldots, Y_k$ in common. Then we have

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

If all the variables are binary, then $\mathbf{f}_1$ and $\mathbf{f}_2$ have 2^{j+k} and 2^{k+l} entries respectively, and the pointwise product has 2^{j+k+l} entries. For example, given two factors $\mathbf{f}_1(A, B)$ and $\mathbf{f}_2(B, C)$ with probability distributions shown below, the pointwise product $\mathbf{f}_1 \times \mathbf{f}_2$ is given as $\mathbf{f}_3(A, B, C)$:

A	B	$\mathbf{f}_1(A,B)$	B	C	$\mathbf{f}_2(B,C)$	A	B	C	$\mathbf{f}_3(A,B,C)$
T	T	.3	T	T	.2	T	T	T	$.3 \times .2$
T	F	.7	T	F	.8	T	T	F	$.3 \times .8$
F	T	.9	F	T	.6	T	F	T	$.7 \times .6$
F	F	.1	F	F	.4	T	F	F	$.7 \times .4$
						F	T	T	$.9 \times .2$
						F	T	F	$.9 \times .8$
						F	F	T	$.1 \times .6$
						F	F	F	$.1 \times .4$

Summing out a variable from a product of factors is also a straightforward computation. The only trick is to notice that any factor that does *not* depend on the variable to be summed out can be moved outside the summation process. For example,

$$\sum_e \mathbf{f}_E(e) \times \mathbf{f}_A(A, B, e) \times \mathbf{f}_J(A) \times \mathbf{f}_M(A) = \mathbf{f}_J(A) \times \mathbf{f}_M(A) \times \sum_e \mathbf{f}_E(e) \times \mathbf{f}_A(A, B, e)\ .$$

Now the pointwise product inside the summation is computed, and the variable is summed out of the resulting matrix:

$$\mathbf{f}_J(A) \times \mathbf{f}_M(A) \times \sum_e \mathbf{f}_E(e) \times \mathbf{f}_A(A, B, e) = \mathbf{f}_J(A) \times \mathbf{f}_M(A) \times \mathbf{f}_{\bar{E},A}(A, B)\ .$$

Notice that matrices are *not* multiplied until we need to sum out a variable from the accumulated product. At that point, we multiply just those matrices that include the variable to be summed out. Given routines for pointwise product and summing out, the variable elimination algorithm itself can be written quite simply, as shown in Figure 14.10.

Let us consider one more query: $P(JohnCalls|Burglary = true)$. As usual, the first step is to write out the nested summation:

$$P(J|b) = \alpha\, P(b) \sum_e P(e) \sum_a P(a|b, e) P(J|a) \sum_m P(m|a)\ .$$

If we evaluate this expression from right to left, we notice something interesting: $\sum_m P(m|a)$ is equal to 1 by definition! Hence, there was no need to include it in the first place; the variable M is *irrelevant* to this query. Another way of saying this is that the result of the query $P(JohnCalls|Burglary = true)$ is unchanged if we remove $MaryCalls$ from the network altogether. In general, we can remove any leaf node that is not a query variable or an evidence

```
function ELIMINATION-ASK(X, e, bn) returns a distribution over X
  inputs: X, the query variable
          e, evidence specified as an event
          bn, a Bayesian network specifying joint distribution P(X_1, ..., X_n)

  factors ← [ ]; vars ← REVERSE(VARS[bn])
  for each var in vars do
      factors ← [MAKE-FACTOR(var, e)|factors]
      if var is a hidden variable then factors ← SUM-OUT(var, factors)
  return NORMALIZE(POINTWISE-PRODUCT(factors))
```

Figure 14.10 The variable elimination algorithm for answering queries on Bayesian networks.

variable. After its removal, there may be some more leaf nodes, and these too may be irrelevant. Continuing this process, we eventually find that *every variable that is not an ancestor of a query variable or evidence variable is irrelevant to the query.* A variable elimination algorithm can therefore remove all these variables before evaluating the query.

The complexity of exact inference

We have argued that variable elimination is more efficient than enumeration because it avoids repeated computations (as well as dropping irrelevant variables). The time and space requirements of variable elimination are dominated by the size of the largest factor constructed during the operation of the algorithm. This in turn is determined by the order of elimination of variables and by the structure of the network.

The burglary network of Figure 14.2 belongs to the family of networks in which there is at most one undirected path between any two nodes in the network. These are called **singly connected** networks or **polytrees**, and they have a particularly nice property: *The time and space complexity of exact inference in polytrees is linear in the size of the network.* Here, the size is defined as the number of CPT entries; if the number of parents of each node is bounded by a constant, then the complexity will also be linear in the number of nodes. These results hold for any ordering consistent with the topological ordering of the network (Exercise 14.7).

SINGLY CONNECTED

POLYTREES

MULTIPLY CONNECTED

For **multiply connected** networks, such as that of Figure 14.11(a), variable elimination can have exponential time and space complexity in the worst case, even when the number of parents per node is bounded. This is not surprising when one considers that, *because it includes inference in propositional logic as a special case, inference in Bayesian networks is NP-hard.* In fact, it can be shown (Exercise 14.8) that the problem is as hard as that of computing the *number* of satisfying assignments for a propositional logic formula. This means that it is #P-hard ("number-P hard")—that is, strictly harder than NP-complete problems.

There is a close connection between the complexity of Bayesian network inference and the complexity of constraint satisfaction problems (CSPs). As we discussed in Chapter 5, the difficulty of solving a discrete CSP is related to how "tree-like" its constraint graph is.

Measures such as **hypertree width**, which bound the complexity of solving a CSP, can also be applied directly to Bayesian networks. Moreover, the variable elimination algorithm can be generalized to solve CSPs as well as Bayesian networks.

Clustering algorithms

The variable elimination algorithm is simple and efficient for answering individual queries. If we want to compute posterior probabilities for all the variables in a network, however, it can be less efficient. For example, in a polytree network, one would need to issue $O(n)$ queries costing $O(n)$ each, for a total of $O(n^2)$ time. Using **clustering** algorithms (also known as **join tree** algorithms), the time can be reduced to $O(n)$. For this reason, these algorithms are widely used in commercial Bayesian network tools.

CLUSTERING

JOIN TREE

The basic idea of clustering is to join individual nodes of the network to form cluster nodes in such a way that the resulting network is a polytree. For example, the multiply connected network shown in Figure 14.11(a) can be converted into a polytree by combining the *Sprinkler* and *Rain* node into a cluster node called *Sprinkler*+*Rain*, as shown in Figure 14.11(b). The two Boolean nodes are replaced by a meganode that takes on four possible values: TT, TF, FT, and FF. The meganode has only one parent, the Boolean variable *Cloudy*, so there are two conditioning cases.

Once the network is in polytree form, a special-purpose inference algorithm is applied. Essentially, the algorithm is a form of constraint propagation (see Chapter 5) where the constraints ensure that neighboring clusters agree on the posterior probability of any variables that they have in common. With careful bookkeeping, this algorithm is able to compute posterior probabilities for all the nonevidence nodes in the network in time $O(n)$, where n is now the size of the modified network. However, the NP-hardness of the problem has not disappeared: if a network requires exponential time and space with variable elimination, then the CPTs in the clustered network will require exponential time and space to construct.

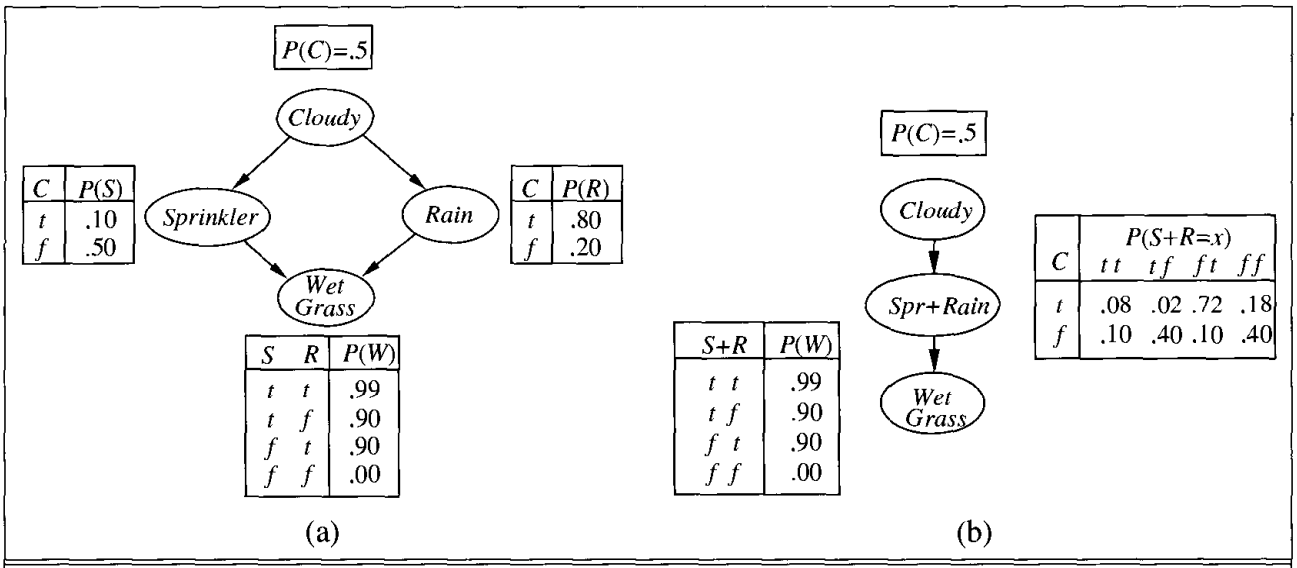


Figure 14.11 (a) A multiply connected network with conditional probability tables. (b) A clustered equivalent of the multiply connected network.

14.5 APPROXIMATE INFERENCE IN BAYESIAN NETWORKS

MONTE CARLO

Given the intractability of exact inference in large, multiply connected networks, it is essential to consider approximate inference methods. This section describes randomized sampling algorithms, also called **Monte Carlo** algorithms, that provide approximate answers whose accuracy depends on the number of samples generated. In recent years, Monte Carlo algorithms have become widely used in computer science to estimate quantities that are difficult to calculate exactly. For example, the simulated annealing algorithm described in Chapter 4 is a Monte Carlo method for optimization problems. In this section, we are interested in sampling applied to the computation of posterior probabilities. We describe two families of algorithms: direct sampling and Markov chain sampling. Two other approaches—variational methods and loopy propagation—are mentioned in the notes at the end of the chapter.

Direct sampling methods

The primitive element in any sampling algorithm is the generation of samples from a known probability distribution. For example, an unbiased coin can be thought of as a random variable $Coin$ with values $\langle heads, tails \rangle$ and a prior distribution $\mathbf{P}(Coin) = \langle 0.5, 0.5 \rangle$. Sampling from this distribution is exactly like flipping the coin: with probability 0.5 it will return $heads$, and with probability 0.5 it will return $tails$. Given a source of random numbers in the range $[0, 1]$, it is a simple matter to sample any distribution on a single variable. (See Exercise 14.9.)

The simplest kind of random sampling process for Bayesian networks generates events from a network that has no evidence associated with it. The idea is to sample each variable in turn, in topological order. The probability distribution from which the value is sampled is conditioned on the values already assigned to the variable's parents. This algorithm is shown in Figure 14.12. We can illustrate its operation on the network in Figure 14.11(a), assuming an ordering $[Cloudy, Sprinkler, Rain, WetGrass]$:

1. Sample from $\mathbf{P}(Cloudy) = \langle 0.5, 0.5 \rangle$; suppose this returns $true$.
2. Sample from $\mathbf{P}(Sprinkler | Cloudy = true) = \langle 0.1, 0.9 \rangle$; suppose this returns $false$.
3. Sample from $\mathbf{P}(Rain | Cloudy = true) = \langle 0.8, 0.2 \rangle$; suppose this returns $true$.
4. Sample from $\mathbf{P}(WetGrass | Sprinkler = false, Rain = true) = \langle 0.9, 0.1 \rangle$; suppose this returns $true$.

In this case, PRIOR-SAMPLE returns the event $[true, false, true, true]$.

It is easy to see that PRIOR-SAMPLE generates samples from the prior joint distribution specified by the network. First, let $S_{PS}(x_1, \ldots, x_n)$ be the probability that a specific event is generated by the PRIOR-SAMPLE algorithm. *Just looking at the sampling process*, we have

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

because each sampling step depends only on the parent values. This expression should look

function PRIOR-SAMPLE(bn) **returns** an event sampled from the prior specified by bn
inputs: bn, a Bayesian network specifying joint distribution $\mathbf{P}(X_1, \ldots, X_n)$

$\mathbf{x} \leftarrow$ an event with n elements
for $i = 1$ **to** n **do**
  $x_i \leftarrow$ a random sample from $\mathbf{P}(X_i \mid parents(X_i))$
return $\mathbf{x}$

Figure 14.12 A sampling algorithm that generates events from a Bayesian network.

familiar, because it is also the probability of the event according to the Bayesian net's representation of the joint distribution, as stated in Equation (14.1). That is, we have

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

This simple fact makes it very easy to answer questions by using samples.

In any sampling algorithm, the answers are computed by counting the actual samples generated. Suppose there are N total samples, and let $N(x_1, \ldots, x_n)$ be the frequency of the specific event $x_1, \ldots, x_n$. We expect this frequency to converge, in the limit, to its expected value according to the sampling probability:

$$\lim_{N \to \infty} \frac{N_{PS}(x_1, \ldots, x_n)}{N} = S_{PS}(x_1, \ldots, x_n) = P(x_1, \ldots, x_n) \ . \tag{14.4}$$

For example, consider the event produced earlier: $[true, false, true, true]$. The sampling probability for this event is

$$S_{PS}(true, false, true, true) = 0.5 \times 0.9 \times 0.8 \times 0.9 = 0.324 \ .$$

Hence, in the limit of large N, we expect 32.4% of the samples to be of this event.

Whenever we use an approximate equality ("$\approx$") in what follows, we mean it in exactly this sense—that the estimated probability becomes exact in the large-sample limit. Such an estimate is called **consistent**. For example, one can produce a consistent estimate of the probability of any partially specified event $x_1, \ldots, x_m$, where $m \leq n$, as follows:

CONSISTENT

$$P(x_1, \ldots, x_m) \approx N_{PS}(x_1, \ldots, x_m)/N \ . \tag{14.5}$$

That is, the probability of the event can be estimated as the fraction of all complete events generated by the sampling process that match the partially specified event. For example, if we generate 1000 samples from the sprinkler network, and 511 of them have $Rain = true$, then the estimated probability of rain, written as $\hat{P}(Rain = true)$, is 0.511.

Rejection sampling in Bayesian networks

REJECTION SAMPLING

Rejection sampling is a general method for producing samples from a hard-to-sample distribution given an easy-to-sample distribution. In its simplest form, it can be used to compute conditional probabilities—that is, to determine $P(X|\mathbf{e})$. The REJECTION-SAMPLING algorithm is shown in Figure 14.13. First, it generates samples from the prior distribution specified

```
function REJECTION-SAMPLING(X, e, bn, N) returns an estimate of P(X|e)
  inputs: X, the query variable
          e, evidence specified as an event
          bn, a Bayesian network
          N, the total number of samples to be generated
  local variables: N, a vector of counts over X, initially zero

  for j = 1 to N do
    x ← PRIOR-SAMPLE(bn)
    if x is consistent with e then
      N[x] ← N[x]+1 where x is the value of X in x
  return NORMALIZE(N[X])
```

Figure 14.13 The rejection sampling algorithm for answering queries given evidence in a Bayesian network.

by the network. Then, it rejects all those that do not match the evidence. Finally, the estimate $\hat{P}(X = x|\mathbf{e})$ is obtained by counting how often $X = x$ occurs in the remaining samples.

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

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

From Equation (14.5), this becomes

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

That is, rejection sampling produces a consistent estimate of the true probability.

Continuing with our example from Figure 14.11(a), let us assume that we wish to estimate $\mathbf{P}(Rain|Sprinkler = true)$, using 100 samples. Of the 100 that we generate, suppose that 73 have $Sprinkler = false$ and are rejected, while 27 have $Sprinkler = true$; of the 27, 8 have $Rain = true$ and 19 have $Rain = false$. Hence,

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

The true answer is $\langle 0.3, 0.7\rangle$. As more samples are collected, the estimate will converge to the true answer. The standard deviation of the error in each probability will be proportional to $1/\sqrt{n}$, where n is the number of samples used in the estimate.

The biggest problem with rejection sampling is that it rejects so many samples! The fraction of samples consistent with the evidence $\mathbf{e}$ drops exponentially as the number of evidence variables grows, so the procedure is simply unusable for complex problems.

Notice that rejection sampling is very similar to the estimation of conditional probabilities directly from the real world. For example, to estimate $\mathbf{P}(Rain|RedSkyAtNight = true)$, one can simply count how often it rains after a red sky is observed the previous evening—ignoring those evenings when the sky is not red. (Here, the world itself plays the role of the

sample generation algorithm.) Obviously, this could take a long time if the sky is very seldom red, and that is the weakness of rejection sampling.

Likelihood weighting

LIKELIHOOD WEIGHTING

Likelihood weighting avoids the inefficiency of rejection sampling by generating only events that are consistent with the evidence **e**. We begin by describing how the algorithm works; then we show that it works correctly—that is, generates consistent probability estimates.

LIKELIHOOD-WEIGHTING (see Figure 14.14) fixes the values for the evidence variables **E** and samples only the remaining variables X and **Y**. This guarantees that each event generated is consistent with the evidence. Not all events are equal, however. Before tallying the counts in the distribution for the query variable, each event is weighted by the *likelihood* that the event accords to the evidence, as measured by the product of the conditional probabilities for each evidence variable, given its parents. Intuitively, events in which the actual evidence appears unlikely should be given less weight.

Let us apply the algorithm to the network shown in Figure 14.11(a), with the query $\mathbf{P}(Rain|Sprinkler = true, WetGrass = true)$. The process goes as follows: First, the weight w is set to 1.0. Then an event is generated:

1. Sample from $\mathbf{P}(Cloudy) = \langle 0.5, 0.5\rangle$; suppose this returns $true$.
2. $Sprinkler$ is an evidence variable with value $true$. Therefore, we set
$$w \leftarrow w \times P(Sprinkler = true|Cloudy = true) = 0.1 .$$
3. Sample from $\mathbf{P}(Rain|Cloudy = true) = \langle 0.8, 0.2\rangle$; suppose this returns $true$.
4. $WetGrass$ is an evidence variable with value $true$. Therefore, we set
$$w \leftarrow w \times P(WetGrass = true|Sprinkler = true, Rain = true) = 0.099 .$$

Here WEIGHTED-SAMPLE returns the event $[true, true, true, true]$ with weight 0.099, and this is tallied under $Rain = true$. The weight is low because the event describes a cloudy day, which makes the sprinkler unlikely to be on.

To understand why likelihood weighting works, we start by examining the sampling distribution S_{WS} for WEIGHTED-SAMPLE. Remember that the evidence variables **E** are fixed with values **e**. We will call the other variables **Z**, that is, $\mathbf{Z} = \{X\} \cup \mathbf{Y}$. The algorithm samples each variable in **Z** given its parent values:

$$S_{WS}(\mathbf{z}, \mathbf{e}) = \prod_{i=1}^{l} P(z_i|parents(Z_i)) . \tag{14.6}$$

Notice that $Parents(Z_i)$ can include both hidden variables and evidence variables. Unlike the prior distribution $P(\mathbf{z})$, the distribution S_{WS} pays some attention to the evidence: the sampled values for each Z_i will be influenced by evidence among Z_i's ancestors. On the other hand, S_{WS} pays less attention to the evidence than does the true posterior distribution $P(\mathbf{z}|\mathbf{e})$, because the sampled values for each Z_i *ignore* evidence among Z_i's non-ancestors.[6]

[6] Ideally, we would like use a sampling distribution equal to the true posterior $P(\mathbf{z}|\mathbf{e})$, to take all the evidence into account. This cannot be done efficiently, however. If it could, then we could approximate the desired probability to arbitrary accuracy with a polynomial number of samples. It can be shown that no such polynomial-time approximation scheme can exist.

function LIKELIHOOD-WEIGHTING($X, \mathbf{e}, bn, N$) **returns** an estimate of $P(X|\mathbf{e})$
  inputs: X, the query variable
    $\mathbf{e}$, evidence specified as an event
    bn, a Bayesian network
    N, the total number of samples to be generated
  local variables: $\mathbf{W}$, a vector of weighted counts over X, initially zero

  for j = 1 to N **do**
    $\mathbf{x}, w \leftarrow$ WEIGHTED-SAMPLE(bn)
    $\mathbf{W}[x] \leftarrow \mathbf{W}[x] + w$ where x is the value of X in $\mathbf{x}$
  return NORMALIZE($\mathbf{W}[X]$)

---

function WEIGHTED-SAMPLE($bn, \mathbf{e}$) **returns** an event and a weight

  $\mathbf{x} \leftarrow$ an event with n elements; $w \leftarrow 1$
  for i = 1 **to** n **do**
    if X_i has a value x_i in $\mathbf{e}$
      then $w \leftarrow w \times P(X_i = x_i \mid parents(X_i))$
      else $x_i \leftarrow$ a random sample from $\mathbf{P}(X_i \mid parents(X_i))$
  return $\mathbf{x}$, w

Figure 14.14 The likelihood weighting algorithm for inference in Bayesian networks.

The likelihood weight w makes up for the difference between the actual and desired sampling distributions. The weight for a given sample $\mathbf{x}$, composed from $\mathbf{z}$ and $\mathbf{e}$, is the product of the likelihoods for each evidence variable given its parents (some or all of which may be among the Z_is):

$$w(\mathbf{z}, \mathbf{e}) = \prod_{i=1}^{m} P(e_i|parents(E_i)) \ . \tag{14.7}$$

Multiplying Equations (14.6) and (14.7), we see that the *weighted* probability of a sample has the particularly convenient form

$$\begin{aligned} S_{WS}(\mathbf{z}, \mathbf{e})w(\mathbf{z}, \mathbf{e}) &= \prod_{i=1}^{l} P(z_i|parents(Z_i)) \prod_{i=1}^{m} P(e_i|parents(E_i)) \\ &= P(\mathbf{z}, \mathbf{e}) \ , \end{aligned} \tag{14.8}$$

because the two products cover all the variables in the network, allowing us to use Equation (14.1) for the joint probability.

Now it is easy to show that likelihood weighting estimates are consistent. For any particular value x of X, the estimated posterior probability can be calculated as follows:

$$\begin{aligned} \hat{P}(x|\mathbf{e}) &= \alpha \sum_{\mathbf{y}} N_{WS}(x, \mathbf{y}, \mathbf{e})w(x, \mathbf{y}, \mathbf{e}) && \text{from LIKELIHOOD-WEIGHTING} \\ &\approx \alpha' \sum_{\mathbf{y}} S_{WS}(x, \mathbf{y}, \mathbf{e})w(x, \mathbf{y}, \mathbf{e}) && \text{for large } N \end{aligned}$$

$$= \alpha' \sum_{\mathbf{y}} P(x, \mathbf{y}, \mathbf{e}) \qquad \text{by Equation (14.8)}$$
$$= \alpha' P(x, \mathbf{e}) = P(x|\mathbf{e}) \, .$$

Hence, likelihood weighting returns consistent estimates.

Because likelihood weighting uses all the samples generated, it can be much more efficient than rejection sampling. It will, however, suffer a degradation in performance as the number of evidence variables increases. This is because most samples will have very low weights and hence the weighted estimate will be dominated by the tiny fraction of samples that accord more than an infinitesimal likelihood to the evidence. The problem is exacerbated if the evidence variables occur late in the variable ordering, because then the samples will be simulations that bear little resemblance to the reality suggested by the evidence.

Inference by Markov chain simulation

MARKOV CHAIN MONTE CARLO

In this section, we describe the **Markov chain Monte Carlo** (MCMC) algorithm for inference in Bayesian networks. We will first describe what the algorithm does, then we will explain why it works and why it has such a complicated name.

The MCMC algorithm

Unlike the other two sampling algorithms, which generate each event from scratch, MCMC generates each event by making a random change to the preceding event. It is therefore helpful to think of the network as being in a particular *current state* specifying a value for every variable. The next state is generated by randomly sampling a value for one of the nonevidence variables X_i, *conditioned on the current values of the variables in the Markov blanket of* X_i. (Recall from page 499 that the Markov blanket of a variable consists of its parents, children, and children's parents.) MCMC therefore wanders randomly around the state space—the space of possible complete assignments—flipping one variable at a time, but keeping the evidence variables fixed.

Consider the query $\mathbf{P}(Rain|Sprinkler = true, WetGrass = true)$ applied to the network in Figure 14.11(a). The evidence variables $Sprinkler$ and $WetGrass$ are fixed to their observed values and the hidden variables $Cloudy$ and $Rain$ are initialized randomly—let us say to $true$ and $false$ respectively. Thus, the initial state is $[true, true, false, true]$. Now the following steps are executed repeatedly:

1. $Cloudy$ is sampled, given the current values of its Markov blanket variables: in this case, we sample from $\mathbf{P}(Cloudy|Sprinkler = true, Rain = false)$. (Shortly, we will show how to calculate this distribution.) Suppose the result is $Cloudy = false$. Then the new current state is $[false, true, false, true]$.
2. $Rain$ is sampled, given the current values of its Markov blanket variables: in this case, we sample from $\mathbf{P}(Rain|Cloudy = false, Sprinkler = true, WetGrass = true)$. Suppose this yields $Rain = true$. The new current state is $[false, true, true, true]$.

Each state visited during this process is a sample that contributes to the estimate for the query variable $Rain$. If the process visits 20 states where $Rain$ is true and 60 states where $Rain$ is

```
function MCMC-ASK(X, e, bn, N) returns an estimate of P(X|e)
  local variables: N[X], a vector of counts over X, initially zero
                   Z, the nonevidence variables in bn
                   x, the current state of the network, initially copied from e

  initialize x with random values for the variables in Z
  for j = 1 to N do
    for each Z_i in Z do
      sample the value of Z_i in x from P(Z_i|mb(Z_i)) given the values of MB(Z_i) in x
      N[x] ← N[x] + 1 where x is the value of X in x
  return NORMALIZE(N[X])
```

Figure 14.15 The MCMC algorithm for approximate inference in Bayesian networks.

false, then the answer to the query is NORMALIZE($\langle 20, 60\rangle$) = $\langle 0.25, 0.75\rangle$. The complete algorithm is shown in Figure 14.15.

Why MCMC works

We will now show that MCMC returns consistent estimates for posterior probabilities. The material in this section is quite technical, but the basic claim is straightforward: *the sampling process settles into a "dynamic equilibrium" in which the long-run fraction of time spent in each state is exactly proportional to its posterior probability.* This remarkable property follows from the specific **transition probability** with which the process moves from one state to another, as defined by the conditional distribution given the Markov blanket of the variable being sampled.

TRANSITION PROBABILITY

Let $q(\mathbf{x} \rightarrow \mathbf{x}')$ be the probability that the process makes a transition from state $\mathbf{x}$ to state $\mathbf{x}'$. This transition probability defines what is called a **Markov chain** on the state space. (Markov chains will also figure prominently in Chapters 15 and 17.) Now suppose that we run the Markov chain for t steps, and let $\pi_t(\mathbf{x})$ be the probability that the system is in state $\mathbf{x}$ at time t. Similarly, let $\pi_{t+1}(\mathbf{x}')$ be the probability of being in state $\mathbf{x}'$ at time $t+1$. Given $\pi_t(\mathbf{x})$, we can calculate $\pi_{t+1}(\mathbf{x}')$ by summing, for all states the system could be in at time t, the probability of being in that state times the probability of making the transition to $\mathbf{x}'$:

MARKOV CHAIN

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

STATIONARY DISTRIBUTION

We will say that the chain has reached its **stationary distribution** if $\pi_t = \pi_{t+1}$. Let us call this stationary distribution π; its defining equation is therefore

$$\pi(\mathbf{x}') = \sum_{\mathbf{x}} \pi(\mathbf{x}) q(\mathbf{x} \rightarrow \mathbf{x}') \qquad \text{for all } \mathbf{x}' \ . \tag{14.9}$$

Under certain standard assumptions about the transition probability distribution q,[7] there is exactly one distribution π satisfying this equation for any given q.

[7] The Markov chain defined by q must be **ergodic**—that is, essentially, every state must be reachable from every other, and there can be no strictly periodic cycles.

Equation (14.9) can be read as saying that the expected "outflow" from each state (i.e., its current "population") is equal to the expected "inflow" from all the states. One obvious way to satisfy this relationship is if the expected flow between any pair of states is the same in both directions. This is the property of **detailed balance**:

DETAILED BALANCE

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

We can show that detailed balance implies stationarity simply by summing over $\mathbf{x}$ in Equation (14.10). We have

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

where the last step follows because a transition from $\mathbf{x}'$ is guaranteed to occur.

Now we will show that the transition probability $q(\mathbf{x} \rightarrow \mathbf{x}')$ defined by the sampling step in MCMC-ASK satisfies the detailed balance equation with a stationary distribution equal to $P(\mathbf{x}|\mathbf{e})$, (the true posterior distribution on the hidden variables). We will do this in two steps. First, we will define a Markov chain in which each variable is sampled conditionally on the current values of *all* the other variables, and we will show that this satisfies detailed balance. Then, we will simply observe that, for Bayesian networks, doing that is equivalent to sampling conditionally on the variable's Markov blanket (see page 499).

Let X_i be the variable to be sampled, and let $\overline{\mathbf{X}_i}$ be all the hidden variables *other than* X_i. Their values in the current state are x_i and $\overline{\mathbf{x}_i}$. If we sample a new value x_i' for X_i conditionally on all the other variables, including the evidence, we have

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

GIBBS SAMPLER

This transition probability is called the **Gibbs sampler** and is a particularly convenient form of MCMC. Now we show that the Gibbs sampler is in detailed balance with the true posterior:

$$\begin{aligned}
&\pi(\mathbf{x})q(\mathbf{x} \rightarrow \mathbf{x}') = P(\mathbf{x}|\mathbf{e})P(x_i'|\overline{\mathbf{x}_i}, \mathbf{e}) = P(x_i, \overline{\mathbf{x}_i}|\mathbf{e})P(x_i'|\overline{\mathbf{x}_i}, \mathbf{e}) \\
&= P(x_i|\overline{\mathbf{x}_i}, \mathbf{e})P(\overline{\mathbf{x}_i}|\mathbf{e})P(x_i'|\overline{\mathbf{x}_i}, \mathbf{e}) \qquad \text{(using the chain rule on the first term)} \\
&= P(x_i|\overline{\mathbf{x}_i}, \mathbf{e})P(x_i', \overline{\mathbf{x}_i}|\mathbf{e}) \qquad \text{(using the chain rule backwards)} \\
&= \pi(\mathbf{x}')q(\mathbf{x}' \rightarrow \mathbf{x}) .
\end{aligned}$$

As stated on page 499, a variable is independent of all other variables given its Markov blanket; hence,

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

where $mb(X_i)$ denotes the values of the variables in X_i's Markov blanket, $MB(X_i)$. As shown in Exercise 14.10, the probability of a variable given its Markov blanket is proportional to the probability of the variable given its parents times the probability of each child given its respective parents:

$$P(x_i'|mb(X_i)) = \alpha\, P(x_i'|parents(X_i)) \times \prod_{Y_j \in Children(X_i)} P(y_j|parents(Y_j)) . \tag{14.11}$$

Hence, to flip each variable X_i, the number of multiplications required is equal to the number of X_i's children.

We have discussed here only one simple variant of MCMC, namely the Gibbs sampler. In its most general form, MCMC is a powerful method for computing with probability models and many variants have been developed, including the simulated annealing algorithm presented in Chapter 4, the stochastic satisfiability algorithms in Chapter 7, and the Metropolis–Hastings sampler in Chapter 15.

14.6 Extending Probability to First-Order Representations

In Chapter 8, we explained the representational advantages possessed by first-order logic in comparison to propositional logic. First-order logic commits to the existence of objects and relations among them and can express facts about *some* or *all* of the objects in a domain. This often results in representations that are vastly more concise than the equivalent propositional descriptions. Now, Bayesian networks are essentially propositional: the set of variables is fixed and finite, and each has a fixed domain of possible values. This fact limits the applicability of Bayesian networks. *If we can find a way to combine probability theory with the expressive power of first-order representations, we expect to be able to increase dramatically the range of problems that can be handled.*

The basic insight required to achieve this goal is the following: In the propositional context, a Bayesian network specifies probabilities over atomic events, each of which specifies a value for each variable in the network. Thus, an atomic event is a **model** or **possible world**, in the terminology of propositional logic. In the first-order context, a model (with its interpretation) specifies a domain of objects, the relations that hold among those objects, and a mapping from the constants and predicates of the knowledge base to the objects and relations in the model. Therefore, *a first-order probabilistic knowledge base should specify probabilities for all possible first-order models.* Let $\mu(M)$ be the probability assigned to model M by the knowledge base. For any first-order sentence ϕ, the probability $P(\phi)$ is given in the usual way by summing over the possible worlds where ϕ is true:

$$P(\phi) = \sum_{M:\phi \text{ is true in } M} \mu(M) \; . \tag{14.12}$$

So far, so good. There is, however, a problem: the set of first-order models is infinite. This means that (1) the summation could be infeasible, and (2) specifying a complete, consistent distribution over an infinite set of worlds could be very difficult.

Let us scale back our ambition, at least temporarily. In particular, let us devise a restricted language for which there are only finitely many models of interest. There are several ways to do this. Here, we present **relational probability models**, or RPMs, which borrow ideas from semantic networks (Chapter 10) and from object-relational databases. Other approaches are discussed in the bibliographical and historical notes.

RELATIONAL PROBABILITY MODEL

RPMs allow constant symbols that name objects. For example, let *ProfSmith* be the name of a professor, and let *Jones* be the name of a student. Each object is an instance of a class; for example, *ProfSmith* is a *Professor* and *Jones* is a *Student*. We assume that the class of every constant symbol is known.

SIMPLE FUNCTION

Our function symbols will be divided into two kinds. The first kind, **simple functions**, maps an object not to another structured object, but to a value from a fixed domain of values, just like a random variable. For example, $Intelligence(Jones)$ and $Funding(ProfSmith)$ might be hi or lo; $Success(Jones)$ and $Fame(ProfSmith)$ may be $true$ or $false$. Function symbols must not be applied to values such as $true$ and $false$, so it is not possible to have nesting of simple functions. In this way, we avoid one source of infinities. The value of a simple function applied to a given object may be observed or unknown; these will be the basic random variables of our representation.[8]

COMPLEX FUNCTION

We also allow **complex functions**, which map objects to other objects. For example, $Advisor(Jones)$ may be $ProfSmith$. Each complex function has a specified domain and range, which are classes. For example, the domain of $Advisor$ is $Student$ and the range is $Professor$. Functions apply only to objects of the right class; for instance, the $Advisor$ of $ProfSmith$ is undefined. Complex functions may be nested: $DeptHead(Advisor(Jones))$ could be $ProfMoore$. We will assume (for now) that the values of all complex functions are known for all constant symbols. Because the KB is finite, this implies that every chain of complex function applications leads to one of a finite number of objects.[9]

The last element we need is the probabilistic information. For each simple function, we specify a set of parents, just as in Bayesian networks. The parents can be other simple functions of the same object; for example, the $Funding$ of a $Professor$ might depend on his or her $Fame$. The parents can also be simple functions of *related* objects—for example, the $Success$ of a student could depend on the $Intelligence$ of the student and the $Fame$ of the student's advisor. These are really *universally quantified* assertions about the parents of all the objects in a class. Thus, we could write

$$\forall x \;\; x \in Student \;\Rightarrow$$
$$Parents(Success(x)) = \{Intelligence(x), Fame(Advisor(x))\} .$$

(Less formally, we can draw diagrams like Figure 14.16(a).) Now we specify the conditional probability distribution for the child, given its parents. For example, we might say that

$$\forall x \;\; x \in Student \;\Rightarrow$$
$$P(Success(x) = true | Intelligence(x) = hi, Fame(Advisor(x)) = true) = 0.95 .$$

Just as in semantic networks, we can attach the conditional distribution to the class itself, so that the instances **inherit** the dependencies and conditional probabilities from the class.

The semantics for the RPM language assumes that every constant symbol refers to a distinct object—the **unique names assumption** described in Chapter 10. Given this assumption and the restrictions listed previously, it can be shown that every RPM generates a fixed, finite set of random variables, each of which is a simple function applied to a constant symbol. Then, provided that the parent–child dependencies are *acyclic*, we can construct an equivalent Bayesian network. That is, the RPM and the Bayesian network specify identical probabili-

[8] They play a role very similar to that of the ground atomic sentences generated in the propositionalization process described in Section 9.1.

[9] This restriction means that we cannot use complex functions such as $Father$ and $Mother$, which lead to potentially infinite chains that would have to end with an unknown object. We revisit this restriction later.

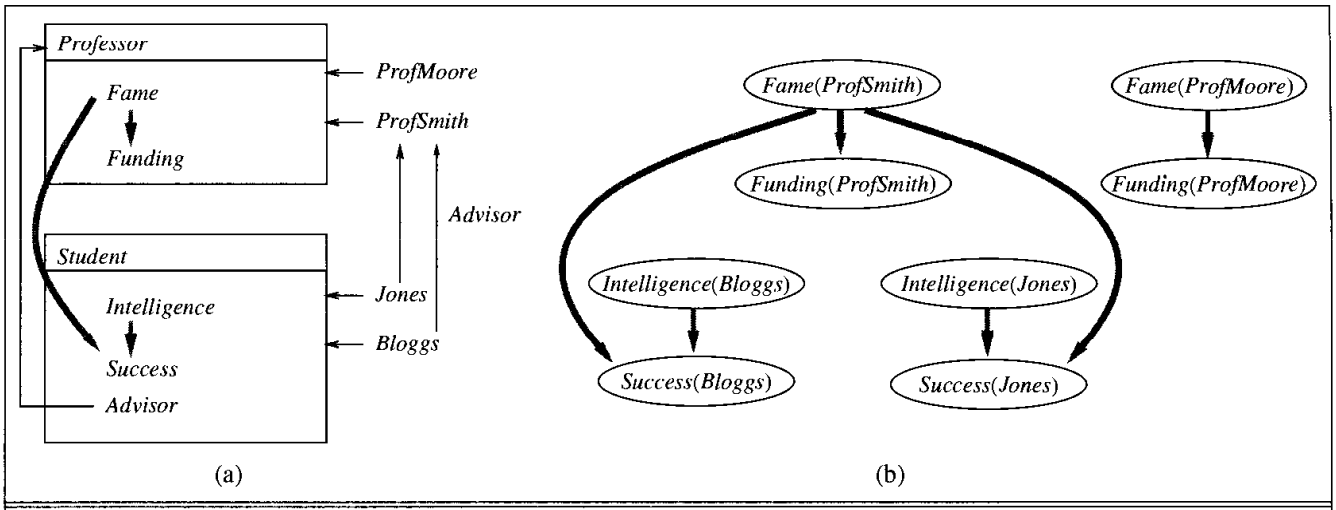


Figure 14.16 (a) An RPM describing two classes: *Professor* and *Student*. There are two professors and two students, and *ProfSmith* is the advisor of both students. (b) The Bayesian network equivalent to the RPM in (a).

ties for each possible world. Figure 14.16(b) shows the Bayesian network corresponding to the RPM in Figure 14.16(a). Notice that the *Advisor* links in the RPM are absent in the Bayesian network. This is because they are fixed and known. They appear *implicitly* in the network topology, however; for example, $Success(Jones)$ has $Fame(ProfSmith)$ as a parent because $Advisor(Jones)$ is $ProfSmith$. In general, the relations that hold among the objects determine the pattern of dependencies among the properties of those objects.

RECURSIVE DEPENDENCY

There are several ways to increase the expressive power of RPMs. We can allow **recursive dependencies** among variables to capture certain kinds of recurring relationships. For example, suppose that addiction to fast food is caused by the $McGene$. Then, for any x, $McGene(x)$ depends on $McGene(Father(x))$ and $McGene(Mother(x))$, which depend in turn on $McGene(Father(Father(x)))$, $McGene(Mother(Father(x)))$, and so on. Even though such knowledge bases correspond to Bayesian networks with infinitely many random variables, solutions can sometimes be obtained from fixed-point equations. For example, the equilibrium distribution of the $McGene$ can be calculated, given the conditional probability of inheritance. Another very important family of recursive knowledge bases consists of the **temporal probability models** described in Chapter 15. In these models, properties of the state at time t depend on properties of the state at time $t - 1$, and so on.

RELATIONAL UNCERTAINTY

RPMs can also be extended to allow for **relational uncertainty**—that is, uncertainty about the values of complex functions. For example, we may not know who $Advisor(Jones)$ is. $Advisor(Jones)$ then becomes a random variable, with possible values $ProfSmith$ and $ProfMoore$. The corresponding network is shown in Figure 14.17.

IDENTITY UNCERTAINTY

There can also be **identity uncertainty**; for example, we might not know whether $Mary$ and $ProfSmith$ are the same person. With identity uncertainty, the number of objects and propositions can vary across possible worlds. A world where $Mary$ and $ProfSmith$ are the same person has one fewer object than a world in which they are different people. This makes the inference process more complicated, but the basic principle established in Equation (14.12) still holds: the probability of any sentence is well defined and can be calculated.

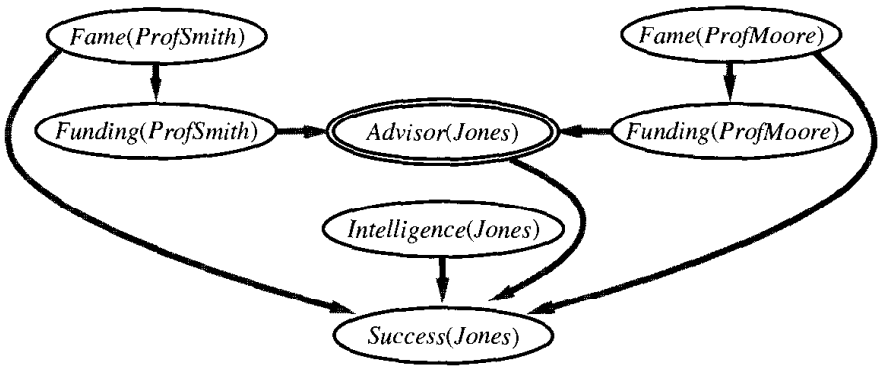


Figure 14.17 Part of the Bayesian network corresponding to an RPM in which $Advisor(Jones)$ is unknown, but is either $ProfSmith$ or $ProfMoore$. The choice of advisor depends on how much funding each professor has. Notice that $Success(Jones)$ will now depend on the $Fame$ of *both* professors, although the value of $Advisor(Jones)$ determines which one actually has an influence.

Identity uncertainty is particularly important for robots and for embedded sensor systems that must keep track of multiple objects. We return to this problem in Chapter 15.

Let us now examine the question of inference. Clearly, inference can be done in the equivalent Bayesian network, provided that we restrict the RPM language so that the equivalent network is finite and has a fixed structure. This is analogous to the way in which first-order logical inference can be done via propositional inference on the equivalent propositional knowledge base. (See Section 9.1.) As in the logical case, the equivalent network could be too large to construct, let alone evaluate. Dense interconnections are also a problem. (See Exercise 14.12.) Approximation algorithms. such as MCMC (Section 14.5), are therefore very useful for RPM inference.

When MCMC is applied to the equivalent Bayesian network for a simple RPM knowledge base with no relational or identity uncertainty, the algorithm samples from the space of possible worlds defined by the values of simple functions of the objects. It is easy to see that this approach can be extended to handle relational and identity uncertainty as well. In that case, a transition between possible worlds might change the value of a simple function or it might change a complex function, and so lead to a change in the dependency structure. Transitions might also change the identity relations among the constant symbols. Thus, MCMC seems to be an elegant way to handle inference for quite expressive first-order probabilistic knowledge bases.

Research in this area is still at an early stage, but already it is becoming clear that first-order probabilistic reasoning yields a tremendous increase in the effectiveness of AI systems at handling uncertain information. Potential applications include computer vision, natural language understanding, information retrieval, and situation assessment. In all of these areas, the set of objects—and hence the set of random variables—is not known in advance, so purely "propositional" methods, such as Bayesian networks, are incapable of representing the situation completely. They have been augmented by search over the space of model, but RPMs allow reasoning about this uncertainty in a single model.

14.7 Other Approaches to Uncertain Reasoning

Other sciences (e.g., physics, genetics, and economics) have long favored probability as a model for uncertainty. In 1819, Pierre Laplace said "Probability theory is nothing but common sense reduced to calculation." In 1850, James Maxwell said "the true logic for this world is the calculus of Probabilities, which takes account of the magnitude of the probability which is, or ought to be, in a reasonable man's mind."

Given this long tradition, it is perhaps surprising that AI has considered many alternatives to probability. The earliest expert systems of the 1970s ignored uncertainty and used strict logical reasoning, but it soon became clear that this was impractical for most real-world domains. The next generation of expert systems (especially in medical domains) used probabilistic techniques. Initial results were promising, but they did not scale up because of the exponential number of probabilities required in the full joint distribution. (Efficient Bayesian network algorithms were unknown then.) As a result, probabilistic approaches fell out of favor from roughly 1975 to 1988, and a variety of alternatives to probability were tried for a variety of reasons:

- One common view is that probability theory is essentially numerical, whereas human judgmental reasoning is more "qualitative." Certainly, we are not consciously aware of doing numerical calculations of degrees of belief. (Neither are we aware of doing unification, yet we seem to be capable of some kind of logical reasoning.) It might be that we have some kind of numerical degrees of belief encoded directly in strengths of connections and activations in our neurons. In that case, the difficulty of conscious access to those strengths is not surprising. One should also note that qualitative reasoning mechanisms can be built directly on top of probability theory, so that the "no numbers" argument against probability has little force. Nonetheless, some qualitative schemes have a good deal of appeal in their own right. One of the best studied is **default reasoning**, which treats conclusions not as "believed to a certain degree," but as "believed until a better reason is found to believe something else." Default reasoning is covered in Chapter 10.
- **Rule-based** approaches to uncertainty also have been tried. Such approaches hope to build on the success of logical rule-based systems, but add a sort of "fudge factor" to each rule to accommodate uncertainty. These methods were developed in the mid-1970s and formed the basis for a large number of expert systems in medicine and other areas.
- One area that we have not addressed so far is the question of **ignorance**, as opposed to uncertainty. Consider the flipping of a coin. If we know that the coin is fair, then a probability of 0.5 for heads is reasonable. If we know that the coin is biased, but we do not know which way, then 0.5 is the only reasonable probability. Obviously, the two cases are different, yet probability seems not to distinguish them. The **Dempster–Shafer theory** uses **interval-valued** degrees of belief to represent an agent's knowledge of the probability of a proposition. Other methods using second-order probabilities are also discussed.

- Probability makes the same ontological commitment as logic: that events are true or false in the world, even if the agent is uncertain as to which is the case. Researchers in **fuzzy logic** have proposed an ontology that allows **vagueness**: that an event can be "sort of" true. Vagueness and uncertainty are in fact orthogonal issues, as we will see.

The next three subsections treat some of these approaches in slightly more depth. We will not provide detailed technical material, but we cite references for further study.

Rule-based methods for uncertain reasoning

Rule-based systems emerged from early work on practical and intuitive systems for logical inference. Logical systems in general, and logical rule-based systems in particular, have three desirable properties:

LOCALITY

◇ **Locality**: In logical systems, whenever we have a rule of the form $A \Rightarrow B$, we can conclude B, given evidence A, *without worrying about any other rules.* In probabilistic systems, we need to consider all the evidence in the Markov blanket.

DETACHMENT

◇ **Detachment**: Once a logical proof is found for a proposition B, the proposition can be used regardless of how it was derived. That is, it can be **detached** from its justification. In dealing with probabilities, on the other hand, the source of the evidence for a belief is important for subsequent reasoning.

TRUTH-FUNCTIONALITY

◇ **Truth-functionality**: In logic, the truth of complex sentences can be computed from the truth of the components. Probability combination does not work this way, except under strong global independence assumptions.

There have been several attempts to devise uncertain reasoning schemes that retain these advantages. The idea is to attach degrees of belief to propositions and rules and to devise purely local schemes for combining and propagating those degrees of belief. The schemes are also truth-functional; for example, the degree of belief in $A \vee B$ is a function of the belief in A and the belief in B.

The bad news for rule-based systems is that the properties of *locality, detachment, and truth-functionality are simply not appropriate for uncertain reasoning.* Let us look at truth-functionality first. Let H_1 be the event that a fair coin flip comes up heads, let T_1 be the event that the coin comes up tails on that same flip, and let H_2 be the event that the coin comes up heads on a second flip. Clearly, all three events have the same probability, 0.5, and so a truth-functional system must assign the same belief to the disjunction of any two of them. But we can see that the probability of the disjunction depends on the events themselves and not just on their probabilities:

$P(A)$	$P(B)$	$P(A \vee B)$
$P(H_1) = 0.5$	$P(H_1) = 0.5$	$P(H_1 \vee H_1) = 0.50$
	$P(T_1) = 0.5$	$P(H_1 \vee T_1) = 1.00$
	$P(H_2) = 0.5$	$P(H_1 \vee H_2) = 0.75$

It gets worse when we chain evidence together. Truth-functional systems have **rules** of the form $A \mapsto B$ that allow us to compute the belief in B as a function of the belief in the rule

and the belief in A. Both forward- and backward-chaining systems can be devised. The belief in the rule is assumed to be constant and is usually specified by the knowledge engineer—for example, as $A \mapsto_{0.9} B$.

Consider the wet-grass situation from Figure 14.11(a). If we wanted to be able to do both causal and diagnostic reasoning, we would need the two rules

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

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

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

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

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

Given these difficulties, how is it possible that truth-functional systems were ever considered useful? The answer lies in restricting the task and in carefully engineering the rule base so that undesirable interactions do not occur. The most famous example of a truth-functional system for uncertain reasoning is the **certainty factors** model, which was developed for the MYCIN medical diagnosis program and was widely used in expert systems of the late 1970s and 1980s. Almost all uses of certainty factors involved rule sets that were either purely diagnostic (as in MYCIN) or purely causal. Furthermore, evidence was entered only at the "roots" of the rule set, and most rule sets were singly connected. Heckerman (1986) has shown that, under these circumstances, a minor variation on certainty-factor inference was exactly equivalent to Bayesian inference on polytrees. In other circumstances, certainty factors could yield disastrously incorrect degrees of belief through overcounting of evidence. As rule sets became larger, undesirable interactions between rules became more common, and practitioners found that the certainty factors of many other rules had to be "tweaked" when new rules were added. Needless to say, the approach is no longer recommended.

CERTAINTY FACTORS

Representing ignorance: Dempster–Shafer theory

DEMPSTER–SHAFER

The **Dempster–Shafer** theory is designed to deal with the distinction between **uncertainty** and **ignorance**. Rather than computing the probability of a proposition, it computes the probability that the evidence supports the proposition. This measure of belief is called a **belief function**, written $Bel(X)$.

BELIEF FUNCTION

We return to coin flipping for an example of belief functions. Suppose a shady character comes up to you and offers to bet you \$10 that his coin will come up heads on the next flip. Given that the coin might or might not be fair, what belief should you ascribe to the event that it comes up heads? Dempster–Shafer theory says that because you have no evidence either way, you have to say that the belief $Bel(Heads) = 0$ and also that $Bel(\neg Heads) = 0$.

This makes Dempster–Shafer reasoning systems skeptical in a way that has some intuitive appeal. Now suppose you have an expert at your disposal who testifies with 90% certainty that the coin is fair (i.e., he is 90% sure that $P(Heads) = 0.5$). Then Dempster–Shafer theory gives $Bel(Heads) = 0.9 \times 0.5 = 0.45$ and likewise $Bel(\neg Heads) = 0.45$. There is still a 10 percentage point "gap" that is not accounted for by the evidence. "Dempster's rule" (Dempster, 1968) shows how to combine evidence to give new values for Bel, and Shafer's work extends this into a complete computational model.

As with default reasoning, there is a problem in connecting beliefs to actions. With probabilities, decision theory says that if $P(Heads) = P(\neg Heads) = 0.5$, then (assuming that winning \$10 and losing \$10 are considered equal magnitude opposites) the reasoner will be indifferent between the action of accepting and declining the bet. A Dempster–Shafer reasoner has $Bel(\neg Heads) = 0$ and thus no reason to accept the bet, but then it also has $Bel(Heads) = 0$ and thus no reason to decline it. Thus, it seems that the Dempster–Shafer reasoner comes to the same conclusion about how to act in this case. Unfortunately, Dempster–Shafer theory allows no definite decision in many other cases where probabilistic inference does yield a specific choice. In fact, the notion of utility in the Dempster–Shafer model is not yet well understood.

One interpretation of Dempster—Shafer theory is that it defines a probability interval: the interval for *Heads* is $[0, 1]$ before our expert testimony and $[0.45, 0.55]$ after. The width of the interval might be an aid in deciding when we need to acquire more evidence: it can tell you that the expert's testimony will help you if you do not know whether the coin is fair, but will not help you if you have already learned that the coin is fair. However, there are no clear guidelines for how to do this, because there is no clear meaning for what the width of an interval means. In the Bayesian approach, this kind of reasoning can be done easily by examining how much one's belief would change if one were to acquire more evidence. For example, knowing whether the coin is fair would have a significant impact on the belief that it will come up heads, and detecting an asymmetric weight would have an impact on the belief that the coin is fair. A complete Bayesian model would include probability estimates for factors such as these, allowing us to express our "ignorance" in terms of how our beliefs would change in the face of future information gathering.

Representing vagueness: Fuzzy sets and fuzzy logic

FUZZY SET THEORY

Fuzzy set theory is a means of specifying how well an object satisfies a vague description. For example, consider the proposition "Nate is tall." Is this true, if Nate is $5'\ 10''$? Most people would hesitate to answer "true" or "false," preferring to say, "sort of." Note that this is not a question of uncertainty about the external world—we are sure of Nate's height. The issue is that the linguistic term "tall" does not refer to a sharp demarcation of objects into two classes—there are *degrees* of tallness. For this reason, *fuzzy set theory is not a method for uncertain reasoning at all.* Rather, fuzzy set theory treats *Tall* as a fuzzy predicate and says that the truth value of $Tall(Nate)$ is a number between 0 and 1, rather than being just *true* or *false*. The name "fuzzy set" derives from the interpretation of the predicate as implicitly defining a set of its members—a set that does not have sharp boundaries.

FUZZY LOGIC

Fuzzy logic is a method for reasoning with logical expressions describing membership in fuzzy sets. For example, the complex sentence $Tall(Nate) \wedge Heavy(Nate)$ has a fuzzy truth value that is a function of the truth values of its components. The standard rules for evaluating the fuzzy truth, T, of a complex sentence are

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

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

FUZZY CONTROL

Fuzzy control is a methodology for constructing control systems in which the mapping between real-valued input and output parameters is represented by fuzzy rules. Fuzzy control has been very successful in commercial products such as automatic transmissions, video cameras, and electric shavers. Critics (see, e.g., Elkan, 1993) argue that these applications are successful because they have small rule bases, no chaining of inferences, and tunable parameters that can be adjusted to improve the system's performance. The fact that they are implemented with fuzzy operators might be incidental to their success; the key is simply to provide a concise and intuitive way to specify a smoothly interpolated, real-valued function.

There have been attempts to provide an explanation of fuzzy logic in terms of probability theory. One idea is to view assertions such as "Nate is Tall" as discrete observations made concerning a continuous hidden variable, Nate's actual $Height$. The probability model specifies $P(\text{Observer says Nate is tall} \mid Height)$, perhaps using a **probit distribution** as described on page 503. A posterior distribution over Nate's height can then be calculated in the usual way, for example if the model is part of a hybrid Bayesian network. Such an approach is not truth-functional, of course. For example, the conditional distribution

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

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

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

RANDOM SETS

Both the hybrid Bayesian network approach and the random sets approach appear to capture aspects of fuzziness without introducing degrees of truth. Nonetheless, there remain many open issues concerning the proper representation of linguistic observations and continuous quantities—issues that have been neglected by most outside the fuzzy community.

14.8 SUMMARY

This chapter has described **Bayesian networks**, a well-developed representation for uncertain knowledge. Bayesian networks play a role roughly analogous to that of propositional logic for definite knowledge.

- A Bayesian network is a directed acyclic graph whose nodes correspond to random variables; each node has a conditional distribution for the node, given its parents.
- Bayesian networks provide a concise way to represent **conditional independence** relationships in the domain.
- A Bayesian network specifies a full joint distribution; each joint entry is defined as the product of the corresponding entries in the local conditional distributions. A Bayesian network is often exponentially smaller than the full joint distribution.
- Many conditional distributions can be represented compactly by canonical families of distributions. **Hybrid Bayesian networks**, which include both discrete and continuous variables, use a variety of canonical distributions.
- Inference in Bayesian networks means computing the probability distribution of a set of query variables, given a set of evidence variables. Exact inference algorithms, such as **variable elimination**, evaluate sums of products of conditional probabilities as efficiently as possible.
- In **polytrees** (singly connected networks), exact inference takes time linear in the size of the network. In the general case, the problem is intractable.
- Stochastic approximation techniques such as **likelihood weighting** and **Markov chain Monte Carlo** can give reasonable estimates of the true posterior probabilities in a network and can cope with much larger networks than can exact algorithms.
- Probability theory can be combined with representational ideas from first-order logic to produce very powerful systems for reasoning under uncertainty. **Relational probability models** (RPMs) include representational restrictions that guarantee a well-defined probability distribution that can be expressed as an equivalent Bayesian network.
- Various alternative systems for reasoning under uncertainty have been suggested. Generally speaking, **truth-functional** systems are not well suited for such reasoning.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

The use of networks to represent probabilistic information began early in the 20th century, with the work of Sewall Wright on the probabilistic analysis of genetic inheritance and animal growth factors (Wright, 1921, 1934). One of his networks appears on the cover of this book. I. J. Good (1961), in collaboration with Alan Turing, developed probabilistic representations and Bayesian inference methods that could be regarded as a forerunner of modern Bayesian

networks—although the paper is not often cited in this context.[10] The same paper is the original source for the noisy-OR model.

The **influence diagram** representation for decision problems, which incorporated a DAG representation for random variables, was used in decision analysis in the late 1970s (see Chapter 16), but only enumeration was used for evaluation. Judea Pearl developed the message-passing method for carrying out inference in tree networks (Pearl, 1982a) and polytree networks (Kim and Pearl, 1983) and explained the importance of constructing causal rather than diagnostic probability models, in contrast to the certainty-factor systems then in vogue. The first expert system using Bayesian networks was CONVINCE (Kim, 1983; Kim and Pearl, 1987). More recent systems include the MUNIN system for diagnosing neuromuscular disorders (Andersen *et al.*, 1989) and the PATHFINDER system for pathology (Heckerman, 1991). By far the most widely used Bayesian network systems have been the diagnosis-and-repair modules (e.g., the Printer Wizard) in Microsoft Windows (Breese and Heckerman, 1996) and the Office Assistant in Microsoft Office (Horvitz *et al.*, 1998).

Pearl (1986) developed a clustering algorithm for exact inference in general Bayesian networks, utilizing a conversion to a directed polytree of clusters in which message passing was used to achieve consistency over variables shared between clusters. A similar approach, developed by the statisticians David Spiegelhalter and Steffen Lauritzen (Spiegelhalter, 1986; Lauritzen and Spiegelhalter, 1988), is based on conversion to an undirected (Markov) network. This approach is implemented in the HUGIN system, an efficient and widely used tool for uncertain reasoning (Andersen *et al.*, 1989). Ross Shachter, working in the influence diagram community, developed an exact method based on goal-directed reduction of the network, using posterior-preserving transformations (Shachter, 1986).

The variable elimination method described in the chapter is closest in spirit to Shachter's method, from which emerged the symbolic probabilistic inference (SPI) algorithm (Shachter *et al.*, 1990). SPI attempts to optimize the evaluation of expression trees such as that shown in Figure 14.8. The algorithm we describe is closest to that developed by Zhang and Poole (1994, 1996). Criteria for pruning irrelevant variables were developed by Geiger *et al.* (1990) and by Lauritzen *et al.* (1990); the criterion we give is a simple special case of these. Rina Dechter (1999) shows how the variable elimination idea is essentially identical to **nonserial dynamic programming** (Bertele and Brioschi, 1972), an algorithmic approach that can be applied to solve a range of inference problems in Bayesian networks—for example, finding the **most probable explanation** for a set of observations. This connects Bayesian network algorithms to related methods for solving CSPs and gives a direct measure of the complexity of exact inference in terms of the **hypertree width** of the network.

NONSERIAL DYNAMIC PROGRAMMING

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

[10] I. J. Good was chief statistician for Turing's code-breaking team in World War II. In *2001: A Space Odyssey* (Clarke, 1968a), Good and Minsky are credited with making the breakthrough that led to the development of the HAL 9000 computer.

cHUGIN system (Olesen, 1993). The probit distribution was studied first by Finney (1947), who called it the sigmoid distribution. It has been used widely for modeling discrete choice phenomena and can be extended to handle more than two choices (Daganzo, 1979). Bishop (1995) gives a justification for the use of the logit distribution.

Cooper (1990) showed that the general problem of inference in unconstrained Bayesian networks is NP-hard, and Paul Dagum and Mike Luby (1993) showed the corresponding approximation problem to be NP-hard. Space complexity is also a serious problem in both clustering and variable elimination methods. The method of **cutset conditioning**, which was developed for CSPs in Chapter 5, avoids the construction of exponentially large tables. In a Bayesian network, a cutset is a set of nodes that, when instantiated, reduces the remaining nodes to a polytree that can be solved in linear time and space. The query is answered by summing over all the instantiations of the cutset, so the overall space requirement is still linear (Pearl, 1988). Darwiche (2001) describes a recursive conditioning algorithm that allows a complete range of space/time tradeoffs.

The development of fast approximation algorithms for Bayesian network inference is a very active area, with contributions from statistics, computer science, and physics. The rejection sampling method is a general technique that is long known to statisticians; it was first applied to Bayesian networks by Max Henrion (1988), who called it **logic sampling**. Likelihood weighting, which was developed by Fung and Chang (1989) and Shachter and Peot (1989), is an example of the well-known statistical method of **importance sampling**. A large-scale application of likelihood weighting to medical diagnosis appears in Shwe and Cooper (1991). Cheng and Druzdzel (2000) describe an adaptive version of likelihood weighting that works well even when the evidence has very low prior likelihood.

Markov chain Monte Carlo (MCMC) algorithms began with the Metropolis algorithm, due to Metropolis *et al.* (1953), which was also the source of the simulated annealing algorithm described in Chapter 4. The Gibbs sampler was devised by Geman and Geman (1984) for inference in undirected Markov networks. The application of MCMC to Bayesian networks is due to Pearl (1987). The papers collected by Gilks *et al.* (1996) cover a wide variety of applications of MCMC, several of which were developed in the well-known BUGS package (Gilks *et al.*, 1994).

VARIATIONAL APPROXIMATION

There are two very important families of approximation methods that we did not cover in the chapter. The first is the family of **variational approximation** methods, which can be used to simplify complex calculations of all kinds. The basic idea is to propose a reduced version of the original problem that is simple to work with, but that resembles the original problem as closely as possible. The reduced problem is described by some **variational parameters** $\boldsymbol{\lambda}$ that are adjusted to minimize a distance function D between the original and the reduced problem, often by solving the system of equations $\partial D/\partial \boldsymbol{\lambda} = 0$. In many cases, strict upper and lower bounds can be obtained. Variational methods have long been used in statistics (Rustagi, 1976). In statistical physics, the **mean field** method is a particular variational approximation in which the individual variables making up the model are assumed to be completely independent. This idea was applied to solve large undirected Markov networks (Peterson and Anderson, 1987; Parisi, 1988). Saul *et al.* (1996) developed the mathematical foundations for applying variational methods to Bayesian networks and obtained

VARIATIONAL PARAMETERS

MEAN FIELD

accurate lower-bound approximations for sigmoid networks with the use of mean-field methods. Jaakkola and Jordan (1996) extended the methodology to obtain both lower and upper bounds. Variational approaches are surveyed by Jordan *et al.* (1999).

A second important family of approximation algorithms is based on Pearl's polytree message-passing algorithm (1982a). This algorithm can be applied to general networks, as suggested by Pearl (1988). The results might be incorrect, or the algorithm might fail to terminate, but in many cases, the values obtained are close to the true values. Little attention was paid to this so-called **belief propagation** (or **loopy propagation**) approach until McEliece *et al.* (1998) observed that message passing in a multiply-connected Bayesian network was exactly the computation performed by the **turbo decoding** algorithm (Berrou *et al.*, 1993), which provided a major breakthrough in the design of efficient error-correcting codes. The implication is that loopy propagation is both fast and accurate on the very large and very highly connected networks used for decoding and might therefore be useful more generally. Murphy *et al.* (1999) present an empirical study of where it does work. Yedidia *et al.* (2001) make further connections between loopy propagation and ideas from statistical physics.

BELIEF PROPAGATION

TURBO DECODING

The connection between probability and first-order languages was first studied by Carnap (1950). Gaifman (1964) and Scott and Krauss (1966) defined a language in which probabilities could be associated with first-order sentences and for which models were probability measures on possible worlds. Within AI, this idea was developed for propositional logic by Nilsson (1986) and for first-order logic by Halpern (1990). The first extensive investigation of knowledge representation issues in such languages was carried out by Bacchus (1990), and the paper by Wellman *et al.* (1992) surveys early implementation approaches based on the construction of equivalent propositional Bayesian networks. More recently, researchers have come to understand the importance of *complete* knowledge bases—that is, knowledge bases that, like Bayesian networks, define a unique joint distribution over all possible worlds. Methods for doing this have been based on probabilistic versions of logic programming (Poole, 1993; Sato and Kameya, 1997) or semantic networks (Koller and Pfeffer, 1998). Relational probability models of the kind described in this chapter are investigated in depth by Pfeffer (2000). Pasula and Russell (2001) examine both issues of relational and identity uncertainty within RPMs and the use of MCMC inference.

As explained in Chapter 13, early probabilistic systems fell out of favor in the early 1970s, leaving a partial vacuum to be filled by alternative methods. Certainty factors were invented for use in the medical expert system MYCIN (Shortliffe, 1976), which was intended both as an engineering solution and as a model of human judgment under uncertainty. The collection *Rule-Based Expert Systems* (Buchanan and Shortliffe, 1984) provides a complete overview of MYCIN and its descendants (see also Stefik, 1995). David Heckerman (1986) showed that a slightly modified version of certainty factor calculations gives correct probabilistic results in some cases, but results in serious overcounting of evidence in other cases. The PROSPECTOR expert system (Duda *et al.*, 1979) used a rule-based approach in which the rules were justified by a (seldom tenable) global independence assumption.

Dempster–Shafer theory originates with a paper by Arthur Dempster (1968) proposing a generalization of probability to interval values and a combination rule for using them. Later work by Glenn Shafer (1976) led to the Dempster-Shafer theory's being viewed as a

competing approach to probability. Ruspini *et al.* (1992) analyze the relationship between the Dempster-Shafer theory and standard probability theory. Shenoy (1989) has proposed a method for decision making with Dempster–Shafer belief functions.

Fuzzy sets were developed by Lotfi Zadeh (1965) in response to the perceived difficulty of providing exact inputs to intelligent systems. The text by Zimmermann (2001) provides a thorough introduction to fuzzy set theory; papers on fuzzy applications are collected in Zimmermann (1999). As we mentioned in the text, fuzzy logic has often been perceived incorrectly as a direct competitor to probability theory, whereas in fact it addresses a different set of issues. **Possibility theory** (Zadeh, 1978) was introduced to handle uncertainty in fuzzy systems and has much in common with probability. Dubois and Prade (1994) provide a thorough survey of the connections between possibility theory and probability theory.

POSSIBILITY THEORY

The resurgence of probability depended mainly on the discovery of Bayesian networks as a method for representing and using conditional independence information. This resurgence did not come without a fight; Peter Cheeseman's (1985) pugnacious "In Defense of Probability," and his later article "An Inquiry into Computer Understanding" (Cheeseman, 1988, with commentaries) give something of the flavor of the debate. One of the principal objections of the logicists was that the numerical calculations that probability theory was thought to require were not apparent to introspection and presumed an unrealistic level of precision in our uncertain knowledge. The development of **qualitative probabilistic networks** (Wellman, 1990a) provided a purely qualitative abstraction of Bayesian networks, using the notion of positive and negative influences between variables. Wellman shows that in many cases such information is sufficient for optimal decision making without the need for the precise specification of probability values. Work by Adnan Darwiche and Matt Ginsberg (1992) extracts the basic properties of conditioning and evidence combination from probability theory and shows that they can also be applied in logical and default reasoning.

The heart disease treatment system described in the chapter is due to Lucas (1996). Other fielded applications of Bayesian networks include the work at Microsoft on inferring computer user goals from their actions (Horvitz *et al.*, 1998) and on filtering junk email (Sahami *et al.*, 1998), the Electric Power Research Institute's work on monitoring power generators (Morjaria *et al.*, 1995), and NASA's work on displaying time-critical information at Mission Control in Houston (Horvitz and Barry, 1995).

Some important early papers on uncertain reasoning methods in AI are collected in the anthologies *Readings in Uncertain Reasoning* (Shafer and Pearl, 1990) and *Uncertainty in Artificial Intelligence* (Kanal and Lemmer, 1986). The most important single publication in the growth of Bayesian networks was undoubtedly the text *Probabilistic Reasoning in Intelligent Systems* (Pearl, 1988). Several excellent texts, including Lauritzen (1996), Jensen (2001) and Jordan (2003), contain more recent material. New research on probabilistic reasoning appears both in mainstream AI journals such as *Artificial Intelligence* and the *Journal of AI Research*, and in more specialized journals, such as the *International Journal of Approximate Reasoning*. Many papers on **graphical models**, which include Bayesian networks, appear in statistical journals. The proceedings of the conferences on Uncertainty in Artificial Intelligence (UAI), Neural Information Processing Systems (NIPS), and Artificial Intelligence and Statistics (AISTATS) are excellent sources for current research.

EXERCISES

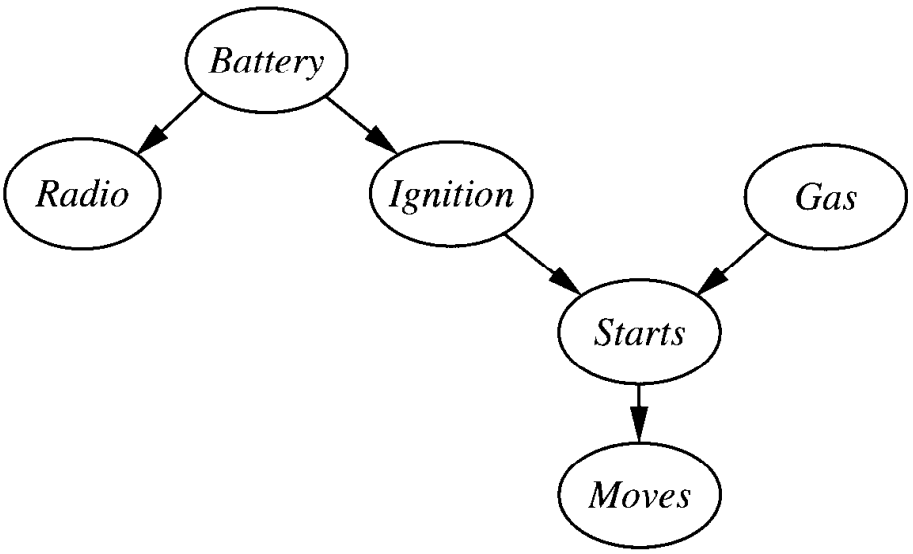


Figure 14.18 A Bayesian network describing some features of a car's electrical system and engine. Each variable is Boolean, and the *true* value indicates that the corresponding aspect of the vehicle is in working order.

14.1 Consider the network for car diagnosis shown in Figure 14.18.

a. Extend the network with the Boolean variables *IcyWeather* and *StarterMotor*.

b. Give reasonable conditional probability tables for all the nodes.

c. How many independent values are contained in the joint probability distribution for eight Boolean nodes, assuming that no conditional independence relations are known to hold among them?

d. How many independent probability values do your network tables contain?

e. The conditional distribution for *Starts* could be described as a **noisy-AND** distribution. Define this family in general and relate it to the noisy-OR distribution.

14.2 In your local nuclear power station, there is an alarm that senses when a temperature gauge exceeds a given threshold. The gauge measures the temperature of the core. Consider the Boolean variables A (alarm sounds), F_A (alarm is faulty), and F_G (gauge is faulty) and the multivalued nodes G (gauge reading) and T (actual core temperature).

a. Draw a Bayesian network for this domain, given that the gauge is more likely to fail when the core temperature gets too high.

b. Is your network a polytree?

c. Suppose there are just two possible actual and measured temperatures, normal and high; the probability that the gauge gives the correct temperature is x when it is working, but y when it is faulty. Give the conditional probability table associated with G.

d. Suppose the alarm works correctly unless it is faulty, in which case it never sounds. Give the conditional probability table associated with A.

e. Suppose the alarm and gauge are working and the alarm sounds. Calculate an expression for the probability that the temperature of the core is too high, in terms of the various conditional probabilities in the network.

14.3 Two astronomers in different parts of the world make measurements M_1 and M_2 of the number of stars N in some small region of the sky, using their telescopes. Normally, there is a small possibility e of error by up to one star in each direction. Each telescope can also (with a much smaller probability f) be badly out of focus (events F_1 and F_2), in which case the scientist will undercount by three or more stars (or, if N is less than 3, fail to detect any stars at all). Consider the three networks shown in Figure 14.19.

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

b. Which is the best network? Explain.

c. Write out a conditional distribution for $\mathbf{P}(M_1|N)$, for the case where $N \in \{1, 2, 3\}$ and $M_1 \in \{0, 1, 2, 3, 4\}$. Each entry in the conditional distribution should be expressed as a function of the parameters e and/or f.

d. Suppose $M_1 = 1$ and $M_2 = 3$. What are the *possible* numbers of stars if we assume no prior constraint on the values of N?

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

14.4 Consider the network shown in Figure 14.19(ii), and assume that the two telescopes work identically. $N \in \{1, 2, 3\}$ and $M_1, M_2 \in \{0, 1, 2, 3, 4\}$, with the symbolic CPTs as described in Exercise 14.3. Using the enumeration algorithm, calculate the probability distribution $\mathbf{P}(N|M_1 = 2, M_2 = 2)$.

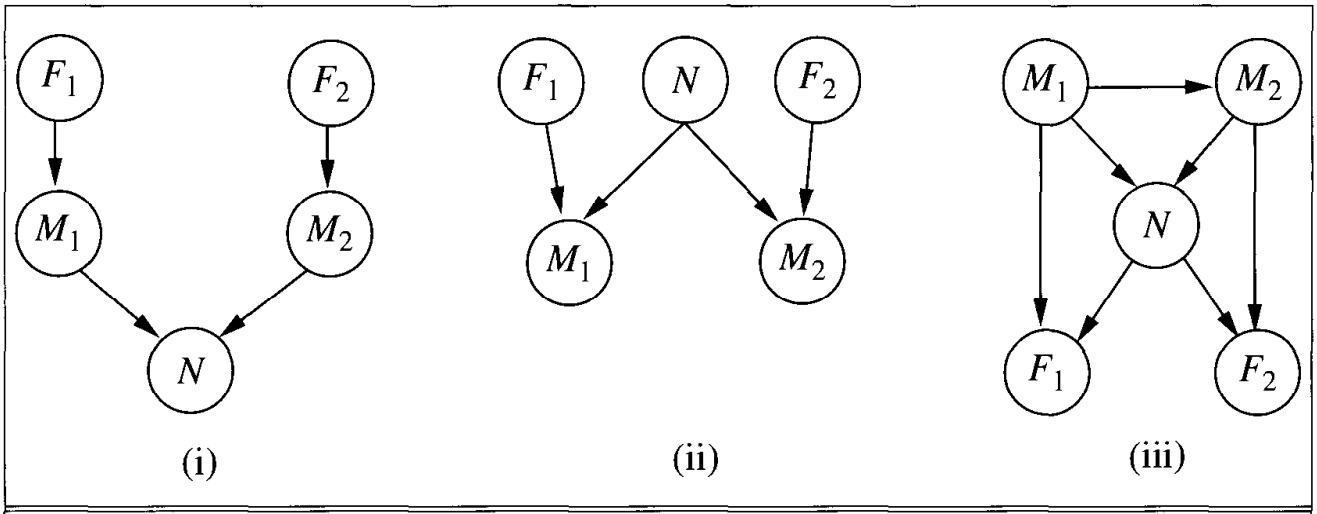


Figure 14.19 Three possible networks for the telescope problem.

14.5 Consider the family of linear Gaussian networks, as illustrated on page 502.

a. In a two-variable network, let X_1 be the parent of X_2, let X_1 have a Gaussian prior, and let $\mathbf{P}(X_2|X_1)$ be a linear Gaussian distribution. Show that the joint distribution $P(X_1, X_2)$ is a multivariate Gaussian, and calculate its covariance matrix.

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

14.6 The probit distribution defined on page 503 describes the probability distribution for a Boolean child, given a single continuous parent.

a. How might the definition be extended to cover multiple continuous parents?

b. How might it be extended to handle a *multivalued* child variable? Consider both cases where the child's values are ordered (as in selecting a gear while driving, depending on speed, slope, desired acceleration, etc.) and cases where they are unordered (as in selecting bus, train, or car to get to work). [*Hint*: Consider ways to divide the possible values into two sets, to mimic a Boolean variable.]

14.7 This exercise is concerned with the variable elimination algorithm in Figure 14.10.

a. Section 14.4 applies variable elimination to the query

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

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

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

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

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

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

a. Prove that any 3-SAT problem can be reduced to exact inference in a Bayesian network constructed to represent the particular problem and hence that exact inference is NP-hard. [*Hint*: Consider a network with one variable for each proposition symbol, one for each clause, and one for the conjunction of clauses.]

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

14.9 Consider the problem of generating a random sample from a specified distribution on a single variable. You can assume that a random number generator is available that returns a random number uniformly distributed between 0 and 1.

a. Let X be a discrete variable with $P(X = x_i) = p_i$ for $i \in \{1, \ldots, k\}$. The **cumulative distribution** of X gives the probability that $X \in \{x_1, \ldots, x_j\}$ for each possible j. Ex-

CUMULATIVE DISTRIBUTION

plain how to calculate the cumulative distribution in $O(k)$ time and how to generate a single sample of X from it. Can the latter be done in less than $O(k)$ time?

b. Now suppose we want to generate N samples of X, where $N \gg k$. Explain how to do this with an expected runtime per sample that is *constant* (i.e., independent of k).

c. Now consider a continuous-valued variable with a parametrized distribution (e.g., Gaussian). How can samples be generated from such a distribution?

d. Suppose you want to query a continuous-valued variable and you are using a sampling algorithm such as LIKELIHOODWEIGHTING to do the inference. How would you have to modify the query-answering process?

14.10 The **Markov blanket** of a variable is defined on page 499.

a. Prove that a variable is independent of all other variables in the network, given its Markov blanket.

b. Derive Equation (14.11).

14.11 Consider the query $\mathbf{P}(Rain|Sprinkler = true, WetGrass = true)$ in Figure 14.11(a) and how MCMC can answer it.

a. How many states does the Markov chain have?

b. Calculate the **transition matrix Q** containing $q(\mathbf{y} \rightarrow \mathbf{y}')$ for all $\mathbf{y}, \mathbf{y}'$.

c. What does $\mathbf{Q}^2$, the square of the transition matrix, represent?

d. What about $\mathbf{Q}^n$ as $n \rightarrow \infty$?

e. Explain how to do probabilistic inference in Bayesian networks, assuming that $\mathbf{Q}^n$ is available. Is this a practical way to do inference?

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

a. Construct a relational probability model to describe this domain, and suggest numerical values for all the necessary probability distributions.

b. Construct the equivalent Bayesian network.

c. Suppose that in the first two matches A beats B and draws with C. Using an exact inference algorithm of your choice, compute the posterior distribution for the outcome of the third match.

d. Suppose there are n teams in the league and we have the results for all but the last match. How does the complexity of predicting the last game vary with n?

e. Investigate the application of MCMC to this problem. How quickly does it converge in practice and how well does it scale?

18 LEARNING FROM OBSERVATIONS

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

The idea behind learning is that percepts should be used not only for acting, but also for improving the agent's ability to act in the future. Learning takes place as the agent observes its interactions with the world and its own decision-making processes. Learning can range from trivial memorization of experience, as exhibited by the wumpus-world agent in Chapter 10, to the creation of entire scientific theories, as exhibited by Albert Einstein. This chapter describes **inductive learning** from observations. In particular, we describe how to learn simple theories in propositional logic. We also give a theoretical analysis that explains why inductive learning works.

18.1 FORMS OF LEARNING

In Chapter 2, we saw that a learning agent can be thought of as containing a **performance element** that decides what actions to take and a **learning element** that modifies the performance element so that it makes better decisions. (See Figure 2.15.) Machine learning researchers have come up with a large variety of learning elements. To understand them, it will help to see how their design is affected by the context in which they will operate. The design of a learning element is affected by three major issues:

- Which *components* of the performance element are to be learned.
- What *feedback* is available to learn these components.
- What *representation* is used for the components.

We now analyze each of these issues in turn. We have seen that there are many ways to build the performance element of an agent. Chapter 2 described several agent designs (Figures 2.9, 2.11, 2.13, and 2.14). The components of these agents include the following:

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 from appropriate feedback. Consider, for example, an agent training to become a taxi driver. Every time the instructor shouts "Brake!" the agent can learn a condition–action rule for when to brake (component 1). 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).

The *type of feedback* available for learning is usually the most important factor in determining the nature of the learning problem that the agent faces. The field of machine learning usually distinguishes three cases: **supervised**, **unsupervised**, and **reinforcement** learning.

SUPERVISED LEARNING

The problem of **supervised learning** involves learning a *function* from examples of its inputs and outputs. Cases (1), (2), and (3) are all instances of supervised learning problems. In (1), the agent learns condition–action rule for braking—this is a function from states to a Boolean output (to brake or not to brake), In (2), the agent learns a function from images to a Boolean output (whether the image contains a bus). In (3), the theory of braking is a function from states and braking actions to, say, stopping distance in feet. Notice that in cases (1) and (2), a teacher provided the correct output value of the examples; in the third, the output value was available directly from the agent's percepts. For fully observable environments, it will always be the case that an agent can observe the effects of its actions and hence can use supervised learning methods to learn to predict them. For partially observable environments, the problem is more difficult, because the immediate effects might be invisible.

UNSUPERVISED LEARNING

The problem of **unsupervised learning** involves learning patterns in the input when no specific output values are supplied. For example, a taxi agent might gradually develop a concept of "good traffic days" and "bad traffic days" without ever being given labelled examples of each. A purely unsupervised learning agent cannot learn what to do, because it has no information as to what constitutes a correct action or a desirable state. We will study unsupervised learning primarily in the context of probabilistic reasoning systems (Chapter 20.

REINFORCEMENT LEARNING

The problem of **reinforcement learning**, which we cover in Chapter 21, is the most general of the three categories. Rather than being told what to do by a teacher, a reinforcement learning agent must learn from **reinforcement**.[1] For example, the lack of a tip at the end of the journey (or a hefty bill for rear-ending the car in front) give the agent some indication that its behavior is undesirable. Reinforcement learning typically includes the subproblem of learning how the environment works.

REINFORCEMENT

The *representation of the learned information* also plays a very important role in determining how the learning algorithm must work. Any of the components of an agent can be represented using any of the representation schemes in this book. We have seen sev-

[1] The term **reward** as used in Chapter 17 is a synonym for **reinforcement**.

eral examples: linear weighted polynomials for utility functions in game-playing programs; propositional and first-order logical sentences for all of the components in a logical agent; and probabilistic descriptions such as Bayesian networks for the inferential components of a decision-theoretic agent. Effective learning algorithms have been devised for all of these. This chapter will cover methods for propositional logic, Chapter 19 describes methods for first-order logic, and Chapter 20 covers methods for Bayesian networks and for neural networks (which include linear polynomials as a special case).

The last major factor in the design of learning systems is the *availability of prior knowledge*. The majority of learning research in AI, computer science, and psychology has studied the case in which the agent begins with no knowledge at all about what it is trying to learn. It has access only to the examples presented by its experience. Although this is an important special case, it is by no means the general case. Most human learning takes place in the context of a good deal of background knowledge. Some psychologists and linguists claim that even newborn babies exhibit knowledge of the world. Whatever the truth of this claim, there is no doubt that prior knowledge can help enormously in learning. A physicist examining a stack of bubble-chamber photographs might be able to induce a theory positing the existence of a new particle of a certain mass and charge; but an art critic examining the same stack might learn nothing more than that the "artist" must be some sort of abstract expressionist. Chapter 19 shows several ways in which learning is helped by the use of existing knowledge; it also shows how knowledge can be *compiled* in order to speed up decision making. Chapter 20 shows how prior knowledge helps in the learning of probabilistic theories.

18.2 Inductive Learning

An algorithm for deterministic supervised learning is given as input the correct value of the unknown function for particular inputs and must try to recover the unknown function or something close to it. More formally, we say that an **example** is a pair $(x, f(x))$, where x is the input and $f(x)$ is the output of the function applied to x. The task of **pure inductive inference** (or **induction**) is this:

EXAMPLE

PURE INDUCTIVE INFERENCE

> Given a collection of examples of f, return a function h that approximates f.

The function h is called a **hypothesis**. The reason that learning is difficult, from a conceptual point of view, is that it is not easy to tell whether any particular h is a good approximation of f. A good hypothesis will **generalize** well—that is, will predict unseen examples correctly. This is the fundamental **problem of induction**. The problem has been studied for centuries; Section 18.5 provides a partial solution.

HYPOTHESIS

GENERALIZATION

PROBLEM OF INDUCTION

Figure 18.1 shows a familiar example: fitting a function of a single variable to some data points. The examples are $(x, f(x))$ pairs, where both x and $f(x)$ are real numbers. We choose the **hypothesis space H**—the set of hypotheses we will consider—to be the set of polynomials of degree at most k, such as $3x^2 + 2$, $x^{17} - 4x^3$, and so on. Figure 18.1(a) shows some data with an exact fit by a straight line (a polynomial of degree 1). The line is called a **consistent** hypothesis because it agrees with all the data. Figure 18.1(b) shows a

HYPOTHESIS SPACE

CONSISTENT

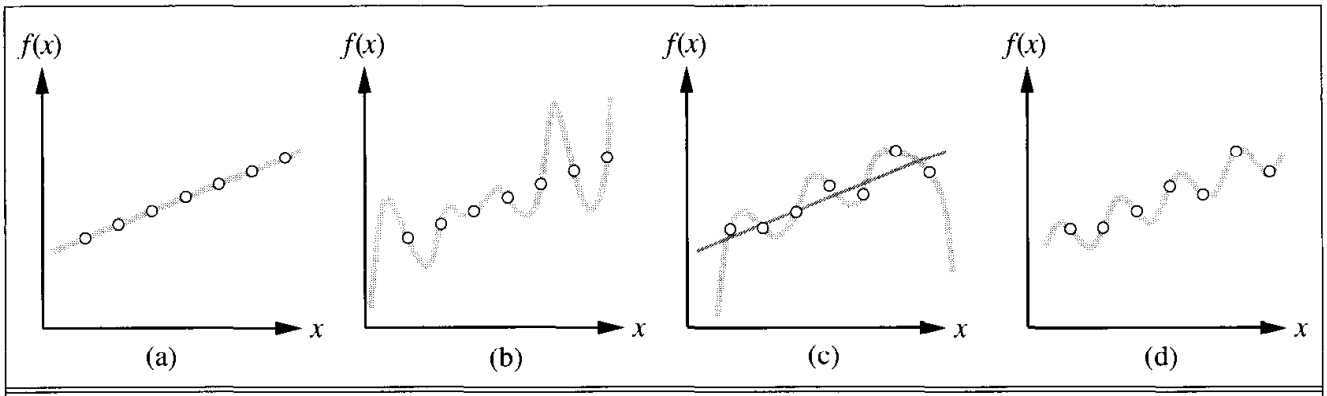


Figure 18.1 (a) Example $(x, f(x))$ pairs and a consistent, linear hypothesis. (b) A consistent, degree-7 polynomial hypothesis for the same data set. (c) A different data set that admits an exact degree-6 polynomial fit or an approximate linear fit. (d) A simple, exact sinusoidal fit to the same data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

high-degree polynomial that is also consistent with the same data. This illustrates the first issue in inductive learning: *how do we choose from among multiple consistent hypotheses?* One answer is **Ockham's**[2] **razor**: prefer the *simplest* hypothesis consistent with the data.

OCKHAM'S RAZOR

Intuitively, this makes sense, because hypotheses that are no simpler than the data themselves are failing to extract any *pattern* from the data. Defining simplicity is not easy, but it seems reasonable to say that a degree-1 polynomial is simpler than a degree-12 polynomial.

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 (with 7 parameters) for an exact fit. There are just 7 data points, so the polynomial has as many parameters as there are data points; thus, it does not seem to be finding any pattern in the data and we do not expect it to generalize well. It might be better to fit a simple straight line that is not exactly consistent but might make reasonable predictions. This amounts to accepting the possibility that the true function is not deterministic (or, roughly equivalently, that the true inputs are not fully observed).

For nondeterministic functions, there is an inevitable tradeoff between the complexity of the hypothesis and the degree of fit to the data. Chapter 20 explains how to make this tradeoff using probability theory.

One should keep in mind that the possibility or impossibility of finding a simple, consistent hypothesis depends strongly on the hypothesis space chosen. Figure 18.1(d) shows that the data in (c) can be fit exactly by a simple function of the form $ax + b + c\sin x$. This example shows the importance of the choice of hypothesis space. A hypothesis space consisting of polynomials of finite degree cannot represent sinusoidal functions accurately, so a learner using that hypothesis space will not be able to learn from sinusoidal data. We say that a learning problem is **realizable** if the hypothesis space contains the true function; otherwise, it is **unrealizable**. Unfortunately, we cannot always tell whether a given learning problem is realizable, because the true function is not known. One way to get around this barrier is to use *prior knowledge* to derive a hypothesis space in which we know the true function must lie. This topic is covered in Chapter 19.

REALIZABLE

UNREALIZABLE

[2] Named after the 14th-century English philosopher, William of Ockham. The name is often misspelled as "Occam," perhaps from the French rendering, "Guillaume d'Occam."

Another approach is to use the largest possible hypothesis space. For example, why not let **H** be the class of all Turing machines? After all, every computable function can be represented by some Turing machine, and that is the best we can do. The 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 simple, consistent hypotheses within that space.* For example, fitting straight lines to data is very easy; fitting high-degree polynomials is harder; and fitting Turing machines is very hard indeed because determining whether a given Turing machine is consistent with the data is not even decidable in general. A second reason to prefer simple hypothesis spaces is that the resulting hypotheses may be simpler to use—that is, it is faster to compute $h(x)$ when h is a linear function than when it is an arbitrary Turing machine program.

For these reasons, most work on learning has focused on relatively simple representations. In this chapter, we concentrate on propositional logic and related languages. Chapter 19 looks at learning theories in first-order logic. We will see that the expressiveness–complexity tradeoff is not as simple as it first seems: it is often the case, as we saw in Chapter 8, that an expressive language makes it possible for a *simple* theory to fit the data, whereas restricting the expressiveness of the language means that any consistent theory 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. In such cases, it should be possible to learn much faster by using the more expressive language.

18.3 LEARNING DECISION TREES

Decision tree induction is one of the simplest, and yet most successful forms of learning algorithm. It serves as a good introduction to the area of inductive learning, and is easy to implement. We first describe the performance element, and then show how to learn it. Along the way, we will introduce ideas that appear in all areas of inductive learning.

Decision trees as performance elements

DECISION TREE
ATTRIBUTES
CLASSIFICATION
REGRESSION
POSITIVE
NEGATIVE

A **decision tree** takes as input an object or situation described by a set of **attributes** and returns a "decision"—the predicted output value for the input. The input attributes can be discrete or continuous. For now, we assume discrete inputs. The output value can also be discrete or continuous; learning a discrete-valued function is called **classification** learning; learning a continuous function is called **regression**. We will concentrate on *Boolean* classification, wherein each example is classified as true (**positive**) or false (**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 properties, and the branches from the node are labeled with the possible values of the test. Each leaf node in the tree specifies the value to be returned if that leaf is reached. The decision tree representation seems to be very 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

A somewhat simpler example is provided by the problem of whether to wait for a table at a restaurant. The aim here is to learn a definition for the **goal predicate** *WillWait*. In setting this up as a learning problem, we first have to state what attributes are available to describe examples in the domain. In Chapter 19, we will see how to automate this task; for now, let's suppose we decide on the following list of attributes:

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, >60).

The decision tree usually used by one of us (SR) for this domain is shown in Figure 18.2. Notice that the tree does not use the *Price* and *Type* attributes, in effect considering them to be irrelevant. 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).

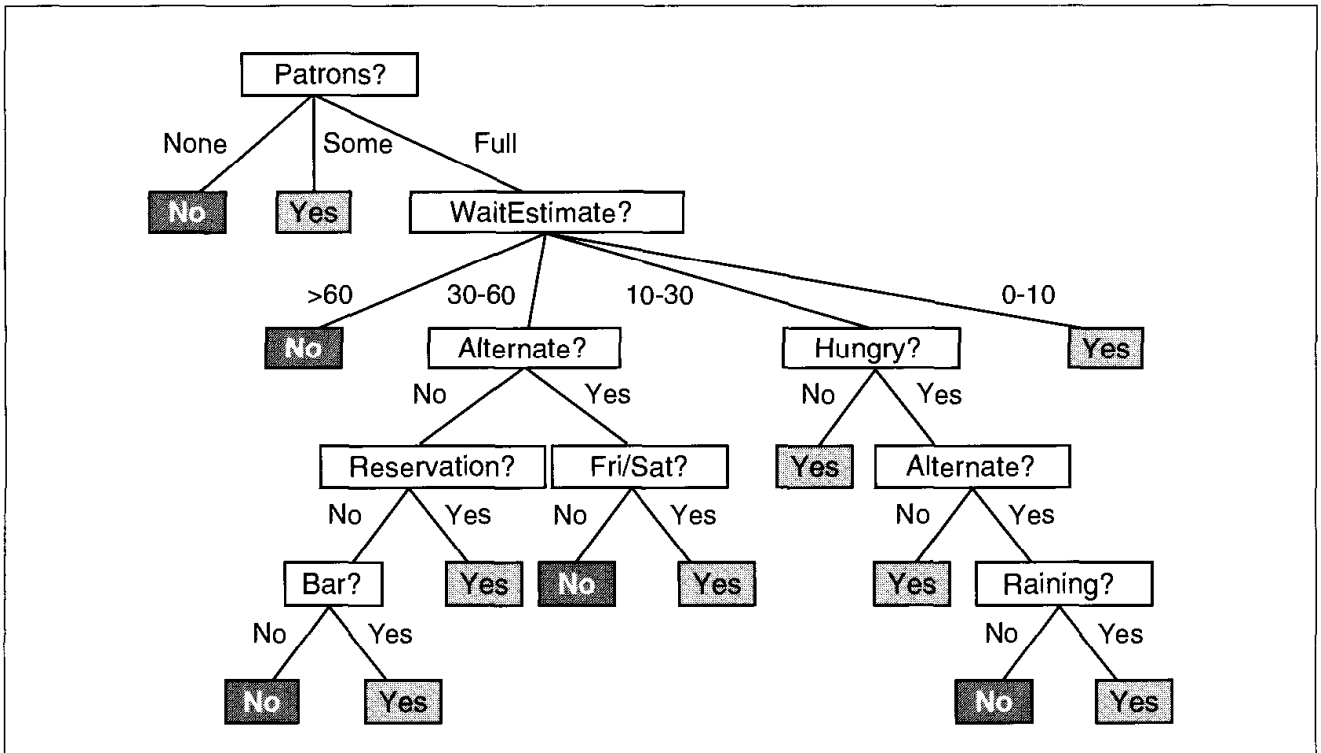


Figure 18.2 A decision tree for deciding whether to wait for a table.

Expressiveness of decision trees

Logically speaking, any particular decision tree hypothesis for the *WillWait* goal predicate can be seen as an assertion of the form

$$\forall s \;\; WillWait(s) \;\Leftrightarrow\; (P_1(s) \vee P_2(s) \vee \cdots \vee P_n(s)) \,,$$

where each condition $P_i(s)$ is a conjunction of tests corresponding to a path from the root of the tree to a leaf with a positive outcome. Although this looks like a first-order sentence, it is, in a sense, propositional, because it contains just one variable and all the predicates are unary. The decision tree is really describing a relationship between *WillWait* and some logical combination of attribute values. We cannot use decision trees to represent tests that refer to two or more different objects—for example,

$$\exists r_2 \;\; Nearby(r_2, r) \wedge Price(r, p) \wedge Price(r_2, p_2) \wedge Cheaper(p_2, p)$$

(is there a cheaper restaurant nearby?). Obviously, we could add another Boolean attribute with the name *CheaperRestaurantNearby*, but it is intractable to add *all* such attributes. Chapter 19 will delve further into the problem of learning in first-order logic proper.

Decision trees *are* fully expressive within the class of propositional languages; that is, any Boolean function can be written as a decision tree. This can be done trivially by having each row in the truth table for the function correspond to a path in the tree. This would yield an exponentially large decision tree representation because the truth table has exponentially many rows. Clearly, decision trees can represent many functions with much smaller trees.

PARITY FUNCTION

For some kinds of functions, however, this is a real problem. For example, if the function is the **parity function**, which returns 1 if and only if an even number of inputs are 1, then an exponentially large decision tree will be needed. It is also difficult to use a decision tree to represent a **majority function**, which returns 1 if more than half of its inputs are 1.

MAJORITY FUNCTION

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 very 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. The truth table has 2^n rows, because each input case is described by n attributes. We can consider the "answer" column of the table as a 2^n-bit number that defines the function. No matter what representation we use for functions, some of the functions (almost all of them, in fact) are going to require at least that many bits to represent.

If it takes 2^n bits to define the function, then there are 2^{2^n} different functions on n attributes. This is a scary number. For example, with just six Boolean attributes, there are $2^{2^6} = 18,446,744,073,709,551,616$ different functions to choose from. We will need some ingenious algorithms to find consistent hypotheses in such a large space.

Inducing decision trees from examples

An example for a Boolean decision tree consists of a vector of input attributes, X, and a single Boolean output value y. A set of examples $(X_1, y_1) \ldots, (X_{12}, y_{12})$ is shown in Figure 18.3.

Example	Attributes										Goal
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
X_1	*Yes*	*No*	*No*	*Yes*	*Some*	*$$$*	*No*	*Yes*	*French*	*0–10*	*Yes*
X_2	*Yes*	*No*	*No*	*Yes*	*Full*	*$*	*No*	*No*	*Thai*	*30–60*	*No*
X_3	*No*	*Yes*	*No*	*No*	*Some*	*$*	*No*	*No*	*Burger*	*0–10*	*Yes*
X_4	*Yes*	*No*	*Yes*	*Yes*	*Full*	*$*	*Yes*	*No*	*Thai*	*10–30*	*Yes*
X_5	*Yes*	*No*	*Yes*	*No*	*Full*	*$$$*	*No*	*Yes*	*French*	*>60*	*No*
X_6	*No*	*Yes*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Italian*	*0–10*	*Yes*
X_7	*No*	*Yes*	*No*	*No*	*None*	*$*	*Yes*	*No*	*Burger*	*0–10*	*No*
X_8	*No*	*No*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Thai*	*0–10*	*Yes*
X_9	*No*	*Yes*	*Yes*	*No*	*Full*	*$*	*Yes*	*No*	*Burger*	*>60*	*No*
X_{10}	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$$$*	*No*	*Yes*	*Italian*	*10–30*	*No*
X_{11}	*No*	*No*	*No*	*No*	*None*	*$*	*No*	*No*	*Thai*	*0–10*	*No*
X_{12}	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$*	*No*	*No*	*Burger*	*30–60*	*Yes*

Figure 18.3 Examples for the restaurant domain.

The positive examples are the ones in which the goal *WillWait* is true $(X_1, X_3, \ldots)$; the negative examples are the ones in which it is false $(X_2, X_5, \ldots)$. The complete set of examples is called the **training set**.

TRAINING SET

The problem of finding a decision tree that agrees with the training set might seem difficult, but in fact there is a trivial solution. We could simply construct a decision tree that has one path to a leaf for each example, where the path tests each attribute in turn and follows the value for the example and the leaf has the classification of the example. When given the same example again,[3] the decision tree will come up with the right classification. Unfortunately, it will not have much to say about any other cases!

The problem with this trivial tree is that it just memorizes the observations. It does not extract any pattern from the examples, so we cannot expect it to be able to extrapolate to examples it has not seen. Applying Ockham's razor, we should find instead the *smallest* decision tree that is consistent with the examples. Unfortunately, for any reasonable definition of "smallest," finding the smallest tree is an intractable problem. With some simple heuristics, however, we can do a good job of finding a "smallish" one. The basic idea behind the DECISION-TREE-LEARNING algorithm is to test the most important attribute first. By "most important," 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 small.

Figure 18.4 shows how the algorithm gets started. We are given 12 training examples, which we classify into positive and negative sets. We then decide which attribute to use as the first test in the tree. 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 and negative examples. On the other hand, in Figure 18.4(b) we see that *Patrons* is a fairly important

[3] The same example *or an example with the same description*—this distinction is very important, and we will return to it in Chapter 19.

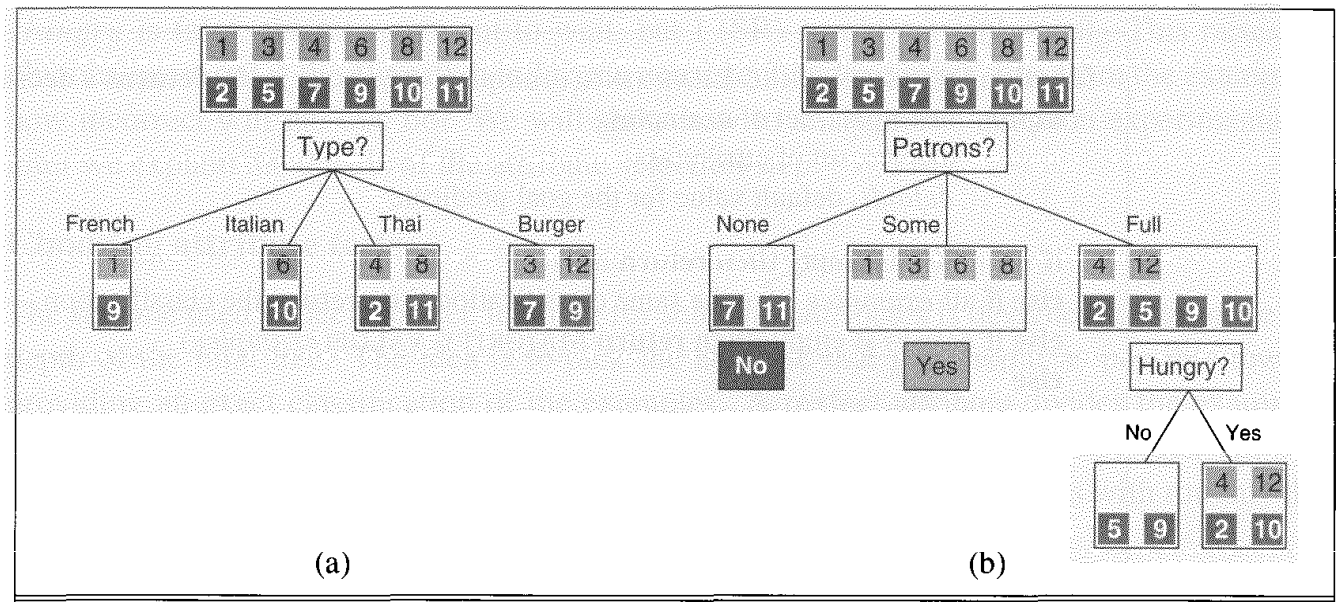


Figure 18.4 Splitting the examples by testing on attributes. (a) Splitting on *Type* brings us no nearer to distinguishing between positive and negative examples. (b) Splitting on *Patrons* does a good job of separating positive and negative examples. After splitting on *Patrons*, *Hungry* is a fairly good second test.

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 fewer attribute. There are four cases to consider for these recursive problems:

1. 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.
2. If all the remaining examples are positive (or all negative), then we are done: we can answer *Yes* or *No*. Figure 18.4(b) shows examples of this in the *None* and *Some* cases.
3. If there are no examples left, it means that no such example has been observed, and we return a default value calculated from the majority classification at the node's parent.
4. If there are no attributes left, but both positive and negative examples, we have a problem. It means that these examples have exactly the same description, but different classifications. This happens when some of the data are incorrect; we say there is **noise** in the data. It also happens either when the attributes do not give enough information to describe the situation fully, or when the domain is truly nondeterministic. One simple way out of the problem is to use a majority vote.

NOISE

The DECISION-TREE-LEARNING algorithm is shown in Figure 18.5. The details of the method for CHOOSE-ATTRIBUTE are given in the next subsection.

The final tree produced by the algorithm applied to the 12-example data set is shown in Figure 18.6. The tree is clearly different from the original tree shown in Figure 18.2, despite the fact that the data were actually generated from an agent using the original tree. One might conclude that the learning algorithm is not doing a very good job of learning the correct

```
function DECISION-TREE-LEARNING(examples, attribs, default) returns a decision tree
  inputs: examples, set of examples
          attribs, set of attributes
          default, default value for the goal predicate

  if examples is empty then return default
  else if all examples have the same classification then return the classification
  else if attribs is empty then return MAJORITY-VALUE(examples)
  else
      best ← CHOOSE-ATTRIBUTE(attribs, examples)
      tree ← a new decision tree with root test best
      m ← MAJORITY-VALUE(examples)
      for each value v_i of best do
          examples_i ← {elements of examples with best = v_i}
          subtree ← DECISION-TREE-LEARNING(examples_i, attribs − best, m)
          add a branch to tree with label v_i and subtree subtree
      return tree
```

Figure 18.5 The decision tree learning algorithm.

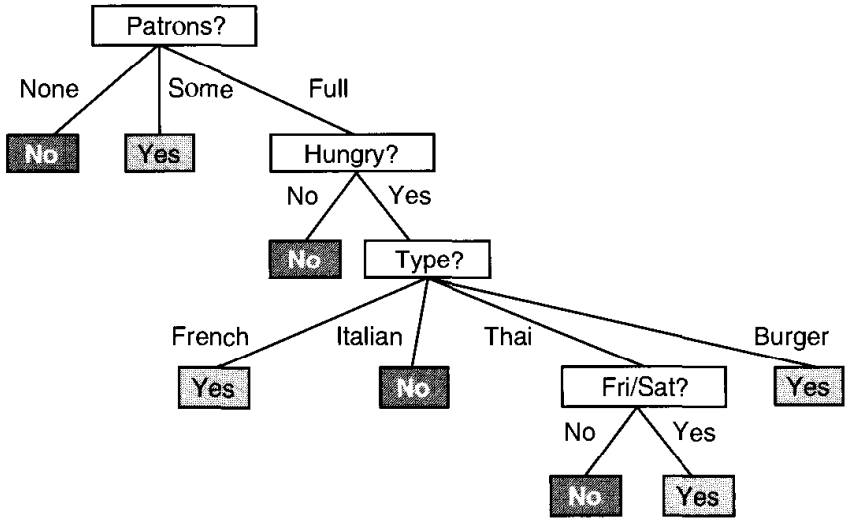


Figure 18.6 The decision tree induced from the 12-example training set.

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 agrees 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.

Of course, if we were to gather more examples, we might induce a tree more similar to the original. The tree in Figure 18.6 is bound to make a mistake; for example, it has never seen a case where the wait is 0–10 minutes but the restaurant is full. For a case where

Hungry is false, the tree says not to wait, but I (SR) would certainly wait. This raises an obvious question: if the algorithm induces a consistent, but incorrect, tree from the examples, how incorrect will the tree be? We will show how to analyze this question experimentally, after we explain the details of the attribute selection step.

Choosing attribute tests

The scheme used in decision tree learning for selecting attributes is designed to 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 that are all positive or all negative. 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.

All we need, then, is a formal measure of "fairly good" and "really useless" and we can implement the CHOOSE-ATTRIBUTE function of Figure 18.5. The measure should have its maximum value when the attribute is perfect and its minimum value when the attribute is of no use at all. One suitable measure is the expected amount of **information** provided by the attribute, where we use the term in the mathematical sense first defined in Shannon and Weaver (1949). To understand the notion of information, think about it as providing the answer to a question—for example, whether a coin will come up heads. The amount of information contained in the answer depends on one's prior knowledge. The less you know, the more information is provided. Information theory measures information content in **bits**. One bit of information is enough to answer a yes/no question about which one has no idea, such as the flip of a fair coin. In general, if the possible answers v_i have probabilities $P(v_i)$, then the information content I of the actual answer is given by

INFORMATION

$$I(P(v_1), \ldots, P(v_n)) = \sum_{i=1}^{n} -P(v_i) \log_2 P(v_i) \; .$$

To check this equation, for the tossing of a fair coin, we get

$$I\left(\tfrac{1}{2}, \tfrac{1}{2}\right) = -\tfrac{1}{2}\log_2 \tfrac{1}{2} - \tfrac{1}{2}\log_2 \tfrac{1}{2} = 1 \text{ bit.}$$

If the coin is loaded to give 99% heads, we get $I\,(1/100, 99/100) = 0.08$ bits, and as the probability of heads goes to 1, the information of the actual answer goes to 0.

For decision tree learning, the question that needs answering is; for a given example, what is the correct classification? A correct decision tree will answer this question. An estimate of the probabilities of the possible answers before any of the attributes have been tested is given by the proportions of positive and negative examples in the training set. Suppose the training set contains p positive examples and n negative examples. Then an estimate of the information contained in a correct answer is

$$I\left(\tfrac{p}{p+n}, \tfrac{n}{p+n}\right) = -\tfrac{p}{p+n}\log_2 \tfrac{p}{p+n} - \tfrac{n}{p+n}\log_2 \tfrac{n}{p+n} \; .$$

The restaurant training set in Figure 18.3 has $p = n = 6$, so we need 1 bit of information.

Now a test on a single attribute A will not usually tell us this much information, but it will give us some of it. We can measure exactly how much by looking at how much

information we still need *after* the attribute test. Any attribute A divides the training set E into subsets $E_1, \ldots, E_v$ according to their values for A, where A can have v distinct values. Each subset E_i has p_i positive examples and n_i negative examples, so if we go along that branch, we will need an additional $I(p_i/(p_i+n_i), n_i/(p_i+n_i))$ bits of information to answer the question. A randomly chosen example from the training set has the ith value for the attribute with probability $(p_i+n_i)/(p+n)$, so on average, after testing attribute A, we will need

$$Remainder(A) = \sum_{i=1}^{v} \frac{p_i+n_i}{p+n} I\left(\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i}\right)$$

INFORMATION GAIN

bits of information to classify the example. The **information gain** from the attribute test is the difference between the original information requirement and the new requirement:

$$Gain(A) = I\left(\frac{p}{p+n}, \frac{n}{p+n}\right) - Remainder(A) .$$

The heuristic used in the CHOOSE-ATTRIBUTE function is just to choose the attribute with the largest gain. Returning to the attributes considered in Figure 18.4, we have

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

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

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

Assessing the performance of the learning algorithm

A learning algorithm is good if it produces hypotheses that do a good job of predicting the classifications of unseen examples. In Section 18.5, we will see how prediction quality can be estimated in advance. For now, we will look at a methodology for assessing prediction quality after the fact.

Obviously, a prediction is good if it turns out to be true, so we can assess the quality of a hypothesis by checking its predictions against the correct classification once we know it. We

TEST SET

do this on a set of examples known as the **test set**. If we train on all our available examples, then we will have to go out and get some more to test on, so often it is more convenient to adopt the following methodology:

1. Collect a large set of examples.
2. Divide it into two disjoint sets: the **training set** and the **test set**.
3. Apply the learning algorithm to the training set, generating a hypothesis h.
4. Measure the percentage of examples in the test set that are correctly classified by h.
5. Repeat steps 2 to 4 for different sizes of training sets and different randomly selected training sets of each size.

The result of this procedure is a set of data that can be processed to give the average prediction quality as a function of the size of the training set. This function can be plotted on a graph,

LEARNING CURVE

giving what is called the **learning curve** for the algorithm on the particular domain. The

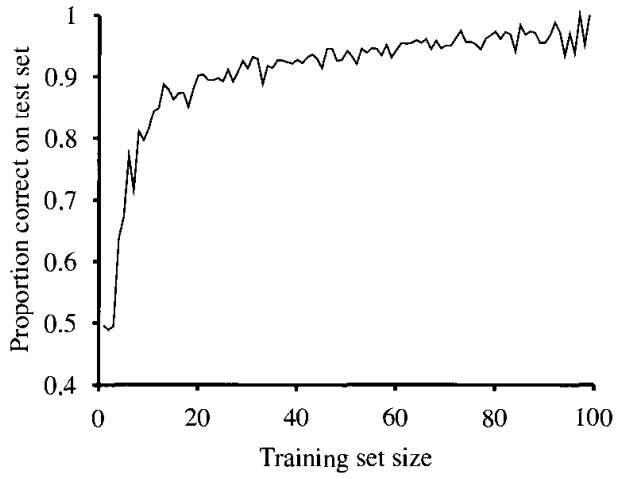


Figure 18.7 A learning curve for the decision tree algorithm on 100 randomly generated examples in the restaurant domain. The graph summarizes 20 trials.

learning curve for DECISION-TREE-LEARNING with the restaurant examples is shown in Figure 18.7. Notice that, as the training set grows, the prediction quality increases. (For this reason, such curves are also called **happy graphs**.) This is a good sign that there is indeed some pattern in the data and the learning algorithm is picking it up.

Obviously, the learning algorithm must not be allowed to "see" the test data before the learned hypothesis is tested on them. Unfortunately, it is all too easy to fall into the trap of **peeking** at the test data. Peeking typically happens as follows: A learning algorithm can have various "knobs" that can be twiddled to tune its behavior—for example, various different criteria for choosing the next attribute in decision tree learning. We generate hypotheses for various different settings of the knobs, measure their performance on the test set, and report the prediction performance of the best hypothesis. Alas, peeking has occurred! The reason is that the hypothesis was selected *on the basis of its test set performance*, so information about the test set has leaked into the learning algorithm. The moral of this tale is that any process that involves comparing the performance of hypotheses on a test set must use a *new* test set to measure the performance of the hypothesis that is finally selected. In practice, this is too difficult, so people continue to run experiments on tainted sets of examples.

PEEKING

Noise and overfitting

We saw earlier that if there are two or more examples with the same description (in terms of the attributes) but different classifications, then the DECISION-TREE-LEARNING algorithm must fail to find a decision tree consistent with all the examples. The solution we mentioned before is to have each leaf node report either the majority classification for its set of examples, if a deterministic hypothesis is required, or report the estimated probabilities of each classification using the relative frequencies. Unfortunately, this is far from the whole story. It is quite possible, and in fact likely, that even when vital information is missing, the decision tree learning algorithm will find a decision tree that is consistent with all the examples. This is because the algorithm can use the *irrelevant* attributes, if any, to make spurious distinctions among the examples.

Consider the problem of trying to predict the roll of a die. Suppose that experiments are carried out during an extended period of time with various dice and that the attributes describing each training example are as follows:

1. *Day*: the day on which the die was rolled (Mon, Tue, Wed, Thu).
2. *Month*: the month in which the die was rolled (Jan or Feb).
3. *Color*: the color of the die (Red or Blue).

As long as no two examples have identical descriptions, DECISION-TREE-LEARNING will find an exact hypothesis. The more attributes there are, the more likely it is that an exact hypothesis will be found. Any such hypothesis will be totally spurious. What we would like is that DECISION-TREE-LEARNING return a single leaf node with probabilities close to 1/6 for each roll, once it has seen enough examples.

Whenever there is a large set of possible hypotheses, one has to be careful not to use the resulting freedom to find meaningless "regularity" in the data. This problem is called **overfitting**. A very general phenomenon, overfitting occurs even when the target function is not at all random. It afflicts every kind of learning algorithm, not just decision trees.

OVERFITTING

A complete mathematical treatment of overfitting is beyond the scope of this book. Here we present a simple technique called **decision tree pruning** to deal with the problem. Pruning works by preventing recursive splitting on attributes that are not clearly relevant, even when the data at that node in the tree are not uniformly classified. The question is, how do we detect an irrelevant attribute?

DECISION TREE PRUNING

Suppose we split a set of examples using an irrelevant attribute. Generally speaking, we would expect the resulting subsets to have roughly the same proportions of each class as the original set. In this case, the information gain will be close to zero.[4] 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 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 each subset, p_i and n_i, with the expected numbers, $\hat{p}_i$ and $\hat{n}_i$, assuming true irrelevance:

$$\hat{p}_i = p \times \frac{p_i + n_i}{p + n} \qquad \hat{n}_i = n \times \frac{p_i + n_i}{p + n} .$$

[4] In fact, the gain will be positive unless the proportions are all exactly the same. (See Exercise 18.10.)

A convenient measure of the total deviation is given by

$$D = \sum_{i=1}^{v} \frac{(p_i - \hat{p}_i)^2}{\hat{p}_i} + \frac{(n_i - \hat{n}_i)^2}{\hat{n}_i} .$$

Under the null hypothesis, the value of D is distributed according to the χ^2 (chi-squared) distribution with $v - 1$ degrees of freedom. The probability that the attribute is really irrelevant can be calculated with the help of standard χ^2 tables or with statistical software. Exercise 18.11 asks you to make the appropriate changes to DECISION-TREE-LEARNING to implement this form of pruning, which is known as χ^2 **pruning**.

χ^2 PRUNING

With pruning, noise can be tolerated: classification errors give a linear increase in prediction error, whereas errors in the descriptions of examples have an asymptotic effect that gets worse as the tree shrinks down to smaller sets. Trees constructed with pruning perform significantly better than trees constructed without pruning when the data contain a large amount of noise. The pruned trees are often much smaller and hence easier to understand.

CROSS-VALIDATION

Cross-validation is another technique that reduces overfitting. It can be applied to any learning algorithm, not just decision tree learning. The basic idea is to estimate how well each hypothesis will predict unseen data. This is done by setting aside some fraction of the known data and using it to test the prediction performance of a hypothesis induced from the remaining data. K-fold cross-validation means that you run k experiments, each time setting aside a different $1/k$ of the data to test on, and average the results. Popular values for k are 5 and 10. The extreme is $k = n$, also known as leave-one-out cross-validation. Cross-validation can be used in conjunction with any tree-construction method (including pruning) in order to select a tree with good prediction performance. To avoid peeking, we must then measure this performance with a new test set.

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 each, 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 object 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 18.12.
- ◇ **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, we could use an attribute, such as $RestaurantName$, that has a different value for every example. Then each subset of examples would be a singleton with a unique classification, so the information gain measure would have its highest value for this attribute. Nonetheless, the attribute could be irrelevant or useless. One solution is to use the **gain ratio** (Exercise 18.13).

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 dynamic programming methods exist for finding good split points, but it is still by far 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 value, such as the price of a work of art, rather than a discrete classification, then we need a **regression tree**. Such a tree has at each leaf a linear function of some subset of numerical attributes, rather than a single value. For example, the branch for hand-colored engravings might end with a linear function of area, age, and number of colors. The learning algorithm must decide when to stop splitting and begin applying linear regression using the remaining attributes (or some subset thereof).

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 several hundred 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 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 neural networks (see Chapter 20).

18.4 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 whole collection, or **ensemble**, of hypotheses from the hypothesis space and combine their predictions. For example, we might generate a hundred different decision trees from the same training set and have them vote on the best classification for a new example.

ENSEMBLE LEARNING

The motivation for ensemble learning is simple. Consider an ensemble of $M = 5$ hypotheses 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 misclassify it.* The hope is that this is much less likely than a misclassification by a single hypothesis. Suppose we assume that each hypothesis h_i in the ensemble has an error of p—that is, the probability that a randomly chosen example is misclassified by h_i 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 example, a simple calculation (Exercise 18.14) 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

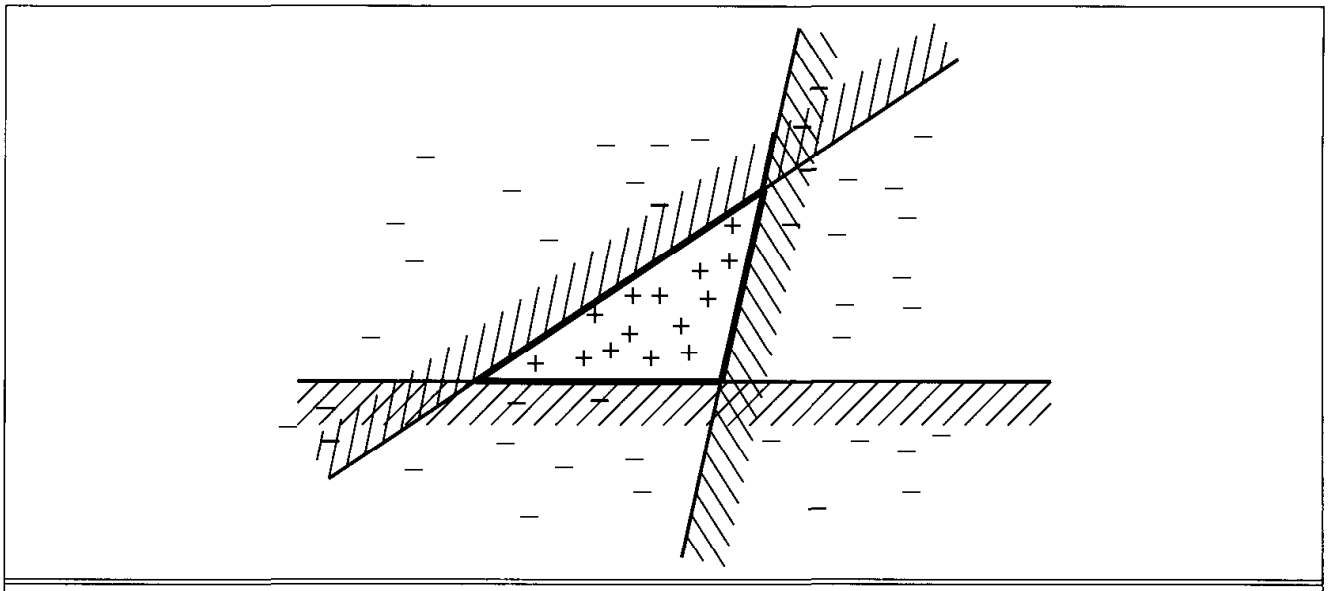


Figure 18.8 Illustration of the increased expressive power obtained by ensemble learning. We take three linear threshold hypotheses, each of which classifies positively on the non-shaded side, and classify as positive any example classified positively by all three. The resulting triangular region is a hypothesis not expressible in the original hypothesis space.

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 space as the set of all possible ensembles constructible from hypotheses in the original space. Figure 18.8 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.[5]

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 examples 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 correctly classified examples. From this new weighted training set, we generate hypothesis h_2. The process continues in this way until we have generated M hypotheses, where M is

[5] For learning algorithms in which this is not possible, one can instead create a **replicated training set** where the ith example appears w_j times, using randomization to handle fractional weights.

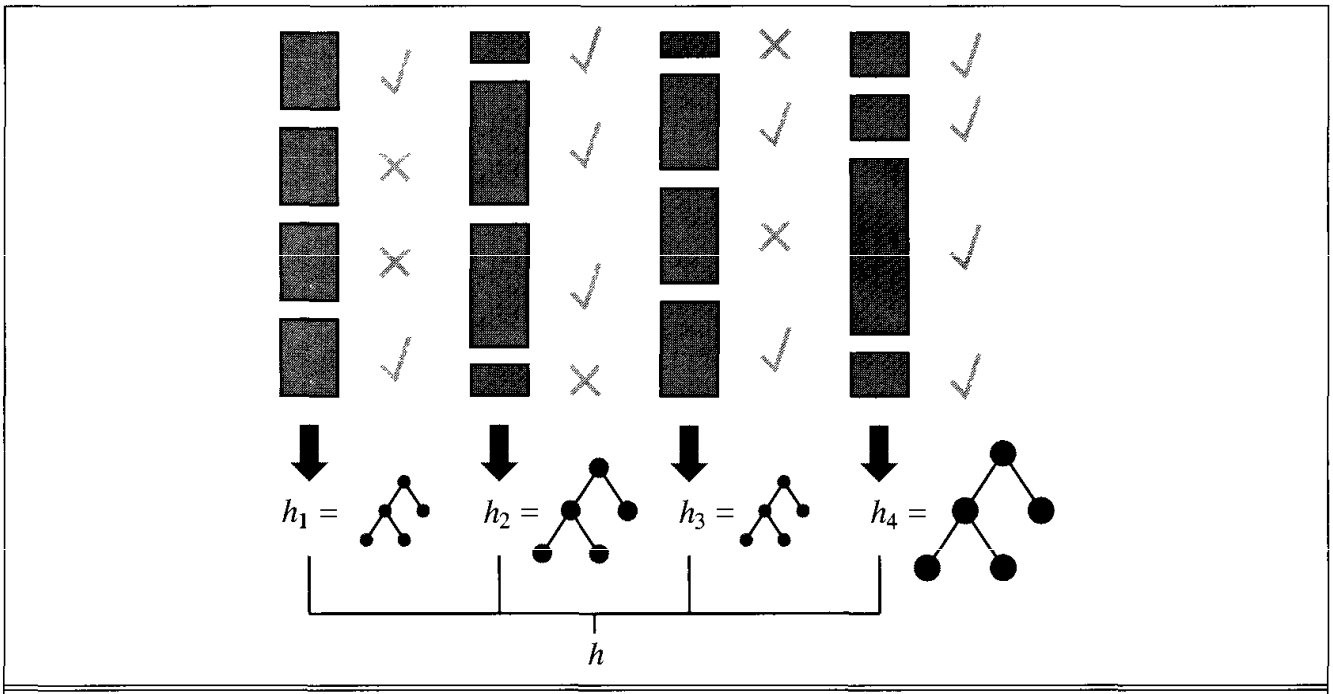


Figure 18.9 How the boosting algorithm works. Each shaded rectangle corresponds to an example; the height of the rectangle corresponds to the weight. The ticks and crosses indicate whether the example was classified correctly by the current hypothesis. The size of the decision tree indicates the weight of that hypothesis in the final ensemble.

an input to the boosting algorithm. The final ensemble hypothesis is a weighted-majority combination of all the M hypotheses, each weighted according to how well it performed on the training set. Figure 18.9 shows how the algorithm works conceptually. There are many variants of the basic boosting idea with different ways of adjusting the weights and combining the hypotheses. One specific algorithm, called ADABOOST, is shown in Figure 18.10. While the details of the weight adjustments are not so important, ADABOOST does have a very important property: if the input learning algorithm L is a **weak learning** algorithm—which means that L always returns a hypothesis with weighted error on the training set that is slightly better than random guessing (i.e., 50% for Boolean classification)—then ADABOOST will return a hypothesis that *classifies the training data perfectly* for large enough M. 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.

WEAK LEARNING

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 18.11(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 $M = 5$), the performance is better, reaching 93% after 100 examples.

DECISION STUMP

An interesting thing happens as the ensemble size M increases. Figure 18.11(b) shows the training set performance (on 100 examples) as a function of M. Notice that the error reaches zero (as the boosting theorem tells us) when M is 20; that is, a weighted-majority

function ADABOOST(*examples*, L, M) **returns** a weighted-majority hypothesis
  inputs: *examples*, set of N labelled examples $(x_1, y_1), \ldots, (x_N, y_N)$
    L, a learning algorithm
    M, the number of hypotheses in the ensemble
  local variables: **w**, a vector of N example weights, initially $1/N$
    h, a vector of M hypotheses
    z, a vector of M hypothesis weights

  for $m = 1$ **to** M **do**
    $\mathbf{h}[m] \leftarrow L(examples, \mathbf{w})$
    $error \leftarrow 0$
    for $j = 1$ **to** N **do**
      if $\mathbf{h}[m](x_j) \neq y_j$ **then** $error \leftarrow error + \mathbf{w}[j]$
    for $j = 1$ **to** N **do**
      if $\mathbf{h}[m](x_j) = y_j$ **then** $\mathbf{w}[j] \leftarrow \mathbf{w}[j] \cdot error/(1 - error)$
    $\mathbf{w} \leftarrow$ NORMALIZE(**w**)
    $\mathbf{z}[m] \leftarrow \log(1 - error)/error$
  return WEIGHTED-MAJORITY(**h**, **z**)

Figure 18.10 The ADABOOST variant of the boosting method for ensemble learning. The algorithm generates hypotheses by successively reweighting the trainng 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**.

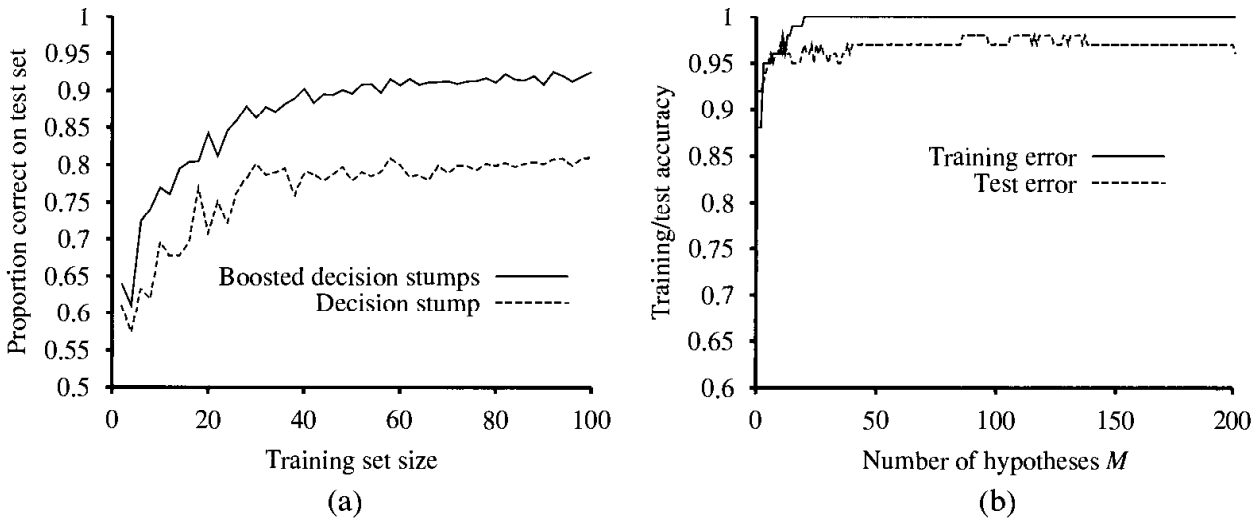


Figure 18.11 (a) Graph showing the performance of boosted decision stumps with $M = 5$ versus decision stumps on the restaurant data. (b) The proportion correct on the training set and the test set as a function of M, the number of hypotheses in the ensemble. Notice that the test set accuracy improves slightly even after the training accuracy reaches 1, i.e., after the ensemble fits the data exactly.

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 $M = 20$, the test performance is 0.95 (or 0.05 error), and the performance increases to 0.98 as late as $M = 137$, 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 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.

18.5 Why Learning Works: Computational Learning Theory

The main unanswered question posed in Section 18.2 was this: how can one be sure that one's learning algorithm has produced a theory that will correctly predict the future? 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. Until we find answers, machine learning will, at best, be puzzled by its own success.

COMPUTATIONAL LEARNING THEORY

The approach taken in this section is based on **computational learning theory**, a field at the intersection of AI, statistics, and theoretical computer science. The underlying principle is the following: *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.

PROBABLY APPROXIMATELY CORRECT

PAC-LEARNING

There are some subtleties in the preceding argument. The main question is the connection between the training and the test examples; after all, we want the hypothesis to be approximately correct on the test set, not just on the training set. The key assumption is that the training and test sets are drawn randomly and independently from the same population of examples with the *same probability distribution.* This is called the **stationarity** assumption. Without the stationarity assumption, the theory can make no claims at all about the future, because there would be no necessary connection between future and past. The stationarity assumption amounts to supposing that the process that selects examples is not malevolent. Obviously, if the training set consists only of weird examples—two-headed dogs, for instance—then the learning algorithm cannot help but make unsuccessful generalizations about how to recognize dogs.

STATIONARITY

How many examples are needed?

In order to put these insights into practice, we will need some notation:

- Let $\mathbf{X}$ be the set of all possible examples.
- Let D be the distribution from which examples are drawn.
- Let $\mathbf{H}$ be the set of possible hypotheses.
- Let N be the number of examples in the training set.

Initially, we will assume that the true function f is a member of $\mathbf{H}$. Now we can define the
ERROR **error** of a hypothesis h with respect to the true function f given a distribution D over the examples as the probability that h is different from f on an example:

$$\text{error}(h) = P(h(x) \neq f(x) | x \text{ drawn from } D) .$$

This is the same quantity being measured experimentally by the learning curves shown earlier.

A hypothesis h is called **approximately correct** if $\text{error}(h) \leq \epsilon$, where ϵ is a small constant. The plan of attack is to show 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
ϵ-BALL is called the **ϵ-ball** around the true function f. Figure 18.12 shows the set of all hypotheses $\mathbf{H}$, divided into the ϵ-ball around f and the remainder, which we call $\mathbf{H}_{\text{bad}}$.

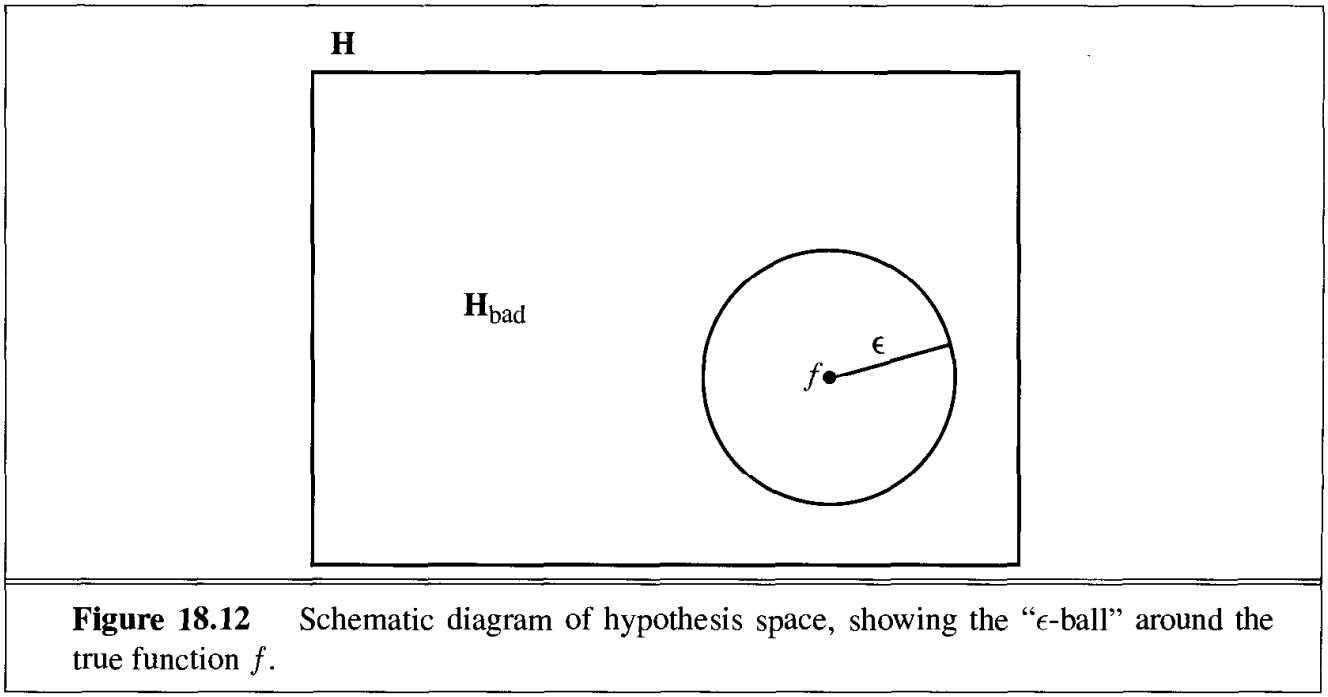


Figure 18.12 Schematic diagram of hypothesis space, showing the "ϵ-ball" around the true function f.

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

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

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

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

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

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

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

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

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

It appears, then, that the key question is the size of the hypothesis space. As we saw earlier, if $\mathbf{H}$ is the set of all Boolean functions on n attributes, then $|\mathbf{H}| = 2^{2^n}$. Thus, the sample complexity of the space grows as 2^n. Because the number of possible examples is also 2^n, this says that any learning algorithm for the space of all Boolean functions will do no better than a lookup table if it merely returns a hypothesis that is consistent with all known examples. Another way to see this is to observe that for any unseen example, the hypothesis space will contain as many consistent hypotheses that predict a positive outcome as it does hypotheses that predict a negative outcome.

The dilemma we face, then, is that unless we restrict the space of functions the algorithm can consider, it will not be able to learn; but if we do restrict the space, we might eliminate the true function altogether. There are two ways to "escape" this dilemma. The first way is to insist that the algorithm return not just any consistent hypothesis, but preferably a simple one (as is done in decision tree learning). The theoretical analysis of such algorithms is beyond the scope of this book, but in cases where finding simple consistent hypotheses is tractable, the sample complexity results are generally better than for analyses based only on consistency. The second escape, which we pursue here, is to focus on learnable subsets of the entire set of Boolean functions. The idea is that in most cases we do not need the full expressive power of Boolean functions, and can get by with more restricted languages. We now examine one such restricted language in more detail.

Learning decision lists

DECISION LIST

A **decision list** is a logical expression of a restricted form. It 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.[6] Decision lists resemble decision trees, but their overall structure is simpler. In contrast, the individual tests are more complex. Figure 18.13 shows a decision list that represents the following hypothesis:

$$\forall x \;\; WillWait(x) \Leftrightarrow Patrons(x, Some) \vee (Patrons(x, Full) \wedge Fri/Sat(x)) \,.$$

If we allow tests of arbitrary size, then decision lists can represent any Boolean function (Exercise 18.15). On the other hand, if we restrict the size of each test to at most k literals,

[6] A decision list is therefore identical in structure to a COND statement in Lisp.

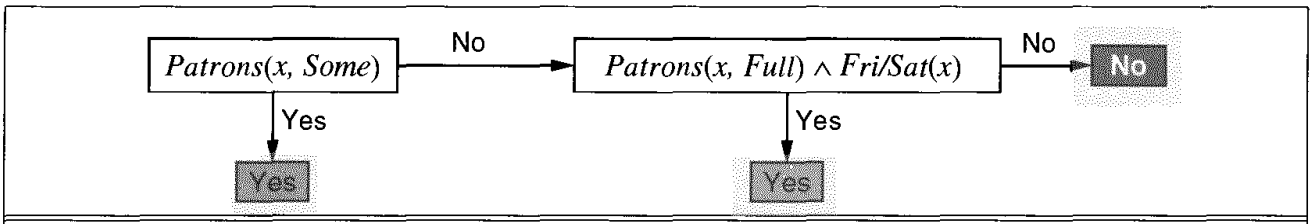


Figure 18.13 A decision list for the restaurant problem.

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 18.13 is in 2-DL. It is easy to show (Exercise 18.15) 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 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 18.14.

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,

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 18.14 An algorithm for learning decision lists.

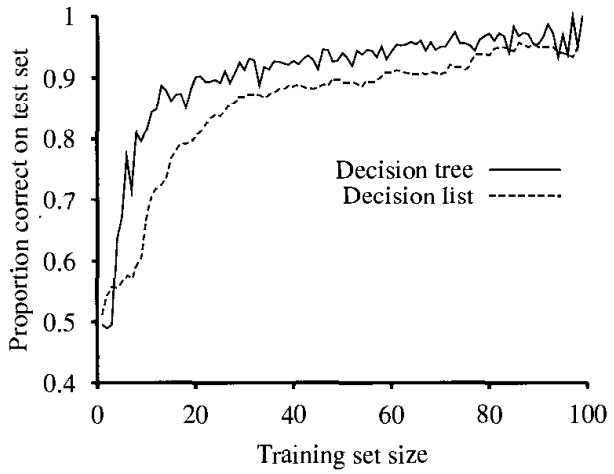


Figure 18.15 Learning curve for DECISION-LIST-LEARNING algorithm on the restaurant data. The curve for DECISION-TREE-LEARNING is shown for comparison.

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 18.15 suggests.

Discussion

IDENTIFICATION IN THE LIMIT

Computational learning theory has generated a new way of looking at the problem of learning. In the early 1960s, the theory of learning focused on the problem of **identification in the limit**. According to this notion, an identification algorithm must return a hypothesis that exactly matches the true function. One way to do that is as follows: First, order all the hypotheses in **H** according to some measure of simplicity. Then, choose the simplest hypothesis consistent with all the examples so far. As new examples arrive, the method will abandon a simpler hypothesis that is invalidated and adopt a more complex one instead. Once it reaches the true function, it will never abandon it. Unfortunately, in many hypothesis spaces, the number of examples and the computation time required to reach the true function are enormous. Thus, computational learning theory does not insist that the learning agent find the "one true

law" governing its environment, but instead that it find a hypothesis with a certain degree of predictive accuracy. Computational learning theory also brings sharply into focus the tradeoff between the expressiveness of the hypothesis language and the complexity of learning, and has led directly to an important class of learning algorithms called support vector machines.

The PAC-learning results we have shown are worst-case complexity results and do not necessarily reflect the average-case sample complexity as measured by the learning curves we have shown. An average-case analysis must also make assumptions about the distribution of examples and the distribution of true functions that the algorithm will have to learn. As these issues become better understood, computational learning theory continues to provide valuable guidance to machine learning researchers who are interested in predicting or modifying the learning ability of their algorithms. Besides decision lists, results have been obtained for almost all known subclasses of Boolean functions, for sets of first-order logical sentences (see Chapter 19), and for neural networks (see Chapter 20). The results show that the pure inductive learning problem, where the agent begins with no prior knowledge about the target function, is generally very hard. As we show in Chapter 19, the use of prior knowledge to guide inductive learning makes it possible to learn quite large sets of sentences from reasonable numbers of examples, even in a language as expressive as first-order logic.

18.6 SUMMARY

This chapter has concentrated on inductive learning of deterministic functions from examples. The main points were as follows:

- Learning takes many forms, depending on the nature of the performance element, the component to be improved, and the available feedback.
- If the available feedback, either from a teacher or from the environment, provides the correct value for the examples, the learning problem is called **supervised learning**. The task, also called **inductive learning**, is then to learn a function from examples of its inputs and outputs. Learning a discrete-valued function is called **classification**; learning a continuous function is called **regression**.
- Inductive learning involves finding a **consistent** hypothesis that agrees 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.
- Ensemble methods such as **boosting** often perform better than individual methods.
- **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.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Chapter 1 outlined the history of philosophical investigations into inductive learning. William of Ockham (1280–1349), the most influential philosopher of his century and a major contributer 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.

EPAM, the "Elementary Perceiver And Memorizer" (Feigenbaum, 1961), was one of the earliest systems to use decision trees (or **discrimination nets**). EPAM was intended as a cognitive-simulation model of human concept learning. CLS (Hunt *et al.*, 1966) used a heuristic look-ahead method to construct decision trees. ID3 (Quinlan, 1979) added the crucial idea of using information content to provide the heuristic function. Information theory itself 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.

Many other algorithmic approaches to learning have been tried. The **current-best-hypothesis** approach maintains a single hypothesis, specializing it when it proves too broad and generalizing it when it proves too narrow. This 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.*, 1986b).

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 the chapter originated with theoretical work by Schapire (1990). The ADABOOST algorithm was developed by Freund and Schapire (1996) and analyzed theoretically by Schapire (1999). Friedman *et al.* (2000) explain boosting from a statistician's viewpoint.

BAGGING

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).

KOLMOGOROV COMPLEXITY

Whereas the identification-in-the-limit approach concentrates on eventual convergence, the study of **Kolmogorov complexity** or **algorithmic complexity**, developed independently by Solomonoff (1964) 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) 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.

MINIMUM DESCRIPTION LENGTH

Computational learning theory—that is, the theory of PAC-learning—was inaugurated by Leslie Valiant (1984). Valiant's 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).

UNIFORM CONVERGENCE THEORY
VC DIMENSION

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 $\ln |\mathbf{H}|$ 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). Subsequent developments in VC theory led to the invention of the **support vector machine** or SVM (Boser *et al.*, 1992; Vapnik, 1998), which we describe in Chapter 20.

A large number of important papers on machine learning have been collected in *Readings in Machine Learning* (Shavlik and Dietterich, 1990). The two volumes *Machine Learning 1* (Michalski *et al.*, 1983) and *Machine Learning 2* (Michalski *et al.*, 1986a) also contain many important papers, as well as huge bibliographies. Weiss and Kulikowski (1991) provide a broad introduction to function-learning methods from machine learning, statistics, and neural networks. The STATLOG project (Michie *et al.*, 1994) is by far the most exhaustive investigation into the comparative performance of learning algorithms. Good current research in machine learning is published in the annual proceedings of the International Conference on Machine Learning and the conference on Neural Information Processing Systems, in *Machine Learning* and the *Journal of Machine Learning Research*, and in mainstream AI journals. Work in computational learning theory also appears in the annual ACM Workshop on Computational Learning Theory (COLT), and is described in the texts by Kearns and Vazirani (1994) and Anthony and Bartlett (1999).

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, identifying each of the components of the model as appropriate.

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 Draw a decision tree for the problem of deciding whether to move forward at a road intersection, given that the light has just turned green.

18.4 We never test the same attribute twice along one path in a decision tree. Why not?

18.5 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?

18.6 A good "straw man" learning algorithm is as follows: create a table out of all the training examples. Identify which output occurs most often among the training examples; call it d. Then when given an input that is not in the table, just return d. For inputs that *are* in the table, return the output associated with it (or the most frequent output, if there is more than one). Implement this algorithm and see how well it does on the restaurant domain. This should give you an idea of the baseline for the domain—the minimal performance that any algorithm should be able to obtain.

18.7 Suppose you are running a learning experiment on a new algorithm. You have a data set consisting of 25 examples of each of two classes. You plan to use leave-one-out cross-validation. As a baseline, you run your experimental setup on 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. Can you explain why?

18.8 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.

CLASS PROBABILITY

b. Show that the **class probability** $p/(p+n)$ minimizes the sum of squared errors.

18.9 Suppose that a learning algorithm is trying to find a consistent hypothesis when the classifications of examples are actually random. There are n Boolean attributes, and examples are drawn uniformly from the set of 2^n possible examples. Calculate the number of examples required before the probability of finding a contradiction in the data reaches 0.5.

18.10 Suppose that an attribute splits the set of examples E into subsets E_i and that each subset has p_i positive examples and n_i negative examples. Show that the attribute has strictly positive information gain unless the ratio $p_i/(p_i + n_i)$ is the same for all i.

18.11 Modify DECISION-TREE-LEARNING to include χ^2-pruning. You might wish to consult Quinlan (1986) for details.

18.12 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.

18.13 In the chapter, 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.

18.14 Consider an ensemble learning algorithm that uses simple majority voting among M 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 M and ϵ, and evaluate it for the cases where $M = 5$, 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?

18.15 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.

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 three chapters, 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 a bias such as "simpler is better." 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 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.

PRIOR KNOWLEDGE

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.

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 an object that 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 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 object is given by the sentence

$$WillWait(X_1) .$$

We will use the generic notation $Q(X_i)$ if the example is positive, and $\neg Q(X_i)$ if the example is negative. The complete training set is then just the conjunction of all the description and classification sentences.

The aim of inductive learning in the logical setting is to find an equivalent logical expression for the goal predicate Q that we can use to classify examples correctly. Each hypothesis proposes such an expression, which we call a **candidate definition** of the goal predicate. Using C_i to denote the candidate definition, each hypothesis H_i is a sentence of the form $\forall x \;\; Q(x) \;\Leftrightarrow\; C_i(x)$. For example, a decision tree asserts that the goal predicate is true of an object if only if one of the branches leading to *true* is satisfied. Thus, the Figure 18.6 expresses the following logical definition (which we will call H_r for future reference):

CANDIDATE DEFINITION

$$\begin{aligned} \forall r \;\; WillWait(r) \;\Leftrightarrow\; & Patrons(r, Some) \\ & \vee\; Patrons(r, Full) \wedge Hungry(r) \wedge Type(r, French) \\ & \vee\; Patrons(r, Full) \wedge Hungry(r) \wedge Type(r, Thai) \\ & \qquad \wedge Fri/Sat(r) \\ & \vee\; Patrons(r, Full) \wedge Hungry(r) \wedge Type(r, Burger) . \end{aligned} \tag{19.1}$$

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.

EXTENSION

The hypothesis space **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_i is consistent with the entire training set, it has to be consistent with each example. What would it mean for it to be inconsistent with an example? This can happen in one of two ways:

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 $Patrons(X_{13}, Full) \wedge Wait(X_{13}, 0\text{-}10) \wedge \neg Hungry(X_{13}) \wedge \ldots \wedge WillWait(X_{13})$

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 $H_1 \vee H_2 \vee H_3 \vee H_4$. Then if I_1 is inconsistent with H_2 and H_3, the logical inference system can deduce the new hypothesis space $H_1 \vee H_4$.

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.

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 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.

[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.

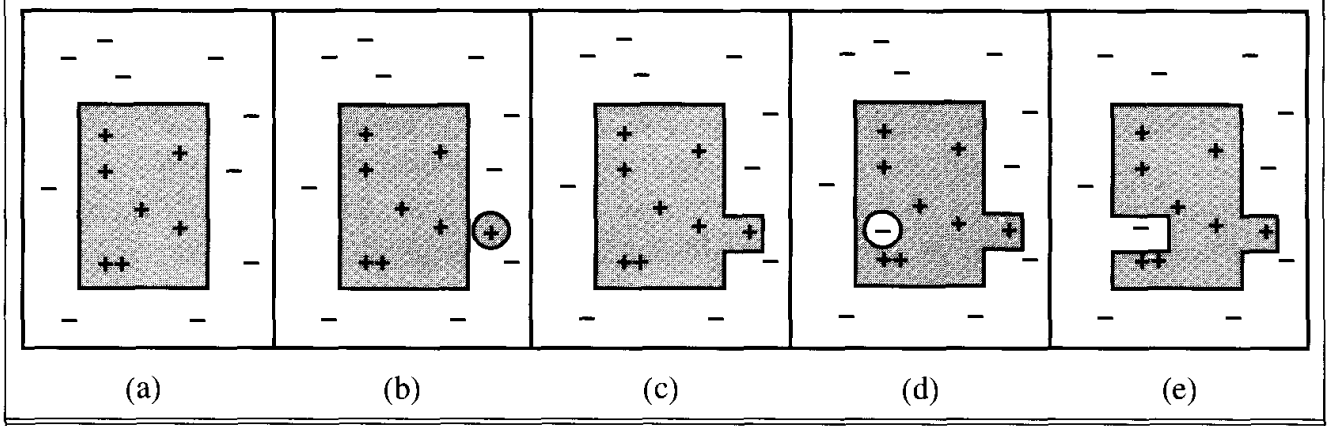


Figure 19.1 (a) A consistent hypothesis. (b) A false negative. (c) The hypothesis is generalized. (d) A false positive. (e) The hypothesis is specialized.

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.

function CURRENT-BEST-LEARNING(*examples*) **returns** a hypothesis

$H \leftarrow$ any hypothesis consistent with the first example in *examples*
for each remaining example in *examples* **do**
  if e is false positive for H **then**
    $H \leftarrow$ **choose** a specialization of H consistent with *examples*
  else if e is false negative for H **then**
    $H \leftarrow$ **choose** a generalization of H consistent with *examples*
  if no consistent specialization/generalization can be found **then fail**
return H

Figure 19.2 The current-best-hypothesis learning algorithm. It searches for a consistent hypothesis and backtracks when no consistent specialization/generalization can be found.

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

$$\forall x \;\; C_2(x) \;\Rightarrow\; C_1(x) \; .$$

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

DROPPING CONDITIONS

$Alternate(x) \wedge Patrons(x, Some)$, 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. $Alternate(X_1)$ 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. 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:

$$H_4 : \forall x \ \ WillWait(x) \Leftrightarrow Patrons(x, Some) \vee (Patrons(x, Full) \wedge Fri/Sat(x)) .$$

Already, the hypothesis is starting to look reasonable. Obviously, there are other possibilities consistent with the first four examples; here are two of them:

$$H'_4 : \forall x \ \ WillWait(x) \Leftrightarrow \neg WaitEstimate(x, 30\text{-}60) .$$

$$H''_4 : \forall x \ \ WillWait(x) \Leftrightarrow Patrons(x, Some) \vee (Patrons(x, Full) \wedge WaitEstimate(x, 10\text{-}30)) .$$

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 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 instances and a large space, however, some difficulties arise:

1. Checking all the previous instances 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.

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 instance 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

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 the *examples*.

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 anyway. It is also a **least-commitment** algorithm because it makes no arbitrary choices (cf. the partial-order planning algorithm in Chapter 11). But there is an obvious problem. We already said that the 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 is 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

BOUNDARY SET

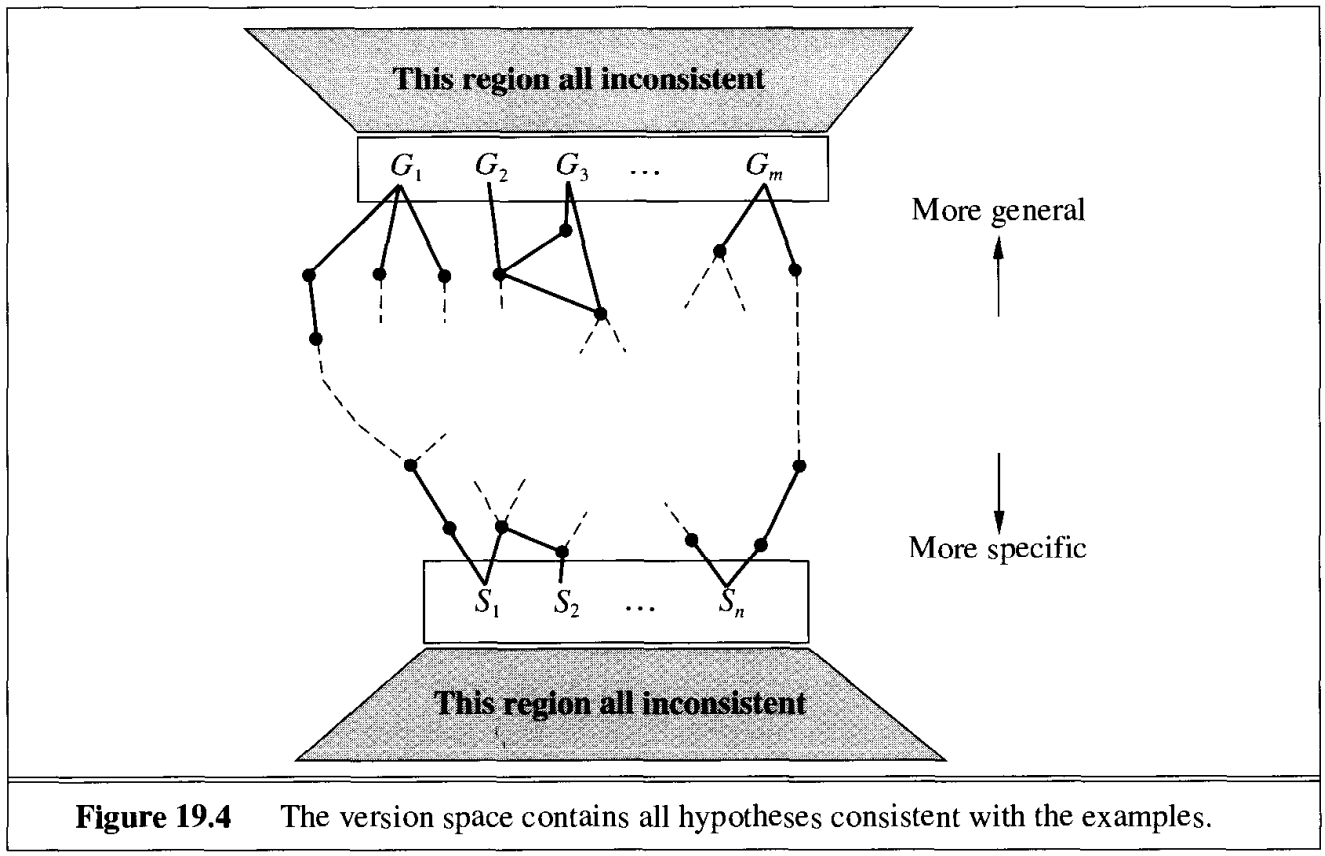


Figure 19.4 The version space contains all hypotheses consistent with the examples.

G-SET
S-SET

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:

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" be-

tween 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.

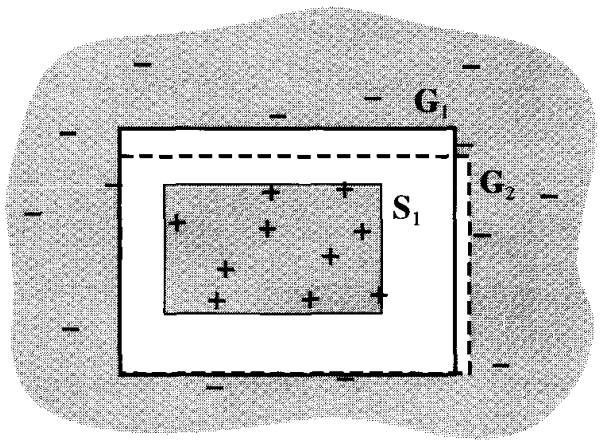


Figure 19.5 The extensions of the members of G and S. No known examples lie in between the two sets of boundaries.

We need to worry about the members S_i and G_i of the S- and G-sets. For each one, the new instance 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 instance until one of three things happens:

1. We have exactly one concept 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 with 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 of G-set may grow exponentially in the number of attributes, even though efficient learning algorithms exist for those hypothesis spaces.

To date, no completely successful solution has been found for the problem of noise. The problem of disjunction can be addressed by allowing limited forms of disjunction or by including a **generalization hierarchy** of more general predicates. For example, instead of using the disjunction $WaitEstimate(x, 30\text{-}60) \vee WaitEstimate(x, >60)$, we might use the single literal $LongWait(x)$. The set of generalization and specialization operations can be easily extended to handle this.

GENERALIZATION HIERARCHY

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.

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 679), 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.

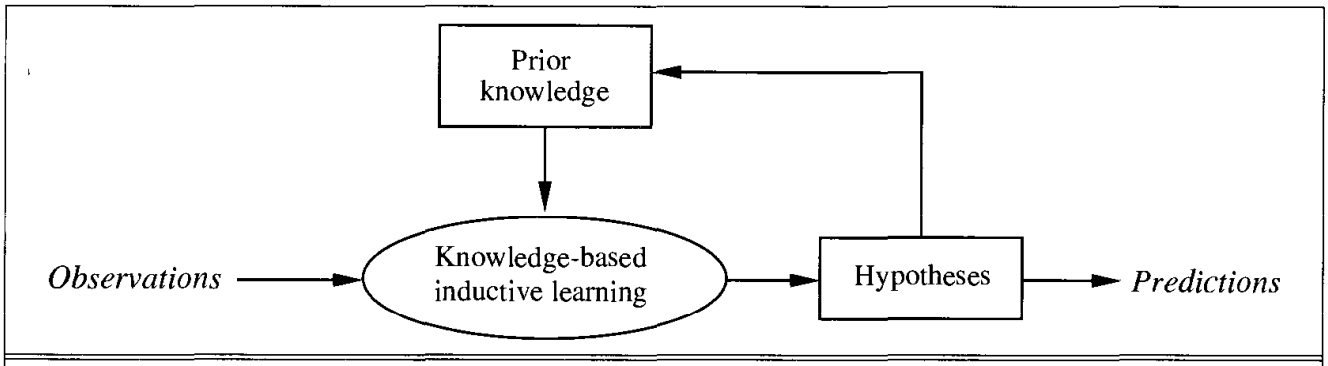


Figure 19.6 A cumulative learning process uses, and adds to, its stock of background knowledge over time.

If we want to build an autonomous learning agent that uses background knowledge, the agent must have some method for obtaining 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.

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 traveller 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.*

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

$$Hypothesis \wedge Descriptions \models Classifications$$
$$Background \models Hypothesis\ .$$

Because EBL uses Equation (19.3), it was initially thought to be a better 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 instance.* 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.

RELEVANCE

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:

$$\begin{aligned} &Hypothesis \wedge Descriptions \models Classifications \ , \\ &Background \wedge Descriptions \wedge Classifications \models Hypothesis \ . \end{aligned} \tag{19.4}$$

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:

$$Background \wedge Hypothesis \wedge Descriptions \models Classifications \ . \tag{19.5}$$

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

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:

INDUCTIVE LOGIC PROGRAMMING

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

As we explained in the introduction to this chapter, 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 9.15). If we differentiate an expression such as X^2 with respect to X, we obtain $2X$. (Notice that 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 $\text{ASK}(Derivative(X^2, X) = d, KB)$, 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[2] that for any arithmetic unknown u, the derivative of u^2 with respect to u is $2u$. 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 Mathe-*

[2] Of course, a general rule for u^n can also be produced, but the current example suffices to make the point.

matica, 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.

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:

$$Rewrite(u, v) \wedge Simplify(v, w) \Rightarrow Simplify(u, w) .$$
$$Primitive(u) \Rightarrow Simplify(u, u) .$$
$$ArithmeticUnknown(u) \Rightarrow Primitive(u) .$$
$$Number(u) \Rightarrow Primitive(u) .$$
$$Rewrite(1 \times u, u) .$$
$$Rewrite(0 + u, u) .$$
$$\vdots$$

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 (with the necessary bindings) and form a general rule for the goal predicate:

$$Rewrite(1 \times (0 + z), 0 + z) \wedge Rewrite(0 + z, z) \wedge ArithmeticUnknown(z)$$
$$\Rightarrow Simplify(1 \times (0 + z), z) .$$

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

$$ArithmeticUnknown(z) \Rightarrow Simplify(1 \times (0 + z), z) .$$

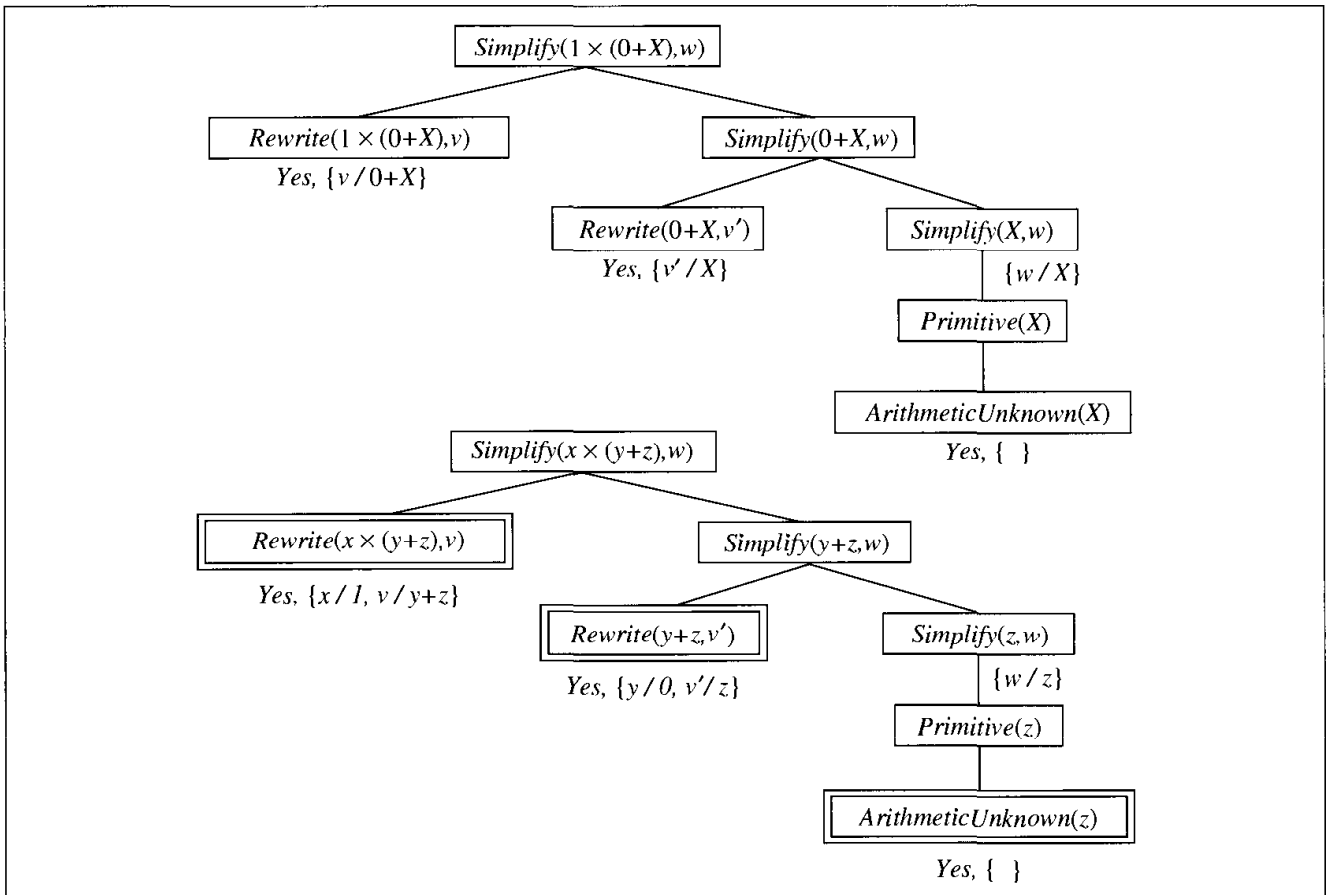


Figure 19.7 Proof trees for the simplification problem. The first tree shows the proof for the original problem instance, from which we can derive

$$ArithmeticUnknown(z) \Rightarrow Simplify(1 \times (0 + z), z) .$$

The second shows the proof for a problem instance with all constants replaced by variables, from which we can derive a variety of other rules.

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 that are true regardless of the values of the variables in the goal.

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 2 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:[3]

$$Nationality(x, n) \wedge Nationality(y, n) \wedge Language(x, l) \Rightarrow Language(y, l) \ . \quad (19.6)$$

(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

$$Nationality(Fernando, Brazil) \wedge Language(Fernando, Portuguese) \ ,$$

the following conclusion is entailed (see Exercise 19.1):

$$Nationality(x, Brazil) \Rightarrow Language(x, Portuguese) \ .$$

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 are called **functional dependencies** or **determinations**. They occur so commonly in certain kinds of applications (e.g., defining database designs) that a special syntax is used to write them. We adopt the notation of Davies (1985):

FUNCTIONAL DEPENDENCIES

DETERMINATIONS

$$Nationality(x, n) \succ Language(x, l) \ .$$

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:

$$Material(x, m) \wedge Temperature(x, t) \succ Conductance(x, \rho) \ ;$$
$$Material(x, m) \wedge Temperature(x, t) \succ Density(x, d) \ .$$

The corresponding generalizations follow logically from the determinations and observations.

[3] We assume for the sake of simplicity that a person speaks only one language. Clearly, the rule also would have to be amended for countries such as Switzerland and India.

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, $\log(|\mathbf{H}|)$ examples are required to converge to a reasonable hypothesis, where $|\mathbf{H}|$ 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, $|\mathbf{H}| = O(2^{2^n})$, 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 $O(2^{n-d})$. For biased hypothesis spaces, such as a conjunctively biased space, the reduction will be less dramatic, but still significant.

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 target predicate—that is, has the same classification. For example, suppose we have the following examples of conductance measurements on material samples:

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

```
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, ..., 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.

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 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 $\binom{n}{p} = O(n^p)$ 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

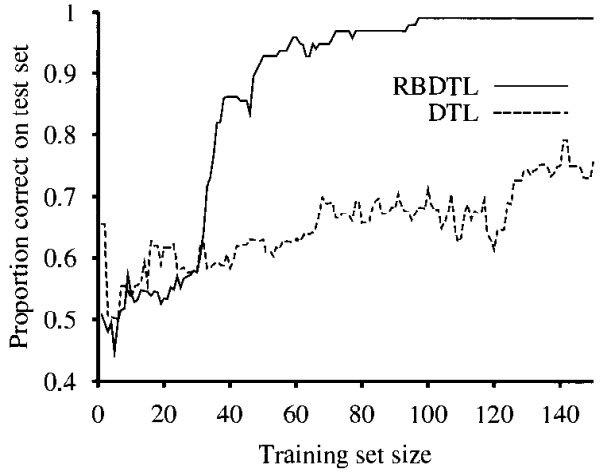


Figure 19.9 A performance comparison between RBDTL and DECISION-TREE-LEARNING on randomly generated data for a target function that depends on only 5 of 16 attributes.

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 theories as logic programs.[4] It has gained popularity for three reasons. First, ILP offers a rigorous approach to

[4] It might be appropriate at this point for the reader to refer to Chapter 9 for some of the underlying concepts, including Horn clauses, conjunctive normal form, unification, and resolution.

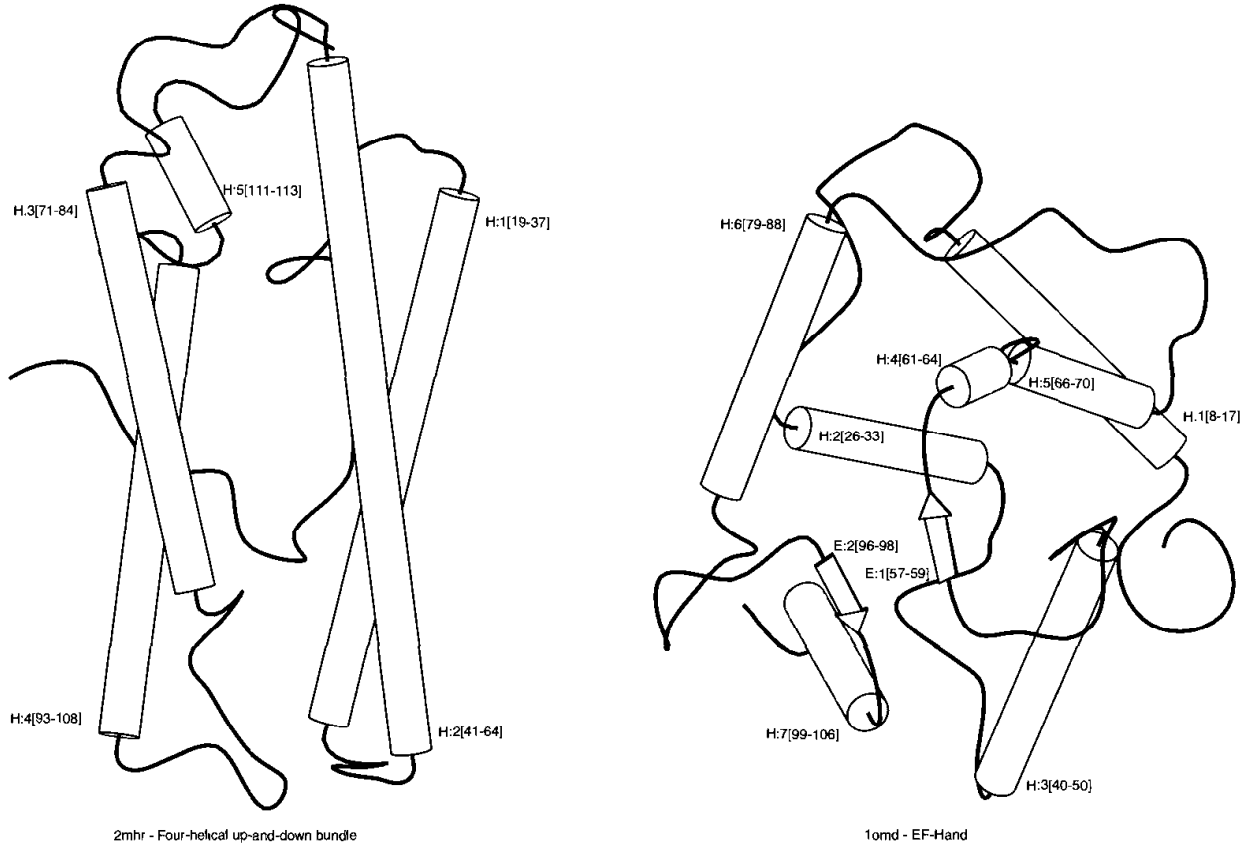


Figure 19.10 (a) and (b) show positive and negative examples, respectively, of the "four-helical up-and-down bundle" concept in the domain of protein folding. Each example structure is coded into a logical expression of about 100 conjuncts such as $TotalLength(D2mhr, 118) \wedge NumberHelices(D2mhr, 6) \wedge \ldots$. From these descriptions and from classifications such as $Fold(\text{FOUR-HELICAL-UP-AND-DOWN-BUNDLE}, D2mhr)$, the inductive logic programming system PROGOL (Muggleton, 1995) learned the following rule:

$$Fold(\text{FOUR-HELICAL-UP-AND-DOWN-BUNDLE}, p) \Leftarrow$$
$$Helix(p, h_1) \wedge Length(h_1, \text{HIGH}) \wedge Position(p, h_1, n)$$
$$\wedge (1 \leq n \leq 3) \wedge Adjacent(p, h_1, h_2) \wedge Helix(p, h_2) .$$

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

> The 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.

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.

An example

Recall from Equation (19.5) that the general knowledge-based induction problem is to "solve" the entailment constraint

$$Background \wedge Hypothesis \wedge Descriptions \models Classifications$$

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 8.11, shown here in Figure 19.11. The corresponding descriptions are as follows:

$$\begin{array}{lll} Father(Philip, Charles) & Father(Philip, Anne) & \ldots \\ Mother(Mum, Margaret) & Mother(Mum, Elizabeth) & \ldots \\ Married(Diana, Charles) & Married(Elizabeth, Philip) & \ldots \\ Male(Philip) & Male(Charles) & \ldots \\ Female(Beatrice) & Female(Margaret) & \ldots \end{array}$$

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 complete set of *Classifications* contains $20 \times 20 = 400$ conjuncts of the form

$$\begin{array}{ll} Grandparent(Mum, Charles) & Grandparent(Elizabeth, Beatrice) \ \ldots \\ \neg Grandparent(Mum, Harry) & \neg Grandparent(Spencer, Peter) \ \ldots \end{array}$$

We could of course learn from a subset of this complete set.

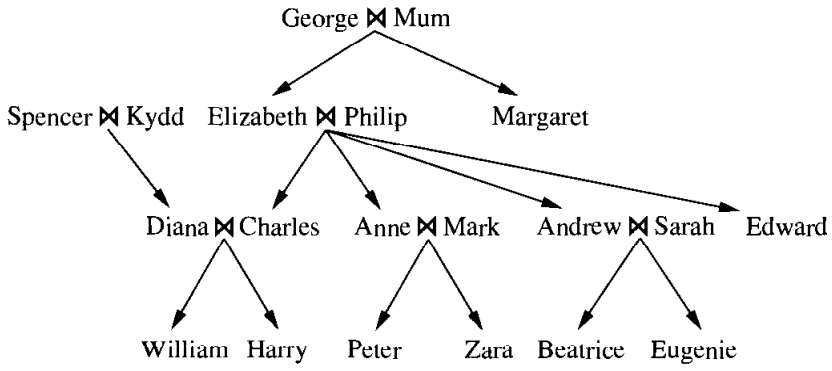


Figure 19.11 A typical family tree.

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 for *Hypothesis* is the following:

$$\begin{aligned} Grandparent(x,y) \quad &\Leftrightarrow \quad [\exists\, z \;\; Mother(x,z) \wedge Mother(z,y)] \\ &\vee \quad [\exists\, z \;\; Mother(x,z) \wedge Father(z,y)] \\ &\vee \quad [\exists\, z \;\; Father(x,z) \wedge Mother(z,y)] \\ &\vee \quad [\exists\, z \;\; Father(x,z) \wedge Father(z,y)] \,. \end{aligned}$$

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 "*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 sketched in the introduction. It has been one of the hardest problems in machine learning, but some ILP techniques provide effective mechanisms for achieving it.

CONSTRUCTIVE INDUCTION

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.

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

$$\langle George, Anne\rangle, \langle Philip, Peter\rangle, \langle Spencer, Harry\rangle, \ldots$$

and negative examples are

$$\langle George, Elizabeth\rangle, \langle Harry, Zara\rangle, \langle Charles, Philip\rangle, \ldots$$

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, extended with negated literals 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:

$$\begin{aligned} Father(x, y) &\Rightarrow Grandfather(x, y) \ . \\ Parent(x, z) &\Rightarrow Grandfather(x, y) \ . \\ Father(x, z) &\Rightarrow Grandfather(x, y) \ . \end{aligned}$$

(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, 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

```
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 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
  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-LITERAL and CHOOSE-LITERAL are explained in the text.

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 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 660) to decide which literal to add. The exact details are not so 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. For an explanation of the connection between noise and clause length, see page 715.

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.

Inductive learning with inverse deduction

INVERSE RESOLUTION

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 $Background \wedge Hypothesis \wedge Descriptions$, 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

$$\neg Parent(Elizabeth, y) \vee Grandparent(George, y)$$

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

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

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 $\neg Parent(Elizabeth, y) \vee Grandparent(George, y)$ for C_1 in the last step of Figure 19.13, the inverse resolution step might have chosen any of the following sentences:

$$\neg Parent(Elizabeth, Anne) \vee Grandparent(George, Anne) .$$
$$\neg Parent(z, Anne) \vee Grandparent(George, Anne) .$$
$$\neg Parent(z, y) \vee Grandparent(George, y) .$$
$$\vdots$$

(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

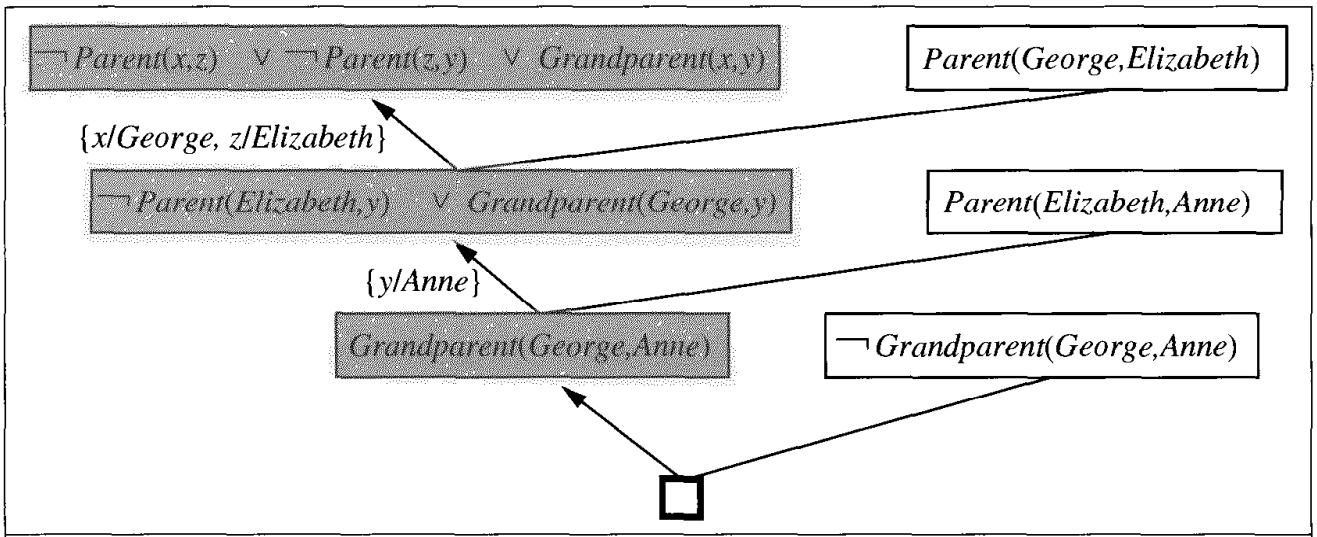


Figure 19.13 Early steps in an inverse resolution process. The shaded clauses are generated by inverse resolution steps from the clause to the right and the clause below. The unshaded clauses are from the *Descriptions* and *Classifications*.

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 that allows proof trees to have only a linear branching structure (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

INVERSE ENTAILMENT

$$Background \wedge Hypothesis \wedge Descriptions \models Classifications$$

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 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 (Lavrač and Džeroski, 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 11.

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.

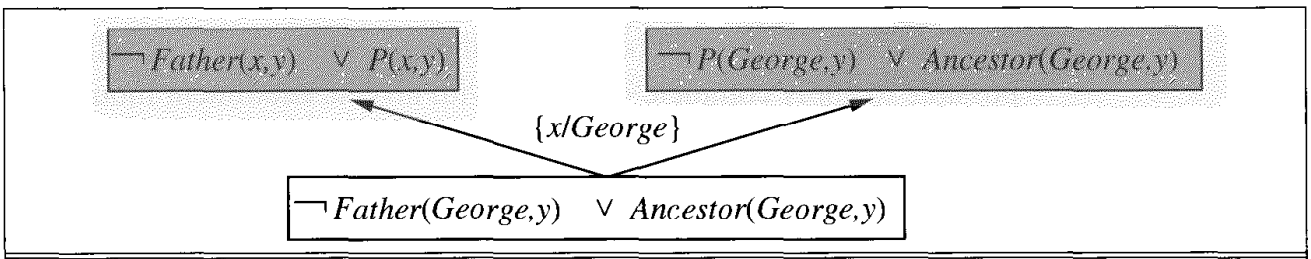


Figure 19.14 An inverse resolution step that generates a new predicate P.

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 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 which allow it to outperform linear regression, neural nets, and decision trees. King *et al.* (1992) showed how to predict the therapeutic efficacy of various drugs from their molecular structures. For all these examples it appears that both the ability 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 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.

GRUE

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).

More recently, rigorous analysis has led to a better understanding of the potential costs and benefits of EBL in terms of problem-solving speed. Minton (1988) showed that, without extensive extra work, EBL could easily slow down a program significantly. Tambe *et al.* (1990) found a similar problem with chunking and proposed a reduction in the expressive power of the rule language in order to minimize the cost of matching rules against working memory. This work has strong parallels with recent results on the complexity of inference in restricted versions of first-order logic. (See Chapter 9.) Formal probabilistic analysis of the expected payoff of EBL can be found in Greiner (1989) and Subramanian and Feldman (1990). An excellent survey appears in Dietterich (1990).

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).

ANALOGICAL REASONING

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 were given a more logical flavor by Bobrow and Raphael (1974). Dependencies were independently rediscovered and given a full logical analysis by Davies and Russell (Davies, 1985; Davies and Russell, 1987). They were used for declarative bias 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 in Almuallim and Dietterich (1991). Tadepalli (1993) describes an 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.[5] Wirth and O'Rorke (1991) also cover predicate invention. 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. Where FOIL was top-down, CIGOL and GOLEM worked bottom-up. 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 Lavrač and Džeroski (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 hopelessly complex. However, with better understanding of the importance of various kinds of syntactic restrictions on clauses, positive results have been obtained even for clauses with recursion (Dzeroski *et al.*, 1992). Learnability results for ILP are surveyed by Kietz and Dzeroski (1994) and Cohen and Page (1995).

[5] The inverse resolution method also appears in (Russell, 1986), with a simple algorithm given in a footnote.

DISCOVERY SYSTEMS

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 694 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. Zip code determines the state (U.S.).

b. Design and denomination determine the mass of a coin.

c. For a given program, input determines output.

d. Climate, food intake, exercise, and metabolism determine weight gain and loss.

e. 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.

20 STATISTICAL LEARNING METHODS

In which we view learning as a form of uncertain reasoning from observations.

Part V pointed out the prevalence of uncertainty in real environments. Agents can handle uncertainty by using the methods of probability and decision theory, but first they must learn their probabilistic theories of the world from experience. This chapter explains how they can do that. We will see how to formulate the learning task itself as a process of probabilistic inference (Section 20.1). We will see that a Bayesian view of learning is extremely powerful, providing general solutions to the problems of noise, overfitting, and optimal prediction. It also takes into account the fact that a less-than-omniscient agent can never be certain about which theory of the world is correct, yet must still make decisions by using some theory of the world.

We describe methods for learning probability models—primarily Bayesian networks—in Sections 20.2 and 20.3. Section 20.4 looks at learning methods that store and recall specific instances. Section 20.5 covers **neural network** learning and Section 20.6 introduces **kernel machines**. Some of the material in this chapter is fairly mathematical (requiring a basic understanding of multivariate calculus), although the general lessons can be understood without plunging into the details. It may benefit the reader at this point to review the material in Chapters 13 and 14 and to peek at the mathematical background in Appendix A.

20.1 STATISTICAL LEARNING

The key concepts in this chapter, just as in Chapter 18, are **data** and **hypotheses**. Here, the data are **evidence**—that is, instantiations of some or all of the random variables describing the domain. The hypotheses are probabilistic theories of how the domain works, including logical theories as a special case.

Let us consider a *very* simple example. Our favorite Surprise candy comes in two flavors: cherry (yum) and lime (ugh). The candy manufacturer has a peculiar sense of humor and wraps each piece of candy in the same opaque wrapper, regardless of flavor. The candy is sold in very large bags, of which there are known to be five kinds—again, indistinguishable from the outside:

h_1: 100% cherry
h_2: 75% cherry + 25% lime
h_3: 50% cherry + 50% lime
h_4: 25% cherry + 75% lime
h_5: 100% lime

Given a new bag of candy, the random variable H (for *hypothesis*) denotes the type of the bag, with possible values h_1 through h_5. H is not directly observable, of course. As the pieces of candy are opened and inspected, data are revealed—$D_1, D_2, \ldots, D_N$, where each D_i is a random variable with possible values *cherry* and *lime*. The basic task faced by the agent is to predict the flavor of the next piece of candy.[1] Despite its apparent triviality, this scenario serves to introduce many of the major issues. The agent really does need to infer a theory of its world, albeit a very simple one.

BAYESIAN LEARNING

Bayesian learning simply calculates the probability of each hypothesis, given the data, and makes predictions on that basis. That is, the predictions are made by using *all* the hypotheses, weighted by their probabilities, rather than by using just a single "best" hypothesis. In this way, learning is reduced to probabilistic inference. Let $\mathbf{D}$ represent all the data, with observed value $\mathbf{d}$; then the probability of each hypothesis is obtained by Bayes' rule:

$$P(h_i|\mathbf{d}) = \alpha P(\mathbf{d}|h_i)P(h_i) \; . \tag{20.1}$$

Now, suppose we want to make a prediction about an unknown quantity X. Then we have

$$\mathbf{P}(X|\mathbf{d}) = \sum_i \mathbf{P}(X|\mathbf{d}, h_i)\mathbf{P}(h_i|\mathbf{d}) = \sum_i \mathbf{P}(X|h_i)P(h_i|\mathbf{d}) \; , \tag{20.2}$$

where we have assumed that each hypothesis determines a probability distribution over X. This equation shows that predictions are weighted averages over the predictions of the individual hypotheses. The hypotheses themselves are essentially "intermediaries" between the raw data and the predictions. The key quantities in the Bayesian approach are the **hypothesis prior**, $P(h_i)$, and the **likelihood** of the data under each hypothesis, $P(\mathbf{d}|h_i)$.

HYPOTHESIS PRIOR

LIKELIHOOD

For our candy example, we will assume for the time being that the prior distribution over $h_1, \ldots, h_5$ is given by $\langle 0.1, 0.2, 0.4, 0.2, 0.1 \rangle$, as advertised by the manufacturer. The likelihood of the data is calculated under the assumption that the observations are **i.i.d.**—that is, independently and identically distributed—so that

I.I.D.

$$P(\mathbf{d}|h_i) = \prod_j P(d_j|h_i) \; . \tag{20.3}$$

For example, suppose the bag is really an all-lime bag (h_5) and the first 10 candies are all lime; then $P(\mathbf{d}|h_3)$ is 0.5^{10}, because half the candies in an h_3 bag are lime.[2] Figure 20.1(a) shows how the posterior probabilities of the five hypotheses change as the sequence of 10 lime candies is observed. Notice that the probabilities start out at their prior values, so h_3 is initially the most likely choice and remains so after 1 lime candy is unwrapped. After 2

[1] Statistically sophisticated readers will recognize this scenario as a variant of the **urn-and-ball** setup. We find urns and balls less compelling than candy; furthermore, candy lends itself to other tasks, such as deciding whether to trade the bag with a friend—see Exercise 20.3.

[2] We stated earlier that the bags of candy are very large; otherwise, the i.i.d. assumption fails to hold. Technically, it is more correct (but less hygienic) to rewrap each candy after inspection and return it to the bag.

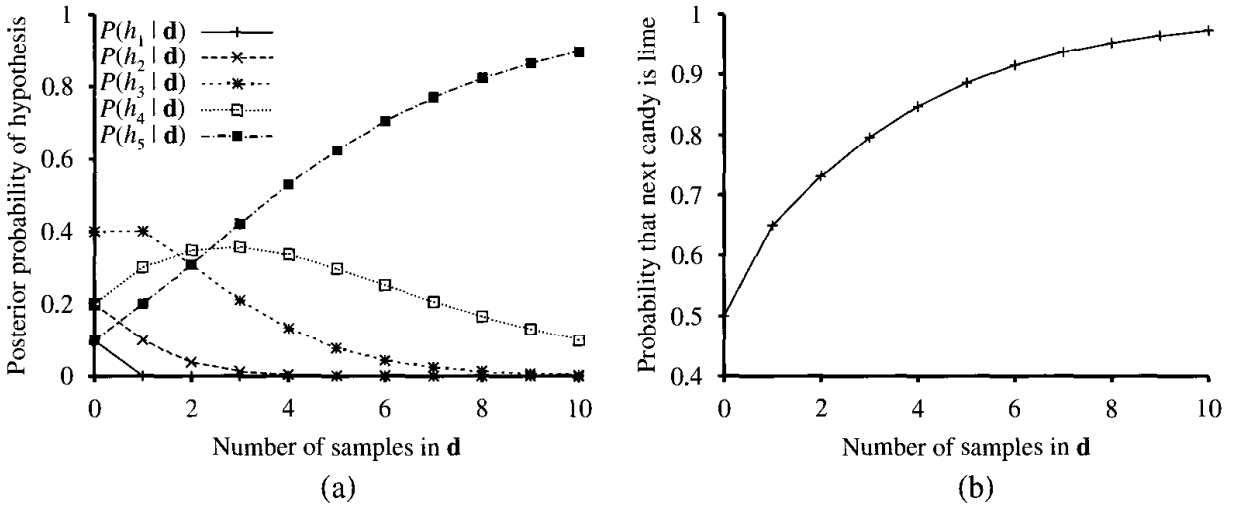


Figure 20.1 (a) Posterior probabilities $P(h_i|d_1, \ldots, d_N)$ from Equation (20.1). The number of observations N ranges from 1 to 10, and each observation is of a lime candy. (b) Bayesian prediction $P(d_{N+1} = lime|d_1, \ldots, d_N)$ from Equation (20.2).

lime candies are unwrapped, h_4 is most likely; after 3 or more, h_5 (the dreaded all-lime bag) is the most likely. After 10 in a row, we are fairly certain of our fate. Figure 20.1(b) shows the predicted probability that the next candy is lime, based on Equation (20.2). As we would expect, it increases monotonically toward 1.

The example shows that *the true hypothesis eventually dominates the Bayesian prediction.* This is characteristic of Bayesian learning. For any fixed prior that does not rule out the true hypothesis, the posterior probability of any false hypothesis will eventually vanish, simply because the probability of generating "uncharacteristic" data indefinitely is vanishingly small. (This point is analogous to one made in the discussion of PAC learning in Chapter 18.) More importantly, the Bayesian prediction is *optimal*, whether the data set be small or large. Given the hypothesis prior, any other prediction will be correct less often.

The optimality of Bayesian learning comes at a price, of course. For real learning problems, the hypothesis space is usually very large or infinite, as we saw in Chapter 18. In some cases, the summation in Equation (20.2) (or integration, in the continuous case) can be carried out tractably, but in most cases we must resort to approximate or simplified methods.

MAXIMUM A POSTERIORI

A very common approximation—one that is usually adopted in science—is to make predictions based on a single *most probable* hypothesis—that is, an h_i that maximizes $P(h_i|\mathbf{d})$. This is often called a **maximum a posteriori** or MAP (pronounced "em-ay-pee") hypothesis. Predictions made according to an MAP hypothesis h_{MAP} are approximately Bayesian to the extent that $\mathbf{P}(X|\mathbf{d}) \approx \mathbf{P}(X|h_{\text{MAP}})$. In our candy example, $h_{\text{MAP}} = h_5$ after three lime candies in a row, so the MAP learner then predicts that the fourth candy is lime with probability 1.0—a much more dangerous prediction than the Bayesian prediction of 0.8 shown in Figure 20.1. As more data arrive, the MAP and Bayesian predictions become closer, because the competitors to the MAP hypothesis become less and less probable. Although our example doesn't show it, finding MAP hypotheses is often much easier than Bayesian learn-

ing, because it requires solving an optimization problem instead of a large summation (or integration) problem. We will see examples of this later in the chapter.

In both Bayesian learning and MAP learning, the hypothesis prior $P(h_i)$ plays an important role. We saw in Chapter 18 that **overfitting** can occur when the hypothesis space is too expressive, so that it contains many hypotheses that fit the data set well. Rather than placing an arbitrary limit on the hypotheses to be considered, Bayesian and MAP learning methods use the prior to *penalize complexity*. Typically, more complex hypotheses have a lower prior probability—in part because there are usually many more complex hypotheses than simple hypotheses. On the other hand, more complex hypotheses have a greater capacity to fit the data. (In the extreme case, a lookup table can reproduce the data exactly with probability 1.) Hence, the hypothesis prior embodies a trade-off between the complexity of a hypothesis and its degree of fit to the data.

We can see the effect of this trade-off most clearly in the logical case, where H contains only *deterministic* hypotheses. In that case, $P(\mathbf{d}|h_i)$ is 1 if h_i is consistent and 0 otherwise. Looking at Equation (20.1), we see that h_{MAP} will then be the *simplest logical theory that is consistent with the data.* Therefore, maximum *a posteriori* learning provides a natural embodiment of Ockham's razor.

Another insight into the trade-off between complexity and degree of fit is obtained by taking the logarithm of Equation (20.1). Choosing h_{MAP} to maximize $P(\mathbf{d}|h_i)P(h_i)$ is equivalent to minimizing

$$-\log_2 P(\mathbf{d}|h_i) - \log_2 P(h_i) \ .$$

Using the connection between information encoding and probability that we introduced in Chapter 18, we see that the $-\log_2 P(h_i)$ term equals the number of bits required to specify the hypothesis h_i. Furthermore, $-\log_2 P(\mathbf{d}|h_i)$ is the additional number of bits required to specify the data, given the hypothesis. (To see this, consider that no bits are required if the hypothesis predicts the data exactly—as with h_5 and the string of lime candies—and $\log_2 1 = 0$.) Hence, MAP learning is choosing the hypothesis that provides maximum *compression* of the data. The same task is addressed more directly by the **minimum description length**, or MDL, learning method, which attempts to minimize the size of hypothesis and data encodings rather than work with probabilities.

MINIMUM DESCRIPTION LENGTH

A final simplification is provided by assuming a **uniform** prior over the space of hypotheses. In that case, MAP learning reduces to choosing an h_i that maximizes $P(\mathbf{d}|H_i)$. This is called a **maximum-likelihood** (ML) hypothesis, h_{ML}. Maximum-likelihood learning is very common in statistics, a discipline in which many researchers distrust the subjective nature of hypothesis priors. It is a reasonable approach when there is no reason to prefer one hypothesis over another *a priori*—for example, when all hypotheses are equally complex. It provides a good approximation to Bayesian and MAP learning when the data set is large, because the data swamps the prior distribution over hypotheses, but it has problems (as we shall see) with small data sets.

MAXIMUM-LIKELIHOOD

20.2 LEARNING WITH COMPLETE DATA

PARAMETER LEARNING
COMPLETE DATA

Our development of statistical learning methods begins with the simplest task: **parameter learning** with **complete data**. A parameter learning task involves finding the numerical parameters for a probability model whose structure is fixed. For example, we might be interested in learning the conditional probabilities in a Bayesian network with a given structure. Data are complete when each data point contains values for every variable in the probability model being learned. Complete data greatly simplify the problem of learning the parameters of a complex model. We will also look briefly at the problem of learning structure.

Maximum-likelihood parameter learning: Discrete models

Suppose we buy a bag of lime and cherry candy from a new manufacturer whose lime–cherry proportions are completely unknown—that is, the fraction could be anywhere between 0 and 1. In that case, we have a continuum of hypotheses. The **parameter** in this case, which we call θ, is the proportion of cherry candies, and the hypothesis is h_θ. (The proportion of limes is just $1-\theta$.) If we assume that all proportions are equally likely *a priori*, then a maximum-likelihood approach is reasonable. If we model the situation with a Bayesian network, we need just one random variable, *Flavor* (the flavor of a randomly chosen candy from the bag). It has values *cherry* and *lime*, where the probability of *cherry* is θ (see Figure 20.2(a)). Now suppose we unwrap N candies, of which c are cherries and $\ell = N - c$ are limes. According to Equation (20.3), the likelihood of this particular data set is

$$P(\mathbf{d}|h_\theta) = \prod_{j=1}^{N} P(d_j|h_\theta) = \theta^c \cdot (1-\theta)^\ell \,.$$

LOG LIKELIHOOD

The maximum-likelihood hypothesis is given by the value of θ that maximizes this expression. The same value is obtained by maximizing the **log likelihood**,

$$L(\mathbf{d}|h_\theta) = \log P(\mathbf{d}|h_\theta) = \sum_{j=1}^{N} \log P(d_j|h_\theta) = c\log\theta + \ell\log(1-\theta) \,.$$

(By taking logarithms, we reduce the product to a sum over the data, which is usually easier to maximize.) To find the maximum-likelihood value of θ, we differentiate L with respect to θ and set the resulting expression to zero:

$$\frac{dL(\mathbf{d}|h_\theta)}{d\theta} = \frac{c}{\theta} - \frac{\ell}{1-\theta} = 0 \qquad \Rightarrow \qquad \theta = \frac{c}{c+\ell} = \frac{c}{N} \,.$$

In English, then, the maximum-likelihood hypothesis h_{ML} asserts that the actual proportion of cherries in the bag is equal to the observed proportion in the candies unwrapped so far!

It appears that we have done a lot of work to discover the obvious. In fact, though, we have laid out one standard method for maximum-likelihood parameter learning:

1. Write down an expression for the likelihood of the data as a function of the parameter(s).
2. Write down the derivative of the log likelihood with respect to each parameter.
3. Find the parameter values such that the derivatives are zero.

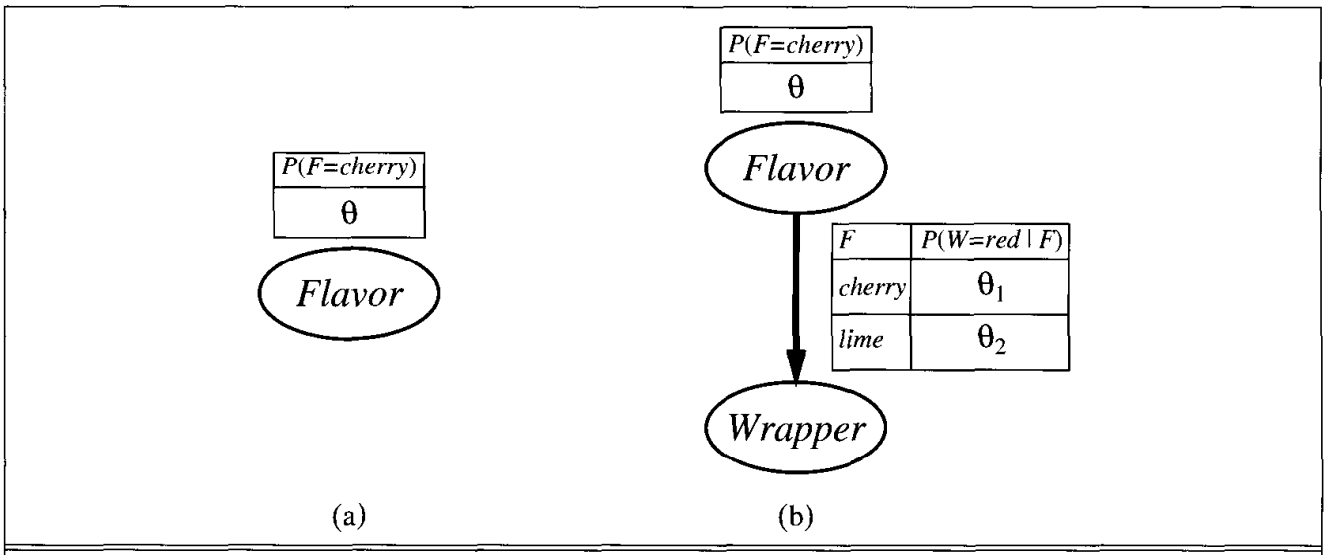


Figure 20.2 (a) Bayesian network model for the case of candies with an unknown proportion of cherries and limes. (b) Model for the case where the wrapper color depends (probabilistically) on the candy flavor.

The trickiest step is usually the last. In our example, it was trivial, but we will see that in many cases we need to resort to iterative solution algorithms or other numerical optimization techniques, as described in Chapter 4. The example also illustrates a significant problem with maximum-likelihood learning in general: *when the data set is small enough that some events have not yet been observed—for instance, no cherry candies—the maximum likelihood hypothesis assigns zero probability to those events.* Various tricks are used to avoid this problem, such as initializing the counts for each event to 1 instead of zero.

Let us look at another example. Suppose this new candy manufacturer wants to give a little hint to the consumer and uses candy wrappers colored red and green. The *Wrapper* for each candy is selected *probabilistically*, according to some unknown conditional distribution, depending on the flavor. The corresponding probability model is shown in Figure 20.2(b). Notice that it has three parameters: θ, θ_1, and θ_2. With these parameters, the likelihood of seeing, say, a cherry candy in a green wrapper can be obtained from the standard semantics for Bayesian networks (page 495):

$$\begin{aligned} &P(Flavor = cherry, \ Wrapper = green | h_{\theta,\theta_1,\theta_2}) \\ &= P(Flavor = cherry | h_{\theta,\theta_1,\theta_2}) P(\ Wrapper = green | Flavor = cherry, h_{\theta,\theta_1,\theta_2}) \\ &= \theta \cdot (1 - \theta_1) \ . \end{aligned}$$

Now, we unwrap N candies, of which c are cherries and ℓ are limes. The wrapper counts are as follows: r_c of the cherries have red wrappers and g_c have green, while r_ℓ of the limes have red and g_ℓ have green. The likelihood of the data is given by

$$P(\mathbf{d} | h_{\theta,\theta_1,\theta_2}) = \theta^c (1 - \theta)^\ell \cdot \theta_1^{r_c} (1 - \theta_1)^{g_c} \cdot \theta_2^{r_\ell} (1 - \theta_2)^{g_\ell} \ .$$

This looks pretty horrible, but taking logarithms helps:

$$L = [c \log \theta + \ell \log(1 - \theta)] + [r_c \log \theta_1 + g_c \log(1 - \theta_1)] + [r_\ell \log \theta_2 + g_\ell \log(1 - \theta_2)] \ .$$

The benefit of taking logs is clear: the log likelihood is the sum of three terms, each of which contains a single parameter. When we take derivatives with respect to each parameter and set

them to zero, we get three independent equations, each containing just one parameter:

$$\begin{array}{lll} \frac{\partial L}{\partial \theta} = \frac{c}{\theta} - \frac{\ell}{1-\theta} = 0 & \Rightarrow & \theta = \frac{c}{c+\ell} \\ \frac{\partial L}{\partial \theta_1} = \frac{r_c}{\theta_1} - \frac{g_c}{1-\theta_1} = 0 & \Rightarrow & \theta_1 = \frac{r_c}{r_c+g_c} \\ \frac{\partial L}{\partial \theta_2} = \frac{r_\ell}{\theta_2} - \frac{g_\ell}{1-\theta_2} = 0 & \Rightarrow & \theta_2 = \frac{r_\ell}{r_\ell+g_\ell} \, . \end{array}$$

The solution for θ is the same as before. The solution for θ_1, the probability that a cherry candy has a red wrapper, is the observed fraction of cherry candies with red wrappers, and similarly for θ_2.

These results are very comforting, and it is easy to see that they can be extended to any Bayesian network whose conditional probabilities are represented as tables. The most important point is that, *with complete data, the maximum-likelihood parameter learning problem for a Bayesian network decomposes into separate learning problems, one for each parameter.*[3] The second point is that the parameter values for a variable, given its parents, are just the observed frequencies of the variable values for each setting of the parent values. As before, we must be careful to avoid zeroes when the data set is small.

Naive Bayes models

Probably the most common Bayesian network model used in machine learning is the **naive Bayes** model. In this model, the "class" variable C (which is to be predicted) is the root and the "attribute" variables X_i are the leaves. The model is "naive" because it assumes that the attributes are conditionally independent of each other, given the class. (The model in Figure 20.2(b) is a naive Bayes model with just one attribute.) Assuming Boolean variables, the parameters are

$$\theta = P(C = true), \theta_{i1} = P(X_i = true | C = true), \theta_{i2} = P(X_i = true | C = false).$$

The maximum-likelihood parameter values are found in exactly the same way as for Figure 20.2(b). Once the model has been trained in this way, it can be used to classify new examples for which the class variable C is unobserved. With observed attribute values $x_1, \ldots, x_n$, the probability of each class is given by

$$\mathbf{P}(C|x_1, \ldots, x_n) = \alpha \, \mathbf{P}(C) \prod_i \mathbf{P}(x_i|C) \, .$$

A deterministic prediction can be obtained by choosing the most likely class. Figure 20.3 shows the learning curve for this method when it is applied to the restaurant problem from Chapter 18. The method learns fairly well but not as well as decision-tree learning; this is presumably because the true hypothesis—which is a decision tree—is not representable exactly using a naive Bayes model. Naive Bayes learning turns out to do surprisingly well in a wide range of applications; the boosted version (Exercise 20.5) is one of the most effective general-purpose learning algorithms. Naive Bayes learning scales well to very large problems: with n Boolean attributes, there are just $2n + 1$ parameters, and *no search is required to find h_{ML}, the maximum-likelihood naive Bayes hypothesis.* Finally, naive Bayes learning has no difficulty with noisy data and can give probabilistic predictions when appropriate.

[3] See Exercise 20.7 for the nontabulated case, where each parameter affects several conditional probabilities.

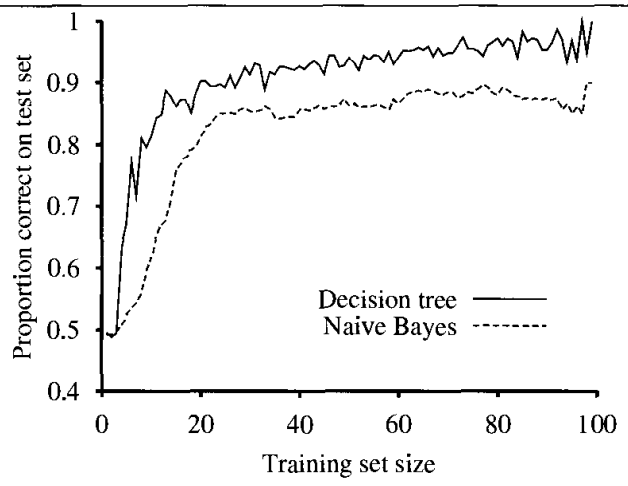


Figure 20.3 The learning curve for naive Bayes learning applied to the restaurant problem from Chapter 18; the learning curve for decision-tree learning is shown for comparison.

Maximum-likelihood parameter learning: Continuous models

Continuous probability models such as the **linear-Gaussian** model were introduced in Section 14.3. Because continuous variables are ubiquitous in real-world applications, it is important to know how to learn continuous models from data. The principles for maximum-likelihood learning are identical to those of the discrete case.

Let us begin with a very simple case: learning the parameters of a Gaussian density function on a single variable. That is, the data are generated as follows:

$$P(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}} .$$

The parameters of this model are the mean μ and the standard deviation σ. (Notice that the normalizing "constant" depends on σ, so we cannot ignore it.) Let the observed values be $x_1, \ldots, x_N$. Then the log likelihood is

$$L = \sum_{j=1}^{N} \log \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x_j-\mu)^2}{2\sigma^2}} = N(-\log\sqrt{2\pi} - \log\sigma) - \sum_{j=1}^{N} \frac{(x_j-\mu)^2}{2\sigma^2} .$$

Setting the derivatives to zero as usual, we obtain

$$\begin{aligned} \frac{\partial L}{\partial \mu} &= -\tfrac{1}{\sigma^2}\textstyle\sum_{j=1}^{N}(x_j-\mu) = 0 \quad &\Rightarrow \quad \mu &= \frac{\sum_j x_j}{N} \\ \frac{\partial L}{\partial \sigma} &= -\tfrac{N}{\sigma} + \tfrac{1}{\sigma^3}\textstyle\sum_{j=1}^{N}(x_j-\mu)^2 = 0 \quad &\Rightarrow \quad \sigma &= \sqrt{\frac{\sum_j (x_j-\mu)^2}{N}} . \end{aligned} \tag{20.4}$$

That is, the maximum-likelihood value of the mean is the sample average and the maximum-likelihood value of the standard deviation is the square root of the sample variance. Again, these are comforting results that confirm "commonsense" practice.

Now consider a linear Gaussian model with one continuous parent X and a continuous child Y. As explained on page 502, Y has a Gaussian distribution whose mean depends linearly on the value of X and whose standard deviation is fixed. To learn the conditional

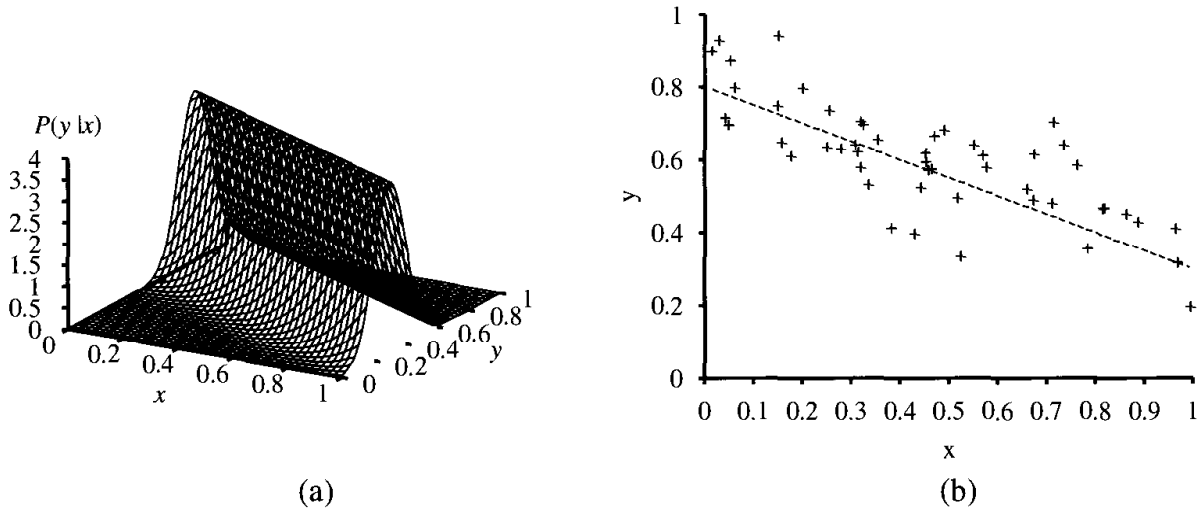


Figure 20.4 (a) A linear Gaussian model described as $y = \theta_1 x + \theta_2$ plus Gaussian noise with fixed variance. (b) A set of 50 data points generated from this model.

distribution $P(Y|X)$, we can maximize the conditional likelihood

$$P(y|x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(y-(\theta_1 x+\theta_2))^2}{2\sigma^2}} . \tag{20.5}$$

Here, the parameters are θ_1, θ_2, and σ. The data are a collection of (x_j, y_j) pairs, as illustrated in Figure 20.4. Using the usual methods (Exercise 20.6), we can find the maximum-likelihood values of the parameters. Here, we want to make a different point. If we consider just the parameters θ_1 and θ_2 that define the linear relationship between x and y, it becomes clear that maximizing the log likelihood with respect to these parameters is the same as *minimizing* the numerator in the exponent of Equation (20.5):

$$E = \sum_{j=1}^{N} (y_j - (\theta_1 x_j + \theta_2))^2 .$$

ERROR

SUM OF SQUARED ERRORS

LINEAR REGRESSION

The quantity $(y_j - (\theta_1 x_j + \theta_2))$ is the **error** for (x_j, y_j)—that is, the difference between the actual value y_j and the predicted value $(\theta_1 x_j + \theta_2)$—so E is the well-known **sum of squared errors**. This is the quantity that is minimized by the standard **linear regression** procedure. Now we can understand why: minimizing the sum of squared errors gives the maximum-likelihood straight-line model, *provided that the data are generated with Gaussian noise of fixed variance.*

Bayesian parameter learning

Maximum-likelihood learning gives rise to some very simple procedures, but it has some serious deficiencies with small data sets. For example, after seeing one cherry candy, the maximum-likelihood hypothesis is that the bag is 100% cherry (i.e., $\theta = 1.0$). Unless one's hypothesis prior is that bags must be either all cherry or all lime, this is not a reasonable conclusion. The Bayesian approach to parameter learning places a hypothesis prior over the possible values of the parameters and updates this distribution as data arrive.

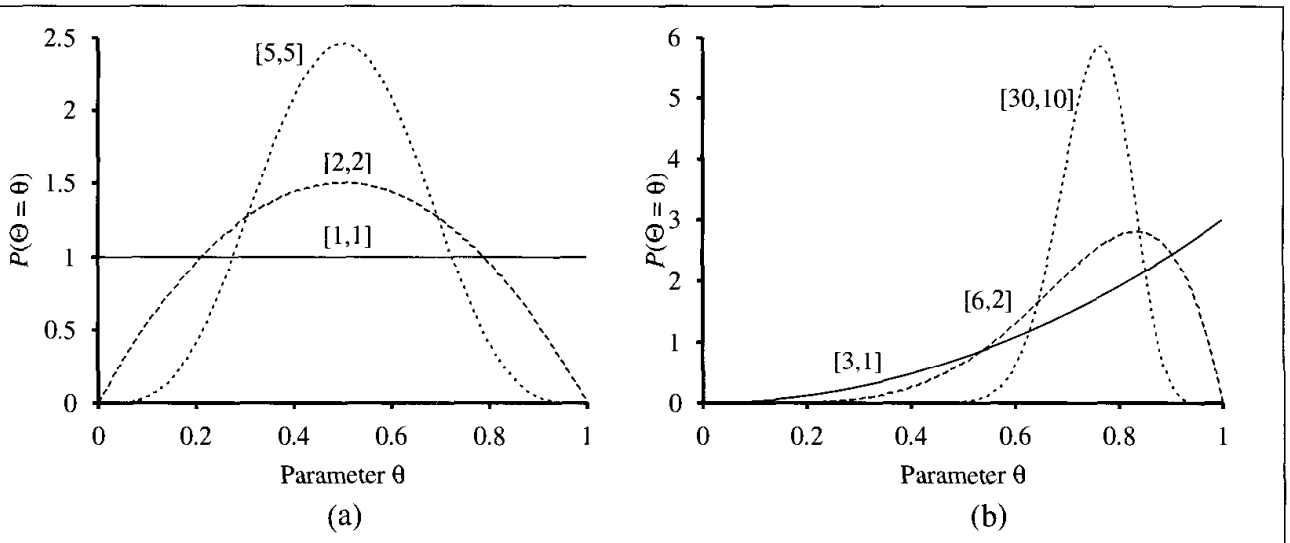


Figure 20.5 Examples of the beta$[a, b]$ distribution for different values of $[a, b]$.

The candy example in Figure 20.2(a) has one parameter, θ: the probability that a randomly selected piece of candy is cherry flavored. In the Bayesian view, θ is the (unknown) value of a random variable Θ; the hypothesis prior is just the prior distribution $\mathbf{P}(\Theta)$. Thus, $P(\Theta = \theta)$ is the prior probability that the bag has a fraction θ of cherry candies.

If the parameter θ can be any value between 0 and 1, then $\mathbf{P}(\Theta)$ must be a continuous distribution that is nonzero only between 0 and 1 and that integrates to 1. The uniform density $P(\theta) = U[0, 1](\theta)$ is one candidate. (See Chapter 13.) It turns out that the uniform density is a member of the family of **beta distributions**. Each beta distribution is defined by two **hyperparameters**[4] a and b such that

BETA DISTRIBUTIONS

HYPERPARAMETER

$$\text{beta}[a, b](\theta) = \alpha\, \theta^{a-1}(1-\theta)^{b-1}\,, \tag{20.6}$$

for θ in the range $[0, 1]$. The normalization constant α depends on a and b. (See Exercise 20.8.) Figure 20.5 shows what the distribution looks like for various values of a and b. The mean value of the distribution is $a/(a + b)$, so larger values of a suggest a belief that Θ is closer to 1 than to 0. Larger values of $a + b$ make the distribution more peaked, suggesting greater certainty about the value of Θ. Thus, the beta family provides a useful range of possibilities for the hypothesis prior.

Besides its flexibility, the beta family has another wonderful property: if Θ has a prior beta$[a, b]$, then, after a data point is observed, the posterior distribution for Θ is also a beta distribution. The beta family is called the **conjugate prior** for the family of distributions for a Boolean variable.[5] Let's see how this works. Suppose we observe a cherry candy; then

CONJUGATE PRIOR

$$\begin{aligned} P(\theta|D_1 = cherry) &= \alpha\, P(D_1 = cherry|\theta)P(\theta) \\ &= \alpha'\, \theta \cdot \text{beta}[a, b](\theta) = \alpha'\, \theta \cdot \theta^{a-1}(1-\theta)^{b-1} \\ &= \alpha'\, \theta^{a}(1-\theta)^{b-1} = \text{beta}[a+1, b](\theta)\,. \end{aligned}$$

[4] They are called hyperparameters because they parameterize a distribution over θ, which is itself a parameter.

[5] Other conjugate priors include the **Dirichlet** family for the parameters of a discrete multivalued distribution and the **Normal–Wishart** family for the parameters of a Gaussian distribution. See Bernardo and Smith (1994).

Thus, after seeing a cherry candy, we simply increment the a parameter to get the posterior; similarly, after seeing a lime candy, we increment the b parameter. Thus, we can view the a and b hyperparameters as **virtual counts**, in the sense that a prior $\text{beta}[a, b]$ behaves exactly as if we had started out with a uniform prior $\text{beta}[1, 1]$ and seen $a - 1$ actual cherry candies and $b - 1$ actual lime candies.

VIRTUAL COUNTS

By examining a sequence of beta distributions for increasing values of a and b, keeping the proportions fixed, we can see vividly how the posterior distribution over the parameter Θ changes as data arrive. For example, suppose the actual bag of candy is 75% cherry. Figure 20.5(b) shows the sequence $\text{beta}[3, 1]$, $\text{beta}[6, 2]$, $\text{beta}[30, 10]$. Clearly, the distribution is converging to a narrow peak around the true value of Θ. For large data sets, then, Bayesian learning (at least in this case) converges to give the same results as maximum-likelihood learning.

The network in Figure 20.2(b) has three parameters, θ, θ_1, and θ_2, where θ_1 is the probability of a red wrapper on a cherry candy and θ_2 is the probability of a red wrapper on a lime candy. The Bayesian hypothesis prior must cover all three parameters—that is, we need to specify $\mathbf{P}(\Theta, \Theta_1, \Theta_2)$. Usually, we assume **parameter independence**:

PARAMETER INDEPENDENCE

$$\mathbf{P}(\Theta, \Theta_1, \Theta_2) = \mathbf{P}(\Theta)\mathbf{P}(\Theta_1)\mathbf{P}(\Theta_2) \ .$$

With this assumption, each parameter can have its own beta distribution that is updated separately as data arrive.

Once we have the idea that unknown parameters can be represented by random variables such as Θ, it is natural to incorporate them into the Bayesian network itself. To do this, we also need to make copies of the variables describing each instance. For example, if we have observed three candies then we need $Flavor_1$, $Flavor_2$, $Flavor_3$ and $Wrapper_1$, $Wrapper_2$, $Wrapper_3$. The parameter variable Θ determines the probability of each $Flavor_i$ variable:

$$P(Flavor_i = cherry | \Theta = \theta) = \theta \ .$$

Similarly, the wrapper probabilities depend on Θ_1 and Θ_2, For example,

$$P(Wrapper_i = red | Flavor_i = cherry, \Theta_1 = \theta_1) = \theta_1 \ .$$

Now, the entire Bayesian learning process can be formulated as an *inference* problem in a suitably constructed Bayes net, as shown in Figure 20.6. Prediction for a new instance is done simply by adding new instance variables to the network, some of which are queried. This formulation of learning and prediction makes it clear that Bayesian learning requires no extra "principles of learning." Furthermore, *there is, in essence, just one learning algorithm,* i.e., the inference algorithm for Bayesian networks.

Learning Bayes net structures

So far, we have assumed that the structure of the Bayes net is given and we are just trying to learn the parameters. The structure of the network represents basic causal knowledge about the domain that is often easy for an expert, or even a naive user, to supply. In some cases, however, the causal model may be unavailable or subject to dispute—for example, certain corporations have long claimed that smoking does not cause cancer—so it is important to

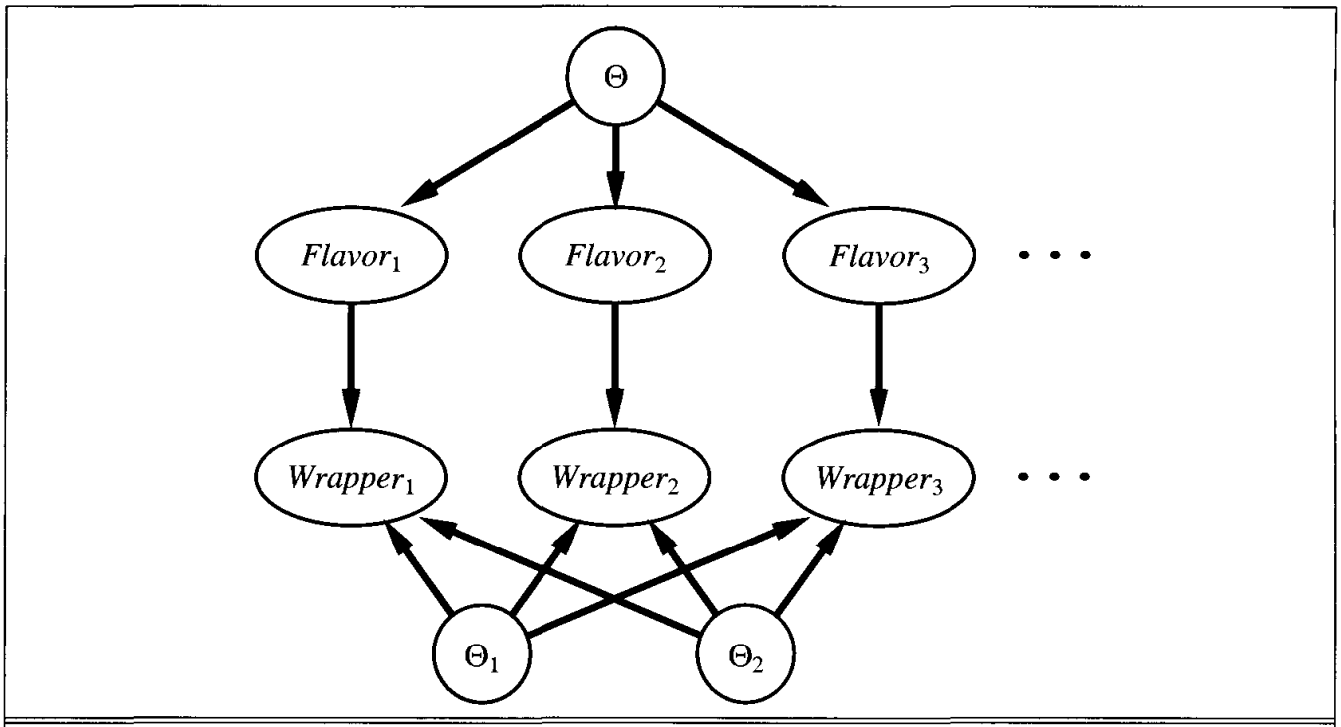


Figure 20.6 A Bayesian network that corresponds to a Bayesian learning process. Posterior distributions for the parameter variables Θ, Θ_1, and Θ_2 can be inferred from their prior distributions and the evidence in the $Flavor_i$ and $Wrapper_i$ variables.

understand how the structure of a Bayes net can be learned from data. At present, structural learning algorithms are in their infancy, so we will give only a brief sketch of the main ideas.

The most obvious approach is to *search* for a good model. We can start with a model containing no links and begin adding parents for each node, fitting the parameters with the methods we have just covered and measuring the accuracy of the resulting model. Alternatively, we can start with an initial guess at the structure and use hill-climbing or simulated annealing search to make modifications, retuning the parameters after each change in the structure. Modifications can include reversing, adding, or deleting arcs. We must not introduce cycles in the process, so many algorithms assume that an ordering is given for the variables, and that a node can have parents only among those nodes that come earlier in the ordering (just as in the construction process Chapter 14). For full generality, we also need to search over possible orderings.

There are two alternative methods for deciding when a good structure has been found. The first is to test whether the conditional independence assertions implicit in the structure are actually satisfied in the data. For example, the use of a naive Bayes model for the restaurant problem assumes that

$$\mathbf{P}(Fri/Sat, Bar \mid WillWait) = \mathbf{P}(Fri/Sat \mid WillWait)\mathbf{P}(Bar \mid WillWait)$$

and we can check in the data that the same equation holds between the corresponding conditional frequencies. Now, even if the structure describes the true causal nature of the domain, statistical fluctuations in the data set mean that the equation will never be satisfied *exactly*, so we need to perform a suitable statistical test to see if there is sufficient evidence that the independence hypothesis is violated. The complexity of the resulting network will depend

on the threshold used for this test—the stricter the independence test, the more links will be added and the greater the danger of overfitting.

An approach more consistent with the ideas in this chapter is to the degree to which the proposed model explains the data (in a probabilistic sense). We must be careful how we measure this, however. If we just try to find the maximum-likelihood hypothesis, we will end up with a fully connected network, because adding more parents to a node cannot decrease the likelihood (Exercise 20.9). We are forced to penalize model complexity in some way. The MAP (or MDL) approach simply subtracts a penalty from the likelihood of each structure (after parameter tuning) before comparing different structures. The Bayesian approach places a joint prior over structures and parameters. There are usually far too many structures to sum over (superexponential in the number of variables), so most practitioners use MCMC to sample over structures.

Penalizing complexity (whether by MAP or Bayesian methods) introduces an important connection between the optimal structure and the nature of the representation for the conditional distributions in the network. With tabular distributions, the complexity penalty for a node's distribution grows exponentially with the number of parents, but with, say, noisy-OR distributions, it grows only linearly. This means that learning with noisy-OR (or other compactly parameterized) models tends to produce learned structures with more parents than does learning with tabular distributions.

20.3 Learning with Hidden Variables: The EM Algorithm

LATENT VARIABLES

The preceding section dealt with the fully observable case. Many real-world problems have **hidden variables** (sometimes called **latent variables**) which are not observable in the data that are available for learning. For example, medical records often include the observed symptoms, the treatment applied, and perhaps the outcome of the treatment, but they seldom contain a direct observation of the disease itself![6] One might ask, "If the disease is not observed, why not construct a model without it?" The answer appears in Figure 20.7, which shows a small, fictitious diagnostic model for heart disease. There are three observable predisposing factors and three observable symptoms (which are too depressing to name). Assume that each variable has three possible values (e.g., *none*, *moderate*, and *severe*). Removing the hidden variable from the network in (a) yields the network in (b); the total number of parameters increases from 78 to 708. Thus, *latent variables can dramatically reduce the number of parameters required to specify a Bayesian network.* This, in turn, can dramatically reduce the amount of data needed to learn the parameters.

Hidden variables are important, but they do complicate the learning problem. In Figure 20.7(a), for example, it is not obvious how to learn the conditional distribution for *HeartDisease*, given its parents, because we do not know the value of *HeartDisease* in each case; the same problem arises in learning the distributions for the symptoms. This section

[6] Some records contain the diagnosis suggested by the physician, but this is a causal consequence of the symptoms, which are in turn caused by the disease.

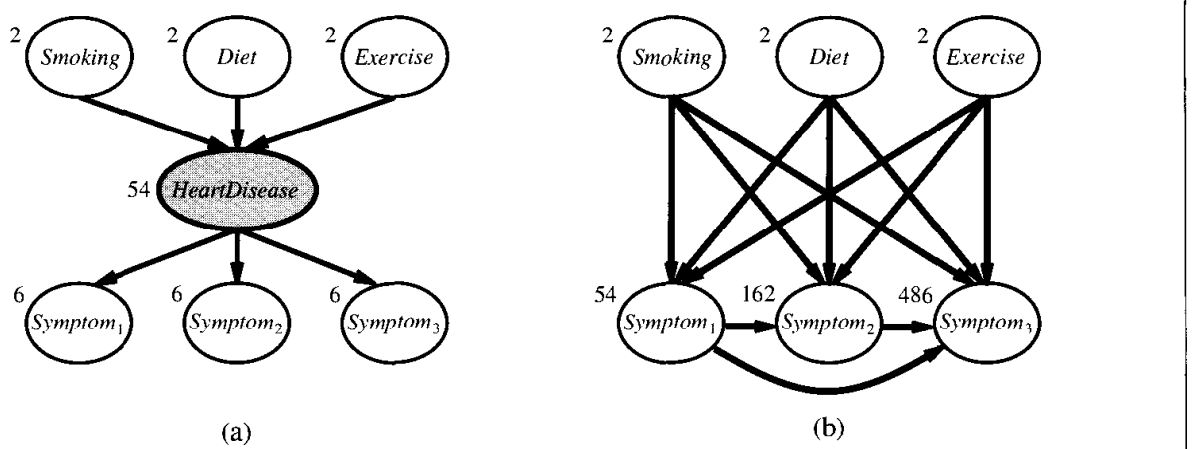


Figure 20.7 (a) A simple diagnostic network for heart disease, which is assumed to be a hidden variable. Each variable has three possible values and is labeled with the number of independent parameters in its conditional distribution; the total number is 78. (b) The equivalent network with *HeartDisease* removed. Note that the symptom variables are no longer conditionally independent given their parents. This network requires 708 parameters.

EXPECTATION-MAXIMIZATION

describes an algorithm called **expectation–maximization**, or EM, that solves this problem in a very general way. We will show three examples and then provide a general description. The algorithm seems like magic at first, but once the intuition has been developed, one can find applications for EM in a huge range of learning problems.

Unsupervised clustering: Learning mixtures of Gaussians

UNSUPERVISED CLUSTERING

Unsupervised clustering is the problem of discerning multiple categories in a collection of objects. The problem is unsupervised because the category labels are not given. For example, suppose we record the spectra of a hundred thousand stars; are there different *types* of stars revealed by the spectra, and, if so, how many and what are their characteristics? We are all familiar with terms such as "red giant" and "white dwarf," but the stars do not carry these labels on their hats—astronomers had to perform unsupervised clustering to identify these categories. Other examples include the identification of species, genera, orders, and so on in the Linnæan taxonomy of organisms and the creation of natural kinds to categorize ordinary objects (see Chapter 10).

Unsupervised clustering begins with data. Figure 20.8(a) shows 500 data points, each of which specifies the values of two continuous attributes. The data points might correspond to stars, and the attributes might correspond to spectral intensities at two particular frequencies. Next, we need to understand what kind of probability distribution might have generated the data. Clustering presumes that the data are generated from a **mixture distribution**, P. Such a distribution has k **components**, each of which is a distribution in its own right. A data point is generated by first choosing a component and then generating a sample from that component. Let the random variable C denote the component, with values $1, \ldots, k$; then the mixture

MIXTURE DISTRIBUTION

COMPONENT

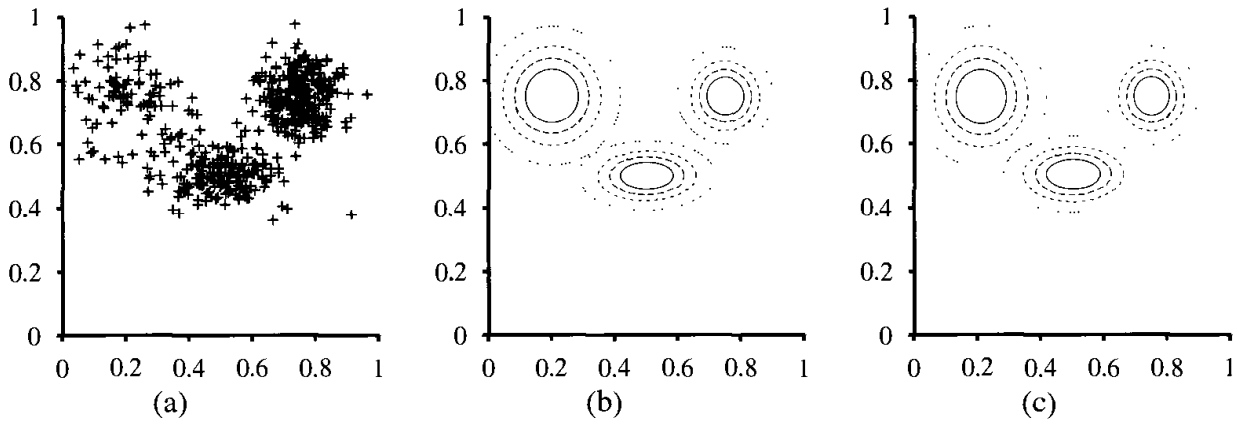


Figure 20.8 (a) 500 data points in two dimensions, suggesting the presence of three clusters. (b) A Gaussian mixture model with three components; the weights (left-to-right) are 0.2, 0.3, and 0.5. The data in (a) were generated from this model. (c) The model reconstructed by EM from the data in (b).

distribution is given by

$$P(\mathbf{x}) = \sum_{i=1}^{k} P(C=i)\, P(\mathbf{x}|C=i)\,,$$

where $\mathbf{x}$ refers to the values of the attributes for a data point. For continuous data, a natural choice for the component distributions is the multivariate Gaussian, which gives the so-called **mixture of Gaussians** family of distributions. The parameters of a mixture of Gaussians are $w_i = P(C=i)$ (the weight of each component), $\boldsymbol{\mu}_i$ (the mean of each component), and $\boldsymbol{\Sigma}_i$ (the covariance of each component). Figure 20.8(b) shows a mixture of three Gaussians; this mixture is in fact the source of the data in (a).

MIXTURE OF GAUSSIANS

The unsupervised clustering problem, then, is to recover a mixture model like the one in Figure 20.8(b) from raw data like that in Figure 20.8(a). Clearly, if we *knew* which component generated each data point, then it would be easy to recover the component Gaussians: we could just select all the data points from a given component and then apply (a multivariate version of) Equation (20.4) for fitting the parameters of a Gaussian to a set of data. On the other hand, if we *knew* the parameters of each component, then we could, at least in a probabilistic sense, assign each data point to a component. The problem is that we know neither the assignments nor the parameters.

The basic idea of EM in this context is to *pretend* that we know the the parameters of the model and then to infer the probability that each data point belongs to each component. After that, we refit the components to the data, where each component is fitted to the entire data set with each point weighted by the probability that it belongs to that component. The process iterates until convergence. Essentially, we are "completing" the data by inferring probability distributions over the hidden variables—which component each data point belongs to—based on the current model. For the mixture of Gaussians, we initialize the mixture model parameters arbitrarily and then iterate the following two steps:

1. E-step: Compute the probabilities $p_{ij} = P(C=i|\mathbf{x}_j)$, the probability that datum $\mathbf{x}_j$ was generated by component i. By Bayes' rule, we have $p_{ij} = \alpha P(\mathbf{x}_j|C=i)P(C=i)$. The term $P(\mathbf{x}_j|C=i)$ is just the probability at $\mathbf{x}_j$ of the ith Gaussian, and the term $P(C=i)$ is just the weight parameter for the ith Gaussian. Define $p_i = \sum_j p_{ij}$.
2. M-step: Compute the new mean, covariance, and component weights as follows:

$$\begin{aligned}\boldsymbol{\mu}_i &\leftarrow \sum_j p_{ij}\mathbf{x}_j / p_i \\ \boldsymbol{\Sigma}_i &\leftarrow \sum_j p_{ij}\mathbf{x}_j\mathbf{x}_j^\top / p_i \\ w_i &\leftarrow p_i \,.\end{aligned}$$

INDICATOR VARIABLE

The E-step, or *expectation* step, can be viewed as computing the expected values p_{ij} of the hidden **indicator variables** Z_{ij}, where Z_{ij} is 1 if datum $\mathbf{x}_j$ was generated by the ith component and 0 otherwise. The M-step, or *maximization* step, finds the new values of the parameters that maximize the log likelihood of the data, given the expected values of the hidden indicator variables.

The final model that EM learns when it is applied to the data in Figure 20.8(a) is shown in Figure 20.8(c); it is virtually indistinguishable from the original model from which the data were generated. Figure 20.9(a) plots the log likelihood of the data according to the current model as EM progresses. There are two points to notice. First, the log likelihood for the final learned model slightly *exceeds* that of the original model, from which the data were generated. This might seem surprising, but it simply reflects the fact that the data were generated randomly and might not provide an exact reflection of the underlying model. The second point is that *EM increases the log likelihood of the data at every iteration.* This fact can be proved in general. Furthermore, under certain conditions, EM can be proven to reach a local maximum in likelihood. (In rare cases, it could reach a saddle point or even a local minimum.) In this sense, EM resembles a gradient-based hill-climbing algorithm, but notice that it has no "step size" parameter!

Things do not always go as well as Figure 20.9(a) might suggest. It can happen, for example, that one Gaussian component shrinks so that it covers just a single data point. Then its variance will go to zero and its likelihood will go to infinity! Another problem is that two components can "merge," acquiring identical means and variances and sharing their data points. These kinds of degenerate local maxima are serious problems, especially in high dimensions. One solution is to place priors on the model parameters and to apply the MAP version of EM. Another is to restart a component with new random parameters if it gets too small or too close to another component. It also helps to initialize the parameters with reasonable values.

Learning Bayesian networks with hidden variables

To learn a Bayesian network with hidden variables, we apply the same insights that worked for mixtures of Gaussians. Figure 20.10 represents a situation in which there are two bags of candies that have been mixed together. Candies are described by three features: in addition to the *Flavor* and the *Wrapper*, some candies have a *Hole* in the middle and some do not.

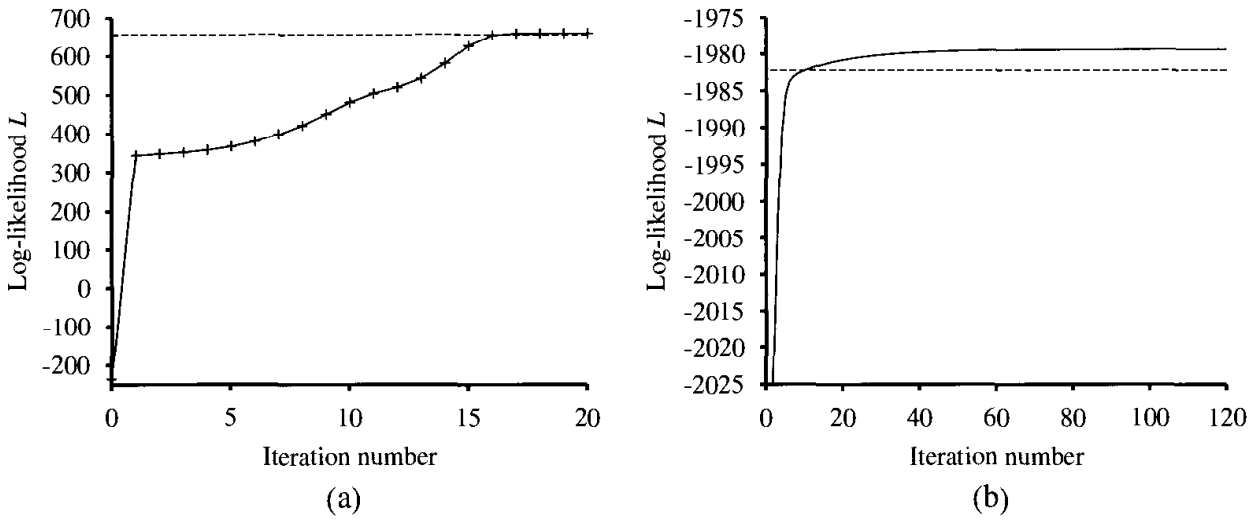


Figure 20.9 Graphs showing the log-likelihood of the data, L, as a function of the EM iteration. The horizontal line shows the log-likelihood according to the true model. (a) Graph for the Gaussian mixture model in Figure 20.8. (b) Graph for the Bayesian network in Figure 20.10(a).

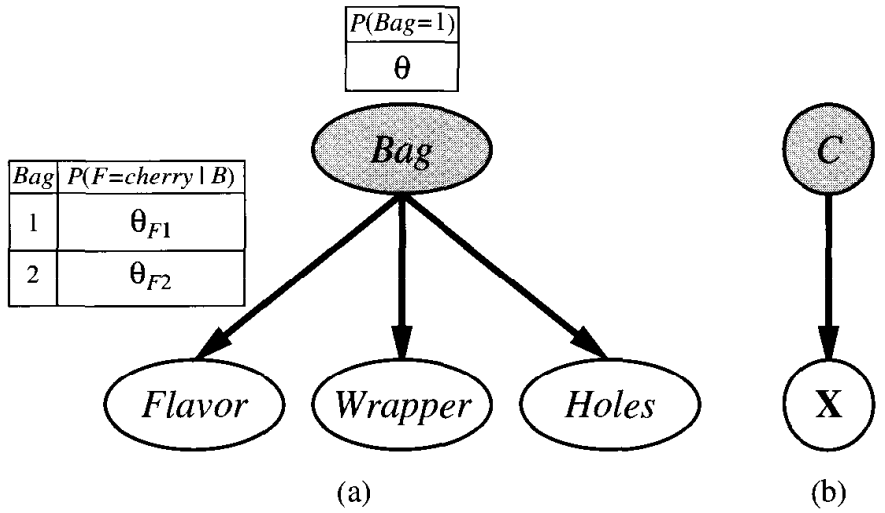


Figure 20.10 (a) A mixture model for candy. The proportions of different flavors, wrappers, and numbers of holes depend on the bag, which is not observed. (b) Bayesian network for a Gaussian mixture. The mean and covariance of the observable variables **X** depend on the component C.

The distribution of candies in each bag is described by a **naive Bayes** model: the features are independent, given the bag, but the conditional probability distribution for each feature depends on the bag. The parameters are as follows: θ is the prior probability that a candy comes from Bag 1; θ_{F1} and θ_{F2} are the probabilities that the flavor is cherry, given that the candy comes from Bag 1 and Bag 2 respectively; θ_{W1} and θ_{W2} give the probabilities that the wrapper is red; and θ_{H1} and θ_{H2} give the probabilities that the candy has a hole. Notice that

the overall model is a mixture model. (In fact, we can also model the mixture of Gaussians as a Bayesian network, as shown in Figure 20.10(b).) In the figure, the bag is is a hidden variable because, once the candies have been mixed together, we no longer know which bag each candy came from. In such a case, can we recover the descriptions of the two bags by observing candies from the mixture?

Let us work through an iteration of EM for this problem. First, let's look at the data. We generated 1000 samples from a model whose true parameters are

$$\theta = 0.5,\ \theta_{F1} = \theta_{W1} = \theta_{H1} = 0.8, \theta_{F2} = \theta_{W2} = \theta_{H2} = 0.3\ . \tag{20.7}$$

That is, the candies are equally likely to come from either bag; the first is mostly cherries with red wrappers and holes; the second is mostly limes with green wrappers and no holes. The counts for the eight possible kinds of candy are as follows:

	$W = red$		$W = green$	
	$H = 1$	$H = 0$	$H = 1$	$H = 0$
$F = cherry$	273	93	104	90
$F = lime$	79	100	94	167

We start by initializing the parameters. For numerical simplicity, we will choose[7]

$$\theta^{(0)} = 0.6,\ \theta^{(0)}_{F1} = \theta^{(0)}_{W1} = \theta^{(0)}_{H1} = 0.6, \theta^{(0)}_{F2} = \theta^{(0)}_{W2} = \theta^{(0)}_{H2} = 0.4\ . \tag{20.8}$$

First, let us work on the θ parameter. In the fully observable case, we would estimate this directly from the *observed* counts of candies from bags 1 and 2. Because the bag is a hidden variable, we calculate the *expected* counts instead. The expected count $\hat{N}(Bag = 1)$ is the sum, over all candies, of the probability that the candy came from bag 1:

$$\theta^{(1)} = \hat{N}(Bag = 1)/N = \sum_{j=1}^{N} P(Bag = 1 | flavor_j, wrapper_j, holes_j)/N\ .$$

These probabilities can be computed by any inference algorithm for Bayesian networks. For a naive Bayes model such as the one in our example, we can do the inference "by hand," using Bayes' rule and applying conditional independence:

$$\theta^{(1)} = \frac{1}{N}\sum_{j=1}^{N} \frac{P(flavor_j | Bag = 1)P(wrapper_j | Bag = 1)P(holes_j | Bag = 1)P(Bag = 1)}{\sum_i P(flavor_j | Bag = i)P(wrapper_j | Bag = i)P(holes_j | Bag = i)P(Bag = i)}\ .$$

(Notice that the normalizing constant also depends on the parameters.) Applying this formula to, say, the 273 red-wrapped cherry candies with holes, we get a contribution of

$$\frac{273}{1000} \cdot \frac{\theta^{(0)}_{F1}\theta^{(0)}_{W1}\theta^{(0)}_{H1}\theta^{(0)}}{\theta^{(0)}_{F1}\theta^{(0)}_{W1}\theta^{(0)}_{H1}\theta^{(0)} + \theta^{(0)}_{F2}\theta^{(0)}_{W2}\theta^{(0)}_{H2}(1 - \theta^{(0)})} \approx 0.22797\ .$$

Continuing with the other seven kinds of candy in the table of counts, we obtain $\theta^{(1)} = 0.6124$.

[7] It is better in practice to choose them randomly, to avoid local maxima due to symmetry.

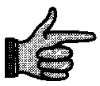
Now let us consider the other parameters, such as θ_{F1}. In the fully observable case, we would estimate this directly from the *observed* counts of cherry and lime candies from bag 1. The *expected* count of cherry candies from bag 1 is given by

$$\sum_{j:Flavor_j = cherry} P(Bag = 1 | Flavor_j = cherry, wrapper_j, holes_j) \ .$$

Again, these probabilities can be calculated by any Bayes net algorithm. Completing this process, we obtain the new values of all the parameters:

$$\begin{aligned} &\theta^{(1)} = 0.6124,\ \theta^{(1)}_{F1} = 0.6684,\ \theta^{(1)}_{W1} = 0.6483,\ \theta^{(1)}_{H1} = 0.6558, \\ &\theta^{(1)}_{F2} = 0.3887, \theta^{(1)}_{W2} = 0.3817,\ \theta^{(1)}_{H2} = 0.3827 \ . \end{aligned} \tag{20.9}$$

The log likelihood of the data increases from about -2044 initially to about -2021 after the first iteration, as shown in Figure 20.9(b). That is, the update improves the likelihood itself by a factor of about $e^{23} \approx 10^{10}$. By the tenth iteration, the learned model is a better fit than the original model ($L = -1982.214$). Thereafter, progress becomes very slow. This is not uncommon with EM, and many practical systems combine EM with a gradient-based algorithm such as Newton–Raphson (see Chapter 4) for the last phase of learning.



The general lesson from this example is that *the parameter updates for Bayesian network learning with hidden variables are directly available from the results of inference on each example. Moreover, only* local *posterior probabilities are needed for each parameter.* For the general case in which we are learning the conditional probability parameters for each variable X_i, given its parents —that is, $\theta_{ijk} = P(X_i = x_{ij} | Pa_i = pa_{ik})$—the update is given by the normalized expected counts as follows:

$$\theta_{ijk} \leftarrow \hat{N}(X_i = x_{ij}, Pa_i = pa_{ik}) / \hat{N}(Pa_i = pa_{ik}) \ .$$

The expected counts are obtained by summing over the examples, computing the probabilities $P(X_i = x_{ij}, Pa_i = pa_{ik})$ for each by using any Bayes net inference algorithm. For the exact algorithms—including variable elimination—all these probabilities are obtainable directly as a by-product of standard inference, with no need for extra computations specific to learning. Moreover, the information needed for learning is available *locally* for each parameter.

Learning hidden Markov models

Our final application of EM involves learning the transition probabilities in hidden Markov models (HMMs). Recall from Chapter 15 that a hidden Markov model can be represented by a dynamic Bayes net with a single discrete state variable, as illustrated in Figure 20.11. Each data point consists of an observation *sequence* of finite length, so the problem is to learn the transition probabilities from a set of observation sequences (or possibly from just one long sequence).

We have already worked out how to learn Bayes nets, but there is one complication: in Bayes nets, each parameter is distinct; in a hidden Markov model, on the other hand, the individual transition probabilities from state i to state j at time t, $\theta_{ijt} = P(X_{t+1} = j | X_t = i)$, are *repeated* across time—that is, $\theta_{ijt} = \theta_{ij}$ for all t. To estimate the transition probability

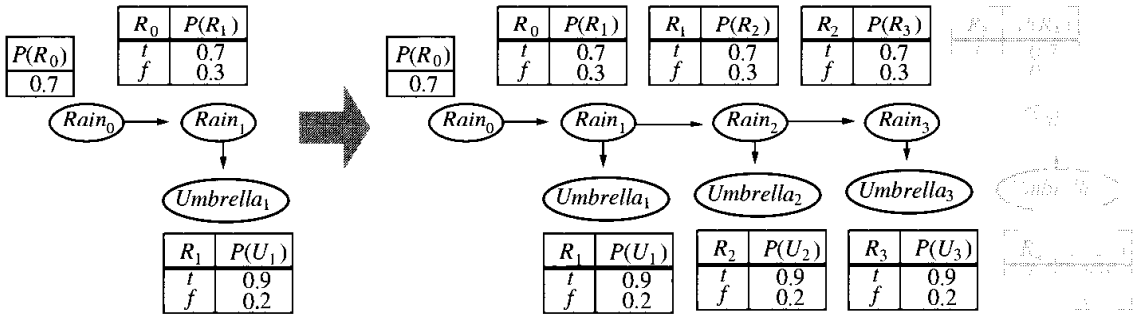


Figure 20.11 An unrolled dynamic Bayesian network that represents a hidden Markov model (repeat of Figure 15.14).

from state i to state j, we simply calculate the expected proportion of times that the system undergoes a transition to state j when in state i:

$$\theta_{ij} \leftarrow \sum_t \hat{N}(X_{t+1}=j, X_t=i) / \sum_t \hat{N}(X_t=i) \ .$$

Again, the expected counts are computed by any HMM inference algorithm. The **forward–backward** algorithm shown in Figure 15.4 can be modified very easily to compute the necessary probabilities. One important point is that the probabilities required are those obtained by **smoothing** rather than **filtering**; that is, we need to pay attention to subsequent evidence in estimating the probability that a particular transition occurred. As we said in Chapter 15, the evidence in a murder case is usually obtained *after* the crime (i.e., the transition from state i to state j) occurs.

The general form of the EM algorithm

We have seen several instances of the EM algorithm. Each involves computing expected values of hidden variables for each example and then recomputing the parameters, using the expected values as if they were observed values. Let $\mathbf{x}$ be all the observed values in all the examples, let $\mathbf{Z}$ denote all the hidden variables for all the examples, and let $\boldsymbol{\theta}$ be all the parameters for the probability model. Then the EM algorithm is

$$\boldsymbol{\theta}^{(i+1)} = \operatorname*{argmax}_{\boldsymbol{\theta}} \sum_{\mathbf{z}} P(\mathbf{Z}=\mathbf{z}|\mathbf{x}, \boldsymbol{\theta}^{(i)}) L(\mathbf{x}, \mathbf{Z}=\mathbf{z}|\boldsymbol{\theta}) \ .$$

This equation is the EM algorithm in a nutshell. The E-step is the computation of the summation, which is the expectation of the log likelihood of the "completed" data with respect to the distribution $P(\mathbf{Z}=\mathbf{z}|\mathbf{x}, \boldsymbol{\theta}^{(i)})$, which is the posterior over the hidden variables, given the data. The M-step is the maximization of this expected log likelihood with respect to the parameters. For mixtures of Gaussians, the hidden variables are the Z_{ij}s, where Z_{ij} is 1 if example j was generated by component i. For Bayes nets, the hidden variables are the values of the unobserved variables for each example. For HMMs, the hidden variables are the $i \rightarrow j$ transitions. Starting from the general form, it is possible to derive an EM algorithm for a specific application once the appropriate hidden variables have been identified.

As soon as we understand the general idea of EM, it becomes easy to derive all sorts of variants and improvements. For example, in many cases the E-step—the computation of

posteriors over the hidden variables—is intractable, as in large Bayes nets. It turns out that one can use an *approximate* E-step and still obtain an effective learning algorithm. With a sampling algorithm such as MCMC (see Section 14.5), the learning process is very intuitive: each state (configuration of hidden and observed variables) visited by MCMC is treated exactly as if it were a complete observation. Thus, the parameters can be updated directly after each MCMC transition. Other forms of approximate inference, such as variational and loopy methods, have also proven effective for learning very large networks.

Learning Bayes net structures with hidden variables

In Section 20.2, we discussed the problem of learning Bayes net structures with complete data. When hidden variables are taken into consideration, things get more difficult. In the simplest case, the hidden variables are listed along with the observed variables; although their values are not observed, the learning algorithm is told that they exist and must find a place for them in the network structure. For example, an algorithm might try to learn the structure shown in Figure 20.7(a), given the information that *HeartDisease* (a three-valued variable) should be included in the model. If the learning algorithm is not told this information, then there are two choices: either pretend that the data is really complete—which forces the algorithm to learn the parameter-intensive model in Figure 20.7(b)—or *invent* new hidden variables in order to simplify the model. The latter approach can be implemented by including new modification choices in the structure search: in addition to modifying links, the algorithm can add or delete a hidden variable or change its arity. Of course, the algorithm will not know that the new variable it has invented is called *HeartDisease*; nor will it have meaningful names for the values. Fortunately, newly invented hidden variables will usually be connected to pre-existing variables, so a human expert can often inspect the local conditional distributions involving the new variable and ascertain its meaning.

As in the complete-data case, pure maximum-likelihood structure learning will result in a completely connected network (moreover, one with no hidden variables), so some form of complexity penalty is required. We can also apply MCMC to approximate Bayesian learning. For example, we can learn mixtures of Gaussians with an unknown number of components by sampling over the number; the approximate posterior distribution for the number of Gaussians is given by the sampling frequencies of the MCMC process.

So far, the process we have discussed has an outer loop that is a structural search process and an inner loop that is a parametric optimization process. For the complete-data case, the inner loop is very fast—just a matter of extracting conditional frequencies from the data set. When there are hidden variables, the inner loop may involve many iterations of EM or a gradient-based algorithm, and each iteration involves the calculation of posteriors in a Bayes net, which is itself an NP-hard problem. To date, this approach has proved impractical for learning complex models. One possible improvement is the so-called **structural EM** algorithm, which operates in much the same way as ordinary (parametric) EM except that the algorithm can update the structure as well as the parameters. Just as ordinary EM uses the current parameters to compute the expected counts in the E-step and then applies those counts in the M-step to choose new parameters, structural EM uses the current structure to compute

STRUCTURAL EM

expected counts and then applies those counts in the M-step to evaluate the likelihood for potential new structures. (This contrasts with the outer-loop/inner-loop method, which computes new expected counts for each potential structure.) In this way, structural EM may make several structural alterations to the network without once recomputing the expected counts, and is capable of learning nontrivial Bayes net structures. Nonetheless, much work remains to be done before we can say that the structure learning problem is solved.

20.4 INSTANCE-BASED LEARNING

So far, our discussion of statistical learning has focused primarily on fitting the parameters of a *restricted* family of probability models to an *unrestricted* data set. For example, unsupervised clustering using mixtures of Gaussians assumes that the data are explained by the *sum* a *fixed* number of *Gaussian* distributions. We call such methods **parametric learning**. Parametric learning methods are often simple and effective, but assuming a particular restricted family of models often oversimplifies what's happening in the real world, from where the data come. Now, it is true when we have very little data, we cannot hope to learn a complex and detailed model, but it seems silly to keep the hypothesis complexity fixed even when the data set grows very large!

PARAMETRIC LEARNING

NONPARAMETRIC LEARNING

INSTANCE-BASED LEARNING

In contrast to parametric learning, **nonparametric learning** methods allow the hypothesis complexity to grow with the data. The more data we have, the wigglier the hypothesis can be. We will look at two very simple families of nonparametric **instance-based learning** (or **memory-based learning**) methods, so called because they construct hypotheses directly from the training instances themselves.

Nearest-neighbor models

NEAREST-NEIGHBOR

The key idea of **nearest-neighbor** models is that the properties of any particular input point $\mathbf{x}$ are likely to be similar to those of points in the neighborhood of $\mathbf{x}$. For example, if we want to do **density estimation**—that is, estimate the value of an unknown probability density at $\mathbf{x}$—then we can simply measure the density with which points are scattered in the neighborhood of $\mathbf{x}$. This sounds very simple, until we realize that we need to specify exactly what we mean by "neighborhood." If the neighborhood is too small, it won't contain any data points; too large, and it may include *all* the data points, resulting in a density estimate that is the same everywhere. One solution is to define the neighborhood to be just big enough to include k points, where k is large enough to ensure a meaningful estimate. For fixed k, the size of the neighborhood varies—where data are sparse, the neighborhood is large, but where data are dense, the neighborhood is small. Figure 20.12(a) shows an example for data scattered in two dimensions. Figure 20.13 shows the results of k-nearest-neighbor density estimation from these data with $k = 3$, 10, and 40 respectively. For $k = 3$, the density estimate at any point is based on only 3 neighboring points and is highly variable. For $k = 10$, the estimate provides a good reconstruction of the true density shown in Figure 20.12(b). For $k = 40$, the neighborhood becomes too large and structure of the data is altogether lost. In practice, using

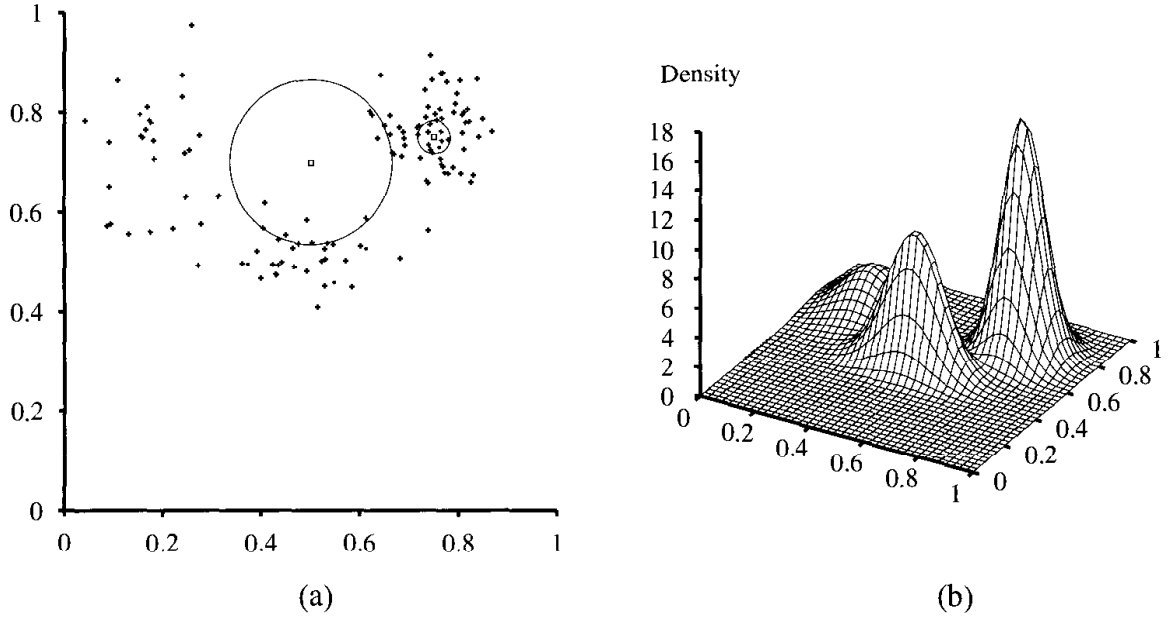


Figure 20.12 (a) A 128-point subsample of the data shown in Figure 20.8(a), together with two query points and their 10-nearest-neighborhoods. (b) A 3-D plot of the mixture of Gaussians from which the data were generated.

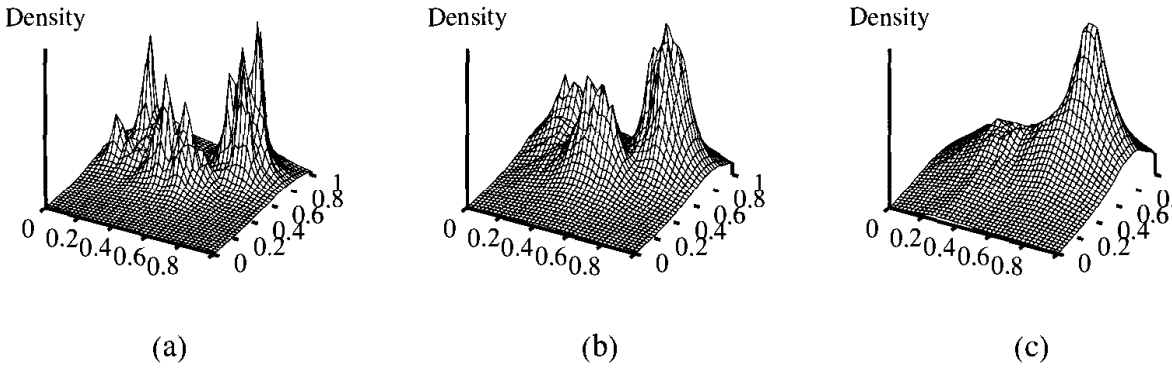


Figure 20.13 Density estimation using k-nearest-neighbors, applied to the data in Figure 20.12(a), for $k = 3$, 10, and 40 respectively.

a value of k somewhere between 5 and 10 gives good results for most low-dimensional data sets. A good value of k can also be chosen by using cross-validation.

To identify the nearest neighbors of a query point, we need a distance metric, $D(\mathbf{x}_1, \mathbf{x}_2)$. The two-dimensional example in Figure 20.12 uses Euclidean distance. This is inappropriate when each dimension of the space is measuring something different—for example, height and weight—because changing the scale of one dimension would change the set of nearest neighbors. One solution is to standardize the scale for each dimension. To do this, we measure

the standard deviation of each feature over the whole data set and express feature values as multiples of the standard deviation for that feature. (This is a special case of the **Mahalanobis distance**, which takes into account the covariance of the features as well.) Finally, for discrete features we can use the **Hamming distance**, which defines $D(\mathbf{x}_1, \mathbf{x}_2)$ to be the number of features on which $\mathbf{x}_1$ and $\mathbf{x}_2$ differ.

MAHALANOBIS DISTANCE

HAMMING DISTANCE

Density estimates like those shown in Figure 20.13 define joint distributions over the input space. Unlike a Bayesian network, however, an instance-based representation cannot contain hidden variables, which means that we cannot perform unsupervised clustering as we did with the mixture-of-Gaussians model. We can still use the density estimate to predict a target value y given input feature values $\mathbf{x}$ by calculating $P(y|\mathbf{x}) = P(y, \mathbf{x})/P(\mathbf{x})$, provided that the training data include values for the target feature.

It is also possible to use the nearest-neighbor idea for direct supervised learning. Given a test example with input $\mathbf{x}$, the output $y = h(\mathbf{x})$ is obtained from the y-values of the k nearest neighbors of $\mathbf{x}$. In the discrete case, we can obtain a single prediction by majority vote. In the continuous case, we can average the k values or do local linear regression, fitting a hyperplane to the k points and predicting the value at $\mathbf{x}$ according to the hyperplane.

The k-nearest-neighbor learning algorithm is very simple to implement, requires little in the way of tuning, and often performs quite well. It is a good thing to try first on a new learning problem. For large data sets, however, we require an efficient mechanism for finding the nearest neighbors of a query point $\mathbf{x}$—simply calculating the distance to every point would take far too long. A variety of ingenious methods have been proposed to make this step efficient by preprocessing the training data. Unfortunately, most of these methods do not scale well with the dimension of the space (i.e., the number of features).

High-dimensional spaces pose an additional problem, namely that nearest neighbors in such spaces are usually a long way away! Consider a data set of size N in the d-dimensional unit hypercube, and assume hypercubic neighborhoods of side b and volume b^d. (The same argument works with hyperspheres, but the formula for the volume of a hypersphere is more complicated.) To contain k points, the average neighborhood must occupy a fraction k/N of the entire volume, which is 1. Hence, $b^d = k/N$, or $b = (k/N)^{1/d}$. So far, so good. Now let the number of features d be 100 and let k be 10 and N be 1,000,000. Then we have $b \approx 0.89$—that is, the neighborhood has to span almost the entire input space! This suggests that nearest-neighbor methods cannot be trusted for high-dimensional data. In low dimensions there is no problem; with $d = 2$ we have $b = 0.003$.

Kernel models

KERNEL MODEL

KERNEL FUNCTION

In a **kernel model**, we view each training instance as generating a little density function—a **kernel function**—of its own. The density estimate as a whole is just the normalized sum of all the little kernel functions. A training instance at $\mathbf{x}_i$ will generate a kernel function $K(\mathbf{x}, \mathbf{x}_i)$ that assigns a probability to each point $\mathbf{x}$ in the space. Thus, the density estimate is

$$P(\mathbf{x}) = \frac{1}{N}\sum_{i=1}^{N} K(\mathbf{x}, \mathbf{x}_i) \ .$$

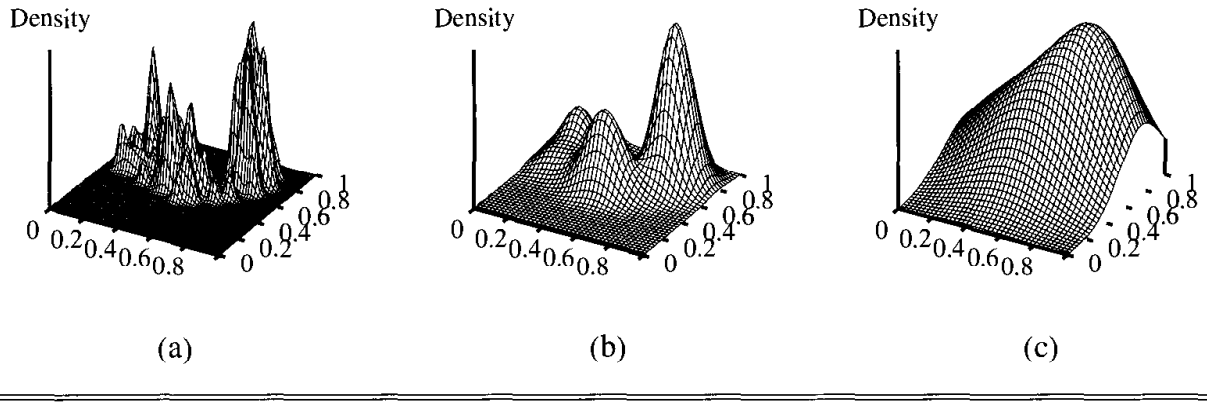


Figure 20.14 Kernel density estimation for the data in Figure 20.12(a), using Gaussian kernels with $w = 0.02$, 0.07, and 0.20 respectively.

The kernel function normally depends only on the *distance* $D(\mathbf{x}, \mathbf{x}_i)$ from $\mathbf{x}$ to the instance $\mathbf{x}_i$. The most popular kernel function is (of course) the Gaussian. For simplicity, we will assume spherical Gaussians with standard deviation w along each axis, i.e.,

$$K(\mathbf{x}, \mathbf{x}_i) = \frac{1}{(w^2\sqrt{2\pi})^d} e^{-\frac{D(\mathbf{x},\mathbf{x}_i)^2}{2w^2}} ,$$

where d is the number of dimensions in $\mathbf{x}$. We still have the problem of choosing a suitable value for w; as before, making the neighborhood too small gives a very spiky estimate—see Figure 20.14(a). In (b), a medium value of w gives a very good reconstruction. In (c), too large a neighborhood results in losing the structure altogether. A good value of w can be chosen by using cross-validation.

Supervised learning with kernels is done by taking a *weighted* combination of *all* the predictions from the training instances. (Compare this with k-nearest-neighbor prediction, which takes an unweighted combination of the nearest k instances.) The weight of the ith instance for a query point $\mathbf{x}$ is given by the value of the kernel $K(\mathbf{x}, \mathbf{x}_i)$. For a discrete prediction, we can take a weighted vote; for a continuous prediction, we can take weighted average or a weighted linear regression. Notice that making predictions with kernels requires looking at *every* training instance. It is possible to combine kernels with nearest-neighbor indexing schemes to make weighted predictions from just the nearby instances.

20.5 NEURAL NETWORKS

A **neuron** is a cell in the brain whose principal function is the collection, processing, and dissemination of electrical signals. Figure 1.2 on page 11 showed a schematic diagram of a typical neuron. The brain's information-processing capacity is thought to emerge primarily from *networks* of such neurons. For this reason, some of the earliest AI work aimed to create artificial **neural networks**. (Other names for the field include **connectionism**, **parallel dis-**

NEURAL NETWORKS

tributed processing, and **neural computation**.) Figure 20.15 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 threshold. 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

Units in neural networks

UNITS
LINKS
ACTIVATION
WEIGHT

Neural networks are composed of nodes or **units** (see Figure 20.15) connected by directed **links**. A link from unit j to unit i serves to propagate the **activation** a_j from j to i. Each link also has a numeric **weight** $W_{j,i}$ associated with it, which determines the strength and sign of the connection. Each unit i first computes a weighted sum of its inputs:

$$in_i = \sum_{j=0}^{n} W_{j,i} a_j \ .$$

ACTIVATION FUNCTION

Then it applies an **activation function** g to this sum to derive the output:

$$a_i = g(in_i) = g\left(\sum_{j=0}^{n} W_{j,i} a_j\right) \ . \tag{20.10}$$

BIAS WEIGHT

Notice that we have included a **bias weight** $W_{0,i}$ connected to a fixed input $a_0 = -1$. We will explain its role in a moment.

The activation function g is designed to meet two desiderata. First, we want the unit to be "active" (near +1) when the "right" inputs are given, and "inactive" (near 0) when the "wrong" inputs are given. Second, the activation needs to be *nonlinear*, otherwise the entire neural network collapses into a simple linear function (see Exercise 20.17). Two choices for g

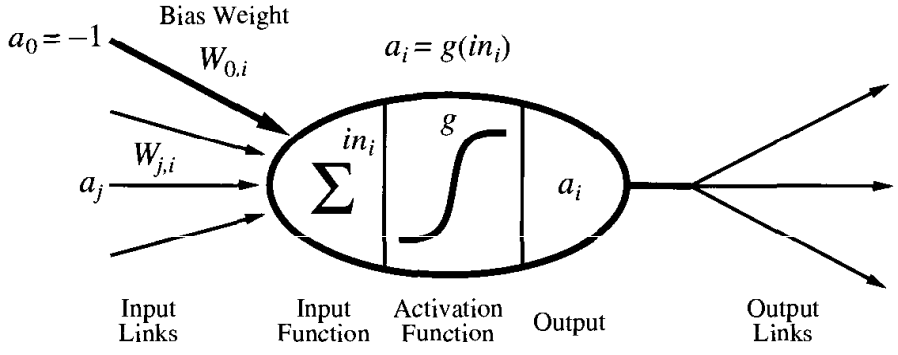


Figure 20.15 A simple mathematical model for a neuron. The unit's output activation is $a_i = g(\sum_{j=0}^{n} W_{j,i} a_j)$, where a_j is the output activation of unit j and $W_{j,i}$ is the weight on the link from unit j to this unit.

THRESHOLD
SIGMOID FUNCTION
LOGISTIC FUNCTION

are shown in Figure 20.16: the **threshold** function and the **sigmoid function** (also known as the **logistic function**). The sigmoid function has the advantage of being differentiable, which we will see later is important for the weight-learning algorithm. Notice that both functions have a threshold (either hard or soft) at zero; the bias weight $W_{0,i}$ sets the *actual* threshold for the unit, in the sense that the unit is activated when the weighted sum of "real" inputs $\sum_{j=1}^{n} W_{j,i} a_j$ (i.e., excluding the bias input) exceeds $W_{0,i}$.

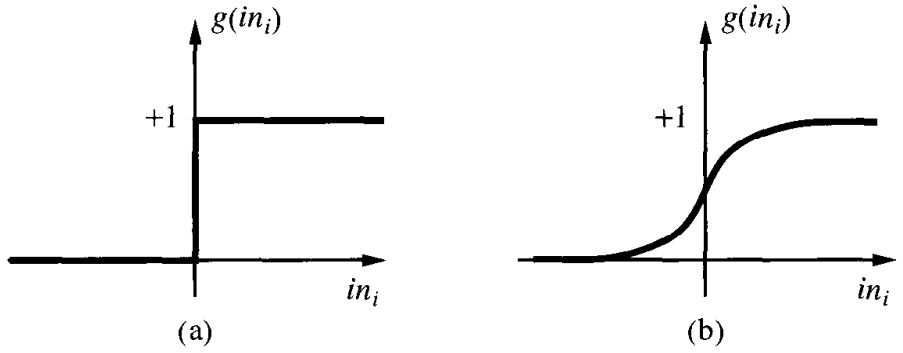


Figure 20.16 (a) The **threshold** activation function, which outputs 1 when the input is positive and 0 otherwise. (Sometimes the **sign** function is used instead, which outputs ± 1 depending on the sign of the input.) (b) The **sigmoid** function $1/(1+e^{-x})$.

We can get a feel for the operation of individual units by comparing them with logic gates. One of the original motivations for the design of individual units (McCulloch and Pitts, 1943) was their ability to represent basic Boolean functions. Figure 20.17 shows how the Boolean functions AND, OR, and NOT can be represented by threshold units with suitable weights. This is important because it means we can use these units to build a network to compute any Boolean function of the inputs.

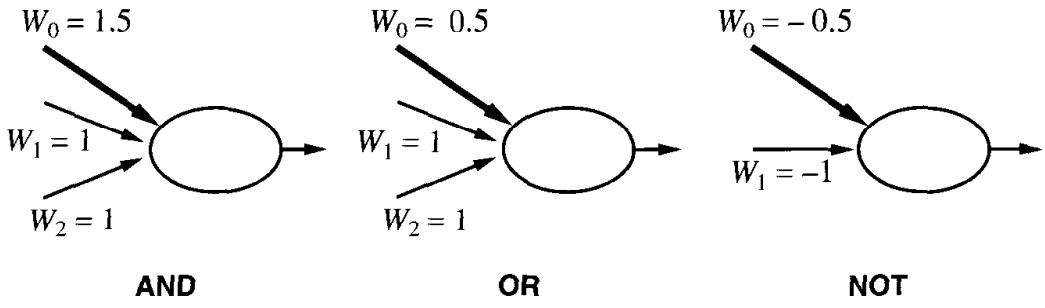


Figure 20.17 Units with a threshold activation function can act as logic gates, given appropriate input and bias weights.

Network structures

FEED-FORWARD NETWORKS
RECURRENT NETWORKS

There are two main categories of neural network structures: acyclic or **feed-forward networks** and cyclic or **recurrent networks**. 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.

Let us look more closely into the assertion that a feed-forward network represents a function of its inputs. Consider the simple network shown in Figure 20.18, which has two input units, two **hidden units**, and an output unit. (To keep things simple, we have omitted the bias units in this example.) Given an input vector $\mathbf{x} = (x_1, x_2)$, the activations of the input units are set to $(a_1, a_2) = (x_1, x_2)$ and the network computes

HIDDEN UNITS

$$\begin{aligned} a_5 &= g(W_{3,5}a_3 + W_{4,5}a_4) \\ &= g(W_{3,5}g(W_{1,3}a_1 + W_{2,3}a_2) + W_{4,5}g(W_{1,4}a_1 + W_{2,4}a_2)) \,. \end{aligned} \tag{20.11}$$

That is, by expressing the output of each hidden unit as a function of *its* inputs, we have shown that output of the network as a whole, a_5, is a function of the network's inputs. Furthermore, we see that the weights in the network act as *parameters* of this function; writing $\mathbf{W}$ for the parameters, the network computes a function $h_{\mathbf{W}}(\mathbf{x})$. By adjusting the weights, we change the function that the network represents. This is how learning occurs in neural networks.

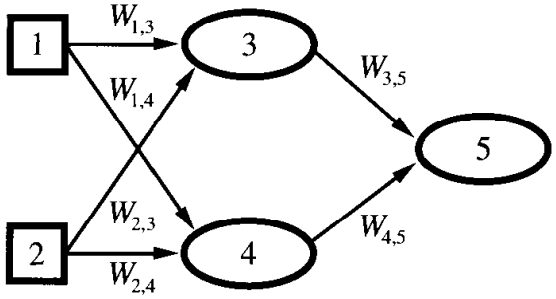


Figure 20.18 A very simple neural network with two inputs, one hidden layer of two units, and one output.

A neural network can be used for classification or regression. For Boolean classification with continuous outputs (e.g., with sigmoid units), it is traditional to have a single output unit, with a value over 0.5 interpreted as one class and a value below 0.5 as the other. For k-way classification, one could divide the single output unit's range into k portions, but it is more common to have k separate output units, with the value of each one representing the relative likelihood of that class given the current input.

LAYERS

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, which have no hidden units, and multilayer networks, which have one or more layers of hidden units.

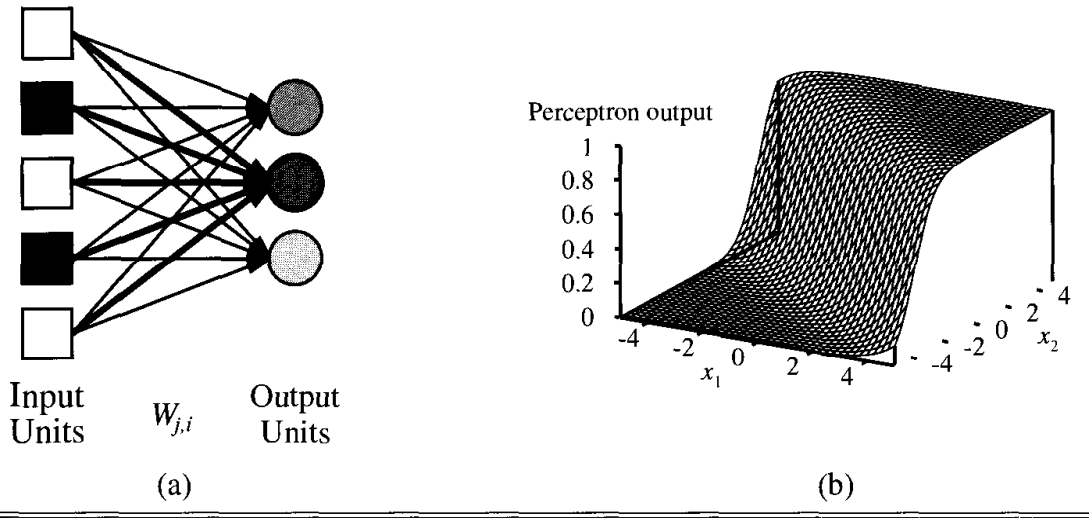


Figure 20.19 (a) A perceptron network consisting of three perceptron output units that share five inputs. Looking at a particular output unit (say the second one, outlined in bold), we see that the weights on its incoming links have no effect on the other output units. (b) A graph of the output of a two-input perceptron unit with a sigmoid activation function.

Single layer feed-forward neural networks (perceptrons)

SINGLE-LAYER NEURAL NETWORK
PERCEPTRON

A network with all the inputs connected directly to the outputs is called a **single-layer neural network**, or a **perceptron** network. Since each output unit is independent of the others—each weight affects only one of the outputs—we can limit our study to perceptrons with a single output unit, as explained in Figure 20.19(a).

Let us begin by examining the hypothesis space that a perceptron can represent. With a threshold activation function, we can view the perceptron as representing a Boolean function. In addition to the elementary Boolean functions AND, OR, and NOT (Figure 20.17), 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 $W_j = 1$ and threshold $W_0 = n/2$. A decision tree would need $O(2^n)$ nodes to represent this function.

Unfortunately, there are many Boolean functions that the threshold perceptron cannot represent. Looking at Equation (20.10), we see that the threshold perceptron returns 1 if and only if the weighted sum of its inputs (including the bias) is positive:

$$\sum_{j=0}^{n} W_j x_j > 0 \qquad \text{or} \qquad \mathbf{W} \cdot \mathbf{x} > 0 \ .$$

Now, the equation $\mathbf{W} \cdot \mathbf{x} = 0$ defines a *hyperplane* in the input space, so the perceptron returns 1 if and only if the input is on one side of that hyperplane. For this reason, the threshold perceptron is called a **linear separator**. Figure 20.20(a) and (b) show this hyperplane (a line, in two dimensions) for the perceptron representations of the AND and OR functions of two inputs. 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 can represent these functions because there is some line that separates all the white dots from all the black

LINEAR SEPARATOR

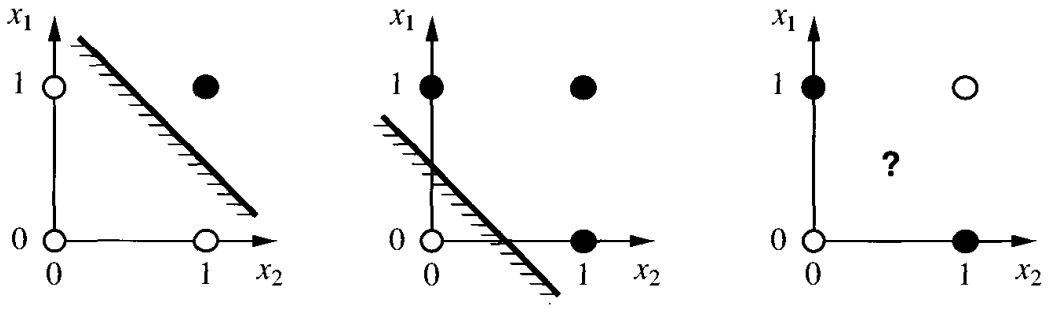


Figure 20.20 Linear separability in threshold perceptrons. Black dots indicate a point in the input space where the value of the function is 1, and white dots indicate a point where the value is 0. The perceptron returns 1 on the region on the non-shaded side of the line. In (c), no such line exists that correctly classifies the inputs.

LINEARLY SEPARABLE

dots. Such functions are called **linearly separable**. Figure 20.20(c) shows an example of a function that is *not* linearly separable—the XOR function. Clearly, there is no way for a threshold perceptron to learn this function. In general, *threshold perceptrons can represent only linearly separable functions.* These constitute just a small fraction of all functions; Exercise 20.14 asks you to quantify this fraction. Sigmoid perceptrons are similarly limited, in the sense that they represent only "soft" linear separators. (See Figure 20.19(b).)

Despite their limited expressive power, threshold perceptrons have some advantages. In particular, *there is a simple learning algorithm that will fit a threshold perceptron to any linearly separable training set.* Rather than present this algorithm, we will *derive* a closely related algorithm for learning in sigmoid perceptrons.

The idea behind this algorithm, and indeed behind most algorithms for neural network learning, is to adjust the weights of the network to minimize some measure of the error on the training set. Thus, learning is formulated as an optimization search in **weight space**.[8] The "classical" measure of error is the **sum of squared errors**, which we used for linear regression on page 720. The squared error for a single training example with input $\mathbf{x}$ and true output y is written as

WEIGHT SPACE

$$E = \frac{1}{2}Err^2 \equiv \frac{1}{2}(y - h_{\mathbf{W}}(\mathbf{x}))^2 ,$$

where $h_{\mathbf{W}}(\mathbf{x})$ is the output of the perceptron on the example.

We can use gradient descent to reduce the squared error by calculating the partial derivative of E with respect to each weight. We have

$$\begin{aligned}\frac{\partial E}{\partial W_j} &= Err \times \frac{\partial Err}{\partial W_j} \\ &= Err \times \frac{\partial}{\partial W_j}\left(y - g\left(\sum_{j=0}^{n} W_j x_j\right)\right) \\ &= -Err \times g'(in) \times x_j ,\end{aligned}$$

[8] See Section 4.4 for general optimization techniques applicable to continuous spaces.

where g' is the derivative of the activation function.[9] In the gradient descent algorithm, where we want to *reduce* E, we update the weight as follows:

$$W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j \,, \tag{20.12}$$

where α is the **learning rate**. Intuitively, this makes a lot of sense. If the error $Err = y - h_{\mathbf{W}}(\mathbf{x})$ is positive, then the network output is too small and so the weights are *increased* for the positive inputs and *decreased* for the negative inputs. The opposite happens when the error is negative.[10]

The complete algorithm is shown in Figure 20.21. It runs the training examples through the net one at a time, adjusting the weights slightly after each example to reduce the error. Each cycle through the examples is called an **epoch**. Epochs are repeated until some stopping criterion is reached—typically, that the weight changes have become very small. Other methods calculate the gradient for the whole training set by adding up all the gradient contributions in Equation (20.12) before updating the weights. The **stochastic gradient** method selects examples randomly from the training set rather than cycling through them.

EPOCH

STOCHASTIC GRADIENT

function PERCEPTRON-LEARNING(*examples*, *network*) **returns** a perceptron hypothesis
inputs: *examples*, a set of examples, each with input $\mathbf{x} = x_1, \ldots, x_n$ and output y
network, a perceptron with weights W_j, $j = 0 \ldots n$, and activation function g

repeat
for each e **in** *examples* **do**
$in \leftarrow \sum_{j=0}^{n} W_j x_j[e]$
$Err \leftarrow y[e] - g(in)$
$W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j[e]$
until some stopping criterion is satisfied
return NEURAL-NET-HYPOTHESIS(*network*)

Figure 20.21 The gradient descent learning algorithm for perceptrons, assuming a differentiable activation function g. For threshold perceptrons, the factor $g'(in)$ is omitted from the weight update. NEURAL-NET-HYPOTHESIS returns a hypothesis that computes the network output for any given example.

Figure 20.22 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

[9] For the sigmoid, this derivative is given by $g' = g(1 - g)$.

[10] For threshold perceptrons, where $g'(in)$ is undefined, the original **perceptron learning rule** developed by Rosenblatt (1957) is identical to Equation (20.12) except that $g'(in)$ is omitted. Since $g'(in)$ is the same for all weights, its omission changes only the magnitude and not the direction of the overall weight update for each example.

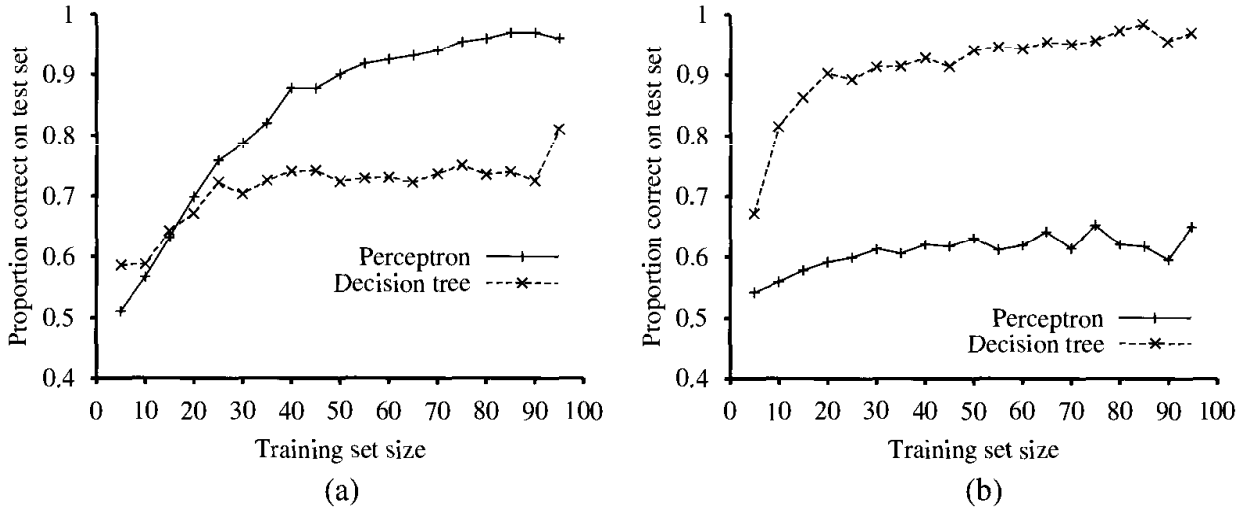


Figure 20.22 Comparing the performance of perceptrons and decision trees. (a) Perceptrons are better at learning the majority function of 11 inputs. (b) Decision trees are better at learning the *WillWait* predicate in the restaurant example.

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%.

So far, we have treated perceptrons as deterministic functions with possibly erroneous outputs. It is also possible to interpret the output of a sigmoid perceptron as a *probability*—specifically, the probability that the true output is 1 given the inputs. With this interpretation, one can use the sigmoid as a canonical representation for conditional distributions in Bayesian networks (see Section 14.3). One can also derive a learning rule using the standard method of maximizing the (conditional) log likelihood of the data, as described earlier in this chapter. Let's see how this works.

Consider a single training example with true output value T, and let p be the probability returned by the perceptron for this example. If $T = 1$, the conditional probability of the datum is p, and if $T = 0$, the conditional probability of the datum is $(1 - p)$. Now we can use a simple trick to write the log likelihood in a form that is differentiable. The trick is that a 0/1 variable in the *exponent* of an expression acts as an **indicator variable**: p^T is p if $T = 1$ and 1 otherwise; similarly $(1 - p)^{(1-T)}$ is $(1 - p)$ if $T = 0$ and 1 otherwise. Hence, we can write the log likelihood of the datum as

INDICATOR VARIABLE

$$L = \log p^T (1 - p)^{(1-T)} = T \log p + (1 - T) \log(1 - p) \ . \tag{20.13}$$

Thanks to the properties of the sigmoid function, the gradient reduces to a very simple formula (Exercise 20.16):

$$\frac{\partial L}{\partial W_j} = Err \times a_j \ .$$

Notice that *the weight-update vector for maximum likelihood learning in sigmoid perceptrons is essentially identical to the update vector for squared error minimization.* Thus, we could say that perceptrons have a probabilistic interpretation even when the learning rule is derived from a deterministic viewpoint.

Multilayer feed-forward neural networks

Now we will consider networks with hidden units. The most common case involves a single hidden layer,[11] as in Figure 20.24. The advantage of adding hidden layers is that it enlarges the space of hypotheses that the network can represent. Think of each hidden unit as a perceptron that represents a soft threshold function in the input space, as shown in Figure 20.19(b). Then, think of an output unit as as 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 20.23(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 20.23(b).

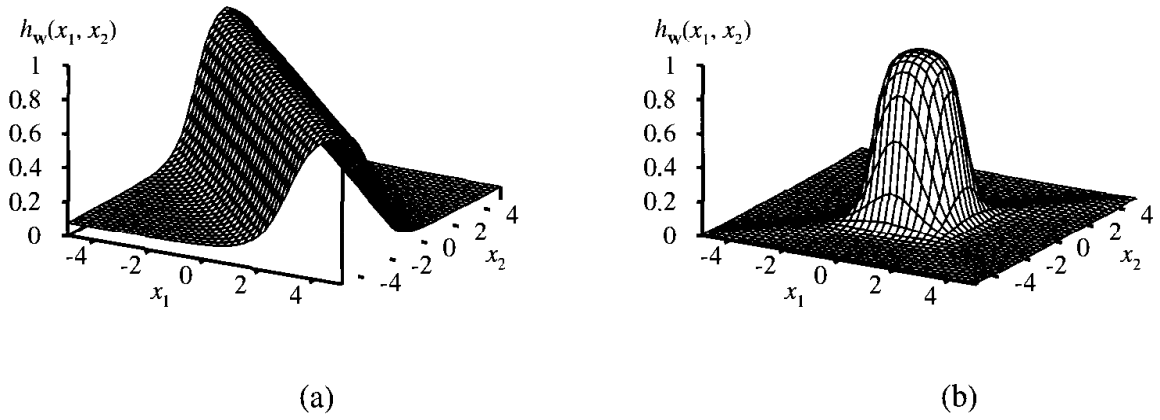


Figure 20.23 (a) The result of combining two opposite-facing soft threshold functions to produce a ridge. (b) The result of combining two ridges to produce a bump.

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.[12] Unfortunately, for any *particular* network structure, it is harder to characterize exactly which functions can be represented and which ones cannot.

Suppose we want to construct a hidden layer network for the restaurant problem. We have 10 attributes describing each example, so we will need 10 input units. How many hidden units are needed? In Figure 20.24, we show a network with four hidden units. This turns out to be about right for this problem. The problem of choosing the right number of hidden units in advance is still not well understood. (See page 748.)

Learning algorithms for multilayer networks are similar to the perceptron learning algorithm show in Figure 20.21. One minor difference is that we may have several outputs, so

[11] Some people call this a three-layer network, and some call it a two-layer network (because the inputs aren't "real" units). We will avoid confusion and call it a "single-hidden-layer network."

[12] 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.

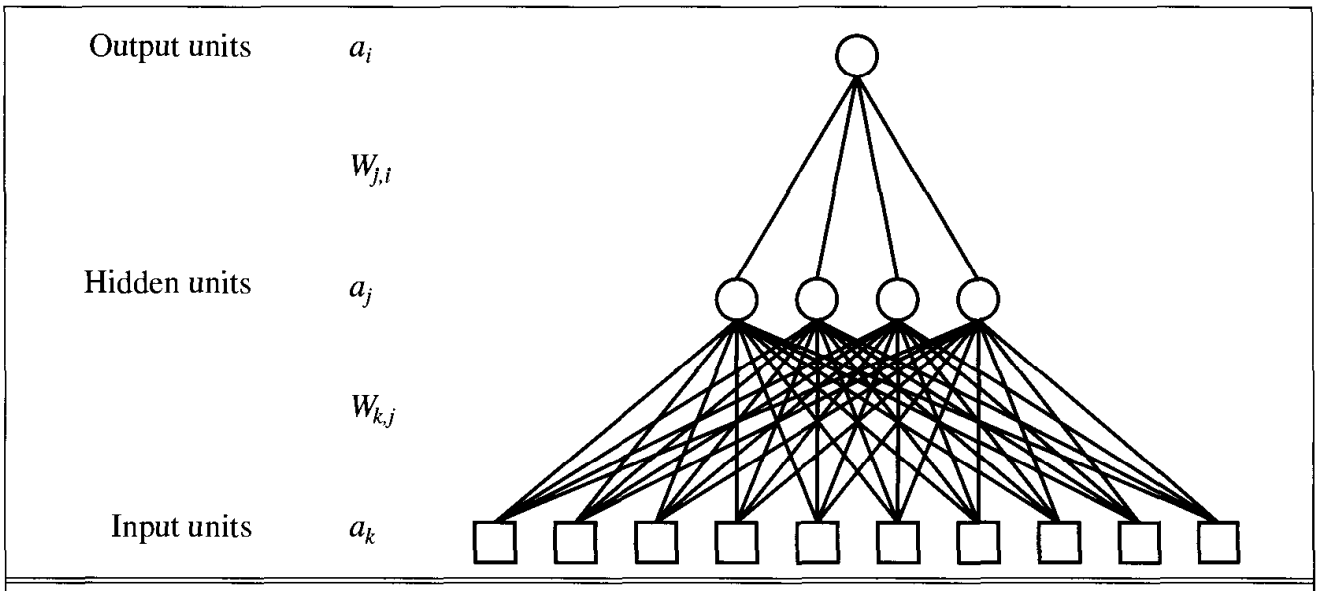


Figure 20.24 A multilayer neural network with one hidden layer and 10 inputs, suitable for the restaurant problem.

we have an output vector $\mathbf{h_W}(\mathbf{x})$ rather than a single value, and each example has an output vector $\mathbf{y}$. The major difference is that, 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 does not say what value the hidden nodes should have. 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 (20.12). We have multiple output units, so let Err_i be ith component of the error vector $\mathbf{y} - \mathbf{h_W}$. We will also find it convenient to define a modified error $\Delta_i = Err_i \times g'(in_i)$, so that the weight-update rule becomes

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i \ . \tag{20.14}$$

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_i in each of the output nodes to which it connects. Thus, the Δ_i 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:

$$\Delta_j = g'(in_j) \sum_i W_{j,i} \Delta_i \ . \tag{20.15}$$

Now the weight-update rule for the weights between the inputs and the hidden layer is almost identical to the update rule for the output layer:

$$W_{k,j} \leftarrow W_{k,j} + \alpha \times a_k \times \Delta_j \ .$$

The back-propagation process can be summarized as follows:

```
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_{j,i}, activation function g

  repeat
    for each e in examples do
      for each node j in the input layer do a_j ← x_j[e]
      for ℓ = 2 to L do
        in_i ← Σ_j W_{j,i} a_j
        a_i ← g(in_i)
      for each node i in the output layer do
        Δ_i ← g'(in_i) × (y_i[e] − a_i)
      for ℓ = L − 1 to 1 do
        for each node j in layer ℓ do
          Δ_j ← g'(in_j) Σ_i W_{j,i} Δ_i
          for each node i in layer ℓ + 1 do
            W_{j,i} ← W_{j,i} + α × a_j × Δ_i
  until some stopping criterion is satisfied
  return NEURAL-NET-HYPOTHESIS(network)
```

Figure 20.25 The back-propagation algorithm for learning in multilayer networks.

- 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 20.25.

For the mathematically inclined, we will now derive the back-propagation equations from first principles. The squared error on a single example is defined as

$$E = \frac{1}{2}\sum_i (y_i - a_i)^2 ,$$

where the sum is over the nodes in the output layer. To obtain the gradient with respect to a specific weight $W_{j,i}$ in the output layer, we need only expand out the activation a_i as all other terms in the summation are unaffected by $W_{j,i}$:

$$\begin{aligned}
\frac{\partial E}{\partial W_{j,i}} &= -(y_i - a_i)\frac{\partial a_i}{\partial W_{j,i}} = -(y_i - a_i)\frac{\partial g(in_i)}{\partial W_{j,i}} \\
&= -(y_i - a_i)g'(in_i)\frac{\partial in_i}{\partial W_{j,i}} = -(y_i - a_i)g'(in_i)\frac{\partial}{\partial W_{j,i}}\left(\sum_j W_{j,i}a_j\right) \\
&= -(y_i - a_i)g'(in_i)a_j = -a_j\Delta_i ,
\end{aligned}$$

with Δ_i defined as before. To obtain the gradient with respect to the $W_{k,j}$ weights connecting the input layer to the hidden layer, we have to keep the entire summation over i because each output value a_i may be affected by changes in $W_{k,j}$. We also have to expand out the activations a_j. We will show the derivation in gory detail because it is interesting to see how the derivative operator propagates back through the network:

$$\begin{aligned}
\frac{\partial E}{\partial W_{k,j}} &= -\sum_i (y_i - a_i)\frac{\partial a_i}{\partial W_{k,j}} = -\sum_i (y_i - a_i)\frac{\partial g(in_i)}{\partial W_{k,j}} \\
&= -\sum_i (y_i - a_i) g'(in_i)\frac{\partial in_i}{\partial W_{k,j}} = -\sum_i \Delta_i \frac{\partial}{\partial W_{k,j}}\left(\sum_j W_{j,i} a_j\right) \\
&= -\sum_i \Delta_i W_{j,i}\frac{\partial a_j}{\partial W_{k,j}} = -\sum_i \Delta_i W_{j,i}\frac{\partial g(in_j)}{\partial W_{k,j}} \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j)\frac{\partial in_j}{\partial W_{k,j}} \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j)\frac{\partial}{\partial W_{k,j}}\left(\sum_k W_{k,j} a_k\right) \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j) a_k = -a_k \Delta_j \,,
\end{aligned}$$

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 20.25.

Having made it through (or skipped over) all the mathematics, let's see how a single-hidden-layer network performs on the restaurant problem. In Figure 20.26, we show two

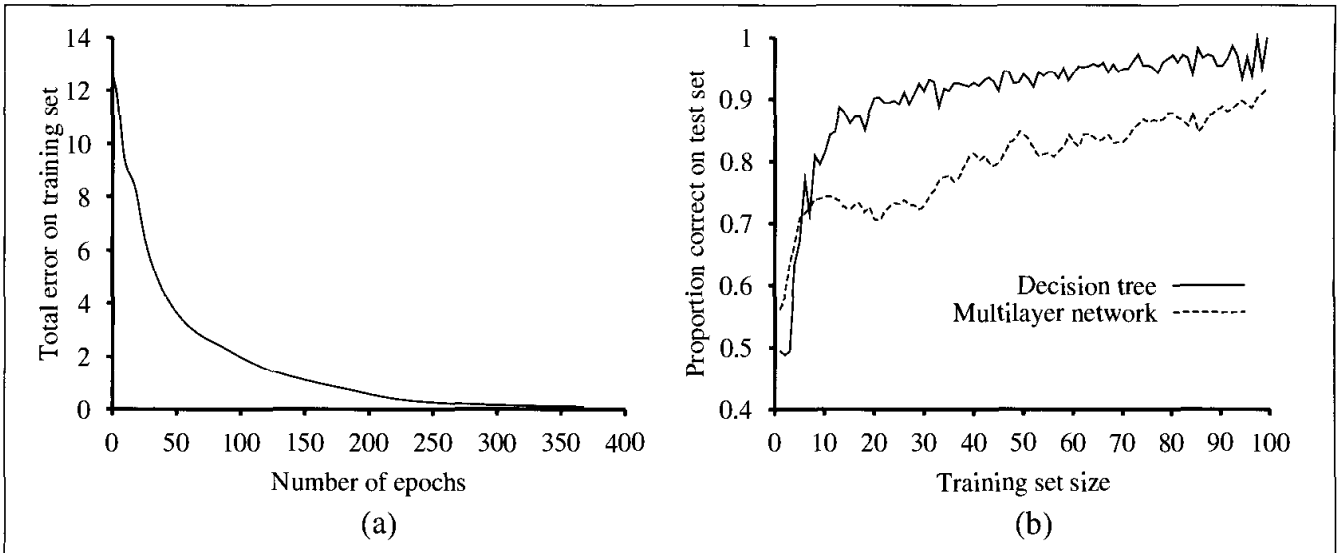


Figure 20.26 (a) Training curve showing the gradual reduction in error as weights are modified over several epochs, for a given set of examples in the restaurant domain. (b) Comparative learning curves showing that decision-tree learning does slightly better than back-propagation in a multilayer network.

TRAINING CURVE

curves. The first is a **training curve**, which shows the mean squared error on a given training set of 100 restaurant examples 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 20.7 looks at one such application in more depth.

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.[13] 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 652), 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 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.

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.

OPTIMAL BRAIN DAMAGE

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.

Several algorithms have been proposed for growing a larger network from a smaller one.

TILING

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.

[13] 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 20.17) with far fewer parameters.

20.6 KERNEL MACHINES

Our discussion of neural networks left us with a dilemma. Single-layer networks have a simple and efficient learning algorithm, but have very limited expressive power—they can learn only linear decision boundaries in the input space. Multilayer networks, on the other hand, are much more expressive—they can represent general nonlinear functions—but are very hard to train because of the abundance of local minima and the high dimensionality of the weight space. In this section, we will explore a relatively new family of learning methods called **support vector machines** (SVMs) or, more generally, **kernel machines**. To some extent, kernel machines give us the best of both worlds. That is, these methods use an efficient training algorithm *and* can represent complex, nonlinear functions.

SUPPORT VECTOR MACHINE

KERNEL MACHINE

The full treatment of kernel machines is beyond the scope of the book, but we can illustrate the main idea through an example. Figure 20.27(a) shows a two-dimensional 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 using some computed features—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

$$f_1 = x_1^2 , \qquad f_2 = x_2^2 , \qquad f_3 = \sqrt{2} x_1 x_2 . \tag{20.16}$$

We will see shortly where these came from, but, for now, just look at what happens. Figure 20.27(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 always be linearly separable. Here, we used only three dimensions,[14] but if we have N data points then, except in special cases, they will always be separable in a space of $N - 1$ dimensions or more (Exercise 20.21).

So, is that it? Do we just produce loads of computed features and then find a linear separator in the corresponding high-dimensional space? Unfortunately, it's not that easy. Remember that a linear separator in a space of d dimensions is defined by an equation with d parameters, so we are in serious danger of overfitting the data if $d \approx N$, the number of data points. (This is like overfitting data with a high-degree polynomial, which we discussed in Chapter 18.) For this reason, kernel machines usually find the *optimal* linear separator—the one that has the largest **margin** between it and the positive examples on one side and the negative examples on the other. (See Figure 20.28.) It can be shown, using arguments from computational learning theory (Section 18.5), that this separator has desirable properties in terms of robust generalization to new examples.

MARGIN

QUADRATIC PROGRAMMING

Now, how do we find this separator? It turns out that this is a **quadratic programming** optimization problem. Suppose we have examples $\mathbf{x}_i$ with classifications $y_i = \pm 1$ and we want to find an optimal separator in the input space; then the quadratic programming problem

[14] The reader may notice that we could have used just f_1 and f_2, but the 3D mapping illustrates the idea better.

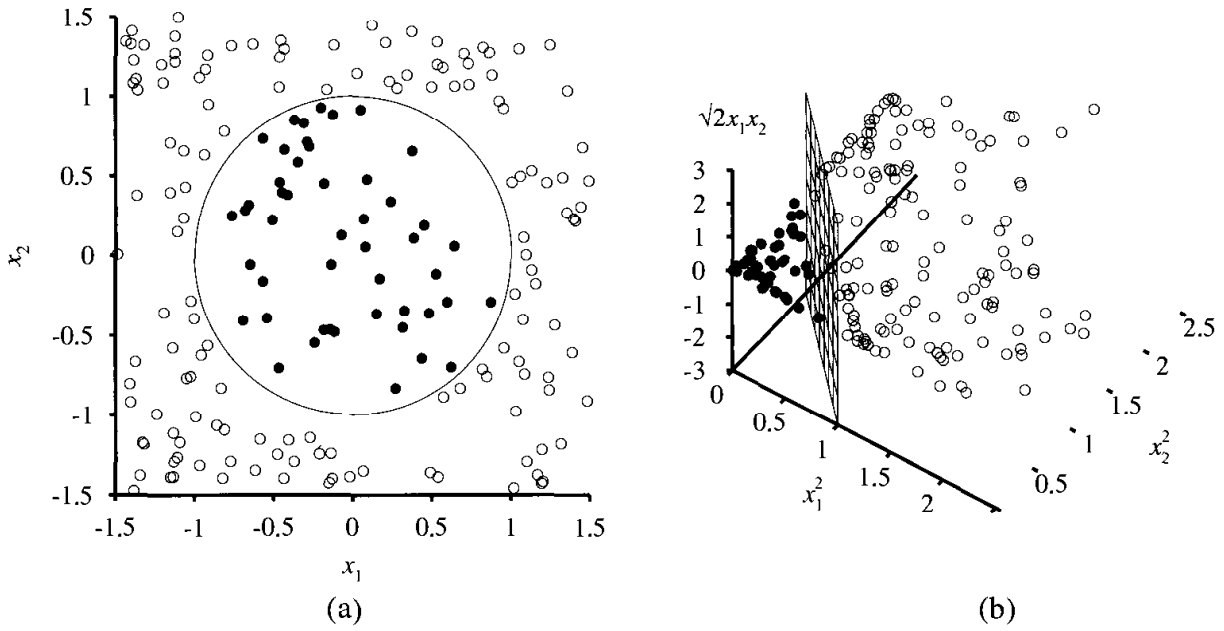


Figure 20.27 (a) A two-dimensional training with positive examples as black circles and negative examples as white circles. The true decision boundary, $x_1^2 + x_2^2 \leq 1$, is also shown. (b) The same data after mapping into a three-dimensional input space $(x_1^2, x_2^2, \sqrt{2}x_1x_2)$. The circular decision boundary in (a) becomes a linear decision boundary in three dimensions.

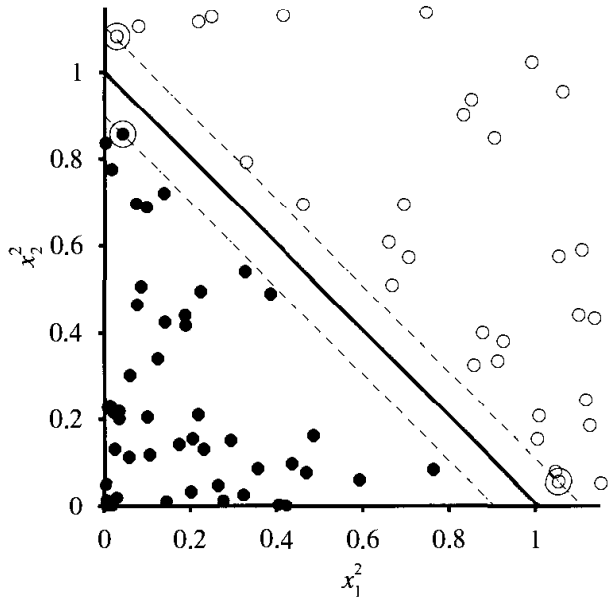


Figure 20.28 A close-up, projected onto the first two dimensions, of the optimal separator shown in Figure 20.27(b). The separator is shown as a heavy line, with the closest points—the **support vectors**—marked with circles. The **margin** is the separation between the positive and negative examples.

is to find values of the parameters α_i that maximize the expression

$$\sum_i \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j) \tag{20.17}$$

subject to the constraints $\alpha_i \geq 0$ and $\sum_i \alpha_i y_i = 0$. Although the derivation of this expression is not crucial to the story, it does have two important properties. First, the expression 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_is have been calculated, it is

$$h(\mathbf{x}) = \text{sign}\left(\sum_i \alpha_i y_i (\mathbf{x} \cdot \mathbf{x}_i)\right) . \tag{20.18}$$

A final important property of the optimal separator defined by this equation is that the weights α_i associated with each data point are *zero* except for those points closest to the separator—the so-called **support vectors**. (They are called this because they "hold up" the separating plane.) Because there are usually many fewer support vectors than data points, the effective number of parameters defining the optimal separator is usually much less than N.

SUPPORT VECTOR

Now, we would not usually expect to find a linear separator in the input space $\mathbf{x}$, but it is easy to see that we can find linear separators in the high-dimensional feature space $F(\mathbf{x})$ simply by replacing $\mathbf{x}_i \cdot \mathbf{x}_j$ in Equation (20.17) with $F(\mathbf{x}_i) \cdot F(\mathbf{x}_j)$. 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 $F(\mathbf{x}_i) \cdot F(\mathbf{x}_j)$ can often be computed without first computing F for each point. In our three-dimensional feature space defined by Equation (20.16), a little bit of algebra shows that

$$F(\mathbf{x}_i) \cdot F(\mathbf{x}_j) = (\mathbf{x}_i \cdot \mathbf{x}_j)^2 .$$

The expression $(\mathbf{x}_i \cdot \mathbf{x}_j)^2$ is called a **kernel function**, usually written as $K(\mathbf{x}_i, \mathbf{x}_j)$. In the kernel machine context, this means a function that can be applied to pairs of input data to evaluate dot products in some corresponding feature space. So, we can restate the claim at the beginning of this paragraph as follows: we can find linear separators in the high-dimensional feature space $F(\mathbf{x})$ simply by replacing $\mathbf{x}_i \cdot \mathbf{x}_j$ in Equation (20.17) with a kernel function $K(\mathbf{x}_i, \mathbf{x}_j)$. Thus, we can learn in the high-dimensional space but we compute only kernel functions rather than the full list of features for each data point.

The next step, which should by now be obvious, is to see that there's nothing special about the kernel $K(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i \cdot \mathbf{x}_j)^2$. 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 theorem** (1909), tells us that any "reasonable" [15] kernel function corresponds to *some* feature space. These feature spaces can be very large, even for innocuous-looking kernels. For example, the **polynomial kernel**, $K(\mathbf{x}_i, \mathbf{x}_j) = (1 + \mathbf{x}_i \cdot \mathbf{x}_j)^d$, corresponds to a feature space whose dimension is exponential in d. Using such kernels in Equation (20.17), then, *optimal linear separators can be found efficiently in feature spaces with billions (or, in some cases, infinitely many) dimensions.* The resulting linear separators,

MERCER'S THEOREM

POLYNOMIAL KERNEL

[15] Here, "reasonable" means that the matrix $\mathbf{K}_{ij} = K(\mathbf{x}_i, \mathbf{x}_j)$ is positive definite; see Appendix A.

when mapped back to the original input space, can correspond to arbitrarily wiggly, nonlinear boundaries between the positive and negative examples.

We mentioned in the preceding section that kernel machines excel at handwritten digit recognition; they are rapidly being adopted for other applications—especially those with many input features. As part of this process, many new kernels have been designed that work with strings, trees, and other non-numerical data types. It has also been observed that 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 20.17 and 20.18. Once this is 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-neighbor and perceptron learning, among others.

KERNELIZATION

20.7 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 20.29.

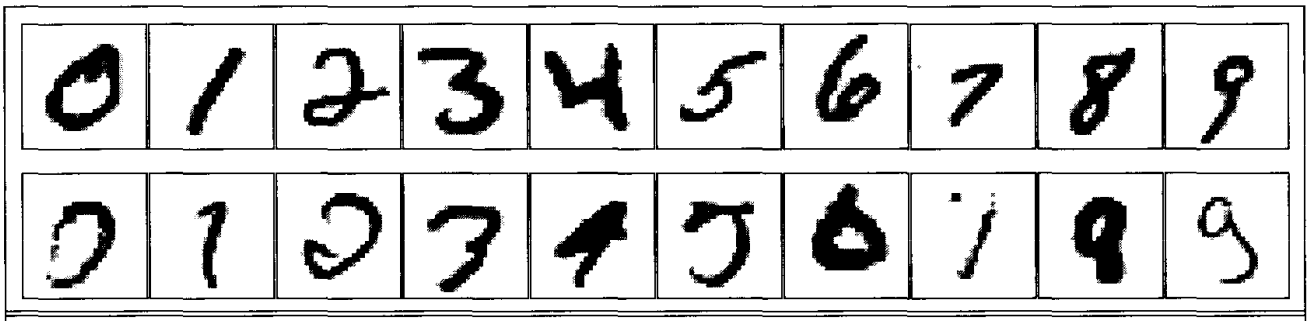


Figure 20.29 Examples from the NIST database of handwritten digits. Top row: examples of digits 0–9 that are easy to identify. Bottom row: more difficult examples of the same digits.

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 runtime 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 their weights for each of the ten digits, and the scores are added to determine the winner. The test error rate was 0.7%.

A **support vector machine** (see Section 20.6) 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, requires 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.

VIRTUAL SUPPORT VECTOR MACHINE

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. (See Chapter 24.) 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, runtime 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
Runtime (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		

20.8 Summary

Statistical learning methods range from simple calculation of averages to the construction of complex models such as Bayesian networks and neural networks. They have applications throughout computer science, engineering, neurobiology, psychology, and physics. This chapter has presented some of the basic ideas and given a flavor of the mathematical underpinnings. The main points are as follows:

- **Bayesian learning** methods formulate learning as a form of probabilistic inference, using the observations to update a prior distribution over hypotheses. This approach provides a good way to implement Ockham's razor, but quickly becomes intractable for complex hypothesis spaces.
- **Maximum a posteriori** (MAP) learning selects a single most likely hypothesis given the data. The hypothesis prior is still used and the method is often more tractable than full Bayesian learning.
- **Maximum likelihood** learning simply selects the hypothesis that maximizes the likelihood of the data; it is equivalent to MAP learning with a uniform prior. In simple cases such as linear regression and fully observable Bayesian networks, maximum likelihood solutions can be found easily in closed form. **Naive Bayes** learning is a particularly effective technique that scales well.
- When some variables are hidden, local maximum likelihood solutions can be found using the EM algorithm. Applications include clustering using mixtures of Gaussians, learning Bayesian networks, and learning hidden Markov models.

- Learning the structure of Bayesian networks is an example of **model selection**. This usually involves a discrete search in the space of structures. Some method is required for trading off model complexity against degree of fit.
- **Instance-based models** represent a distribution using the collection of training instances. Thus, the number of parameters grows with the training set. **Nearest-neighbor** methods look at the instances nearest to the point in question, whereas **kernel** methods form a distance-weighted combination of all the instances.
- **Neural networks** are complex nonlinear functions with many parameters. Their parameters can be learned from noisy data and they have been used for thousands of applications.
- A **perceptron** is a feed-forward neural network with no hidden units that can represent only **linearly separable** functions. If the data are linearly separable, a simple weight-update rule can be used to fit the data exactly.
- **Multilayer 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.

Statistical learning continues to be a very active area of research. Enormous strides have been made in both theory and practice, to the point where it is possible to learn almost any model for which exact or approximate inference is feasible.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

The application of statistical learning techniques in AI was an active area of research in the early years (see Duda and Hart, 1973) but became separated from mainstream AI as the latter field concentrated on symbolic methods. It continued in various forms—some explicitly probabilistic, others not—in areas such as **pattern recognition** (Devroye *et al.*, 1996) and **information retrieval** (Salton and McGill, 1983). A resurgence of interest occurred shortly after the introduction of Bayesian network models in the late 1980s; at roughly the same time, a statistical view of neural network learning began to emerge. In the late 1990s, there was a noticeable convergence of interests in machine learning, statistics, and neural networks, centered on methods for creating large probabilistic models from data.

The naive Bayes model is one of the oldest and simplest forms of Bayesian network, dating back to the 1950s. Its origins were mentioned in the notes at the end of Chapter 13. is partially explained by Domingos and Pazzani (1997). A boosted form of naive Bayes learning won the first KDD Cup data mining competition (Elkan, 1997). Heckerman (1998) gives an excellent introduction to the general problem of Bayes net learning. Bayesian parameter learning with Dirichlet priors for Bayesian networks was discussed by Spiegelhalter *et al.* (1993). The BUGS software package (Gilks *et al.*, 1994) incorporates many of these ideas and provides a very powerful tool for formulating and learning complex probability models. The first algorithms for learning Bayes net structures used conditional independence tests (Pearl, 1988; Pearl and Verma, 1991). Spirtes *et al.* (1993) developed a comprehensive approach

and the TETRAD package for Bayes net learning using similar ideas. Algorithmic improvements since then led to a clear victory in the 2001 KDD Cup data mining competition for a Bayes net learning method (Cheng *et al.*, 2002). (The specific task here was a bioinformatics problem with 139,351 features!) A structure-learning approach based on maximizing likelihood was developed by Cooper and Herskovits (1992) and improved by Heckerman *et al.* (1994). Friedman and Goldszmidt (1996) pointed out the influence of the representation of local conditional distributions on the learned structure.

The general problem of learning probability models with hidden variables and missing data was addressed by the EM algorithm (Dempster *et al.*, 1977), which was abstracted from several existing methods including the Baum–Welch algorithm for HMM learning (Baum and Petrie, 1966). (Dempster himself views EM as a schema rather than an algorithm, since a good deal of mathematical work may be required before it can be applied to a new family of distributions.) EM is now one of the most widely used algorithms in science, and McLachlan and Krishnan (1997) devote an entire book to the algorithm and its properties. The specific problem of learning mixture models, including mixtures of Gaussians, is covered by Titterington *et al.* (1985). Within AI, the first successful system that used EM for mixture modeling was AUTOCLASS (Cheeseman *et al.*, 1988; Cheeseman and Stutz, 1996). AUTOCLASS has been applied to a number of real-world scientific classification tasks, including the discovery of new types of stars from spectral data (Goebel *et al.*, 1989) and new classes of proteins and introns in DNA/protein sequence databases (Hunter and States, 1992).

An EM algorithm for learning Bayes nets with hidden variables was developed by Lauritzen (1995). Gradient-based techniques have also proved effective for Bayes nets as well as dynamic Bayes nets (Russell *et al.*, 1995; Binder *et al.*, 1997a). The structural EM algorithm was developed by (Friedman, 1998). The ability to learn the structure of Bayesian networks is closely connected to the issue of recovering *causal* information from data. That is, is it possible to learn Bayes nets in such a way that the recovered network structure indicates real causal influences? For many years, statisticians avoided this question, believing that observational data (as opposed to data generated from experimental trials) could yield only correlational information—after all, any two variables that appear related might in fact be influenced by third, unknown causal factor rather than influencing each other directly. Pearl (2000) has presented convincing arguments to the contrary, showing that there are in fact many cases where causality can be ascertained and developing the **causal network** formalism to express causes and the effects of intervention as well as ordinary conditional probabilities.

CAUSAL NETWORK

Nearest-neighbor 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. Efficient indexing schemes for finding nearest neighbors are studied within the algorithms community (see, e.g., Indyk, 2000). Kernel density estimation, also called **Parzen window** density estimation, was investigated initially by Rosenblatt (1956) and Parzen (1962). Since that time, a huge literature has developed investigating the properties of various estimators. Devroye (1987) gives a thorough introduction.

The literature on neural networks is rather too large (approximately 100,000 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). 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). Meanwhile, in Britain, W. Ross Ashby (also a pioneer of cybernetics; see Ashby, 1940), Alan Turing, Grey Walter, and others formed the Ratio Club for "those who had Wiener's ideas before Wiener's book appeared." Ashby's *Design for a Brain* (1948, 1952) put forth the idea that intelligence could be created by the use of **homeostatic** devices containing appropriate feedback loops to achieve stable adaptive behavior. 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 the 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).

Support vector machines were originated in the 1990s (Cortes and Vapnik, 1995) and are now the subject of a fast-growing literature, including textbooks such as Cristianini and Shawe-Taylor (2000). They have proven to be very popular and effective for tasks such as text categorization (Joachims, 2001), bioinformatics research (Brown *et al.*, 2000), and natural language processing, such as the handwritten digit recognition of DeCoste and Scholkopf (2002). A related technique that also uses the "kernel trick" to implicitly represent an exponential feature space is the voted perceptron (Collins and Duffy, 2002).

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.

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.[16] 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.

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. 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 labelled training data, and the second phase, although supervised, is efficient. See Bishop (1995) for more details.

HOPFIELD NETWORKS

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 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

ASSOCIATIVE MEMORY

[16] 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.

taken. Notice that the original photographs are not stored separately in the network; each 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 MACHINES

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.)

The first application of the ideas underlying kernel machines was by Aizerman *et al.* (1964), but the full development of the theory, under the heading of support vector machines, is due to Vladimir Vapnik and colleagues (Boser *et al.*, 1992; Vapnik, 1998). Cristianini and Shawe-Taylor (2000) and Schölkopf and Smola (2002) provide rigorous introductions; a friendlier exposition appears in the *AI Magazine* article by Cristianini and Schölkopf (2002).

The material in this chapter brings together work from the fields of statistics, pattern recognition, and neural networks, so the story has been told many times in many ways. Good texts on Bayesian statistics include those by DeGroot (1970), Berger (1985), and Gelman *et al.* (1995). Hastie *et al.* (2001) provide an excellent introduction to statistical learning methods. For pattern classification, the classic text for many years has been Duda and Hart (1973), now updated (Duda *et al.*, 2001). For neural nets, Bishop (1995) and Ripley (1996) are the leading texts. The field of computational neuroscience is covered by Dayan and Abbott (2001). The most important conference on neural networks and related topics is the annual NIPS (Neural Information Processing Conference) conference, whose proceedings are published as the series *Advances in Neural Information Processing Systems*. Papers on learning Bayesian networks also appear in the *Uncertainty in AI* and *Machine Learning* conferences and in several statistics conferences. Journals specific to neural networks include *Neural Computation*, *Neural Networks*, and the *IEEE Transactions on Neural Networks*.

EXERCISES

20.1 The data used for Figure 20.1 can be viewed as being generated by h_5. For each of the other four hypotheses, generate a data set of length 100 and plot the corresponding graphs for $P(h_i|d_1,\ldots,d_m)$ and $P(D_{m+1}=lime|d_1,\ldots,d_m)$. Comment on your results.

20.2 Repeat Exercise 20.1, this time plotting the values of $P(D_{m+1}=lime|h_{\mathrm{MAP}})$ and $P(D_{m+1}=lime|h_{\mathrm{ML}})$.

20.3 Suppose that Ann's utilities for cherry and lime candies are c_A and ℓ_A, whereas Bob's utilities are c_B and ℓ_B. (But once Ann has unwrapped a piece of candy, Bob won't buy it.) Presumably, if Bob likes lime candies much more than Ann, it would be wise to sell for Ann

to sell her bag of candies once she is sufficiently sure of its lime content. On the other hand, if Ann unwraps too many candies in the process, the bag will be worth less. Discuss the problem of determining the optimal point at which to sell the bag. Determine the expected utility of the optimal procedure, given the prior distribution from Section 20.1.

20.4 Two statisticians go to the doctor and are both given the same prognosis: A 40% chance that the problem is the deadly disease A, and a 60% chance of the fatal disease B. Fortunately, there are anti-A and anti-B drugs that are inexpensive, 100% effective, and free of side-effects. The statisticians have the choice of taking one drug, both, or neither. What will the first statistician (an avid Bayesian) do? How about the second statistician, who always uses the maximum likelihood hypothesis?

The doctor does some research and discovers that disease B actually comes in two versions, dextro-B and levo-B, which are equally likely and equally treatable by the anti-B drug. Now that there are three hypotheses, what will the two statisticians do?

20.5 Explain how to apply the boosting method of Chapter 18 to naive Bayes learning. Test the performance of the resulting algorithm on the restaurant learning problem.

20.6 Consider m data points (x_j, y_j), where the y_js are generated from the x_js according to the linear Gaussian model in Equation (20.5). Find the values of θ_1, θ_2, and σ that maximize the conditional log likelihood of the data.

20.7 Consider the noisy-OR model for fever described in Section 14.3. Explain how to apply maximum-likelihood learning to fit the parameters of such a model to a set of complete data. (*Hint*: use the chain rule for partial derivatives.)

20.8 This exercise investigates properties of the Beta distribution defined in Equation (20.6).

a. By integrating over the range $[0, 1]$, show that the normalization constant for the distribution $\text{beta}[a, b]$ is given by $\alpha = \Gamma(a + b)/\Gamma(a)\Gamma(b)$ where $\Gamma(x)$ is the **Gamma function**, defined by $\Gamma(x + 1) = x \cdot \Gamma(x)$ and $\Gamma(1) = 1$. (For integer x, $\Gamma(x + 1) = x!$.)

GAMMA FUNCTION

b. Show that the mean is $a/(a + b)$.

c. Find the mode(s) (the most likely value(s) of θ).

d. Describe the distribution $\text{beta}[\epsilon, \epsilon]$ for very small ϵ. What happens as such a distribution is updated?

20.9 Consider an arbitrary Bayesian network, a complete data set for that network, and the likelihood for the data set according to the network. Give a simple proof that the likelihood of the data cannot decrease if we add a new link to the network and recompute the maximum-likelihood parameter values.

20.10 Consider the application of EM to learn the parameters for the network in Figure 20.10(a), given the true parameters in Equation (20.7).

a. Explain why the EM algorithm would not work if there were just two attributes in the model rather than three.

b. Show the calculations for the first iteration of EM starting from Equation (20.8).

c. What happens if we start with all the parameters set to the same value p? (*Hint*: you may find it helpful to investigate this empirically before deriving the general result.)

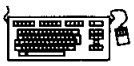
d. Write out an expression for the log likelihood of the tabulated candy data on page 729 in terms of the parameters, calculate the partial derivatives with respect to each parameter, and investigate the nature of the fixed point reached in part (c).

20.11 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.

20.12 Construct a support vector machine that computes the XOR function. It will be convenient to use values of 1 and –1 instead of 1 and 0 for the inputs and for the outputs. So an example looks like $([-1, 1], 1)$ or $([-1, -1], -1)$. It is typical to map an input $\mathbf{x}$ into a space consisting of five dimensions, the two original dimensions x_1 and x_2, and the three combination x_1^2, x_2^2 and $x_1\, x_2$. But for this exercise we will consider only the two dimensions 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.

20.13 A simple perceptron cannot represent XOR (or, generally, the parity function of its inputs). Describe what happens to the weights of a four-input, step-function perceptron, beginning with all weights set to 0.1, as examples of the parity function arrive.

20.14 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?



20.15 Consider the following set of examples, each with six inputs and one target output:

I_1	1	1	1	1	1	1	1	0	0	0	0	0	0	0
I_2	0	0	0	1	1	0	0	1	1	0	1	0	1	1
I_3	1	1	1	0	1	0	0	1	1	0	0	0	1	1
I_4	0	1	0	0	1	0	0	1	0	1	1	1	0	1
I_5	0	0	1	1	0	1	1	0	1	1	0	0	1	0
I_6	0	0	0	1	0	1	0	1	1	0	1	1	1	0
T	1	1	1	1	1	1	0	1	0	0	0	0	0	0

a. Run the perceptron learning rule on these data and show the final weights.

b. Run the decision tree learning rule, and show the resulting decision tree.

c. Comment on your results.

20.16 Starting from Equation (20.13), show that $\partial L / \partial W_j = Err \times a_j$.

20.17 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 $\mathbf{W}$, write down equations for the value of the units in the output layer as a function of $\mathbf{W}$ and the input layer $\mathbf{I}$, without any explicit mention to 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), this time for a network with any number of hidden layers. What can you conclude about linear activation functions?

20.18 Implement a data structure for layered, feed-forward neural networks, remembering to provide the information needed for both forward evaluation and backward propagation. Using this data structure, write a function NEURAL-NETWORK-OUTPUT that takes an example and a network and computes the appropriate output values.

20.19 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 to zero.)

20.20 The network in Figure 20.24 has four hidden nodes. This number was chosen somewhat arbitrarily. Run systematic experiments to measure the learning curves for networks with different numbers of hidden nodes. What is the optimal number? Would it be possible to use a cross-validation method to find the best network before the fact?

20.21 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 labelled 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 $N - 1$. From this it follows that the **VC dimension** (see Chapter 18) of linear halfspaces in dimension $N - 1$ is N.

21 REINFORCEMENT LEARNING

In which we examine how an agent can learn from success and failure, from reward and punishment.

21.1 INTRODUCTION

Chapters 18 and 20 covered learning methods that learn functions and probability models from example. In this chapter, we will study how agents can learn *what to do*, particularly when there is no teacher telling the agent what action to take in each circumstance.

For example, we know an agent can learn to play chess by supervised learning—by being given examples of game situations along with the best moves for those situations. But if there is no friendly teacher providing examples, what can the agent do? By trying random moves, the agent can eventually build a predictive model of its environment: what the board will be like after it makes a given move and even how the opponent is likely to reply in a given situation. The problem is this: *without some feedback about what is good and what is bad, the agent will have no grounds for deciding which move to make.* The agent needs to know that something good has happened when it wins and that something bad has happened when it loses. This kind of feedback is called a **reward**, or **reinforcement**. In games like chess, the reinforcement is received only at the end of the game. In other environments, the rewards come more frequently. In ping-pong, each point scored can be considered a reward; when learning to crawl, any forward motion is an achievement. Our framework for agents regards the reward as *part* of the input percept, but the agent must be "hardwired" to recognize that part as a reward rather than as just another sensory input. Thus, animals seem to be hardwired to recognize pain and hunger as negative rewards and pleasure and food intake as positive rewards. Reinforcement has been carefully studied by animal psychologists for over 60 years.

Rewards were introduced in Chapter 17, where they served to define optimal policies in **Markov decision processes** (MDPs). An optimal policy is a policy that maximizes the expected total reward. The task of **reinforcement learning** is to use observed rewards to learn an optimal (or nearly optimal) policy for the environment. Whereas in Chapter 17 the agent

has a complete model of the environment and knows the reward function, here we assume no prior knowledge of either. Imagine playing a new game whose rules you don't know; after a hundred or so moves, your opponent announces, "You lose." This is reinforcement learning in a nutshell.

In many complex domains, reinforcement learning is the only feasible way to train a program to perform at high levels. For example, in game playing, it is very hard for a human to provide accurate and consistent evaluations of large numbers of positions, which would be needed to train an evaluation function directly from examples. Instead, the program can be told when it has won or lost, and it can use this information to learn an evaluation function that gives reasonably accurate estimates of the probability of winning from any given position. Similarly, it is extremely difficult to program an agent to fly a helicopter; yet given appropriate negative rewards for crashing, wobbling, or deviating from a set course, an agent can learn to fly by itself.

Reinforcement learning might be considered to encompass all of AI: an agent is placed in an environment and must learn to behave successfully therein. To keep the chapter manageable, we will concentrate on simple settings and simple agent designs. For the most part, we will assume a fully observable environment, so that the current state is supplied by each percept. On the other hand, we will assume that the agent does not know how the environment works or what its actions do, and we will allow for probabilistic action outcomes. We will consider three of the agent designs first introduced in Chapter 2:

- A **utility-based agent** learns a utility function on states and uses it to select actions that maximize the expected outcome utility.
- Q-LEARNING ACTION-VALUE A **Q-learning** agent learns an **action-value** function, or Q-function, giving the expected utility of taking a given action in a given state.
- A **reflex agent** learns a policy that maps directly from states to actions.

A utility-based agent must also have a model of the environment in order to make decisions, because it must know the states to which its actions will lead. For example, in order to make use of a backgammon evaluation function, a backgammon program must know what its legal moves are *and how they affect the board position.* Only in this way can it apply the utility function to the outcome states. A Q-learning agent, on the other hand, can compare the values of its available choices without needing to know their outcomes, so it does not need a model of the environment. On the other hand, because they do not know where their actions lead, Q-learning agents cannot look ahead; this can seriously restrict their ability to learn, as we shall see.

PASSIVE LEARNING ACTIVE LEARNING EXPLORATION We begin in Section 21.2 with **passive learning**, where the agent's policy is fixed and the task is to learn the utilities of states (or state–action pairs); this could also involve learning a model of the environment. Section 21.3 covers **active learning**, where the agent must also learn what to do. The principal issue is **exploration**: an agent must experience as much as possible of its environment in order to learn how to behave in it. Section 21.4 discusses how an agent can use inductive learning to learn much faster from its experiences. Section 21.5 covers methods for learning direct policy representations in reflex agents. An understanding of Markov decision processes (Chapter 17) is essential for this chapter.

21.2 PASSIVE REINFORCEMENT LEARNING

To keep things simple, we start with the case of a passive learning agent using a state-based representation in a fully observable environment. In passive learning, the agent's policy π is fixed: in state s, it always executes the action $\pi(s)$. Its goal is simply to learn how good the policy is—that is, to learn the utility function $U^{\pi}(s)$. We will use as our example the 4×3 world introduced in Chapter 17. Figure 21.1 shows a policy for that world and the corresponding utilities. Clearly, the passive learning task is similar to the **policy evaluation** task, part of the **policy iteration** algorithm described in Section 17.3. The main difference is that the passive learning agent does not know the **transition model** $T(s, a, s')$, which specifies the probability of reaching state s' from state s after doing action a; nor does it know the **reward function** $R(s)$, which specifies the reward for each state.

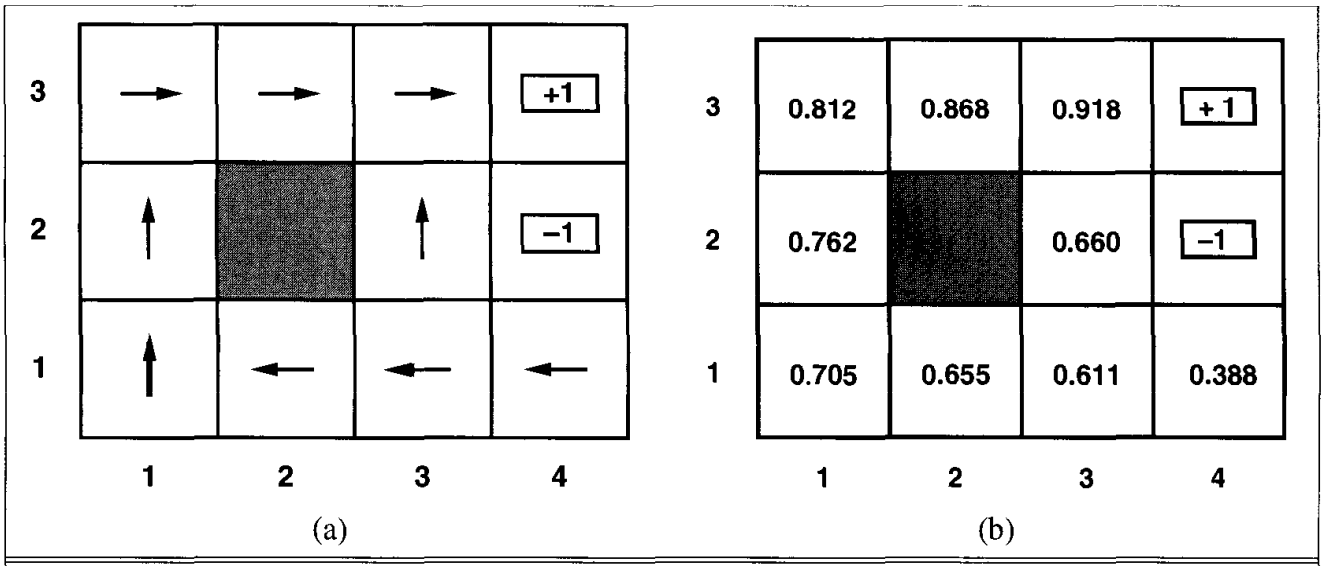


Figure 21.1 (a) A policy π for the 4×3 world; this policy happens to be optimal with rewards of $R(s) = -0.04$ in the nonterminal states and no discounting. (b) The utilities of the states in the 4×3 world, given policy π.

TRIAL

The agent executes a set of **trials** in the environment using its policy π. In each trial, the agent starts in state (1,1) and experiences a sequence of state transitions until it reaches one of the terminal states, (4,2) or (4,3). Its percepts supply both the current state and the reward received in that state. Typical trials might look like this:

$$(1,1)_{\text{-.04}} \rightarrow (1,2)_{\text{-.04}} \rightarrow (1,3)_{\text{-.04}} \rightarrow (1,2)_{\text{-.04}} \rightarrow (1,3)_{\text{-.04}} \rightarrow (2,3)_{\text{-.04}} \rightarrow (3,3)_{\text{-.04}} \rightarrow (4,3)_{\text{+1}}$$
$$(1,1)_{\text{-.04}} \rightarrow (1,2)_{\text{-.04}} \rightarrow (1,3)_{\text{-.04}} \rightarrow (2,3)_{\text{-.04}} \rightarrow (3,3)_{\text{-.04}} \rightarrow (3,2)_{\text{-.04}} \rightarrow (3,3)_{\text{-.04}} \rightarrow (4,3)_{\text{+1}}$$
$$(1,1)_{\text{-.04}} \rightarrow (2,1)_{\text{-.04}} \rightarrow (3,1)_{\text{-.04}} \rightarrow (3,2)_{\text{-.04}} \rightarrow (4,2)_{\text{-1}} .$$

Note that each state percept is subscripted with the reward received. The object is to use the information about rewards to learn the expected utility $U^{\pi}(s)$ associated with each nonterminal state s. The utility is defined to be the expected sum of (discounted) rewards obtained if

policy π is followed. As in Equation (17.3) on page 619, this is written as

$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t) \mid \pi, s_0 = s\right] . \tag{21.1}$$

We will include a **discount factor** γ in all of our equations, but for the 4×3 world we will set $\gamma = 1$.

Direct utility estimation

DIRECT UTILITY ESTIMATION
ADAPTIVE CONTROL THEORY

A simple method for **direct utility estimation** was invented in the late 1950s in the area of **adaptive control theory** by Widrow and Hoff (1960). The idea is that the utility of a state is the expected total reward from that state onward, and each trial provides a *sample* of this value for each state visited. For example, the first trial in the set of three given earlier provides a sample total reward of 0.72 for state (1,1), two samples of 0.76 and 0.84 for (1,2), two samples of 0.80 and 0.88 for (1,3), and so on. Thus, at the end of each sequence, the algorithm calculates the observed reward-to-go for each state and updates the estimated utility for that state accordingly, just by keeping a running average for each state in a table. In the limit of infinitely many trials, the sample average will converge to the true expectation in Equation (21.1).

It is clear that direct utility estimation is just an instance of supervised learning where each example has the state as input and the observed reward-to-go as output. This means that we have reduced reinforcement learning to a standard inductive learning problem, as discussed in Chapter 18. Section 21.4 discusses the use of more powerful kinds of representations for the utility function, such as neural networks. Learning techniques for those representations can be applied directly to the observed data.

Direct utility estimation succeeds in reducing the reinforcement learning problem to an inductive learning problem, about which much is known. Unfortunately, it misses a very important source of information, namely, the fact that the utilities of states are not independent! *The utility of each state equals its own reward plus the expected utility of its successor states.* That is, the utility values obey the Bellman equations for a fixed policy (see also Equation (17.10)):

$$U^{\pi}(s) = R(s) + \gamma \sum_{s'} T(s, \pi(s), s') U^{\pi}(s') . \tag{21.2}$$

By ignoring the connections between states, direct utility estimation misses opportunities for learning. For example, the second of the three trials given earlier reaches the state (3,2), which has not previously been visited. The next transition reaches (3,3), which is known from the first trial to have a high utility. The Bellman equation suggests immediately that (3,2) is also likely to have a high utility, because it leads to (3,3), but direct utility estimation learns nothing until the end of the trial. More broadly, we can view direct utility estimation as searching in a hypothesis space for U that is much larger than it needs to be, in that it includes many functions that violate the Bellman equations. For this reason, the algorithm often converges very slowly.

Adaptive dynamic programming

ADAPTIVE DYNAMIC PROGRAMMING

In order to take advantage of the constraints between states, an agent must learn how states are connected. An **adaptive dynamic programming** (or **ADP**) agent works by learning the transition model of the environment as it goes along and solving the corresponding Markov decision process using a dynamic programming method. For a passive learning agent, this means plugging the learned transition model $T(s, \pi(s), s')$ and the observed rewards $R(s)$ into the Bellman equations (21.2) to calculate the utilities of the states. As we remarked in our discussion of policy iteration in Chapter 17, these equations are linear (no maximization involved) so they can be solved using any linear algebra package. Alternatively, we can adopt the approach of **modified policy iteration** (see page 625), using a simplified value iteration process to update the utility estimates after each change to the learned model. Because the model usually changes only slightly with each observation, the value iteration process can use the previous utility estimates as initial values and should converge quite quickly.

The process of learning the model itself is easy, because the environment is fully observable. This means that we have a supervised learning task where the input is a state–action pair and the output is the resulting state. In the simplest case, we can represent the transition model as a table of probabilities. We keep track of how often each action outcome occurs and estimate the transition probability $T(s, a, s')$ from the frequency with which s' is reached when executing a in s.[1] For example, in the three traces given on page 765, *Right* is executed three times in (1,3) and two out of three times the resulting state is (2,3), so $T((1,3), \mathit{Right}, (2,3))$ is estimated to be 2/3.

The full agent program for a passive ADP agent is shown in Figure 21.2. Its performance on the 4×3 world is shown in Figure 21.3. In terms of how quickly its value estimates improve, the ADP agent does as well as possible, subject to its ability to learn the transition model. In this sense, it provides a standard against which to measure other reinforcement learning algorithms. It is, however, somewhat intractable for large state spaces. In backgammon, for example, it would involve solving roughly 10^{50} equations in 10^{50} unknowns.

Temporal difference learning

It is possible to have (almost) the best of both worlds; that is, one can approximate the constraint equations shown earlier without solving them for all possible states. *The key is to use the observed transitions to adjust the values of the observed states so that they agree with the constraint equations.* Consider, for example, the transition from (1,3) to (2,3) in the second trial on page 765. Suppose that, as a result of the first trial, the utility estimates are $U^\pi(1,3) = 0.84$ and $U^\pi(2,3) = 0.92$. Now, if this transition occurred all the time, we would expect the utilities to obey

$$U^\pi(1,3) = -0.04 + U^\pi(2,3) \ ,$$

so $U^\pi(1,3)$ would be 0.88. Thus, its current estimate of 0.84 might be a little low and should be increased. More generally, when a transition occurs from state s to state s', we apply the

[1] This is the maximum likelihood estimate, as discussed in Chapter 20. A Bayesian update with a Dirichlet prior might work better.

function PASSIVE-ADP-AGENT(*percept*) **returns** an action
inputs: *percept*, a percept indicating the current state s' and reward signal r'
static: π, a fixed policy
mdp, an MDP with model T, rewards R, discount γ
U, a table of utilities, initially empty
N_{sa}, a table of frequencies for state-action pairs, initially zero
$N_{sas'}$, a table of frequencies for state-action-state triples, initially zero
s, a, the previous state and action, initially null

if s' is new **then do** $U[s'] \leftarrow r'$; $R[s'] \leftarrow r'$
if s is not null **then do**
increment $N_{sa}[s, a]$ and $N_{sas'}[s, a, s']$
for each t such that $N_{sas'}[s, a, t]$ is nonzero **do**
$T[s, a, t] \leftarrow N_{sas'}[s, a, t] \,/\, N_{sa}[s, a]$
$U \leftarrow$ VALUE-DETERMINATION(π, U, mdp)
if TERMINAL?$[s']$ **then** $s, a \leftarrow$ null **else** $s, a \leftarrow s', \pi[s']$
return a

Figure 21.2 A passive reinforcement learning agent based on adaptive dynamic programming. To simplify the code, we have assumed that each percept can be divided into a perceived state and a reward signal.

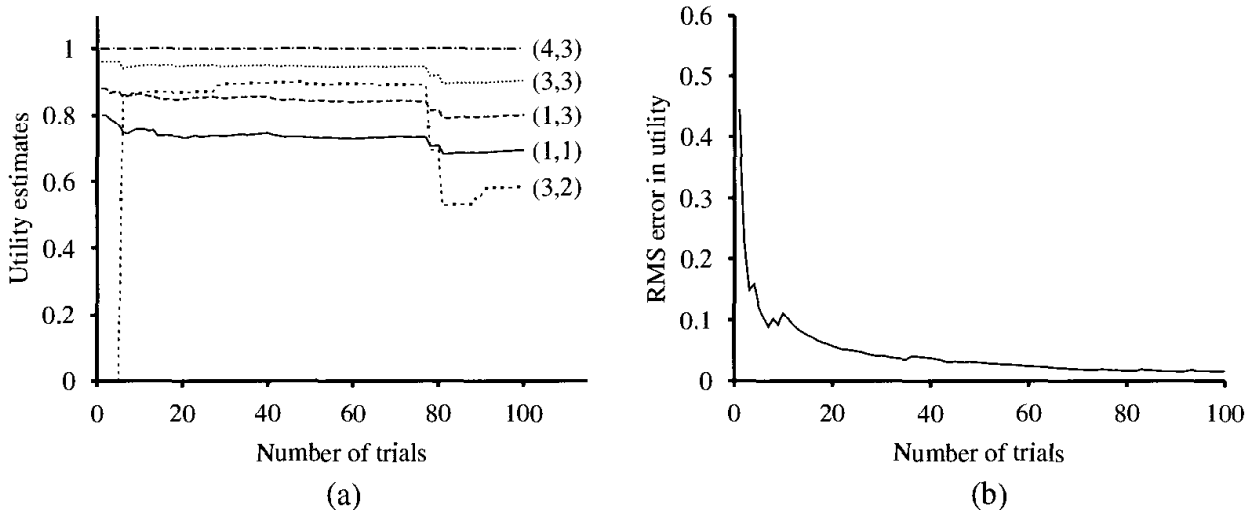


Figure 21.3 The passive ADP learning curves for the 4×3 world, given the optimal policy shown in Figure 21.1. (a) The utility estimates for a selected subset of states, as a function of the number of trials. Notice the large changes occurring around the 78th trial—this is the first time that the agent falls into the -1 terminal state at (4,2). (b) The root-mean-square error in the estimate for $U(1, 1)$, averaged over 20 runs of 100 trials each.

```
function PASSIVE-TD-AGENT(percept) returns an action
  inputs: percept, a percept indicating the current state s' and reward signal r'
  static: π, a fixed policy
          U, a table of utilities, initially empty
          N_s, a table of frequencies for states, initially zero
          s, a, r, the previous state, action, and reward, initially null

  if s' is new then U[s'] ← r'
  if s is not null then do
     increment N_s[s]
     U[s] ← U[s] + α(N_s[s])(r + γ U[s'] − U[s])
  if TERMINAL?[s'] then s, a, r ← null else s, a, r ← s', π[s'], r'
  return a
```

Figure 21.4 A passive reinforcement learning agent that learns utility estimates using temporal differences.

following update to $U^\pi(s)$:

$$U^\pi(s) \leftarrow U^\pi(s) + \alpha(R(s) + \gamma U^\pi(s') - U^\pi(s)) \ . \tag{21.3}$$

TEMPORAL-DIFFERENCE

Here, α is the **learning rate** parameter. Because this update rule uses the difference in utilities between successive states, it is often called the **temporal-difference**, or **TD**, equation.

The basic idea of all temporal-difference methods is, first to define the conditions that hold locally when the utility estimates are correct, and then, to write an update equation that moves the estimates toward this ideal "equilibrium" equation. In the case of passive learning, the equilibrium is given by Equation (21.2). Now Equation (21.3) does in fact cause the agent to reach the equilibrium given by Equation (21.2), but there is some subtlety involved. First, notice that the update involves only the observed successor s', whereas the actual equilibrium conditions involve all possible next states. One might think that this causes an improperly large change in $U^\pi(s)$ when a very rare transition occurs; but, in fact, because rare transitions occur only rarely, the *average value* of $U^\pi(s)$ will converge to the correct value. Furthermore, if we change α from a fixed parameter to a function that decreases as the number of times a state has been visited increases, then $U(s)$ itself will converge to the correct value.[2] This gives us the agent program shown in Figure 21.4. Figure 21.5 illustrates the performance of the passive TD agent on the 4×3 world. It does not learn quite as fast as the ADP agent and shows much higher variability, but it is much simpler and requires much less computation per observation. Notice that *TD does not need a model to perform its updates*. The environment supplies the connection between neighboring states in the form of observed transitions.

The ADP approach and the TD approach are actually closely related. Both try to make local adjustments to the utility estimates in order to make each state "agree" with its successors. One difference is that TD adjusts a state to agree with its *observed* successor (Equa-

[2] Technically, we require that $\sum_{n=1}^{\infty} \alpha(n) = \infty$ and $\sum_{n=1}^{\infty} \alpha^2(n) < \infty$. The decay $\alpha(n) = 1/n$ satisfies these conditions. In Figure 21.5 we have used $\alpha(n) = 60/(59 + n)$.

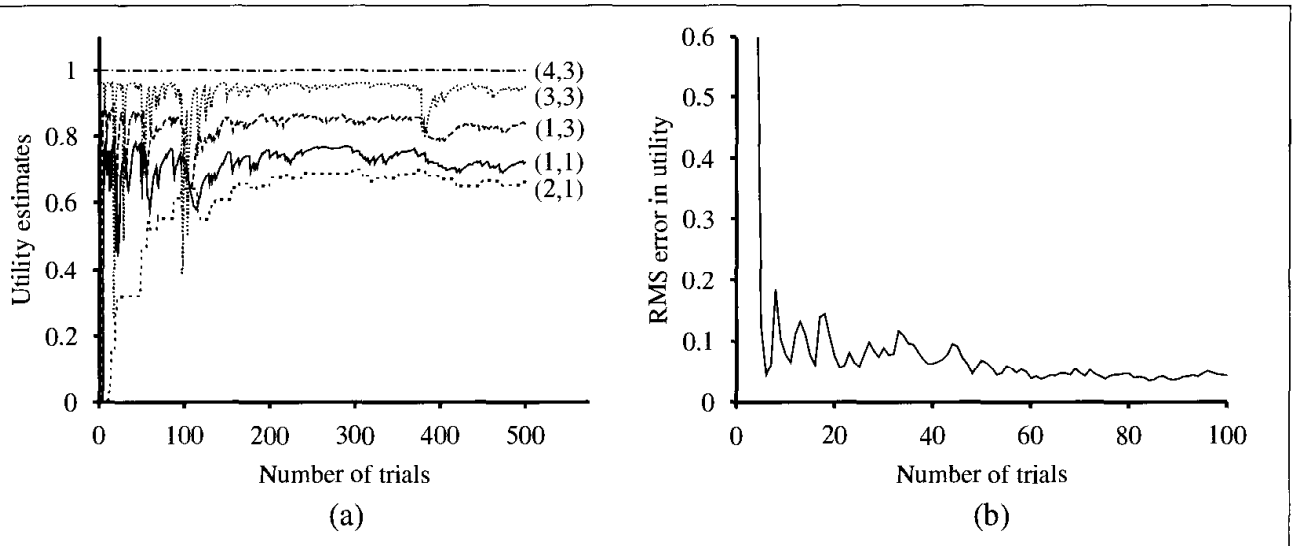


Figure 21.5 The TD learning curves for the 4×3 world. (a) The utility estimates for a selected subset of states, as a function of the number of trials. (b) The root-mean-square error in the estimate for $U(1,1)$, averaged over 20 runs of 500 trials each. Only the first 100 trials are shown to enable comparison with Figure 21.3.

tion (21.3)), whereas ADP adjusts the state to agree with *all* of the successors that might occur, weighted by their probabilities (Equation (21.2)). This difference disappears when the effects of TD adjustments are averaged over a large number of transitions, because the frequency of each successor in the set of transitions is approximately proportional to its probability. A more important difference is that whereas TD makes a single adjustment per observed transition, ADP makes as many as it needs to restore consistency between the utility estimates U and the environment model T. Although the observed transition makes only a local change in T, its effects might need to be propagated throughout U. Thus, TD can be viewed as a crude but efficient first approximation to ADP.

Each adjustment made by ADP could be seen, from the TD point of view, as a result of a "pseudo-experience" generated by simulating the current environment model. It is possible to extend the TD approach to use an environment model to generate several pseudo-experiences—transitions that the TD agent can imagine *might* happen, given its current model. For each observed transition, the TD agent can generate a large number of imaginary transitions. In this way, the resulting utility estimates will approximate more and more closely those of ADP—of course, at the expense of increased computation time.

In a similar vein, we can generate more efficient versions of ADP by directly approximating the algorithms for value iteration or policy iteration. Recall that full value iteration can be intractable when the number of states is large. Many of the adjustment steps, however, are extremely tiny. One possible approach to generating reasonably good answers quickly is to bound the number of adjustments made after each observed transition. One can also use a heuristic to rank the possible adjustments so as to carry out only the most significant ones.

PRIORITIZED SWEEPING

The **prioritized sweeping** heuristic prefers to make adjustments to states whose *likely* successors have just undergone a *large* adjustment in their own utility estimates. Using heuristics like this, approximate ADP algorithms usually can learn roughly as fast as full ADP, in terms

of the number of training sequences, but can be several orders of magnitude more efficient in terms of computation. (See Exercise 21.3.) This enables them to handle state spaces that are far too large for full ADP. Approximate ADP algorithms have an additional advantage: in the early stages of learning a new environment, the environment model T often will be far from correct, so there is little point in calculating an exact utility function to match it. An approximation algorithm can use a minimum adjustment size that decreases as the environment model becomes more accurate. This eliminates the very long value iterations that can occur early in learning due to large changes in the model.

21.3 Active Reinforcement Learning

A passive learning agent has a fixed policy that determines its behavior. An active agent must decide what actions to take. Let us begin with the adaptive dynamic programming agent and consider how it must be modified to handle this new freedom.

First, the agent will need to learn a complete model with outcome probabilities for all actions, rather than just the model for the fixed policy. The simple learning mechanism used by PASSIVE-ADP-AGENT will do just fine for this. Next, we need to take into account the fact that the agent has a choice of actions. The utilities it needs to learn are those defined by the *optimal* policy; they obey the Bellman equations given on page 619, which we repeat here:

$$U(s) = R(s) + \gamma \max_a \sum_{s'} T(s, a, s')U(s') \,. \tag{21.4}$$

These equations can be solved to obtain the utility function U using the value iteration or policy iteration algorithms from Chapter 17. The final issue is what to do at each step. Having obtained a utility function U that is optimal for the learned model, the agent can extract an optimal action by one-step look-ahead to maximize the expected utility; alternatively, if it uses policy iteration, the optimal policy is already available, so it should simply execute the action the optimal policy recommends. Or should it?

Exploration

Figure 21.6 shows the results of one sequence of trials for an ADP agent that follows the recommendation of the optimal policy for the learned model at each step. The agent *does not* learn the true utilities or the true optimal policy! What happens instead is that, in the 39th trial, it finds a policy that reaches the +1 reward along the lower route via (2,1), (3,1), (3,2), and (3,3). (See Figure 21.6.) After experimenting with minor variations, from the 276th trial onward it sticks to that policy, never learning the utilities of the other states and never finding the optimal route via (1,2), (1,3), and (2,3). We call this agent the **greedy agent**. Repeated experiments show that the greedy agent *very seldom* converges to the optimal policy for this environment and sometimes converges to really horrendous policies.

GREEDY AGENT

How can it be that choosing the optimal action leads to suboptimal results? The answer is that the learned model is not the same as the true environment; what is optimal in the

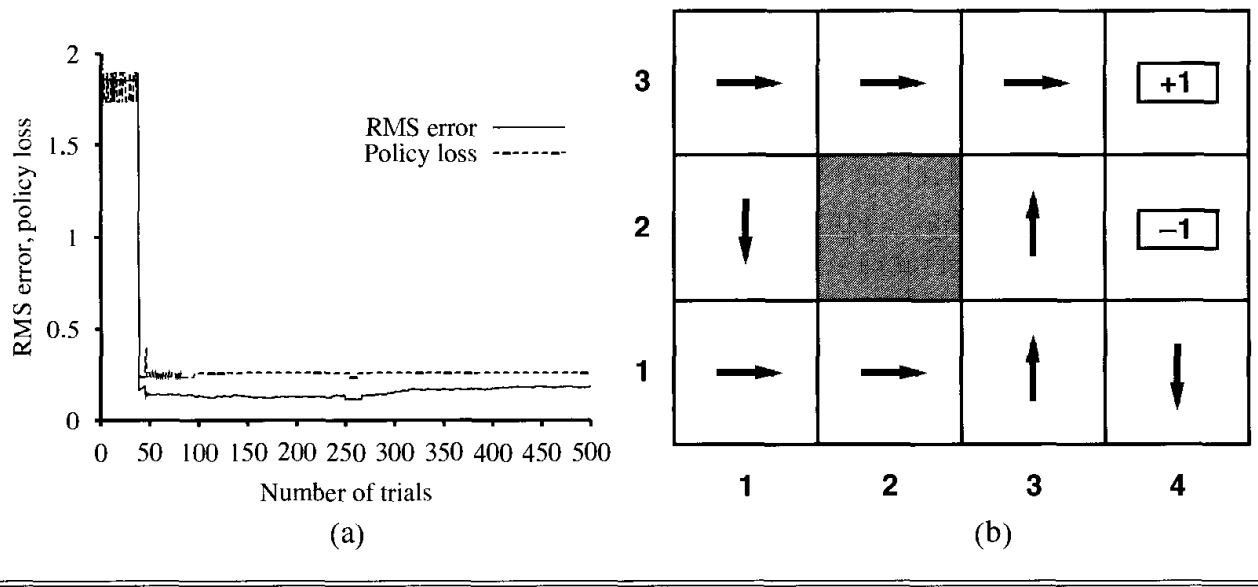


Figure 21.6 Performance of a greedy ADP agent that executes the action recommended by the optimal policy for the learned model. (a) RMS error in the utility estimates averaged over the nine nonterminal squares. (b) The suboptimal policy to which the greedy agent converges in this particular sequence of trials.

learned model can therefore be suboptimal in the true environment. Unfortunately, the agent does not know what the true environment is, so it cannot compute the optimal action for the true environment. What, then, is to be done?

What the greedy agent has overlooked is that actions do more than provide rewards according to the current learned model; they also contribute to learning the true model by affecting the percepts that are received. By improving the model, the agent will receive greater rewards in the future.[3] An agent therefore must make a trade-off between **exploitation** to maximize its reward—as reflected in its current utility estimates—and **exploration** to maximize its long-term well-being. Pure exploitation risks getting stuck in a rut. Pure exploration to improve one's knowledge is of no use if one never puts that knowledge into practice. In the real world, one constantly has to decide between continuing in a comfortable existence and striking out into the unknown in the hopes of discovering a new and better life. With greater understanding, less exploration is necessary.

EXPLOITATION

EXPLORATION

Can we be a little more precise than this? Is there an *optimal* exploration policy? It turns out that this question has been studied in depth in the subfield of statistical decision theory that deals with so-called **bandit problems**. (See sidebar.)

BANDIT PROBLEMS

Although bandit problems are extremely difficult to solve exactly to obtain an *optimal* exploration method, it is nonetheless possible to come up with a *reasonable* scheme that will eventually lead to optimal behavior by the agent. Technically, any such scheme needs to be greedy in the limit of infinite exploration, or **GLIE**. A GLIE scheme must try each action in each state an unbounded number of times to avoid having a finite probability that an optimal action is missed because of an unusually bad series of outcomes. An ADP agent

GLIE

[3] Notice the direct analogy to the theory of information value in Chapter 16.

EXPLORATION AND BANDITS

In Las Vegas, a *one-armed bandit* is a slot machine. A gambler can insert a coin, pull the lever, and collect the winnings (if any). An n-**armed bandit** has n levers. The gambler must choose which lever to play on each successive coin—the one that has paid off best, or maybe one that has not been tried?

The n-armed bandit problem is a formal model for real problems in many vitally important areas, such as deciding on the annual budget for AI research and development. Each arm corresponds to an action (such as allocating $20 million for the development of new AI textbooks), and the payoff from pulling the arm corresponds to the benefits obtained from taking the action (immense). Exploration, whether it is exploration of a new research field or exploration of a new shopping mall, is risky, is expensive, and has uncertain payoffs; on the other hand, failure to explore at all means that one never discovers *any* actions that are worthwhile.

To formulate a bandit problem properly, one must define exactly what is meant by optimal behavior. Most definitions in the literature assume that the aim is to maximize the expected total reward obtained over the agent's lifetime. These definitions require that the expectation be taken over the possible worlds that the agent could be in, as well as over the possible results of each action sequence in any given world. Here, a "world" is defined by the transition model $T(s, a, s')$. Thus, in order to act optimally, the agent needs a prior distribution over the possible models. The resulting optimization problems are usually wildly intractable.

In some cases—for example, when the payoff of each machine is independent and discounted rewards are used—it is possible to calculate a **Gittins index** for each slot machine (Gittins, 1989). The index is a function only of the number of times the slot machine has been played and how much it has paid off. The index for each machine indicates how worthwhile it is to invest more, based on a combination of expected return and expected value of information. Choosing the machine with the highest index value gives an optimal exploration policy. Unfortunately, no way has been found to extend Gittins indices to sequential decision problems.

One can use the theory of n-armed bandits to argue for the reasonableness of the selection strategy in genetic algorithms. (See Chapter 4.) If you consider each arm in an n-armed bandit problem to be a possible string of genes, and the investment of a coin in one arm to be the reproduction of those genes, then genetic algorithms allocate coins optimally, given an appropriate set of independence assumptions.

using such a scheme will eventually learn the true environment model. A GLIE scheme must also eventually become greedy, so that the agent's actions become optimal with respect to the learned (and hence the true) model.

There are several GLIE schemes; one of the simplest is to have the agent choose a random action a fraction $1/t$ of the time and to follow the greedy policy otherwise. While this does eventually converge to an optimal policy, it can be extremely slow. A more sensible approach would give some weight to actions that the agent has not tried very often, while tending to avoid actions that are believed to be of low utility. This can be implemented by altering the constraint equation (21.4) so that it assigns a higher utility estimate to relatively unexplored state–action pairs. Essentially, this amounts to an optimistic prior over the possible environments and causes the agent to behave initially as if there were wonderful rewards scattered all over the place. Let us use $U^+(s)$ to denote the optimistic estimate of the utility (i.e., the expected reward-to-go) of the state s, and let $N(a, s)$ be the number of times action a has been tried in state s. Suppose we are using value iteration in an ADP learning agent; then we need to rewrite the update equation (i.e., Equation (17.6)) to incorporate the optimistic estimate. The following equation does this:

$$U^+(s) \leftarrow R(s) + \gamma \max_a f\left(\sum_{s'} T(s, a, s')U^+(s'),\ N(a, s)\right) . \tag{21.5}$$

EXPLORATION FUNCTION

Here, $f(u, n)$ is called the **exploration function**. It determines how greed (preference for high values of u) is traded off against curiosity (preference for low values of n—actions that have not been tried often). The function $f(u, n)$ should be increasing in u and decreasing in n. Obviously, there are many possible functions that fit these conditions. One particularly simple definition is

$$f(u, n) = \begin{cases} R^+ & \text{if } n < N_e \\ u & \text{otherwise} \end{cases}$$

where R^+ is an optimistic estimate of the best possible reward obtainable in any state and N_e is a fixed parameter. This will have the effect of making the agent try each action–state pair at least N_e times.

The fact that U^+ rather than U appears on the right-hand side of Equation (21.5) is very important. As exploration proceeds, the states and actions near the start state might well be tried a large number of times. If we used U, the more pessimistic utility estimate, then the agent would soon become disinclined to explore further afield. The use of U^+ means that the benefits of exploration are propagated back from the edges of unexplored regions, so that actions that lead *toward* unexplored regions are weighted more highly, rather than just actions that are themselves unfamiliar. The effect of this exploration policy can be seen clearly in Figure 21.7, which shows a rapid convergence toward optimal performance, unlike that of the greedy approach. A very nearly optimal policy is found after just 18 trials. Notice that the utility estimates themselves do not converge as quickly. This is because the agent stops exploring the unrewarding parts of the state space fairly soon, visiting them only "by accident" thereafter. However, it makes perfect sense for the agent not to care about the exact utilities of states that it knows are undesirable and can be avoided.

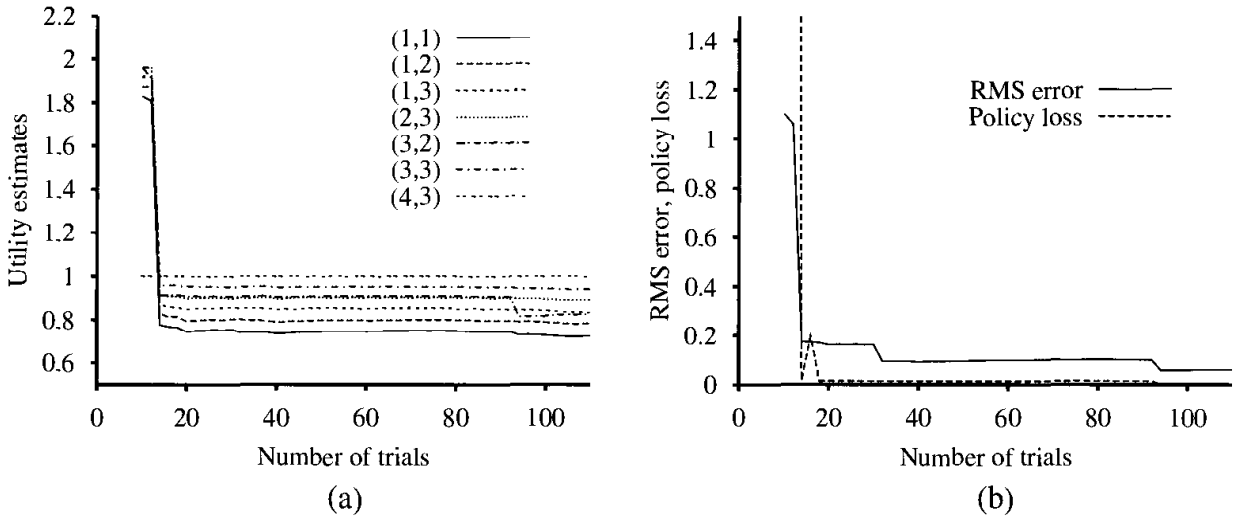


Figure 21.7 Performance of the exploratory ADP agent. using $R^+ = 2$ and $N_e = 5$. (a) Utility estimates for selected states over time. (b) The RMS error in utility values and the associated policy loss.

Learning an Action-Value Function

Now that we have an active ADP agent, let us consider how to construct an active temporal-difference learning agent. The most obvious change from the passive case is that the agent is no longer equipped with a fixed policy, so, if it learns a utility function U, it will need to learn a model in order to be able to choose an action based on U via one-step look-ahead. The model acquisition problem for the TD agent is identical to that for the ADP agent. What of the TD update rule itself? Perhaps surprisingly, the update rule (21.3) remains unchanged. This might seem odd, for the following reason: Suppose the agent takes a step that normally leads to a good destination, but because of nondeterminism in the environment the agent ends up in a catastrophic state. The TD update rule will take this as seriously as if the outcome had been the normal result of the action, whereas one might suppose that, because the outcome was a fluke, the agent should not worry about it too much. In fact, of course, the unlikely outcome will occur only infrequently in a large set of training sequences; hence in the long run its effects will be weighted proportionally to its probability, as we would hope. Once again, it can be shown that the TD algorithm will converge to the same values as ADP as the number of training sequences tends to infinity.

There is an alternative TD method called **Q-learning** that learns an action-value representation instead of learning utilities. We will use the notation $Q(a, s)$ to denote the value of doing action a in state s. Q-values are directly related to utility values as follows:

$$U(s) = \max_a Q(a, s) \,. \tag{21.6}$$

MODEL-FREE

Q-functions may seem like just another way of storing utility information, but they have a very important property: *a TD agent that learns a Q-function does not need a model for either learning or action selection.* For this reason, Q-learning is called a **model-free** method. As with utilities, we can write a constraint equation that must hold at equilibrium when the

function Q-LEARNING-AGENT(*percept*) **returns** an action
inputs: *percept*, a percept indicating the current state s' and reward signal r'
static: Q, a table of action values index by state and action
N_{sa}, a table of frequencies for state-action pairs
s, a, r, the previous state, action, and reward, initially null

if s is not null **then do**
increment $N_{sa}[s, a]$
$Q[a, s] \leftarrow Q[a, s] + \alpha(N_{sa}[s, a])(r + \gamma \max_{a'} Q[a', s'] - Q[a, s])$
if TERMINAL?$[s']$ **then** $s, a, r \leftarrow$ null
else $s, a, r \leftarrow s', \text{argmax}_{a'} f(Q[a', s'], N_{sa}[a', s']), r'$
return a

Figure 21.8 An exploratory Q-learning agent. It is an active learner that learns the value $Q(a, s)$ of each action in each situation. It uses the same exploration function f as the exploratory ADP agent, but avoids having to learn the transition model because the Q-value of a state can be related directly to those of its neighbors.

Q-values are correct:

$$Q(a, s) = R(s) + \gamma \sum_{s'} T(s, a, s') \max_{a'} Q(a', s') \ . \tag{21.7}$$

As in the ADP learning agent, we can use this equation directly as an update equation for an iteration process that calculates exact Q-values, given an estimated model. This does, however, require that a model also be learned because the equation uses $T(s, a, s')$. The temporal-difference approach, on the other hand, requires no model. The update equation for TD Q-learning is

$$Q(a, s) \leftarrow Q(a, s) + \alpha(R(s) + \gamma \max_{a'} Q(a', s') - Q(a, s)) \ , \tag{21.8}$$

which is calculated whenever action a is executed in state s leading to state s'.

The complete agent design for an exploratory Q-learning agent using TD is shown in Figure 21.8. Notice that it uses exactly the same exploration function f as that used by the exploratory ADP agent—hence the need to keep statistics on actions taken (the table N). If a simpler exploration policy is used—say, acting randomly on some fraction of steps, where the fraction decreases over time—then we can dispense with the statistics.

The Q-learning agent learns the optimal policy for the 4×3 world, but does so at a much slower rate than the ADP agent. This is because TD does not enforce consistency among values via the model. The comparison raises a general question: is it better to learn a model and a utility function or to learn an action-value function with no model? In other words, what is the best way to represent the agent function? This is an issue at the foundations of artificial intelligence. As we stated in Chapter 1, one of the key historical characteristics of much of AI research is its (often unstated) adherence to the **knowledge-based** approach. This amounts to an assumption that the best way to represent the agent function is to build a representation of some aspects of the environment in which the agent is situated.

Some researchers, both inside and outside AI, have claimed that the availability of model-free methods such as Q-learning means that the knowledge-based approach is unnecessary. There is, however, little to go on but intuition. Our intuition, for what it's worth, is that as the environment becomes more complex, the advantages of a knowledge-based approach become more apparent. This is borne out even in games such as chess, checkers (draughts), and backgammon (see next section), where efforts to learn an evaluation function by means of a model have met with more success than Q-learning methods.

21.4 GENERALIZATION IN REINFORCEMENT LEARNING

So far, we have assumed that the utility functions and Q-functions learned by the agents are represented in tabular form with one output value for each input tuple. Such an approach works reasonably well for small state spaces, but the time to convergence and (for ADP) the time per iteration increase rapidly as the space gets larger. With carefully controlled, approximate ADP methods, it might be possible to handle 10,000 states or more. This suffices for two-dimensional maze-like environments, but more realistic worlds are out of the question. Chess and backgammon are tiny subsets of the real world, yet their state spaces contain on the order of 10^{50} to 10^{120} states. It would be absurd to suppose that one must visit all these states in order to learn how to play the game!

FUNCTION APPROXIMATION

One way to handle such problems is to use **function approximation**, which simply means using any sort of representation for the function other than a table. The representation is viewed as approximate because it might not be the case that the *true* utility function or Q-function can be represented in the chosen form. For example, in Chapter 6 we described an **evaluation function** for chess that is represented as a weighted linear function of a set of **features** (or **basis functions**) $f_1, \ldots, f_n$:

BASIS FUNCTIONS

$$\hat{U}_\theta(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + \cdots + \theta_n f_n(s) .$$

A reinforcement learning algorithm can learn values for the parameters $\theta = \theta_1, \ldots, \theta_n$ such that the evaluation function $\hat{U}_\theta$ approximates the true utility function. Instead of, say, 10^{120} values in a table, this function approximator is characterized by, say, $n = 20$ parameters—an *enormous* compression. Although no one knows the true utility function for chess, no one believes that it can be represented exactly in 20 numbers. If the approximation is good enough, however, the agent might still play excellent chess.[4]

Function approximation makes it practical to represent utility functions for very large state spaces, but that is not its principal benefit. *The compression achieved by a function approximator allows the learning agent to generalize from states it has visited to states it has not visited.* That is, the most important aspect of function approximation is not that it

[4] We do know that the exact utility function can be represented in a page or two of Lisp, Java, or C++. That is, it can be represented by a program that solves the game exactly every time it is called. We are interested only in function approximators that use a *reasonable* amount of computation. It might in fact be better to learn a very simple function approximator and combine it with a certain amount of look-ahead search. The trade-offs involved are currently not well understood.

requires less space, but that it allows for inductive generalization over input states. To give you some idea of the power of this effect: by examining only one in every 10^{44} of the possible backgammon states, it is possible to learn a utility function that allows a program to play as well as any human (Tesauro, 1992).

On the flip side, of course, there is the problem that there could fail to be any function in the chosen hypothesis space that approximates the true utility function sufficiently well. As in all inductive learning, there is a trade-off between the size of the hypothesis space and the time it takes to learn the function. A larger hypothesis space increases the likelihood that a good approximation can be found, but also means that convergence is likely to be delayed.

Let us begin with the simplest case, which is direct utility estimation. (See Section 21.2.) With function approximation, this is an instance of **supervised learning**. For example, suppose we represent the utilities for the 4×3 world using a simple linear function. The features of the squares are just their x and y coordinates, so we have

$$\hat{U}_\theta(x, y) = \theta_0 + \theta_1 x + \theta_2 y \ . \tag{21.9}$$

Thus, if $(\theta_0, \theta_1, \theta_2) = (0.5, 0.2, 0.1)$, then $\hat{U}_\theta(1, 1) = 0.8$. Given a collection of trials, we obtain a set of sample values of $\hat{U}_\theta(x, y)$, and we can find the best fit, in the sense of minimizing the squared error, using standard linear regression. (See Chapter 20.)

For reinforcement learning, it makes more sense to use an *online* learning algorithm that updates the parameters after each trial. Suppose we run a trial and the total reward obtained starting at (1,1) is 0.4. This suggests that $\hat{U}_\theta(1, 1)$, currently 0.8, is too large and must be reduced. How should the parameters be adjusted to achieve this? As with neural network learning, we write an error function and compute its gradient with respect to the parameters. If $u_j(s)$ is the observed total reward from state s onward in the jth trial, then the error is defined as (half) the squared difference of the predicted total and the actual total: $E_j(s) = (\hat{U}_\theta(s) - u_j(s))^2/2$. The rate of change of the error with respect to each parameter θ_i is $\partial E_j / \partial \theta_i$, so to move the parameter in the direction of decreasing the error, we want

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial E_j(s)}{\partial \theta_i} = \theta_i + \alpha \left(u_j(s) - \hat{U}_\theta(s)\right) \frac{\partial \hat{U}_\theta(s)}{\partial \theta_i} \ . \tag{21.10}$$

WIDROW–HOFF RULE
DELTA RULE

This is called the **Widrow–Hoff rule**, or the **delta rule**, for online least-squares. For the linear function approximator $\hat{U}_\theta(s)$ in Equation (21.9), we get three simple update rules:

$$\begin{aligned}
\theta_0 &\leftarrow \theta_0 + \alpha \left(u_j(s) - \hat{U}_\theta(s)\right) , \\
\theta_1 &\leftarrow \theta_1 + \alpha \left(u_j(s) - \hat{U}_\theta(s)\right) x , \\
\theta_2 &\leftarrow \theta_2 + \alpha \left(u_j(s) - \hat{U}_\theta(s)\right) y .
\end{aligned}$$

We can apply these rules to the example where $\hat{U}_\theta(1, 1)$ is 0.8 and $u_j(1, 1)$ is 0.4. θ_0, θ_1, and θ_2 are all decreased by 0.4α, which reduces the error for (1,1). Notice that changing the θ_is *also changes the values of* $\hat{U}_\theta$ *for every other state*! This is what we mean by saying that function approximation allows a reinforcement learner to generalize from its experiences.

We expect that the agent will learn faster if it uses a function approximator, provided that the hypothesis space is not too large, but includes some functions that are a reasonably good fit to the true utility function. Exercise 21.7 asks you to evaluate the performance of direct utility estimation, both with and without function approximation. The improvement

in the 4×3 world is noticeable but not dramatic, because this is a very small state space to begin with. The improvement is much greater in a 10×10 world with a +1 reward at (10,10). This world is well suited for a linear utility function because the true utility function is smooth and nearly linear. (See Exercise 21.10.) If we put the +1 reward at (5,5), the true utility is more like a pyramid and the function approximator in Equation (21.9) will fail miserably. All is not lost, however! Remember that what matters for linear function approximation is that the function be linear in the *parameters*—the features themselves can be arbitrary nonlinear functions of the state variables. Hence, we can include a term such as $\theta_3 = \sqrt{(x - x_g)^2 + (y - y_g)^2}$ that measures the distance to the goal.

We can apply these ideas equally well to temporal-difference learners. All we need do is adjust the parameters to try to reduce the temporal difference between successive states. The new versions of the TD and Q-learning equations (21.3 and 21.8) are

$$\theta_i \leftarrow \theta_i + \alpha\,[R(s) + \gamma\,\hat{U}_\theta(s') - \hat{U}_\theta(s)]\frac{\partial \hat{U}_\theta(s)}{\partial \theta_i} \tag{21.11}$$

for utilities and

$$\theta_i \leftarrow \theta_i + \alpha\,[R(s) + \gamma \max_{a'} \hat{Q}_\theta(a', s') - \hat{Q}_\theta(a, s)]\frac{\partial \hat{Q}_\theta(a, s)}{\partial \theta_i} \tag{21.12}$$

for Q-values. These update rules can be shown to converge to the closest possible[5] approximation to the true function when the function approximator is *linear* in the parameters. Unfortunately, all bets are off when *nonlinear* functions—such as neural networks—are used. There are some very simple cases in which the parameters can go off to infinity even though there are good solutions in the hypothesis space. There are more sophisticated algorithms that can avoid these problems, but at present reinforcement learning with general function approximators remains a delicate art.

Function approximation can also be very helpful for learning a model of the environment. Remember that learning a model for an *observable* environment is a *supervised* learning problem, because the next percept gives the outcome state. Any of the supervised learning methods in Chapter 18 can be used, with suitable adjustments for the fact that we need to predict a complete state description rather than just a Boolean classification or a single real value. For example, if the state is defined by n Boolean variables, we will need to learn n Boolean functions to predict all the variables. For a *partially observable* environment, the learning problem is much more difficult. If we know what the hidden variables are and how they are causally related to each other and to the observable variables, then we can fix the structure of a dynamic Bayesian network and use the EM algorithm to learn the parameters, as was described in Chapter 20. Inventing the hidden variables and learning the model structure are still open problems.

We now turn to examples of large-scale applications of reinforcement learning. We will see that, in cases where a utility function (and hence a model) is used, the model is usually taken as given. For example, in learning an evaluation function for backgammon, it is normally assumed that the legal moves and their effects are known in advance.

[5] The definition of distance between utility functions is rather technical; see Tsitsiklis and Van Roy (1997).

Applications to game-playing

The first significant application of reinforcement learning was also the first significant learning program of any kind—the checker-playing program written by Arthur Samuel (1959, 1967). Samuel first used a weighted linear function for the evaluation of positions, using up to 16 terms at any one time. He applied a version of Equation (21.11) to update the weights. There were some significant differences, however, between his program and current methods. First, he updated the weights using the difference between the current state and the backed-up value generated by full look-ahead in the search tree. This works fine, because it amounts to viewing the state space at a different granularity. A second difference was that the program did *not* use any observed rewards! That is, the values of terminal states were ignored. This means that it is quite possible for Samuel's program not to converge, or to converge on a strategy designed to lose rather than to win. He managed to avoid this fate by insisting that the weight for material advantage should always be positive. Remarkably, this was sufficient to direct the program into areas of weight space corresponding to good checker play.

Gerry Tesauro's TD-Gammon system (1992) forcefully illustrates the potential of reinforcement learning techniques. In earlier work (Tesauro and Sejnowski, 1989), Tesauro tried learning a neural network representation of $Q(a, s)$ directly from examples of moves labeled with relative values by a human expert. This approach proved extremely tedious for the expert. It resulted in a program, called NEUROGAMMON, that was strong by computer standards, but not competitive with human experts. The TD-Gammon project was an attempt to learn from self-play alone. The only reward signal was given at the end of each game. The evaluation function was represented by a fully connected neural network with a single hidden layer containing 40 nodes. Simply by repeated application of Equation (21.11), TD-Gammon learned to play considerably better than Neurogammon, even though the input representation contained just the raw board position with no computed features. This took about 200,000 training games and two weeks of computer time. Although that may seem like a lot of games, it is only a vanishingly small fraction of the state space. When precomputed features were added to the input representation, a network with 80 hidden units was able, after 300,000 training games, to reach a standard of play comparable to that of the top three human players worldwide. Kit Woolsey, a top player and analyst, said that "There is no question in my mind that its positional judgment is far better than mine."

Application to robot control

CART–POLE
INVERTED PENDULUM

The setup for the famous **cart–pole** balancing problem, also known as the **inverted pendulum**, is shown in Figure 21.9. The problem is to control the position x of the cart so that the pole stays roughly upright ($\theta \approx \pi/2$), while staying within the limits of the cart track as shown. Over two thousand papers in reinforcement learning and control theory have been published on this seemingly simple problem. The cart–pole problem differs from the problems described earlier in that the state variables x, θ, $\dot{x}$, and $\dot{\theta}$ are continuous. The actions are usually discrete: jerk left or jerk right, the so-called **bang-bang control** regime.

BANG-BANG CONTROL

The earliest work on learning for this problem was carried out by Michie and Chambers (1968). Their BOXES algorithm was able to balance the pole for over an hour after only

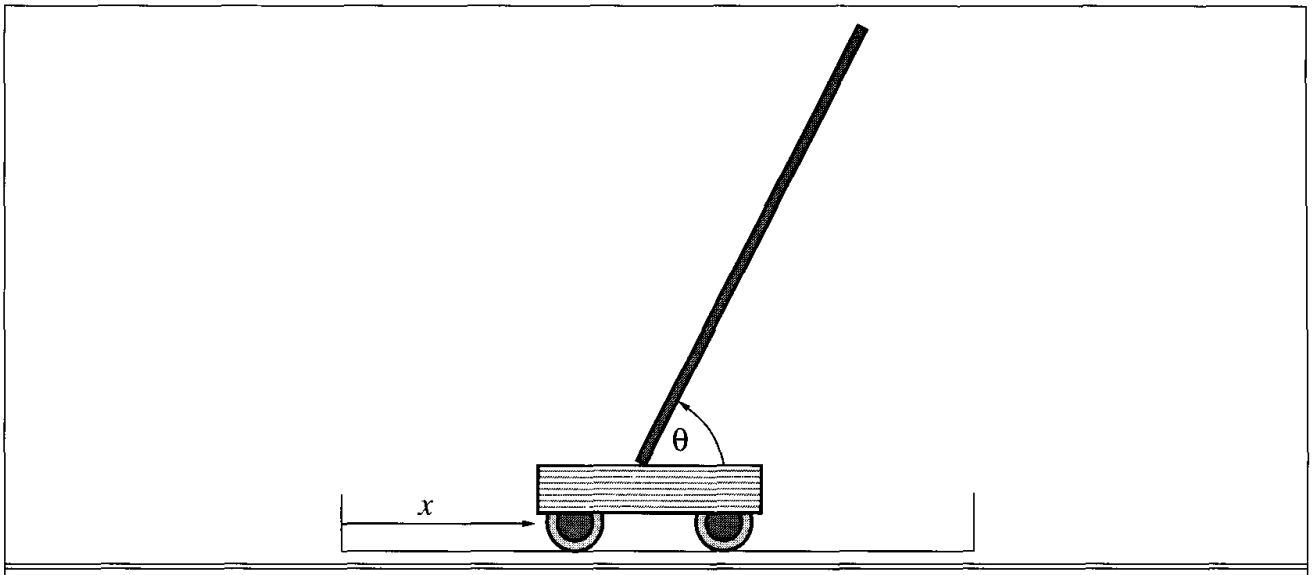


Figure 21.9 Setup for the problem of balancing a long pole on top of a moving cart. The cart can be jerked left or right by a controller that observes x, θ, $\dot{x}$, and $\dot{\theta}$.

about 30 trials. Moreover, unlike many subsequent systems, BOXES was implemented with a real cart and pole, not a simulation. The algorithm first discretized the four-dimensional state space into boxes—hence the name. It then ran trials until the pole fell over or the cart hit the end of the track. Negative reinforcement was associated with the final action in the final box and then propagated back through the sequence. It was found that the discretization caused some problems when the apparatus was initialized in a position different from those used in training, suggesting that generalization was not perfect. Improved generalization and faster learning can be obtained using an algorithm that *adaptively* partitions the state space according to the observed variation in the reward. Nowadays, balancing a *triple* inverted pendulum is a common exercise—a feat far beyond the capabilities of most humans.

21.5 POLICY SEARCH

POLICY SEARCH

The final approach we will consider for reinforcement learning problems is called **policy search**. In some ways, policy search is the simplest of all the methods in this chapter: the idea is to keep twiddling the policy as long as its performance improves, then stop.

Let us begin with the policies themselves. Remember that a policy π is a function that maps states to actions. We are interested primarily in *parameterized* representations of π that have far fewer parameters than there are states in the state space (just as in the preceding section). For example, we could represent π by a collection of parameterized Q-functions, one for each action, and take the action with the highest predicted value:

$$\pi(s) = \max_a \hat{Q}_\theta(a, s) \ . \tag{21.13}$$

Each Q-function could be a linear function of the parameters θ, as in Equation (21.9), or it could be a nonlinear function such as a neural network. Policy search will then adjust the parameters θ to improve the policy. Notice that if the policy is represented by Q-functions,

then policy search results in a process that learns Q-functions. *This process is not the same as Q-learning!* In Q-learning with function approximation, the algorithm finds a value of θ such that $\hat{Q}_\theta$ is "close" to Q^*, the optimal Q-function. Policy search, on the other hand, finds a value of θ that results in good performance; the values found may differ very substantially.[6] Another clear example of the difference is the case where $\pi(s)$ is calculated using, say, depth-10 look-ahead search with an approximate utility function $\hat{U}_\theta$. The value of θ that gives good play may be a long way from making $\hat{U}_\theta$ resemble the true utility function.

One problem with policy representations of the kind given in Equation (21.13) is that the policy is a *discontinuous* function of the parameters when the actions are discrete.[7] That is, there will be values of θ such that an infinitesimal change in θ causes the policy to switch from one action to another. This means that the value of the policy may also change discontinuously, which makes gradient-based search difficult. For this reason, policy search methods often use a **stochastic policy** representation $\pi_\theta(s, a)$, which specifies the *probability* of selecting action a in state s. One popular representation is the **softmax function**:

STOCHASTIC POLICY
SOFTMAX FUNCTION

$$\pi_\theta(s, a) = \exp(\hat{Q}_\theta(a, s))/\sum_{a'} \exp(\hat{Q}_\theta(a', s)) \ .$$

Softmax becomes nearly deterministic if one action is much better than the others, but it always gives a differentiable function of θ; hence, the value of the policy (which depends in a continuous fashion on the action selection probabilities) is a differentiable function of θ.

Now let us look at methods for improving the policy. We start with the simplest case: a deterministic policy and a deterministic environment. In this case, evaluating the policy is trivial: we simply execute it and observe the accumulated reward; this gives us the **policy value** $\rho(\theta)$. Improving the policy is just a standard optimization problem, as described in Chapter 4. We can follow the **policy gradient** vector $\nabla_\theta \rho(\theta)$ provided $\rho(\theta)$ is differentiable. Alternatively, we can follow the **empirical gradient** by hillclimbing—i.e., evaluating the change in policy for small increments in each parameter value. With the usual caveats, this process will converge to a local optimum in policy space.

POLICY VALUE
POLICY GRADIENT

When the environment (or the policy) is stochastic, things get more difficult. Suppose we are trying to do hillclimbing, which requires comparing $\rho(\theta)$ and $\rho(\theta + \Delta\theta)$ for some small $\Delta\theta$. The problem is that the total reward on each trial may vary widely, so estimates of the policy value from a small number of trials will be quite unreliable; trying to compare two such estimates will be even more unreliable. One solution is simply to run lots of trials, measuring the sample variance and using it to determine that enough trials have been run to get a reliable indication of the direction of improvement for $\rho(\theta)$. Unfortunately, this is impractical for many real problems where each trial may be expensive, time-consuming, and perhaps even dangerous.

For the case of a stochastic policy $\pi_\theta(s, a)$, it is possible to obtain an unbiased estimate of the gradient at θ, $\nabla_\theta \rho(\theta)$, directly from the results of trials executed at θ. For simplicity, we will derive this estimate for the simple case of a nonsequential environment in which the

[6] Trivially, the approximate Q-function defined by $\hat{Q}_\theta(a, s) = Q^*(a, s)/10$ gives optimal performance, even though it is not at all close to Q^*.

[7] For a continuous action space, the policy can be a smooth function of the parameters.

reward is obtained immediately after acting in the start state s_0. In this case, the policy value is just the expected value of the reward, and we have

$$\nabla_\theta \rho(\theta) = \nabla_\theta \sum_a \pi_\theta(s_0, a) R(a) = \sum_a (\nabla_\theta \pi_\theta(s_0, a)) R(a) .$$

Now we perform a simple trick so that this summation can be approximated by samples generated from the probability distribution defined by $\pi_\theta(s_0, a)$. Suppose that we have N trials in all and the action taken on the jth trial is a_j. Then

$$\nabla_\theta \rho(\theta) = \sum_a \pi_\theta(s_0, a) \cdot \frac{(\nabla_\theta \pi_\theta(s_0, a)) R(a)}{\pi_\theta(s_0, a)} \approx \frac{1}{N} \sum_{j=1}^{N} \frac{(\nabla_\theta \pi_\theta(s_0, a_j)) R(a_j)}{\pi_\theta(s_0, a_j)} .$$

Thus, the true gradient of the policy value is approximated by a sum of terms involving the gradient of the action selection probability in each trial. For the sequential case, this generalizes to

$$\nabla_\theta \rho(\theta) \approx \frac{1}{N} \sum_{j=1}^{N} \frac{(\nabla_\theta \pi_\theta(s, a_j)) R_j(s)}{\pi_\theta(s, a_j)}$$

for each state s visited, where a_j is executed in s on the jth trial and $R_j(s)$ is the total reward received from state s onwards in the jth trial. The resulting algorithm is called REINFORCE (Williams, 1992); it is usually much more effective than hillclimbing using lots of trials at each value of θ. It is still much slower than necessary, however.

Consider the following task: given two blackjack[8] programs, determine which is best. One way to do this is to have each play against a standard "dealer" for a certain number of hands and then to measure their respective winnings. The problem with this, as we have seen, is that the winnings of each program fluctuate widely depending on whether it receives good or bad cards. An obvious solution is to generate a certain number of hands in advance and *set of hands*. In this way, we eliminate the measurement error due to differences in the cards received. This is the idea behind the PEGASUS algorithm (Ng and Jordan, 2000). The algorithm is applicable to domains for which a simulator is available so that the "random" outcomes of actions can be repeated. The algorithm works by generating in advance N sequences of random numbers, each of which can be used to run a trial of any policy. Policy search is carried out by evaluating each candidate policy using the *same* set of random sequences to determine the action outcomes. It can be shown that the number of random sequences required to ensure that the value of *every* policy is well-estimated depends only on the complexity of the policy space, and not at all on the complexity of the underlying domain. The PEGASUS algorithm has been used to develop effective policies for several domains, including autonomous helicopter flight (see Figure 21.10).

[8] Also known as twenty-one or pontoon.

Figure 21.10 Superimposed time-lapse images of an autonomous helicopter performing a very difficult "nose-in circle" maneuver. The helicopter is under the control of a policy developed by the PEGASUS policy search algorithm. A simulator model was developed by observing the effects of various control manipulations on the real helicopter; then the algorithm was run on the simulator model overnight. A variety of controllers were developed for different maneuvers. In all cases, performance far exceeded that of an expert human pilot using remote control. (Image courtesy of Andrew Ng.)

21.6 SUMMARY

This chapter has examined the reinforcement learning problem: how an agent can become proficient in an unknown environment, given only its percepts and occasional rewards. Reinforcement learning can be viewed as a microcosm for the entire AI problem, but it is studied in a number of simplified settings to facilitate progress. The major points are:

- The overall agent design dictates the kind of information that must be learned. The three main designs we covered were the model-based design, using a model T and a utility function U; the model-free design, using an action-value function Q; and the reflex design, using a policy π.
- Utilities can be learned using three approaches:
  1. **Direct utility estimation** uses the total observed reward-to-go for a given state as direct evidence for learning its utility.
  2. **Adaptive dynamic programming** (ADP) learns a model and a reward function from observations and then uses value or policy iteration to obtain the utilities or an optimal policy. ADP makes optimal use of the local constraints on utilities of states imposed through the neighborhood structure of the environment.
  3. **Temporal-difference** (TD) methods update utility estimates to match those of successor states. They can be viewed as simple approximations to the ADP approach

that require no model for the learning process. Using a learned model to generate pseudoexperiences can, however, result in faster learning.

- Action-value functions, or Q-functions, can be learned by an ADP approach or a TD approach. With TD, Q-learning requires no model in either the learning or action-selection phase. This simplifies the learning problem but potentially restricts the ability to learn in complex environments, because the agent cannot simulate the results of possible courses of action.
- When the learning agent is responsible for selecting actions while it learns, it must trade off the estimated value of those actions against the potential for learning useful new information. An exact solution of the exploration problem is infeasible, but some simple heuristics do a reasonable job.
- In large state spaces, reinforcement learning algorithms must use an approximate functional representation in order to generalize over states. The temporal-difference signal can be used directly to update parameters in representations such as neural networks.
- **Policy search** methods operate directly on a representation of the policy, attempting to improve it based on observed performance. The variance in the performance in a stochastic domain is a serious problem; for simulated domains this can be overcome by fixing the randomness in advance.

Because of its potential for eliminating hand coding of control strategies, reinforcement learning continues to be one of the most active areas of machine learning research. Applications in robotics promise to be particularly valuable; these will require methods for handling *continuous*, *high-dimensional*, *partially observable* environments in which successful behaviors may consist of thousands or even millions of primitive actions.

Bibliographical and Historical Notes

Turing (1948, 1950) proposed the reinforcement learning approach, although he was not convinced of its effectiveness, writing, "the use of punishments and rewards can at best be a part of the teaching process." Arthur Samuel's work (1959) was probably the earliest successful machine learning research. Although this work was informal and had a number of flaws, it contained most of the modern ideas in reinforcement learning, including temporal differencing and function approximation. Around the same time, researchers in adaptive control theory (Widrow and Hoff, 1960), building on work by Hebb (1949), were training simple networks using the delta rule. (This early connection between neural networks and reinforcement learning may have led to the persistent misperception that the latter is a subfield of the former.) The cart–pole work of Michie and Chambers (1968) can also be seen as a reinforcement learning method with a function approximator. The psychological literature on reinforcement learning is much older; Hilgard and Bower (1975) provide a good survey. Direct evidence for the operation of reinforcement learning in animals has been provided by investigations into the foraging behavior of bees; there is a clear neural correlate of the reward signal in the form of a large neuron mapping from the nectar intake sensors directly

to the motor cortex (Montague *et al.*, 1995). Research using single-cell recording suggests that the dopamine system in primate brains implements something resembling value function learning (Schultz *et al.*, 1997).

The connection between reinforcement learning and Markov decision processes was first made by Werbos (1977), but the development of reinforcement learning in AI stems from work at the University of Massachusetts in the early 1980s (Barto *et al.*, 1981). The paper by Sutton (1988) provides a good historical overview. Equation (21.3) in this chapter is a special case for $\lambda = 0$ of Sutton's general TD(λ) algorithm. TD(λ) updates the values of all states in a sequence leading up to each transition by an amount that drops off as λ^t for states t steps in the past. TD(1) is identical to the Widrow–Hoff or delta rule. Boyan (2002), building on work by Bradtke and Barto (1996), argues that TD(λ) and related algorithms make inefficient use of experiences; essentially, they are online regression algorithms that converge much more slowly than offline regression. His LSTD(λ) is an online algorithm that gives the same results as offline regression.

The combination of temporal difference learning with the model-based generation of simulated experiences was proposed in Sutton's DYNA architecture (Sutton, 1990). The idea of prioritized sweeping was introduced independently by Moore and Atkeson (1993) and Peng and Williams (1993). Q-learning was developed in Watkins's Ph.D. thesis (1989).

Bandit problems, which model the problem of exploration for nonsequential decisions, are analyzed in depth by Berry and Fristedt (1985). Optimal exploration strategies for several settings are obtainable using the technique called **Gittins indices** (Gittins, 1989). A variety of exploration methods for sequential decision problems are discussed by Barto *et al.* (1995). Kearns and Singh (1998) and Brafman and Tennenholtz (2000) describe algorithms that explore unknown environments and are guaranteed to converge on near-optimal policies in polynomial time.

Function approximation in reinforcement learning goes back to the work of Samuel, who used both linear and nonlinear evaluation functions and also used feature selection methods to reduce the feature space. Later methods include the **CMAC** (Cerebellar Model Articulation Controller) (Albus, 1975), which is essentially a sum of overlapping local kernel functions, and the associative neural networks of Barto *et al.* (1983). Neural networks are currently the most popular form of function approximator. The best known application is TD-Gammon (Tesauro, 1992, 1995), which was discussed in the chapter. One significant problem exhibited by neural-network-based TD learners is that they tend to forget earlier experiences, especially those in parts of the state space that are avoided once competence is achieved. This can result in catastrophic failure if such circumstances reappear. Function approximation based on **instance-based learning** can avoid this problem (Ormoneit and Sen, 2002; Forbes, 2002).

CMAC

The convergence of reinforcement learning algorithms using function approximation is an extremely technical subject. Results for TD learning have been progressively strengthened for the case of linear function approximators (Sutton, 1988; Dayan, 1992; Tsitsiklis and Van Roy, 1997), but several examples of divergence have been presented for nonlinear functions (see Tsitsiklis and Van Roy, 1997, for a discussion). Papavassiliou and Russell (1999) describe a new type of reinforcement learning that converges with any form of function ap-

proximator, provided that a best-fit approximation can be found for the observed data.

Policy search methods were brought to the fore by Williams (1992), who developed the REINFORCE family of algorithms. Later work by Marbach and Tsitsiklis (1998), Sutton *et al.* (2000), and Baxter and Bartlett (2000) strengthened and generalized the convergence results for policy search. The PEGASUS algorithm is due to Ng and Jordan (2000) although similar techniques appear in Van Roy's PhD thesis (1998). As we mentioned in the chapter, the performance of a *stochastic* policy is a continuous function of its parameters, which helps with gradient-based search methods. This is not the only benefit: Jaakkola *et al.* (1995) argue that stochastic policies actually work better than deterministic policies in partially observable environments, if both are limited to acting based on the current percept. (One reason is that the stochastic policy is less likely to get "stuck" because of some unseen hindrance.) Now, in Chapter 17 we pointed out that optimal policies in partially observable MDPs are deterministic functions of the *belief state* rather than the current percept, so we would expect still better results by keeping track of the belief state using the **filtering** methods of Chapter 15. Unfortunately, belief state space is high-dimensional and continuous, and effective algorithms have not yet been developed for reinforcement learning with belief states.

Real-world environments also exhibit enormous complexity in terms of the number of primitive actions required to achieve significant reward. For example, a robot playing soccer might make a hundred thousand individual leg motions before scoring a goal. One common method, used originally in animal training, is called **reward shaping**. This involves supplying the agent with additional rewards for "making progress." For soccer, these might be given for making contact with the ball or for kicking it toward the goal. Such rewards can speed up learning enormously, and are very simple to provide, but there is a risk that the agent will learn to maximize the pseudorewards rather than the true rewards; for example, standing next to the ball and "vibrating" causes many contacts with the ball. Ng *et al.* (1999) show that the agent will still learn the optimal policy provided that the pseudoreward $F(s, a, s')$ satisfies $F(s, a, s') = \gamma\Phi(s') - \Phi(s)$, where Φ is an arbitrary function of state. Φ can be constructed to reflect any desirable aspects of the state, such as achievement of subgoals or distance to a goal state.

REWARD SHAPING

The generation of complex behaviors can also be facilitated by **hierarchical reinforcement learning** methods, which attempt to solve problems at multiple levels of abstraction—much like the **HTN planning** methods of Chapter 12. For example, "scoring a goal" can be broken down into "obtain possession," "dribble towards the goal," and "shoot;" and each of these can be broken down further into lower-level motor behaviors. The fundamental result in this area is due to Forestier and Varaiya (1978), who proved that lower-level behaviors of arbitrary complexity can be treated just like primitive actions (albeit ones that can take varying amounts of time) from the point of view of the higher-level behavior that invokes them. Current approaches (Parr and Russell, 1998; Dietterich, 2000; Sutton *et al.*, 2000; Andre and Russell, 2002) build on this result to develop methods for supplying an agent with a **partial program** that constrains the agent's behavior to have a particular hierarchical structure. Reinforcement learning is then applied to learn the best behavior consistent with the partial program. The combination of function approximation, shaping, and hierarchical reinforcement learning may enable large-scale problems to be tackled successfully.

HIERARCHICAL REINFORCEMENT LEARNING

PARTIAL PROGRAM

The survey by Kaelbling *et al.* (1996) provides a good entry point to the literature. The text by Sutton and Barto (1998), two of the field's pioneers, focuses on architectures and algorithms, showing how reinforcement learning weaves together the ideas of learning, planning, and acting. The somewhat more technical work by Bertsekas and Tsitsiklis (1996) gives a rigorous grounding in the theory of dynamic programming and stochastic convergence. Reinforcement learning papers are published frequently in *Machine Learning*, in the *Journal of Machine Learning Research*, and in the International Conferences on Machine Learning and the Neural Information Processing Systems meetings.

EXERCISES

21.1 Implement a passive learning agent in a simple environment, such as the 4×3 world. For the case of an initially unknown environment model, compare the learning performance of the direct utility estimation, TD, and ADP algorithms. Do the comparison for the optimal policy and for several random policies. For which do the utility estimates converge faster? What happens when the size of the environment is increased? (Try environments with and without obstacles.)

21.2 Chapter 17 defined a **proper policy** for an MDP as one that is guaranteed to reach a terminal state. Show that it is possible for a passive ADP agent to learn a transition model for which its policy π is improper even if π is proper for the true MDP; with such models, the value determination step may fail if $\gamma = 1$. Show that this problem cannot arise if value determination is applied to the learned model only at the end of a trial.

21.3 Starting with the passive ADP agent, modify it to use an approximate ADP algorithm as discussed in the text. Do this in two steps:

a. Implement a priority queue for adjustments to the utility estimates. Whenever a state is adjusted, all of its predecessors also become candidates for adjustment and should be added to the queue. The queue is initialized with the state from which the most recent transition took place. Allow only a fixed number of adjustments.

b. Experiment with various heuristics for ordering the priority queue, examining their effect on learning rates and computation time.

21.4 The direct utility estimation method in Section 21.2 uses distinguished terminal states to indicate the end of a trial. How could it be modified for environments with discounted rewards and no terminal states?

21.5 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 M, compared with an agent using correct values?

21.6 Adapt the vacuum world (Chapter 2) for reinforcement learning by including rewards for picking up each piece of dirt and for getting home and switching off. Make the world accessible by providing suitable percepts. Now experiment with different reinforcement learn-

ing agents. Is function approximation necessary for success? What sort of approximator works for this application?

21.7 Implement an exploring reinforcement learning agent that uses direct utility estimation. Make two versions—one with a tabular representation and one using the function approximator in Equation (21.9). Compare their performance in three environments:

a. The 4×3 world described in the chapter.

b. A 10×10 world with no obstacles and a +1 reward at (10,10).

c. A 10×10 world with no obstacles and a +1 reward at (5,5).

21.8 Write out the parameter update equations for TD learning with

$$\hat{U}(x, y) = \theta_0 + \theta_1 x + \theta_2 y + \theta_3 \sqrt{(x - x_g)^2 + (y - y_g)^2} \, .$$

21.9 Devise suitable features for stochastic grid worlds (generalizations of the 4×3 world) that contain multiple obstacles and multiple terminal states with $+1$ or -1 rewards.

21.10 Compute the true utility function and the best linear approximation in x and y (as in Equation (21.9)) for the following environments:

a. A 10×10 world with a single $+1$ terminal state at (10,10).

b. As in (a), but add a -1 terminal state at (10,1).

c. As in (b), but add obstacles in 10 randomly selected squares.

d. As in (b), but place a wall stretching from (5,2) to (5,9).

e. As in (a), but with the terminal state at (5,5).

The actions are deterministic moves in the four directions. In each case, compare the results using three-dimensional plots. For each environment, propose additional features (besides x and y) that would improve the approximation and show the results.

21.11 Extend the standard game-playing environment (Chapter 6) to incorporate a reward signal. Put two reinforcement learning agents into the environment (they may, of course, share the agent program) and have them play against each other. Apply the generalized TD update rule (Equation (21.11)) to update the evaluation function. You might wish to start with a simple linear weighted evaluation function and a simple game, such as tic-tac-toe.

21.12 Implement the REINFORCE and PEGASUS algorithms and apply them to the 4×3 world, using a policy family of your own choosing. Comment on the results.

21.13 Investigate the application of reinforcement learning ideas to the modeling of human and animal behavior.

21.14 Is reinforcement learning an appropriate abstract model for evolution? What connection exists, if any, between hardwired reward signals and evolutionary fitness?