7 LOGICAL AGENTS

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

This chapter introduces knowledge-based agents. The concepts that we discuss—the *representation* of knowledge and the *reasoning* processes that bring knowledge to life—are central to the entire field of artificial intelligence.

Humans, it seems, know things and do reasoning. Knowledge and reasoning are also important for artificial agents because they enable successful behaviors that would be very hard to achieve otherwise. We have seen that knowledge of action outcomes enables problem-solving agents to perform well in complex environments. A reflex agents could only find its way from Arad to Bucharest by dumb luck. The knowledge of problem-solving agents is, however, very specific and inflexible. A chess program can calculate the legal moves of its king, but does not know in any useful sense that no piece can be on two different squares at the same time. Knowledge-based agents can benefit from knowledge expressed in very general forms, combining and recombining information to suit myriad purposes. Often, this process can be quite far removed from the needs of the moment—as when a mathematician proves a theorem or an astronomer calculates the earth's life expectancy.

Knowledge and reasoning also play a crucial role in dealing with *partially observable* environments. A knowledge-based agent can combine general knowledge with current percepts to infer hidden aspects of the current state prior to selecting actions. For example, a physician diagnoses a patient—that is, infers a disease state that is not directly observable—prior to choosing a treatment. Some of the knowledge that the physician uses in the form of rules learned from textbooks and teachers, and some is in the form of patterns of association that the physician may not be able to consciously describe. If its inside the physician's head, it counts as knowledge.

Understanding natural language also requires inferring hidden state, namely, the intention of the speaker. When we hear, "John saw the diamond through the window and coveted it," we know "it" refers to the diamond and not the window—we reason, perhaps unconsciously, with our knowledge of relative value. Similarly, when we hear, "John threw the brick through the window and broke it," we know "it" refers to the window. Reasoning allows us to cope with the virtually infinite variety of utterances using a finite store of commonsense knowledge. Problem-solving agents have difficulty with this kind of ambiguity because their representation of contingency problems is inherently exponential.

Our final reason for studying knowledge-based agents is their flexibility. They are able to accept new tasks in the form of explicitly described goals, they can achieve competence quickly by being told or learning new knowledge about the environment, and they can adapt to changes in the environment by updating the relevant knowledge.

We begin in Section 7.1 with the overall agent design. Section 7.2 introduces a simple new environment, the wumpus world, and illustrates the operation of a knowledge-based agent without going into any technical detail. Then, in Section 7.3, we explain the general principles of **logic**. Logic will be the primary vehicle for representing knowledge throughout Part III of the book. The knowledge of logical agents is always *definite*—each proposition is either true or false in the world, although the agent may be agnostic about some propositions.

Logic has the pedagogical advantage of being simple example of a representation for knowledge-based agents, but logic has some severe limitations. Clearly, a large portion of the reasoning carried out by humans and other agents in partially observable environments depends on handling knowledge that is *uncertain*. Logic cannot represent this uncertainty well, so in Part V we cover probability, which can. In Part VI and Part VII we cover many representations, including some based on continuous mathematics such as mixtures of Gaussians, neural networks, and other representations.

Section 7.4 of this chapter defines a simple logic called **propositional logic**. While much less expressive than **first-order logic** (Chapter 8), propositional logic serves to illustrate all the basic concepts of logic. There is also a well-developed technology for reasoning in propositional logic, which we describe in sections 7.5 and 7.6. Finally, Section 7.7 combines the concept of logical agents with the technology of propositional logic to build some simple agents for the wumpus world. Certain shortcomings in propositional logic are identified, motivating the development of more powerful logics in subsequent chapters.

7.1 KNOWLEDGE-BASED AGENTS

KNOWLEDGE BASE
SENTENCE
KNOWLEDGE REPRESENTATION LANGUAGE

The central component of a knowledge-based agent is its **knowledge base**, or KB. Informally, a knowledge base is a set of **sentences**. (Here "sentence" is used as a technical term. It is related but is not identical to the sentences of English and other natural languages.) Each sentence is expressed in a language called a **knowledge representation language** and represents some assertion about the world.

INFERENCE
LOGICAL AGENTS

There must be a way to add new sentences to the knowledge base and a way to query what is known. The standard names for these tasks are TELL and ASK, respectively. Both tasks may involve **inference**—that is, deriving new sentences from old. In **logical agents**, which are the main subject of study in this chapter, inference must obey the fundamental requirement that when one ASKs a question of the knowledge base, the answer should follow from what has been told (or rather, TELLed) to the knowledge base previously. Later in the

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

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

Figure 7.1 A generic knowledge-based agent.

chapter, we will be more precise about the crucial word "follow." For now, take it to mean that the inference process should not just make things up as it goes along.

Figure 7.1 shows the outline of a knowledge-based agent program. Like all our agents, it takes a percept as input and returns an action. The agent maintains a knowledge base, KB, which may initially contain some **background knowledge**. Each time the agent program is called, it does two things. First, it TELLs the knowledge base what it perceives. Second, it ASKs the knowledge base what action it should perform. In the process of answering this query, extensive reasoning may be done about the current state of the world, about the outcomes of possible action sequences, and so on. Once the action is chosen, the agent records its choice with TELL and executes the action. The second TELL is necessary to let the knowledge base know that the hypothetical *action* has actually been executed.

BACKGROUND KNOWLEDGE

The details of the representation language are hidden inside two functions that implement the interface between the sensors and actuators and the core representation and reasoning system. MAKE-PERCEPT-SENTENCE takes a percept and a time and returns a sentence asserting that the agent perceived the percept at the given time. MAKE-ACTION-QUERY takes a time as input and returns a sentence that asks what action should be performed at that time. The details of the inference mechanisms are hidden inside TELL and ASK. Later sections will reveal these details.

The agent in Figure 7.1 appears quite similar to the agents with internal state described in Chapter 2. Because of the definitions of TELL and ASK, however, the knowledge-based agent is not an arbitrary program for calculating actions. It is amenable to a description at the **knowledge level**, where we need specify only what the agent knows and what its goals are, in order to fix its behavior. For example, an automated taxi might have the goal of delivering a passenger to Marin County and might know that it is in San Francisco and that the Golden Gate Bridge is the only link between the two locations. Then we can expect it to cross the Golden Gate Bridge *because it knows that that will achieve its goal*. Notice that this analysis is independent of how the taxi works at the **implementation level**. It doesn't matter whether its geographical knowledge is implemented as linked lists or pixel maps, or whether it reasons by manipulating strings of symbols stored in registers or by propagating noisy signals in a network of neurons.

KNOWLEDGE LEVEL

IMPLEMENTATION LEVEL

As we mentioned in the introduction to the chapter, *one can build a knowledge-based agent simply by* TELL*ing it what it needs to know.* The agent's initial program, before it starts to receive percepts, is built by adding one by one the sentences that represent the designer's knowledge of the environment. Designing the representation language to make it easy to express this knowledge in the form of sentences simplifies the construction problem enormously. This is called the **declarative** approach to system building. In contrast, the **procedural** approach encodes desired behaviors directly as program code; minimizing the role of explicit representation and reasoning can result in a much more efficient system. We will see agents of both kinds in Section 7.7. In the 1970s and 1980s, advocates of the two approaches engaged in heated debates. We now understand that a successful agent must combine both declarative and procedural elements in its design.

DECLARATIVE

In addition to TELLing it what it needs to know, we can provide a knowledge-based agent with mechanisms that allow it to learn for itself. These mechanisms, which are discussed in Chapter 18, create general knowledge about the environment out of a series of percepts. This knowledge can be incorporated into the agent's knowledge base and used for decision making. In this way, the agent can be fully autonomous.

All these capabilities—representation, reasoning, and learning—rest on the centuries-long development of the theory and technology of logic. Before explaining that theory and technology, however, we will create a simple world with which to illustrate them.

7.2 THE WUMPUS WORLD

WUMPUS WORLD

The **wumpus world** is a cave consisting of rooms connected by passageways. Lurking somewhere in the cave is the wumpus, a beast that eats anyone who enters its room. The wumpus can be shot by an agent, but the agent has only one arrow. Some rooms contain bottomless pits that will trap anyone who wanders into these rooms (except for the wumpus, which is too big to fall in). The only mitigating feature of living in this environment is the possibility of finding a heap of gold. Although the wumpus world is rather tame by modern computer game standards, it makes an excellent testbed environment for intelligent agents. Michael Genesereth was the first to suggest this.

A sample wumpus world is shown in Figure 7.2. The precise definition of the task environment is given, as suggested in Chapter 2, by the PEAS description:

◇ **Performance measure**: +1000 for picking up the gold, –1000 for falling into a pit or being eaten by the wumpus, –1 for each action taken and –10 for using up the arrow.

◇ **Environment**: A 4×4 grid of rooms. The agent always starts in the square labeled [1,1], facing to the right. The locations of the gold and the wumpus are chosen randomly, with a uniform distribution, from the squares other than the start square. In addition, each square other than the start can be a pit, with probability 0.2.

◇ **Actuators**: The agent can move forward, turn left by 90°, or turn right by 90°. The agent dies a miserable death if it enters a square containing a pit or a live wumpus. (It is safe, albeit smelly, to enter a square with a dead wumpus.) Moving forward has no effect if there is a wall in front of the agent. The action *Grab* can be used to pick up an object that is in the same square as the agent. The action *Shoot* can be used to fire an arrow in a straight line in the direction the agent is facing. The arrow continues until it either hits (and hence kills) the wumpus or hits a wall. The agent only has one arrow, so only the first *Shoot* action has any effect.

◊ **Sensors**: The agent has five sensors, each of which gives a single bit of information:

- In the square containing the wumpus and in the directly (not diagonally) adjacent squares the agent will perceive a stench.
- In the squares directly adjacent to a pit, the agent will perceive a breeze.
- In the square where the gold is, the agent will perceive a glitter.
- When an agent walks into a wall, it will perceive a bump.
- When the wumpus is killed, it emits a woeful scream that can be perceived anywhere in the cave.

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

Exercise 7.1 asks you to define the wumpus environment along the various dimensions given in Chapter 2. The principal difficulty for the agent is its initial ignorance of the configuration of the environment; overcoming this ignorance seems to require logical reasoning. In most instances of the wumpus world, it is possible for the agent to retrieve the gold safely. Occasionally, the agent must choose between going home empty-handed and risking death to find the gold. About 21% of the environments are utterly unfair, because the gold is in a pit or surrounded by pits.

Let us watch a knowledge-based wumpus agent exploring the environment shown in Figure 7.2. The agent's initial knowledge base contains the rules of the environment, as listed

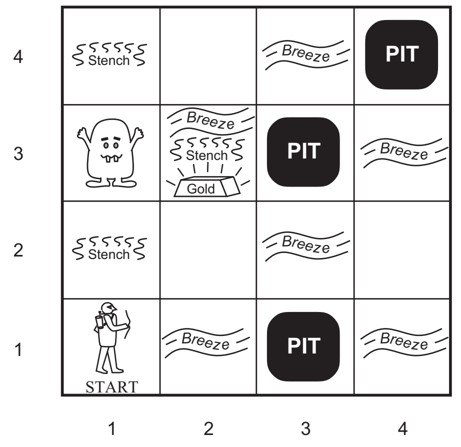


Figure 7.2 A typical wumpus world. The agent is in the bottom left corner.

previously; in particular, it knows that it is in [1,1] and that [1,1] is a safe square. We will see how its knowledge evolves as new percepts arrive and actions are taken.

The first percept is $[None, None, None, None, None]$, from which the agent can conclude that its neighboring squares are safe. Figure 7.3(a) shows the agent's state of knowledge at this point. We list (some of) the sentences in the knowledge base using letters such as B (breezy) and OK (safe, neither pit nor wumpus) marked in the appropriate squares. Figure 7.2, on the other hand, depicts the world itself.

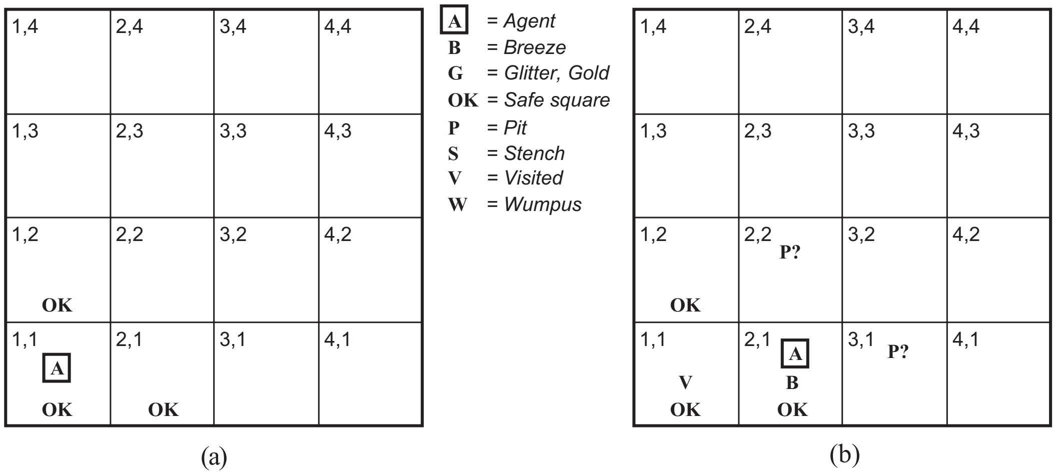


Figure 7.3 The first step taken by the agent in the wumpus world. (a) The initial situation, after percept $[None, None, None, None, None]$. (b) After one move, with percept $[None, Breeze, None, None, None]$.

From the fact that there was no stench or breeze in [1,1], the agent can infer that [1,2] and [2,1] are free of dangers. They are marked with an OK to indicate this. A cautious agent will move only into a square that it knows is OK. Let us suppose the agent decides to move forward to [2,1], giving the scene in Figure 7.3(b).

The agent detects a breeze in [2,1], so there must be a pit in a neighboring square. The pit cannot be in [1,1], by the rules of the game, so there must be a pit in [2,2] or [3,1] or both. The notation $P?$ in Figure 7.3(b) indicates a possible pit in those squares. At this point, there is only one known square that is OK and has not been visited yet. So the prudent agent will turn around, go back to [1,1], and then proceed to [1,2].

The new percept in [1,2] is $[Stench, None, None, None, None]$, resulting in the state of knowledge shown in Figure 7.4(a). The stench in [1,2] means that there must be a wumpus nearby. But the wumpus cannot be in [1,1], by the rules of the game, and it cannot be in [2,2] (or the agent would have detected a stench when it was in [2,1]). Therefore, the agent can infer that the wumpus is in [1,3]. The notation $W!$ indicates this. Moreover, the lack of a $Breeze$ in [1,2] implies that there is no pit in [2,2]. Yet we already inferred that there must be a pit in either [2,2] or [3,1], so this means it must be in [3,1]. This is a fairly difficult inference, because it combines knowledge gained at different times in different places and

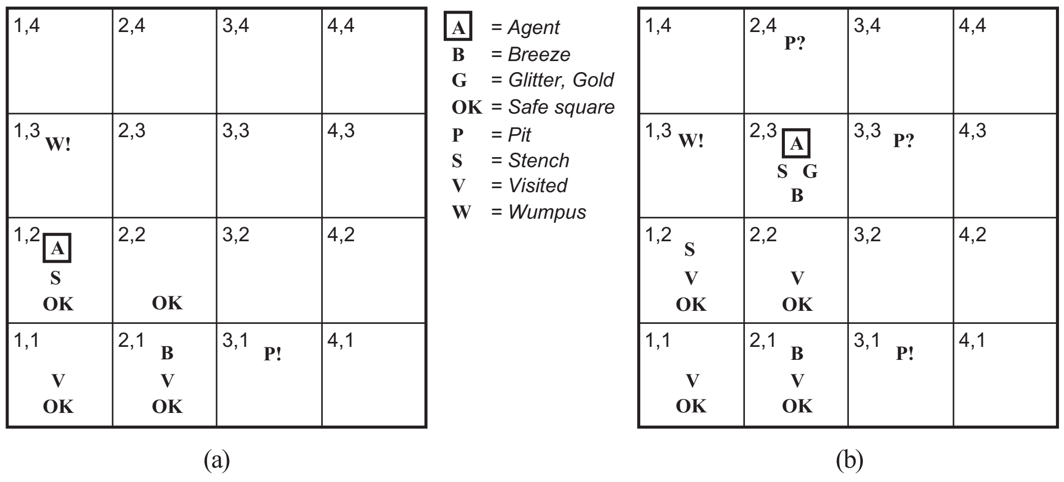


Figure 7.4 Two later stages in the progress of the agent. (a) After the third move, with percept $[Stench, None, None, None, None]$. (b) After the fifth move, with percept $[Stench, Breeze, Glitter, None, None]$.

relies on the lack of a percept to make one crucial step. The inference is beyond the abilities of most animals, but it is typical of the kind of reasoning that a logical agent does.

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

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

7.3 LOGIC

This section provides an overview of all the fundamental concepts of logical representation and reasoning. We postpone the technical details of any particular form of logic until the next section. We will instead use informal examples from the wumpus world and from the familiar realm of arithmetic. We adopt this rather unusual approach because the ideas of logic are far more general and beautiful than is commonly supposed.

SYNTAX

In Section 7.1, we said that knowledge bases consist of sentences. These sentences are expressed according to the **syntax** of the representation language, which specifies all the sentences that are well formed. The notion of syntax is clear enough in ordinary arithmetic: "$x + y = 4$" is a well-formed sentence, whereas "$x2y+ =$" is not. The syntax of logical languages (and of arithmetic, for that matter) is usually designed for writing papers and books. There are literally dozens of different syntaxes, some with lots of Greek letters and exotic mathematical symbols, and some with rather visually appealing diagrams with arrows and bubbles. In all cases, however, sentences in an agent's knowledge base are real physical configurations of (parts of) the agent. Reasoning will involve generating and manipulating those configurations.

SEMANTICS

A logic must also define the **semantics** of the language. Loosely speaking, semantics has to do with the "meaning" of sentences. In logic, the definition is more precise. The semantics of the language defines the **truth** of each sentence with respect to each **possible world**. For example, the usual semantics adopted for arithmetic specifies that the sentence "$x + y = 4$" is true in a world where x is 2 and y is 2, but false in a world where x is 1 and y is 1.[1] In standard logics, every sentence must be either true or false in each possible world—there is no "in between."[2]

TRUTH

POSSIBLE WORLD

MODEL

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

ENTAILMENT

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

$$\alpha \models \beta$$

to mean that the sentence α entails the sentence β. The formal definition of entailment is this: $\alpha \models \beta$ if and only if, in every model in which α is true, β is also true. Another way to say this is that if α is true, then β *must* also be true. Informally, the truth of β is "contained" in the truth of α. The relation of entailment is familiar from arithmetic; we are happy with the idea that the sentence $x + y = 4$ entails the sentence $4 = x + y$. Obviously, in any model where $x + y = 4$—such as the model in which x is 2 and y is 2—it is the case that $4 = x + y$. We will see shortly that a knowledge base can be considered a statement, and we often talk of a knowledge base entailing a sentence.

We can apply the same kind of analysis to the wumpus-world reasoning example given in the preceding section. Consider the situation in Figure 7.3(b): the agent has detected nothing in [1,1] and a breeze in [2,1]. These percepts, combined with the agent's knowledge of the rules of the wumpus world (the PEAS description on page 197), constitute the KB. The

[1] The reader will no doubt have noticed the similarity between the notion of truth of sentences and the notion of satisfaction of constraints in Chapter 5. This is no accident—constraint languages are indeed logics and constraint solving is a form of logical reasoning.

[2] **Fuzzy logic**, discussed in Chapter 14, allows for degrees of truth.

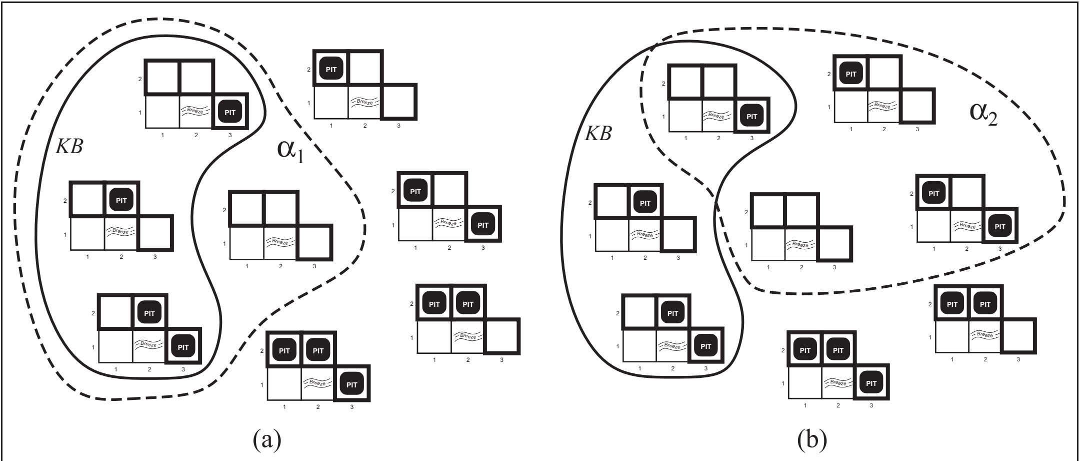


Figure 7.5 Possible models for the presence of pits in squares [1,2], [2,2], and [3,1], given observations of nothing in [1,1] and a breeze in [2,1]. (a) Models of the knowledge base and α_1 (no pit in [1,2]). (b) Models of the knowledge base and α_2 (no pit in [2,2]).

agent is interested (among other things) in whether the adjacent squares [1,2], [2,2], and [3,1] contain pits. Each of the three squares might or might not contain a pit, so (for the purposes of this example) there are $2^3 = 8$ possible models. These are shown in Figure 7.5.[3]

The KB is false in models that contradict what the agent knows—for example, the KB is false in any model in which [1,2] contains a pit, because there is no breeze in [1,1]. There are in fact just three models in which the KB is true, and these are shown as a subset of the models in Figure 7.5. Now let us consider two possible conclusions:

α_1 = "There is no pit in [1,2]."
α_2 = "There is no pit in [2,2]."

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

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

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

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

Hence, $KB \not\models \alpha_2$: the agent *cannot* conclude that there is no pit in [2,2]. (Nor can it conclude that there *is* a pit in [2,2].)[4]

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

LOGICAL INFERENCE
MODEL CHECKING

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

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

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

$$KB \vdash_i \alpha \; ,$$

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

SOUND
TRUTH-PRESERVING

An inference algorithm that derives only entailed sentences is called **sound** or **truth-preserving**. Soundness is a highly desirable property. An unsound inference procedure essentially makes things up as it goes along—it announces the discovery of nonexistent needles. It is easy to see that model checking, when it is applicable,[5] is a sound procedure.

COMPLETENESS

The property of **completeness** is also desirable: an inference algorithm is complete if it can derive any sentence that is entailed. For real haystacks, which are finite in extent, it seems obvious that a systematic examination can always decide whether the needle is in the haystack. For many knowledge bases, however, the haystack of consequences is infinite, and completeness becomes an important issue.[6] Fortunately, there are complete inference procedures for logics that are sufficiently expressive to handle many knowledge bases.

We have described a reasoning process whose conclusions are guaranteed to be true in any world in which the premises are true; in particular, *if KB is true in the* real *world, then any sentence α derived from KB by a sound inference procedure is also true in the real world.* So, while an inference process operates on "syntax"—internal physical configurations such as bits in registers or patterns of electrical blips in brains—the process *corresponds* to the real-world relationship whereby some aspect of the real world is the case[7] by virtue of other aspects of the real world being the case. This correspondence between world and representation is illustrated in Figure 7.6.

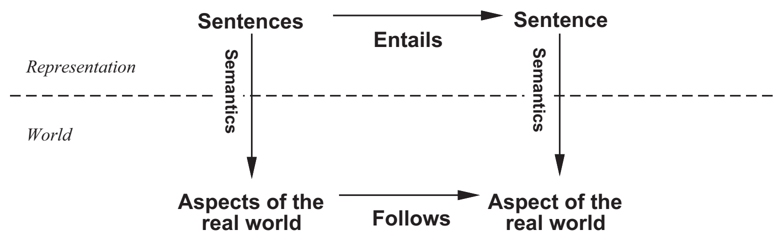


Figure 7.6 Sentences are physical configurations of the agent, and reasoning is a process of constructing new physical configurations from old ones. Logical reasoning should ensure that the new configurations represent aspects of the world that actually follow from the aspects that the old configurations represent.

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

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

[7] As Wittgenstein (1922) put it in his famous *Tractatus*: "The world is everything that is the case."

GROUNDING

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

7.4 Propositional Logic: A Very Simple Logic

PROPOSITIONAL LOGIC

We now present a very simple logic called **propositional logic**.[8] We cover the syntax of propositional logic and its semantics—the way in which the truth of sentences is determined. Then we look at **entailment**—the relation between a sentence and another sentence that follows from it—and see how this leads to a simple algorithm for logical inference. Everything takes place, of course, in the wumpus world.

Syntax

ATOMIC SENTENCES
PROPOSITION SYMBOL

The **syntax** of propositional logic defines the allowable sentences. The **atomic sentences**—the indivisible syntactic elements—consist of a single **proposition symbol**. Each such symbol stands for a proposition that can be true or false. We will use uppercase names for symbols: P, Q, R, and so on. The names are arbitrary but are often chosen to have some mnemonic value to the reader. For example, we might use $W_{1,3}$ to stand for the proposition that the wumpus is in [1,3]. (Remember that symbols such as $W_{1,3}$ are *atomic*, i.e., W, 1, and 3 are not meaningful parts of the symbol.) There are two proposition symbols with fixed meanings: $True$ is the always-true proposition and $False$ is the always-false proposition.

COMPLEX SENTENCES
LOGICAL CONNECTIVES

Complex sentences are constructed from simpler sentences using **logical connectives**. There are five connectives in common use:

NEGATION
LITERAL

$\neg$ (not). A sentence such as $\neg W_{1,3}$ is called the **negation** of $W_{1,3}$. A **literal** is either an atomic sentence (a **positive literal**) or a negated atomic sentence (a **negative literal**).

[8] Propositional logic is also called **Boolean logic**, after the logician George Boole (1815–1864).

CONJUNCTION

$\land$ (and). A sentence whose main connective is $\land$, such as $W_{1,3} \land P_{3,1}$, is called a **conjunction**; its parts are the **conjuncts**. (The $\land$ looks like an "A" for "And.")

DISJUNCTION

$\lor$ (or). A sentence using $\lor$, such as $(W_{1,3} \land P_{3,1}) \lor W_{2,2}$, is a **disjunction** of the **disjuncts** $(W_{1,3} \land P_{3,1})$ and $W_{2,2}$. (Historically, the $\lor$ comes from the Latin "vel," which means "or." For most people, it is easier to remember as an upside-down $\land$.)

IMPLICATION
PREMISE
CONCLUSION

$\Rightarrow$ (implies). A sentence such as $(W_{1,3} \land P_{3,1}) \Rightarrow \neg W_{2,2}$ is called an **implication** (or conditional). Its **premise** or **antecedent** is $(W_{1,3} \land P_{3,1})$, and its **conclusion** or **consequent** is $\neg W_{2,2}$. Implications are also known as **rules** or **if–then** statements. The implication symbol is sometimes written in other books as $\supset$ or $\rightarrow$.

BICONDITIONAL

$\Leftrightarrow$ (if and only if). The sentence $W_{1,3} \Leftrightarrow \neg W_{2,2}$ is a **biconditional**.

Figure 7.7 gives a formal grammar of propositional logic; see page 984 if you are not familiar with the BNF notation.

$$
\begin{array}{rcl}
\mathit{Sentence} & \rightarrow & \mathit{AtomicSentence} \mid \mathit{ComplexSentence} \\
\mathit{AtomicSentence} & \rightarrow & \textbf{True} \mid \textbf{False} \mid \mathit{Symbol} \\
\mathit{Symbol} & \rightarrow & \textbf{P} \mid \textbf{Q} \mid \textbf{R} \mid \ldots \\
\mathit{ComplexSentence} & \rightarrow & \neg\ \mathit{Sentence} \\
& \mid & \textbf{(}\ \mathit{Sentence} \land \mathit{Sentence}\ \textbf{)} \\
& \mid & \textbf{(}\ \mathit{Sentence} \lor \mathit{Sentence}\ \textbf{)} \\
& \mid & \textbf{(}\ \mathit{Sentence} \Rightarrow \mathit{Sentence}\ \textbf{)} \\
& \mid & \textbf{(}\ \mathit{Sentence} \Leftrightarrow \mathit{Sentence}\ \textbf{)}
\end{array}
$$

Figure 7.7 A BNF (Backus–Naur Form) grammar of sentences in propositional logic.

Notice that the grammar is very strict about parentheses: every sentence constructed with binary connectives must be enclosed in parentheses. This ensures that the syntax is completely unambiguous. It also means that we have to write $((A \land B) \Rightarrow C)$ instead of $A \land B \Rightarrow C$, for example. To improve readability, we will often omit parentheses, relying instead on an order of precedence for the connectives. This is similar to the precedence used in arithmetic—for example, $ab + c$ is read as $((ab) + c)$ rather than $a(b + c)$ because multiplication has higher precedence than addition. The order of precedence in propositional logic is (from highest to lowest): $\neg$, $\land$, $\lor$, $\Rightarrow$, and $\Leftrightarrow$. Hence, the sentence

$$\neg P \lor Q \land R \Rightarrow S$$

is equivalent to the sentence

$$((\neg P) \lor (Q \land R)) \Rightarrow S\,.$$

Precedence does not resolve ambiguity in sentences such as $A \land B \land C$, which could be read as $((A \land B) \land C)$ or as $(A \land (B \land C))$. Because these two readings mean the same thing according to the semantics given in the next section, sentences such as $A \land B \land C$ are allowed. We also allow $A \lor B \lor C$ and $A \Leftrightarrow B \Leftrightarrow C$. Sentences such as $A \Rightarrow B \Rightarrow C$ are not allowed because the two readings have different meanings; we insist on parentheses in this case. Finally, we will sometimes use square brackets instead of parentheses when it makes the sentence clearer.

Semantics

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

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

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

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

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

For complex sentences, we have rules such as

- For any sentence s and any model m, the sentence $\neg s$ is true in m if and only if s is false in m.

TRUTH TABLE

Such rules reduce the truth of a complex sentence to the truth of simpler sentences. The rules for each connective can be summarized in a **truth table** that specifies the truth value of a complex sentence for each possible assignment of truth values to its components. Truth tables for the five logical connectives are given in Figure 7.8. Using these tables, the truth value of any sentence s can be computed with respect to any model m by a simple process of recursive evaluation. For example, the sentence $\neg P_{1,2} \land (P_{2,2} \lor P_{3,1})$, evaluated in m_1, gives $true \land (false \lor true) = true \land true = true$. Exercise 7.3 asks you to write the algorithm PL-TRUE?(s, m), which computes the truth value of a propositional logic sentence s in a model m.

Previously we said that a knowledge base consists of a set of sentences. We can now see that a logical knowledge base is a conjunction of those sentences. That is, if we start with an empty KB and do $\text{TELL}(KB, S_1) \ldots \text{TELL}(KB, S_n)$ then we have $KB = S_1 \land \ldots \land S_n$. This means that we can treat knowledge bases and sentences interchangeably.

The truth tables for "and," "or," and "not" are in close accord with our intuitions about the English words. The main point of possible confusion is that $P \lor Q$ is true when P is true

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

Figure 7.8 Truth tables for the five logical connectives. To use the table to compute, for example, the value of $P \vee Q$ when P is true and Q is false, first look on the left for the row where P is $true$ and Q is $false$ (the third row). Then look in that row under the $P \vee Q$ column to see the result: $true$. Another way to look at this is to think of each row as a model, and the entries under each column for that row as saying whether the corresponding sentence is true in that model.

or Q is true *or both*. There is a different connective called "exclusive or" ("xor" for short) that yields false when both disjuncts are true.[9] There is no consensus on the symbol for exclusive or; two choices are $\dot{\vee}$ and $\oplus$.

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

The truth table for a biconditional, $P \Leftrightarrow Q$, shows that it is true whenever both $P \Rightarrow Q$ and $Q \Rightarrow P$ are true. In English, this is often written as "P if and only if Q" or "P iff Q." The rules of the wumpus world are best written using $\Leftrightarrow$. For example, a square is breezy *if* a neighboring square has a pit, and a square is breezy *only if* a neighboring square has a pit. So we need biconditionals such as

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

where $B_{1,1}$ means that there is a breeze in [1,1]. Notice that the one-way implication

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

is true in the wumpus world, but incomplete. It does not rule out models in which $B_{1,1}$ is false and $P_{1,2}$ is true, which would violate the rules of the wumpus world. Another way of putting it is that the implication requires the presence of pits if there is a breeze, whereas the biconditional also requires the absence of pits if there is no breeze.

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

A simple knowledge base

Now that we have defined the semantics for propositional logic, we can construct a knowledge base for the wumpus world. For simplicity, we will deal only with pits; the wumpus itself is left as an exercise. We will provide enough knowledge to carry out the inference that was done informally in Section 7.3.

First, we need to choose our vocabulary of proposition symbols. For each i, j:

- Let $P_{i,j}$ be true if there is a pit in $[i, j]$.
- Let $B_{i,j}$ be true if there is a breeze in $[i, j]$.

The knowledge base includes the following sentences, each one labeled for convenience:

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

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

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

$$R_2 : \quad B_{1,1} \quad \Leftrightarrow \quad (P_{1,2} \lor P_{2,1}) \ .$$
$$R_3 : \quad B_{2,1} \quad \Leftrightarrow \quad (P_{1,1} \lor P_{2,2} \lor P_{3,1}) \ .$$

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

$$R_4 : \quad \neg B_{1,1} \ .$$
$$R_5 : \quad B_{2,1} \ .$$

The knowledge base, then, consists of sentences R_1 through R_5. It can also be considered as a single sentence—the conjunction $R_1 \land R_2 \land R_3 \land R_4 \land R_5$—because it asserts that all the individual sentences are true.

Inference

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

Figure 7.9 reproduces in a more precise form the reasoning illustrated in Figure 7.5. A general algorithm for deciding entailment in propositional logic is shown in Figure 7.10. Like the BACKTRACKING-SEARCH algorithm on page 76, TT-ENTAILS? performs a recursive enumeration of a finite space of assignments to variables. The algorithm is **sound**, because it

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

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

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

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

function TT-CHECK-ALL($KB, \alpha, symbols, model$) returns $true$ or $false$
  if EMPTY?($symbols$) then
      if PL-TRUE?($KB, model$) then return PL-TRUE?($\alpha, model$)
      else return $true$
  else do
      $P \leftarrow$ FIRST($symbols$); $rest \leftarrow$ REST($symbols$)
      return TT-CHECK-ALL($KB, \alpha, rest$, EXTEND($P, true, model$)) and
                 TT-CHECK-ALL($KB, \alpha, rest$, EXTEND($P, false, model$))
```

Figure 7.10 A truth-table enumeration algorithm for deciding propositional entailment. TT stands for truth table. PL-TRUE? returns true if a sentence holds within a model. The variable *model* represents a partial model—an assignment to only some of the variables. The function call EXTEND(*P, true, model*) returns a new partial model in which P has the value *true*.

implements directly the definition of entailment, and **complete**, because it works for any KB and α and always terminates—there are only finitely many models to examine.

Of course, "finitely many" is not always the same as "few." If KB and α contain n symbols in all, then there are 2^n models. Thus, the time complexity of the algorithm is $O(2^n)$. (The space complexity is only $O(n)$ because the enumeration is depth-first.) Later in this

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

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

chapter, we will see algorithms that are much more efficient in practice. Unfortunately, *every known inference algorithm for propositional logic has a worst-case complexity that is exponential in the size of the input.* We do not expect to do better than this because propositional entailment is co-NP-complete. (See Appendix A.)

Equivalence, validity, and satisfiability

Before we plunge into the details of logical inference algorithms, we will need some additional concepts related to entailment. Like entailment, these concepts apply to all forms of logic, but they are best illustrated for a particular logic, such as propositional logic.

LOGICAL EQUIVALENCE

The first concept is **logical equivalence**: two sentences α and β are logically equivalent if they are true in the same set of models. We write this as $\alpha \Leftrightarrow \beta$. For example, we can easily show (using truth tables) that $P \wedge Q$ and $Q \wedge P$ are logically equivalent; other equivalences are shown in Figure 7.11. They play much the same role in logic as arithmetic identities do in ordinary mathematics. An alternative definition of equivalence is as follows: for any two sentences α and β,

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

(Recall that $\models$ means entailment.)

VALIDITY

The second concept we will need is **validity**. A sentence is valid if it is true in *all* models. For example, the sentence $P \vee \neg P$ is valid. Valid sentences are also known as

TAUTOLOGY

tautologies—they are *necessarily* true and hence vacuous. Because the sentence $True$ is true in all models, every valid sentence is logically equivalent to $True$.

DEDUCTION THEOREM

What good are valid sentences? From our definition of entailment, we can derive the **deduction theorem**, which was known to the ancient Greeks:

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

(Exercise 7.4 asks for a proof.) We can think of the inference algorithm in Figure 7.10 as checking the validity of $(KB \Rightarrow \alpha)$. Conversely, every valid implication sentence describes a legitimate inference.

SATISFIABILITY

The final concept we will need is **satisfiability**. A sentence is satisfiable if it is true in *some* model. For example, the knowledge base given earlier, $(R_1 \wedge R_2 \wedge R_3 \wedge R_4 \wedge R_5)$, is satisfiable because there are three models in which it is true, as shown in Figure 7.9. If a sentence α is true in a model m, then we say that m **satisfies** α, or that m **is a model of** α. Satisfiability can be checked by enumerating the possible models until one is found that satisfies the sentence. Determining the satisfiability of sentences in propositional logic was the first problem proved to be NP-complete.

SATISFIES

Many problems in computer science are really satisfiability problems. For example, all the constraint satisfaction problems in Chapter 5 are essentially asking whether the constraints are satisfiable by some assignment. With appropriate transformations, search problems can also be solved by checking satisfiability. Validity and satisfiability are of course connected: α is valid iff $\neg\alpha$ is unsatisfiable; contrapositively, α is satisfiable iff $\neg\alpha$ is not valid. We also have the following useful result:

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

REDUCTIO AD ABSURDUM

REFUTATION

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

7.5 Reasoning Patterns in Propositional Logic

INFERENCE RULES

MODUS PONENS

This section covers standard patterns of inference that can be applied to derive chains of conclusions that lead to the desired goal. These patterns of inference are called **inference rules**. The best-known rule is called **Modus Ponens** and is written as follows:

$$\frac{\alpha \;\Rightarrow\; \beta, \qquad \alpha}{\beta}\,.$$

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

AND-ELIMINATION

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

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

For example, from $(\mathit{WumpusAhead} \wedge \mathit{WumpusAlive})$, $\mathit{WumpusAlive}$ can be inferred.

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

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

$$\frac{\alpha \;\Leftrightarrow\; \beta}{(\alpha \;\Rightarrow\; \beta) \land (\beta \;\Rightarrow\; \alpha)} \qquad \text{and} \qquad \frac{(\alpha \;\Rightarrow\; \beta) \land (\beta \;\Rightarrow\; \alpha)}{\alpha \;\Leftrightarrow\; \beta} \; .$$

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

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

$$R_6: \quad (B_{1,1} \;\Rightarrow\; (P_{1,2} \lor P_{2,1})) \;\land\; ((P_{1,2} \lor P_{2,1}) \;\Rightarrow\; B_{1,1}) \; .$$

Then we apply And-Elimination to R_6 to obtain

$$R_7: \quad ((P_{1,2} \lor P_{2,1}) \;\Rightarrow\; B_{1,1}) \; .$$

Logical equivalence for contrapositives gives

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

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

$$R_9: \quad \neg(P_{1,2} \lor P_{2,1}) \; .$$

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

$$R_{10}: \quad \neg P_{1,2} \land \neg P_{2,1} \; .$$

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

PROOF

The preceding derivation—a sequence of applications of inference rules—is called a **proof**. Finding proofs is exactly like finding solutions to search problems. In fact, if the successor function is defined to generate all possible applications of inference rules, then all of the search algorithms in Chapters 3 and 4 can be applied to find proofs. Thus, searching for proofs is an alternative to enumerating models. The search can go forward from the initial knowledge base, applying inference rules to derive the goal sentence, or it can go backward from the goal sentence, trying to find a chain of inference rules leading from the initial knowledge base. Later in this section, we will see two families of algorithms that use these techniques.

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

MONOTONICITY

This property of logical systems actually follows from a much more fundamental property called **monotonicity**. Monotonicity says that the set of entailed sentences can only *in-*

crease as information is added to the knowledge base.[10] For any sentences α and β,

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

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

Resolution

We have argued that the inference rules covered so far are *sound*, but we have not discussed the question of *completeness* for the inference algorithms that use them. Search algorithms such as iterative deepening search (page 78) are complete in the sense that they will find any reachable goal, but if the available inference rules are inadequate, then the goal is not reachable—no proof exists that uses only those inference rules. For example, if we removed the biconditional elimination rule, the proof in the preceding section would not go through. The current section introduces a single inference rule, **resolution**, that yields a complete inference algorithm when coupled with any complete search algorithm.

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

$$R_{11}: \quad \neg B_{1,2}\ .$$
$$R_{12}: \quad B_{1,2} \Leftrightarrow (P_{1,1} \lor P_{2,2} \lor P_{1,3})\ .$$

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

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

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

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

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

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

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

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

[10] **Nonmonotonic** logics, which violate the monotonicity property, capture a common property of human reasoning: changing one's mind. They are discussed in Section 10.7.

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

UNIT RESOLUTION

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

COMPLEMENTARY LITERALS

CLAUSE

where each ℓ is a literal and ℓ_i and m are **complementary literals** (i.e., one is the negation of the other). Thus, the unit resolution rule takes a **clause**—a disjunction of literals—and a literal and produces a new clause. Note that a single literal can be viewed as a disjunction of one literal, also known as a **unit clause**.

UNIT CLAUSE

RESOLUTION

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

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

where ℓ_i and m_j are complementary literals. If we were dealing only with clauses of length two we could write this as

$$\frac{\ell_1 \vee \ell_2, \qquad \neg\ell_2 \vee \ell_3}{\ell_1 \vee \ell_3} .$$

That is, resolution takes two clauses and produces a new clause containing all the literals of the two original clauses *except* the two complementary literals. For example, we have

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

FACTORING

There is one more technical aspect of the resolution rule: the resulting clause should contain only one copy of each literal.[11] The removal of multiple copies of literals is called **factoring**. For example, if we resolve $(A \vee B)$ with $(A \vee \neg B)$, we obtain $(A \vee A)$, which is reduced to just A.

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

What is more surprising about the resolution rule is that it forms the basis for a family of *complete* inference procedures. *Any complete search algorithm, applying only the resolution rule, can derive any conclusion entailed by any knowledge base in propositional logic.* There is a caveat: resolution is complete in a specialized sense. Given that A is true, we cannot use resolution to automatically generate the consequence $A \vee B$. However, we can use resolution to answer the question of whether $A \vee B$ is true. This is called **refutation completeness**, meaning that resolution can always be used to either confirm or refute a sentence, but it cannot be used to enumerate true sentences. The next two subsections explain how resolution accomplishes this.

REFUTATION COMPLETENESS

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

Conjunctive normal form

The resolution rule applies only to disjunctions of literals, so it would seem to be relevant only to knowledge bases and queries consisting of such disjunctions. How, then, can it lead to a complete inference procedure for all of propositional logic? The answer is that *every sentence of propositional logic is logically equivalent to a conjunction of disjunctions of literals.* A sentence expressed as a conjunction of disjunctions of literals is said to be in **conjunctive normal form** or **CNF**. We will also find it useful later to consider the restricted family of k-**CNF** sentences. A sentence in k-CNF has exactly k literals per clause:

CONJUNCTIVE NORMAL FORM

K-CNF

$$(\ell_{1,1} \vee \ldots \vee \ell_{1,k}) \wedge \ldots \wedge (\ell_{n,1} \vee \ldots \vee \ell n, k)\ .$$

It turns out that every sentence can be transformed into a 3-CNF sentence that has an equivalent set of models.

Rather than prove these assertions (see Exercise 7.10), we describe a simple conversion procedure. We illustrate the procedure by converting R_2, the sentence $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$, into CNF. The steps are as follows:

1. Eliminate $\Leftrightarrow$, replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \;\Rightarrow\; \beta) \wedge (\beta \;\Rightarrow\; \alpha)$.
$$(B_{1,1} \;\Rightarrow\; (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \;\Rightarrow\; B_{1,1})\ .$$
2. Eliminate $\Rightarrow$, replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$:
$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1})\ .$$
3. CNF requires $\neg$ to appear only in literals, so we "move $\neg$ inwards" by repeated application of the following equivalences from Figure 7.11:
$$\begin{array}{ll} \neg(\neg\alpha) \equiv \alpha & \text{(double-negation elimination)} \\ \neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta) & \text{(de Morgan)} \\ \neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta) & \text{(de Morgan)} \end{array}$$
In the example, we require just one application of the last rule:
$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1})\ .$$
4. Now we have a sentence containing nested $\wedge$ and $\vee$ operators applied to literals. We apply the distributivity law from Figure 7.11, distributing $\vee$ over $\wedge$ wherever possible.
$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1})\ .$$

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

A resolution algorithm

Inference procedures based on resolution work by using the principle of proof by contradiction discussed at the end of Section 7.4. That is, to show that $KB \models \alpha$, we show that $(KB \wedge \neg\alpha)$ is unsatisfiable. We do this by proving a contradiction.

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

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

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

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

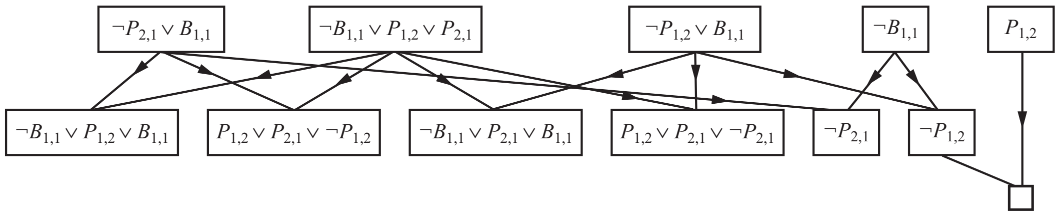

$\neg P_{2,1} \vee B_{1,1}$
$\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}$
$\neg P_{1,2} \vee B_{1,1}$
$\neg B_{1,1}$
$P_{1,2}$
$\neg B_{1,1} \vee P_{1,2} \vee B_{1,1}$
$P_{1,2} \vee P_{2,1} \vee \neg P_{1,2}$
$\neg B_{1,1} \vee P_{2,1} \vee B_{1,1}$
$P_{1,2} \vee P_{2,1} \vee \neg P_{2,1}$
$\neg P_{2,1}$
$\neg P_{1,2}$

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

- there are no new clauses that can be added, in which case α does not entail β or
- an application of the resolution rule derives the *empty* clause, in which case α entails β.

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

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

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

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

Completeness of resolution

RESOLUTION CLOSURE

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

GROUND RESOLUTION THEOREM

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

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

We prove this theorem by demonstrating its contrapositive: if the closure $RC(S)$ does *not* contain the empty clause, then S is satisfiable. In fact, we can construct a model for S with suitable truth values for $P_1, \ldots, P_k$. The construction procedure is as follows:

For i from 1 to k,

- If there is a clause in $RC(S)$ containing the literal $\neg P_i$ such that all its other literals are false under the assignment chosen for $P_1, \ldots, P_{i-1}$, then assign $false$ to P_i.
- Otherwise, assign $true$ to P_i.

It remains to show that this assignment to $P_1, \ldots, P_k$ is a model of S, provided that $RC(S)$ is closed under resolution and does not contain the empty clause. The proof of this is left as an exercise.

Forward and backward chaining

HORN CLAUSES

The completeness of resolution makes it a very important inference method. In many practical situations, however, the full power of resolution is not needed. Real-world knowledge bases often contain only clauses of a restricted kind called **Horn clauses**. A Horn clause is a disjunction of literals of which *at most one is positive*. For example, the clause $(\neg L_{1,1} \vee \neg Breeze \vee B_{1,1})$, where $L_{1,1}$ means that the agent's location is [1,1], is a Horn clause, whereas $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1})$ is not.

The restriction to just one positive literal may seem somewhat arbitrary and uninteresting, but it is actually very important for three reasons:

1. Every Horn clause can be written as an implication whose premise is a conjunction of positive literals and whose conclusion is a single positive literal. (See Exercise 7.12.) For example, the Horn clause $(\neg L_{1,1} \vee \neg Breeze \vee B_{1,1})$ can be written as the implication

$(L_{1,1} \wedge \mathit{Breeze}) \Rightarrow B_{1,1}$. In the latter form, the sentence is much easier to read: it says that if the agent is in [1,1] and there is a breeze, then [1,1] is breezy. People find it easy to read and write sentences in this form for many domains of knowledge.

DEFINITE CLAUSES
HEAD
BODY
FACT

Horn clauses like this one with *exactly* one positive literal are called **definite clauses**. The positive literal is called the **head** and the negative literals form the **body** of the clause. A definite clause with no negative literals simply asserts a given proposition—sometimes called a **fact**. Definite clauses form the basis for **logic programming**, which is discussed in Chapter 9. A Horn clause with *no* positive literals can be written as an implication whose conclusion is the literal False. For example, the clause $(\neg W_{1,1} \vee \neg W_{1,2})$—the wumpus cannot be in both [1,1] and [1,2]—is equivalent to $W_{1,1} \wedge W_{1,2} \Rightarrow \mathit{False}$. Such sentences are called **integrity constraints** in the database world, where they are used to signal errors in the data. In the algorithms that follow, we assume for simplicity that the knowledge base contains only definite clauses and no integrity constraints. We say these knowledge bases are in Horn form.

INTEGRITY CONSTRAINTS

FORWARD CHAINING
BACKWARD CHAINING

2. Inference with Horn clauses can be done through the **forward chaining** and **backward chaining** algorithms, which we explain next. Both of these algorithms are very natural, in that the inference steps are obvious and easy to follow for humans.
3. Deciding entailment with Horn clauses can be done in time that is *linear* in the size of the knowledge base.

This last fact is a pleasant surprise. It means that logical inference is very cheap for many propositional knowledge bases that are encountered in practice.

The forward-chaining algorithm PL-FC-ENTAILS?(KB, q) determines whether a single proposition symbol q—the query—is entailed by a knowledge base of Horn clauses. It begins from known facts (positive literals) in the knowledge base. If all the premises of an implication are known, then its conclusion is added to the set of known facts. For example, if $L_{1,1}$ and Breeze are known and $(L_{1,1} \wedge \mathit{Breeze}) \Rightarrow B_{1,1}$ is in the knowledge base, then $B_{1,1}$ can be added. This process continues until the query q is added or until no further inferences can be made. The detailed algorithm is shown in Figure 7.14; the main point to remember is that it runs in linear time.

AND–OR GRAPH

The best way to understand the algorithm is through an example and a picture. Figure 7.15(a) shows a simple knowledge base of Horn clauses with A and B as known facts. Figure 7.15(b) shows the same knowledge base drawn as an **AND–OR graph**. In AND–OR graphs, multiple links joined by an arc indicate a conjunction—every link must be proved—while multiple links without an arc indicate a disjunction—any link can be proved. It is easy to see how forward chaining works in the graph. The known leaves (here, A and B) are set, and inference propagates up the graph as far as possible. Wherever a conjunction appears, the propagation waits until all the conjuncts are known before proceeding. The reader is encouraged to work through the example in detail.

FIXED POINT

It is easy to see that forward chaining is **sound**: every inference is essentially an application of Modus Ponens. Forward chaining is also **complete**: every entailed atomic sentence will be derived. The easiest way to see this is to consider the final state of the $\mathit{inferred}$ table (after the algorithm reaches a **fixed point** where no new inferences are possible). The table

```
function PL-FC-ENTAILS?($KB$, $q$) returns $true$ or $false$
  inputs: $KB$, the knowledge base, a set of propositional Horn clauses
          $q$, the query, a proposition symbol
  local variables: $count$, a table, indexed by clause, initially the number of premises
                   $inferred$, a table, indexed by symbol, each entry initially $false$
                   $agenda$, a list of symbols, initially the symbols known to be true in $KB$

  while $agenda$ is not empty do
    $p \leftarrow$ POP($agenda$)
    unless $inferred[p]$ do
      $inferred[p] \leftarrow true$
      for each Horn clause $c$ in whose premise $p$ appears do
        decrement $count[c]$
        if $count[c] = 0$ then do
          if HEAD[$c$] = $q$ then return $true$
          PUSH(HEAD[$c$], $agenda$)
  return $false$
```

Figure 7.14 The forward-chaining algorithm for propositional logic. The $agenda$ keeps track of symbols known to be true but not yet "processed." The $count$ table keeps track of how many premises of each implication are as yet unknown. Whenever a new symbol p from the agenda is processed, the count is reduced by one for each implication in whose premise p appears. (These can be identified in constant time if KB is indexed appropriately.) If a count reaches zero, all the premises of the implication are known so its conclusion can be added to the agenda. Finally, we need to keep track of which symbols have been processed; an inferred symbol need not be added to the agenda if it has been processed previously. This avoids redundant work; it also prevents infinite loops that could be caused by implications such as $P \Rightarrow Q$ and $Q \Rightarrow P$.

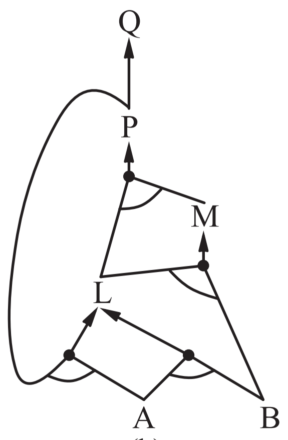
contains $true$ for each symbol inferred during the process, and $false$ for all other symbols. We can view the table as a logical model; moreover, *every definite clause in the original KB is true in this model.* To see this, assume the opposite, namely that some clause $a_1 \wedge \ldots \wedge a_k \Rightarrow b$ is false in the model. Then $a_1 \wedge \ldots \wedge a_k$ must be true in the model and b must be false in the model. But this contradicts our assumption that the algorithm has reached a fixed point! We can conclude, therefore, that the set of atomic sentences inferred at the fixed point defines a model of the original KB. Furthermore, any atomic sentence q that is entailed by the KB must be true in all its models and in this model in particular. Hence, every entailed sentence q must be inferred by the algorithm.

DATA-DRIVEN

Forward chaining is an example of the general concept of **data-driven** reasoning—that is, reasoning in which the focus of attention starts with the known data. It can be used within an agent to derive conclusions from incoming percepts, often without a specific query in mind. For example, the wumpus agent might TELL its percepts to the knowledge base using an incremental forward-chaining algorithm in which new facts can be added to the agenda to initiate new inferences. In humans, a certain amount of data-driven reasoning occurs as new

$P \Rightarrow Q$
$L \wedge M \Rightarrow P$
$B \wedge L \Rightarrow M$
$A \wedge P \Rightarrow L$
$A \wedge B \Rightarrow L$
A
B

(a)



(b)

Figure 7.15 (a) A simple knowledge base of Horn clauses. (b) The corresponding AND–OR graph.

information arrives. For example, if I am indoors and hear rain starting to fall, it might occur to me that the picnic will be canceled. Yet it will probably not occur to me that the seventeenth petal on the largest rose in my neighbor's garden will get wet; humans keep forward chaining under careful control, lest they be swamped with irrelevant consequences.

The backward-chaining algorithm, as its name suggests, works backwards from the query. If the query q is known to be true, then no work is needed. Otherwise, the algorithm finds those implications in the knowledge base that conclude q. If all the premises of one of those implications can be proved true (by backward chaining), then q is true. When applied to the query Q in Figure 7.15, it works back down the graph until it reaches a set of known facts that forms the basis for a proof. The detailed algorithm is left as an exercise; as with forward chaining, an efficient implementation runs in linear time.

GOAL-DIRECTED REASONING

Backward chaining is a form of **goal-directed reasoning**. It is useful for answering specific questions such as "What shall I do now?" and "Where are my keys?" Often, the cost of backward chaining is *much less* than linear in the size of the knowledge base, because the process touches only relevant facts. In general, an agent should share the work between forward and backward reasoning, limiting forward reasoning to the generation of facts that are likely to be relevant to queries that will be solved by backward chaining.

7.6 Effective propositional inference

In this section, we describe two families of efficient algorithms for propositional inference based on model checking: one approach based on backtracking search, and one on hillclimbing search. These algorithms are part of the "technology" of propositional logic. This section can be skimmed on a first reading of the chapter.

The algorithms we describe are for checking satisfiability. We have already noted the connection between finding a satisfying model for a logical sentence and finding a solution for a constraint satisfaction problem, so it is perhaps not surprising that the two families of algorithms closely resemble the backtracking algorithms of Section 5.2 and the local-search algorithms of Section 5.3. They are, however, extremely important in their own right because so many combinatorial problems in computer science can be reduced to checking the satisfiability of a propositional sentence. Any improvement in satisfiability algorithms has huge consequences for our ability to handle complexity in general.

A complete backtracking algorithm

DAVIS–PUTNAM ALGORITHM

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

- *Early termination*: The algorithm detects whether the sentence must be true or false, even with a partially completed model. A clause is true if *any* literal is true, even if the other literals do not yet have truth values; hence, the sentence as a whole could be judged true even before the model is complete. For example, the sentence $(A \vee B) \wedge (A \vee C)$ is true if A is true, regardless of the values of B and C. Similarly, a sentence is false if *any* clause is false, which occurs when each of its literals is false. Again, this can occur long before the model is complete. Early termination avoids examination of entire subtrees in the search space.

PURE SYMBOL

- *Pure symbol heuristic*: A **pure symbol** is a symbol that always appears with the same "sign" in all clauses. For example, in the three clauses $(A \vee \neg B)$, $(\neg B \vee \neg C)$, and $(C \vee A)$, the symbol A is pure because only the positive literal appears, B is pure because only the negative literal appears, and C is impure. It is easy to see that if a sentence has a model, then it has a model with the pure symbols assigned so as to make their literals $true$, because doing so can never make a clause false. Note that, in determining the purity of a symbol, the algorithm can ignore clauses that are already known to be true in the model constructed so far. For example, if the model contains $B = false$, then the clause $(\neg B \vee \neg C)$ is already true, and C becomes pure because it appears only in $(C \vee A)$.
- *Unit clause heuristic*: A **unit clause** was defined earlier as a clause with just one literal. In the context of DPLL, it also means clauses in which all literals but one are already assigned $false$ by the model. For example, if the model contains $B = false$, then $(B \vee \neg C)$ becomes a unit clause because it is equivalent to $(False \vee \neg C)$, or just $\neg C$. Obviously, for this clause to be true, C must be set to $false$. The unit clause heuristic assigns all such symbols before branching on the remainder. One important consequence of the heuristic is that any attempt to prove (by refutation) a literal that is

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

  $clauses$ ← the set of clauses in the CNF representation of $s$
  $symbols$ ← a list of the proposition symbols in $s$
  return DPLL($clauses, symbols$, [ ])
```

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

  if every clause in $clauses$ is true in $model$ then return $true$
  if some clause in $clauses$ is false in $model$ then return $false$
  $P, value$ ← FIND-PURE-SYMBOL($symbols, clauses, model$)
  if $P$ is non-null then return DPLL($clauses, symbols – P$, EXTEND($P, value, model$))
  $P, value$ ← FIND-UNIT-CLAUSE($clauses, model$)
  if $P$ is non-null then return DPLL($clauses, symbols – P$, EXTEND($P, value, model$))
  $P$ ← FIRST($symbols$); $rest$ ← REST($symbols$)
  return DPLL($clauses, rest$, EXTEND($P, true, model$)) or
         DPLL($clauses, rest$, EXTEND($P, false, model$))
```

Figure 7.16 The DPLL algorithm for checking satisfiability of a sentence in propositional logic. FIND-PURE-SYMBOL and FIND-UNIT-CLAUSE are described in the text; each returns a symbol (or null) and the truth value to assign to that symbol. Like TT-ENTAILS?, it operates over partial models.

already in the knowledge base will succeed immediately (Exercise 7.16). Notice also that assigning one unit clause can create another unit clause—for example, when C is set to *false*, $(C \vee A)$ becomes a unit clause, causing true to be assigned to A. This "cascade" of forced assignments is called **unit propagation**. It resembles the process of forward chaining with Horn clauses, and indeed, if the CNF expression contains only Horn clauses then DPLL essentially replicates forward chaining. (See Exercise 7.17.)

UNIT PROPAGATION

The DPLL algorithm is shown in Figure 7.16. We have given the essential skeleton of the algorithm, which describes the search process itself. We have not described the data structures that must be maintained in order to make each search step efficient, nor the tricks that can be added to improve performance: clause learning, variable selection heuristics, and randomized restarts. When these are included DPLL is one of the fastest satisfiability algorithms yet developed, despite its antiquity. The CHAFF implementation is used to solve hardware verification problems with a million variables.

Local-search algorithms

We have seen several local-search algorithms so far in this book, including HILL-CLIMBING (page 112) and SIMULATED-ANNEALING (page 116). These algorithms can be applied directly to satisfiability problems, provided that we choose the right evaluation function. Because the goal is to find an assignment that satisfies every clause, an evaluation function that counts the number of unsatisfied clauses will do the job. In fact, this is exactly the measure

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

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

Figure 7.17 The WALKSAT algorithm for checking satisfiability by randomly flipping the values of variables. Many versions of the algorithm exist.

used by the MIN-CONFLICTS algorithm for CSPs (page 151). All these algorithms take steps in the space of complete assignments, flipping the truth value of one symbol at a time. The space usually contains many local minima, to escape from which various forms of randomness are required. In recent years, there has been a great deal of experimentation to find a good balance between greediness and randomness.

One of the simplest and most effective algorithms to emerge from all this work is called WALKSAT (Figure 7.17). On every iteration, the algorithm picks an unsatisfied clause and picks a symbol in the clause to flip. It chooses randomly between two ways to pick which symbol to flip: (1) a "min-conflicts" step that minimizes the number of unsatisfied clauses in the new state, and (2) a "random walk" step that picks the symbol randomly.

Does WALKSAT actually work? Clearly, if it returns a model, then the input sentence is indeed satisfiable. What if it returns *failure*? Unfortunately, in that case we cannot tell whether the sentence is unsatisfiable or we need to give the algorithm more time. We could try setting max_flips to infinity. In that case, it is easy to show that WALKSAT will eventually return a model (if one exists), provided that the probability $p > 0$. This is because there is always a sequence of flips leading to a satisfying assignment, and eventually the random walk steps will generate that sequence. Alas, if max_flips is infinity and the sentence is unsatisfiable, then the algorithm never terminates!

What this suggests is that local-search algorithms such as WALKSAT are most useful when we expect a solution to exist—for example, the problems discussed in Chapters 3 and 5 usually have solutions. On the other hand, local search cannot always detect *unsatisfiability*, which is required for deciding entailment. For example, an agent cannot *reliably* use local search to prove that a square is safe in the wumpus world. Instead, it can say, "I thought about it for an hour and couldn't come up with a possible world in which the square *isn't* safe." If the local-search algorithm is usually really fast at finding a model when one exists, the agent might be justified in assuming that failure to find a model indicates unsatisfiability. This isn't the same as a proof, of course, and the agent should think twice before staking its life on it.

Hard satisfiability problems

We now look at how DPLL and WALKSAT perform in practice. We are particularly interested in *hard* problems, because *easy* problems can be solved by any old algorithm. In Chapter 5, we saw some surprising discoveries about certain kinds of problems. For example, the n-queens problem—thought to be quite tricky for backtracking search algorithms—turned out to be trivially easy for local-search methods, such as min-conflicts. This is because solutions are very densely distributed in the space of assignments, and any initial assignment is guaranteed to have a solution nearby. Thus, n-queens is easy because it is **underconstrained**.

UNDERCONSTRAINED

When we look at satisfiability problems in conjunctive normal form, an underconstrained problem is one with relatively *few* clauses constraining the variables. For example, here is a randomly generated[12] 3-CNF sentence with five symbols and five clauses:

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

16 of the 32 possible assignments are models of this sentence, so, on average, it would take just two random guesses to find a model.

So where are the hard problems? Presumably, if we *increase* the number of clauses, keeping the number of symbols fixed, we make the problem more constrained, and solutions become harder to find. Let m be the number of clauses and n be the number of symbols. Figure 7.18(a) shows the probability that a random 3-CNF sentence is satisfiable, as a function of the clause/symbol ratio, m/n, with n fixed at 50. As we expect, for small m/n the probability is close to 1, and at large m/n the probability is close to 0. The probability drops fairly sharply around $m/n = 4.3$. CNF sentences near this **critical point** could be described as "nearly satisfiable" or "nearly unsatisfiable." Is this where the hard problems are?

CRITICAL POINT

Figure 7.18(b) shows the runtime for DPLL and WALKSAT around this point, where we have restricted attention to just the *satisfiable* problems. Three things are clear: First, problems near the critical point are *much* more difficult than other random problems. Second, even on the hardest problems, DPLL is quite effective—an average of a few thousand steps compared with $2^{50} \approx 10^{15}$ for truth-table enumeration. Third, WALKSAT is much faster than DPLL throughout the range.

Of course, these results are only for randomly generated problems. Real problems do not necessarily have the same structure—in terms of proportions of positive and negative literals, densities of connections among clauses, and so on—as random problems. Yet, in practice, WALKSAT and related algorithms are very good at solving real problems too—often as good as the best special-purpose algorithms for those tasks. Problems with thousands of symbols and millions of clauses are routinely handled by solvers such as CHAFF. These observations suggest that some combination of the min-conflicts heuristic and random-walk behavior provides a *general-purpose* capability for resolving most situations in which combinatorial reasoning is required.

[12] Each clause contains three randomly selected *distinct* symbols, each of which is negated with 50% probability.

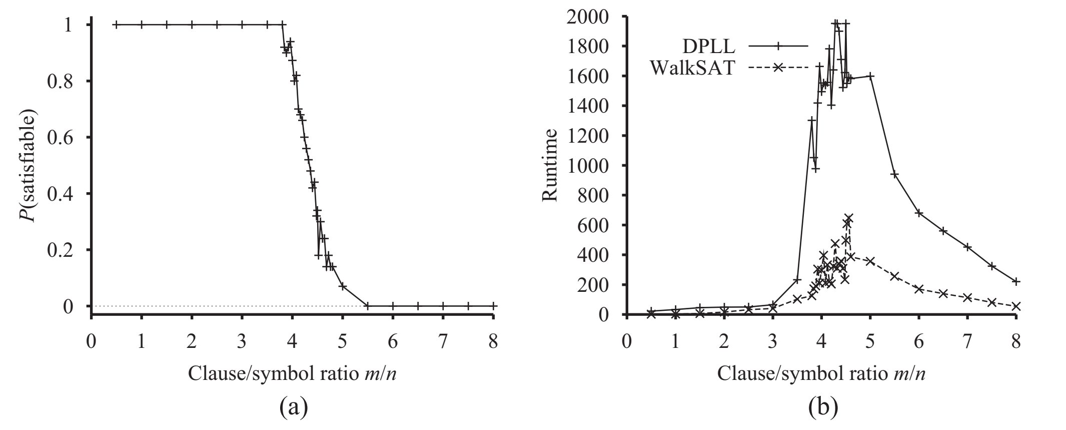


Figure 7.18 (a) Graph showing the probability that a random 3-CNF sentence with $n = 50$ symbols is satisfiable, as a function of the clause/symbol ratio m/n. (b) Graph of the median runtime of DPLL and WALKSAT on 100 *satisfiable* random 3-CNF sentences with $n = 50$, for a narrow range of m/n around the critical point.

7.7 AGENTS BASED ON PROPOSITIONAL LOGIC

In this section, we bring together what we have learned so far in order to construct agents that operate using propositional logic. We will look at two kinds of agents: those which use inference algorithms and a knowledge base, like the generic knowledge-based agent in Figure 7.1, and those which evaluate logical expressions directly in the form of circuits. We will demonstrate both kinds of agents in the wumpus world, and will find that both suffer from serious drawbacks.

Finding pits and wumpuses using logical inference

Let us begin with an agent that reasons logically about the location of pits, wumpuses, and safe squares. It begins with a knowledge base that states the "physics" of the wumpus world. It knows that [1,1] does not contain a pit or a wumpus; that is, $\neg P_{1,1}$ and $\neg W_{1,1}$. For every square $[x, y]$, it knows a sentence stating how a breeze arises:

$$B_{x,y} \Leftrightarrow (P_{x,y+1} \vee P_{x,y-1} \vee P_{x+1,y} \vee P_{x-1,y}) \,. \tag{7.1}$$

For every square $[x, y]$, it knows a sentence stating how a stench arises:

$$S_{x,y} \Leftrightarrow (W_{x,y+1} \vee W_{x,y-1} \vee W_{x+1,y} \vee W_{x-1,y}) \,. \tag{7.2}$$

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

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

Then, we have to say that there is *at most one* wumpus. One way to do this is to say that for any two squares, one of them must be wumpus-free. With n squares, we get $n(n-1)/2$

```
function PL-WUMPUS-AGENT(percept) returns an action
  inputs: percept, a list, [stench,breeze,glitter]
  static: KB, a knowledge base, initially containing the "physics" of the wumpus world
          x, y, orientation, the agent's position (initially 1,1) and orientation (initially right)
          visited, an array indicating which squares have been visited, initially false
          action, the agent's most recent action, initially null
          plan, an action sequence, initially empty

  update x,y,orientation, visited based on action
  if stench then TELL(KB, $S_{x,y}$) else TELL(KB, $\neg S_{x,y}$)
  if breeze then TELL(KB, $B_{x,y}$) else TELL(KB, $\neg B_{x,y}$)
  if glitter then action ← grab
  else if plan is nonempty then action ← POP(plan)
  else if for some fringe square [i,j], ASK(KB, ($\neg P_{i,j} \land \neg W_{i,j}$)) is true or
          for some fringe square [i,j], ASK(KB, ($P_{i,j} \lor W_{i,j}$)) is false then do
     plan ← A*-GRAPH-SEARCH(ROUTE-PROBLEM([x,y], orientation, [i,j],visited))
     action ← POP(plan)
  else action ← a randomly chosen move
  return action
```

Figure 7.19 A wumpus-world agent that uses propositional logic to identify pits, wumpuses, and safe squares. The subroutine ROUTE-PROBLEM constructs a search problem whose solution is a sequence of actions leading from $[x,y]$ to $[i,j]$ and passing through only previously visited squares.

sentences such as $\neg W_{1,1} \lor \neg W_{1,2}$. For a 4×4 world, then, we begin with a total of 155 sentences containing 64 distinct symbols.

The agent program, shown in Figure 7.19, TELLs its knowledge base about each new breeze and stench percept. (It also updates some ordinary program variables to keep track of where it is and where it has been—more on this later.) Then, the program chooses where to look next among the fringe squares—that is, the squares adjacent to those already visited. A fringe square $[i,j]$ is *provably safe* if the sentence $(\neg P_{i,j} \land \neg W_{i,j})$ is entailed by the knowledge base. The next best thing is a *possibly safe* square, for which the agent cannot prove that there *is* a pit or a wumpus—that is, for which $(P_{i,j} \lor W_{i,j})$ is *not* entailed.

The entailment computation in ASK can be implemented using any of the methods described earlier in the chapter. TT-ENTAILS? (Figure 7.10) is obviously impractical, since it would have to enumerate 2^{64} rows. DPLL (Figure 7.16) performs the required inferences in a few milliseconds, thanks mainly to the unit propagation heuristic. WALKSAT can also be used, with the usual caveats about incompleteness. In wumpus worlds, failures to find a model, given 10,000 flips, invariably correspond to unsatisfiability, so no errors are likely due to incompleteness.

PL-WUMPUS-AGENT works quite well in a small wumpus world. There is, however, something deeply unsatisfying about the agent's knowledge base. *KB* contains "physics" sentences of the form given in Equations (7.1) and (7.2) *for every single square*. The larger the environment, the larger the initial knowledge base needs to be. We would much prefer to have just two sentences that say how breezes and stenches arise in *all* squares. These are beyond the powers of propositional logic to express. In the next chapter, we will see a more expressive logical language in which such sentences are easy to express.

Keeping track of location and orientation

The agent program in Figure 7.19 "cheats" because it keeps track of location *outside* the knowledge base, instead of using logical reasoning.[13] To do it "properly," we will need propositions for location. One's first inclination might be to use a symbol such as $L_{1,1}$ to mean that the agent is in [1,1]. Then the initial knowledge base might include sentences like

$$L_{1,1} \land \mathit{FacingRight} \land \mathit{Forward} \;\Rightarrow\; L_{2,1} \;.$$

Instantly, we see that this won't work. If the agent starts in [1,1] facing right and moves forward, the knowledge base will entail both $L_{1,1}$ (the original location) and $L_{1,2}$ (the new location). Yet these propositions cannot both be true! The problem is that the location propositions should refer to two different times. We need $L^1_{1,1}$ to mean that the agent is in [1,1] at time 1, $L^2_{2,1}$ to mean that the agent is in [2,1] at time 2, and so on. The orientation and action propositions also need to depend on time. Therefore, the correct sentence is

$$\begin{array}{l} L^1_{1,1} \land \mathit{FacingRight}^1 \land \mathit{Forward}^1 \;\Rightarrow\; L^2_{2,1} \\ \mathit{FacingRight} \land \mathit{TurnLeft}^1 \;\Rightarrow\; \mathit{FacingUp}^2 \;, \end{array}$$

and so on. It turns out to be quite tricky to build a complete and correct knowledge base for keeping track of everything in the wumpus world; we will defer the full discussion until Chapter 10. The point we want to make here is that the initial knowledge base will contain sentences like the preceding two examples for every time t, as well as for every location. That is, for every time t and location $[x, y]$, the knowledge base contains a sentence of the form

$$L^t_{x,y} \land \mathit{FacingRight}^t \land \mathit{Forward}^t \;\Rightarrow\; L^{t+1}_{x+1,y} \;. \qquad (7.3)$$

Even if we put an upper limit on the number of time steps allowed—100, perhaps—we end up with tens of thousands of sentences. The same problem arises if we add the sentences "as needed" for each new time step. This proliferation of clauses makes the knowledge base unreadable for a human, but fast propositional solvers can still handle the 4×4 Wumpus world with ease (they reach their limit at around 100×100). The circuit-based agents in the next subsection offer a partial solution to this clause proliferation problem, but the full solution will have to wait until we have developed first-order logic in Chapter 8.

Circuit-based agents

CIRCUIT-BASED AGENT

SEQUENTIAL CIRCUIT

GATES

REGISTERS

A **circuit-based agent** is a particular kind of reflex agent with state, as defined in Chapter 2. The percepts are inputs to a **sequential circuit**—a network of **gates**, each of which implements a logical connective, and **registers**, each of which stores the truth value of a single proposition. The outputs of the circuit are registers corresponding to actions—for example,

[13] The observant reader will have noticed that this allowed us to finesse the connection between the raw percepts such as *Breeze* and the location-specific propositions such as $B_{1,1}$.

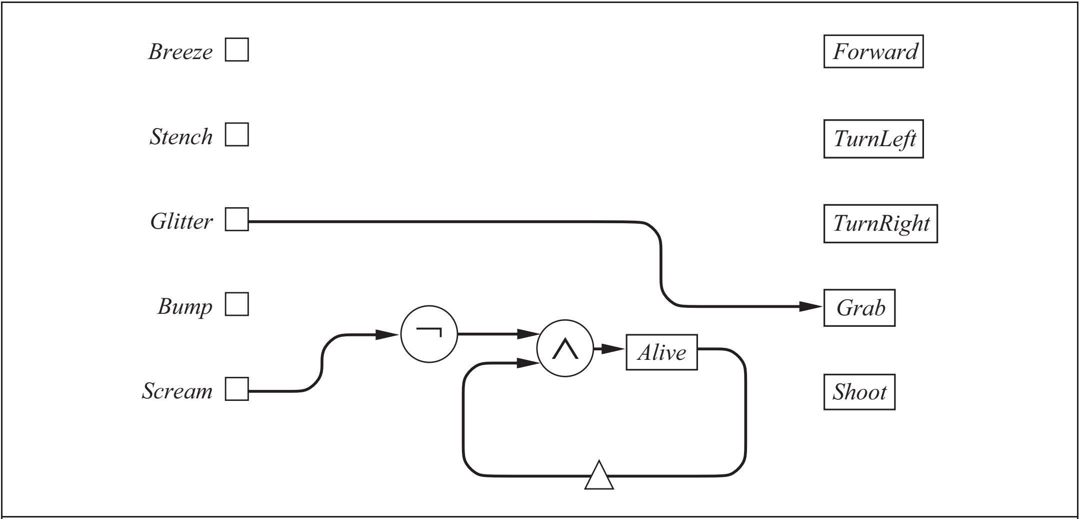


Figure 7.20 Part of a circuit-based agent for the wumpus world, showing inputs, outputs, the circuit for grabbing the gold, and the circuit for determining whether the wumpus is alive. Registers are shown as rectangles and one-step delays are shown as small triangles.

the $Grab$ output is set to $true$ if the agent wants to grab something. If the $Glitter$ input is connected directly to the $Grab$ output, the agent will grab the goal whenever it sees it. (See Figure 7.20.)

DATAFLOW

Circuits are evaluated in a **dataflow** fashion: at each time step, the inputs are set and the signals propagate through the circuit. Whenever a gate has all its inputs, it produces an output. This process is closely related to the process of forward chaining in an AND–OR graph such as Figure 7.15(b).

We said in the preceding section that circuit-based agents handle time more satisfactorily than propositional inference-based agents. This is because the value stored in each register gives the truth value of the corresponding proposition symbol *at the current time* t, rather than having a different copy for each different time step. For example, we might have an $Alive$ register that should contain $true$ when the wumpus is alive and $false$ when it is dead. This register corresponds to the proposition symbol $Alive^t$, so on each time step it refers to a different proposition. The internal state of the agent—i.e., its memory—is maintained by connecting the output of a register back into the circuit through a **delay line**. This delivers the value of the register at the *previous* time step. Figure 7.20 shows an example. The value for $Alive$ is given by the conjunction of the negation of $Scream$ and the delayed value of $Alive$ itself. In terms of propositions, the circuit for $Alive$ implements the biconditional

DELAY LINE

$$Alive^t \Leftrightarrow \neg Scream^t \wedge Alive^{t-1} . \tag{7.4}$$

which says that the wumpus is alive at time t *if and only if* there was no scream perceived at time t (from a scream at $t-1$) *and* it was alive at $t-1$. We assume that the circuit is initialized with $Alive$ set to $true$. Therefore, $Alive$ will remain true until there is a scream, whereupon it will become false and stay false. This is exactly what we want.

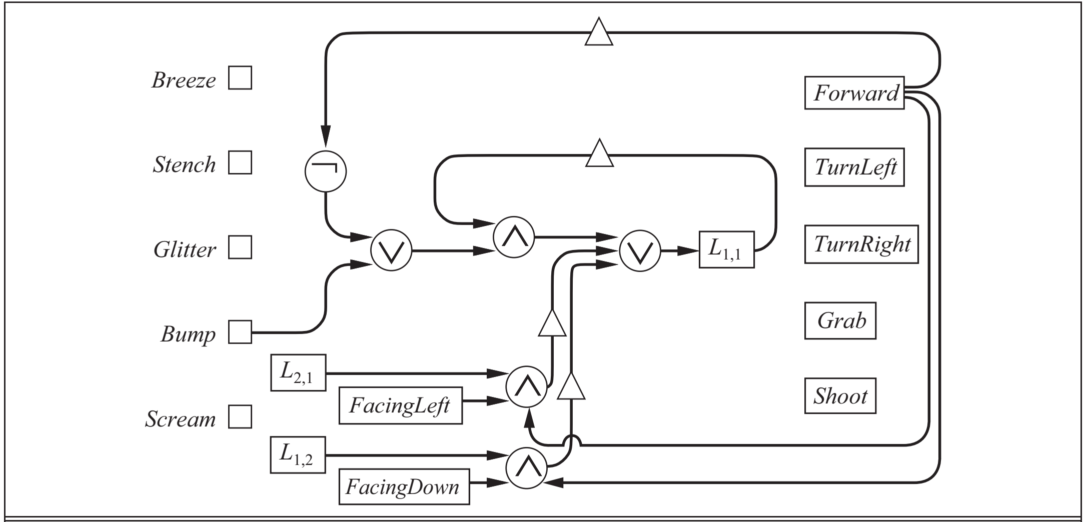


Figure 7.21 The circuit for determining whether the agent is at [1,1]. Every location and orientation register has a similar circuit attached.

The agent's location can be handled in much the same way as the wumpus's health. We need an $L_{x,y}$ register for each x and y; its value should be $true$ if the agent is at $[x, y]$. The circuit that sets the value of $L_{x,y}$ is, however, much more complicated than the circuit for $Alive$. For example, the agent is at [1,1] at time t if (a) it was there at $t - 1$ and either didn't move forward or tried but bumped into a wall; or (b) it was at [1,2] facing down and moved forward; or (c) it was at [2,1] facing left and moved forward:

$$\begin{aligned} L^t_{1,1} \quad \Leftrightarrow \quad & (L^{t-1}_{1,1} \wedge (\neg Forward^{t-1} \vee Bump^t)) \\ \vee \; & (L^{t-1}_{1,2} \wedge (FacingDown^{t-1} \wedge Forward^{t-1})) \\ \vee \; & (L^{t-1}_{2,1} \wedge (FacingLeft^{t-1} \wedge Forward^{t-1}))\,. \end{aligned} \tag{7.5}$$

The circuit for $L_{1,1}$ is shown in Figure 7.21. Every location register has a similar circuit attached to it. Exercise 7.13(b) asks you to design a circuit for the orientation propositions.

The circuits in Figures 7.20 and 7.21 maintain the correct truth values for $Alive$ and $L_{x,y}$ for all time. These propositions are unusual, however, in that *their correct truth values can always be ascertained.* Consider instead the proposition $B_{4,4}$: square [4,4] is breezy. Although this proposition's truth value remains fixed, the agent cannot learn that truth value until it has visited [4,4] (or deduced that there is an adjacent pit). Propositional and first-order logic are designed to represent true, false, and unknown propositions automatically, but circuits are not: the register for $B_{4,4}$ must contain *some* value, either $true$ or $false$, even before the truth has been discovered. The value in the register might well be the wrong one, and this could lead the agent astray. In other words, we need to represent three possible states ($B_{4,4}$ is known true, known false, or unknown) and we only have one bit to do it with.

The solution to this problem is to use two bits instead of one. $B_{4,4}$ is represented by two registers that we will call $K(B_{4,4})$ and $K(\neg B_{4,4})$, where K stands for "known.". (Remember that these are still just symbols with complicated names, even though they look like structured expressions!) When both $K(B_{4,4})$ and $K(\neg B_{4,4})$ are false, it means the truth value of $B_{4,4}$ is unknown. (If both are true, there's a bug in the knowledge base!) Now whenever we would use $B_{4,4}$ in some part of the circuit, we use $K(B_{4,4})$ instead ; and whenever we would use $\neg B_{4,4}$, we use $K(\neg B_{4,4})$. In general, we represent each potentially indeterminate proposition with two **knowledge propositions** that state whether the underlying proposition is known to be true and known to be false.

KNOWLEDGE PROPOSITION

We will see an example of how to use knowledge propositions shortly. First, we need to work out how to determine the truth values of the knowledge propositions themselves. Notice that, whereas $B_{4,4}$ has a fixed truth value, $K(B_{4,4})$ and $K(\neg B_{4,4})$ *do* change as the agent finds out more about the world. For example, $K(B_{4,4})$ starts out false and then becomes true as soon as $B_{4,4}$ can be determined to be true—that is, when the agent is in [4,4] and detects a breeze. It stays true thereafter. So we have

$$K(B_{4,4})^t \Leftrightarrow K(B_{4,4})^{t-1} \vee (L^t_{4,4} \wedge \mathit{Breeze}^t) \; . \qquad (7.6)$$

A similar equation can be written for $K(\neg B_{4,4})^t$.

Now that the agent knows about breezy squares, it can deal with pits. The absence of a pit in a square can be ascertained if and only if one of the neighboring squares is known not to be breezy. For example, we have

$$K(\neg P_{4,4})^t \Leftrightarrow K(\neg B_{3,4})^t \vee K(\neg B_{4,3})^t \; . \qquad (7.7)$$

Determining that there *is* a pit in a square is more difficult—there must be a breeze in an adjacent square that cannot be accounted for by another pit:

$$\begin{array}{rcl} K(P_{4,4})^t & \Leftrightarrow & (K(B_{3,4})^t \wedge K(\neg P_{2,4})^t \wedge K(\neg P_{3,3})^t) \\ & \vee & (K(B_{4,3})^t \wedge K(\neg P_{4,2})^t \wedge K(\neg P_{3,3})^t) \; . \end{array} \qquad (7.8)$$

While the circuits for determining the presence or absence of pits are somewhat hairy, *they have only a constant number of gates for each square.* This property is essential if we are to build circuit-based agents that scale up in a reasonable way. It is really a property of the wumpus world itself; we say that an environment exhibits **locality** if the truth of each proposition of interest can be determined looking only at a constant number of other propositions. Locality is very sensitive to the precise "physics" of the environment. For example, the minesweeper domain (Exercise 7.11) is nonlocal because determining that a mine is in a given square can involve looking at squares arbitrarily far away. For nonlocal domains, circuit-based agents are not always practical.

ACYCLICITY

There is one issue around which we have tiptoed carefully: the question of **acyclicity**. A circuit is acyclic if every path that connects the output of a register back to its input has an intervening delay element. We require that all circuits be acyclic because cyclic circuits, as physical devices, do not work! They can go into unstable oscillations resulting in undefined values. As an example of a cyclic circuit, consider the following augmentation of Equation (7.6):

$$K(B_{4,4})^t \Leftrightarrow K(B_{4,4})^{t-1} \vee (L^t_{4,4} \wedge \mathit{Breeze}^t) \vee K(P_{3,4})^t \vee K(P_{4,3})^t \; . \qquad (7.9)$$

The extra disjuncts, $K(P_{3,4})^t$ and $K(P_{4,3})^t$, allow the agent to determine breeziness from the known presence of adjacent pits, which seems entirely reasonable. Now, unfortunately, breeziness depends on adjacent pits, and pits depend on adjacent breeziness through equations such as Equation (7.8). Therefore, the complete circuit would contain cycles.

The difficulty is not that the augmented Equation (7.9) is *incorrect*. Rather, the problem is that the interlocking dependencies represented by these equations cannot be resolved by the simple mechanism of propagating truth values in the corresponding Boolean circuit. The acyclic version using Equation (7.6), which determines breeziness only from direct observation, is *incomplete* in the sense that at some points the circuit-based agent might know less than an inference-based agent using a complete inference procedure. For example, if there is a breeze in [1,1], the inference-based agent can conclude that there is also a breeze in [2,2], whereas the acyclic circuit-based agent using Equation (7.6) cannot. A complete circuit *can* be built—after all, sequential circuits can emulate any digital computer—but it would be significantly more complex.

A comparison

The inference-based agent and the circuit-based agent represent the declarative and procedural extremes in agent design. They can be compared along several dimensions:

- *Conciseness*: The circuit-based agent, unlike the inference-based agent, need not have separate copies of its "knowledge" for every time step. Instead, it refers only to the current and previous time steps. Both agents need copies of the "physics" (expressed as sentences or circuits) for every square and therefore do not scale well to larger environments. In environments with many objects related in complex ways, the number of propositions will swamp any propositional agent. Such environments require the expressive power of first-order logic. (See Chapter 8.) Propositional agents of both kinds are also poorly suited for expressing or solving the problem of finding a path to a nearby safe square. (For this reason, PL-WUMPUS-AGENT falls back on a search algorithm.)
- *Computational efficiency*: In the *worst* case, inference can take time exponential in the number of symbols, whereas evaluating a circuit takes time linear in the size of the circuit (or linear in the *depth* of the circuit if realized as a physical device). In *practice*, however, we saw that DPLL completed the required inferences very quickly.[14]
- *Completeness*: We suggested earlier that the circuit-based agent might be incomplete because of the acyclicity restriction. The reasons for incompleteness are actually more fundamental. First, remember that a circuit executes in time linear in the circuit size. This means that, for some environments, a circuit that is complete (i.e., one that computes the truth value of every determinable proposition) must be exponentially larger than the inference-based agent's KB. Otherwise, we would have a way to solve the propositional entailment problem in less than exponential time, which is very unlikely. A second reason is the nature of the internal state of the agent. The inference-based agent remembers every percept and knows, either implicitly or explicitly, every sentence that follows from the percepts and initial KB. For example, given $B_{1,1}$, it knows the disjunction $P_{1,2} \lor P_{2,1}$, from which $B_{2,2}$ follows. The circuit-based agent, on the

[14] In fact, all the inferences done by a circuit can be done in linear time by DPLL! This is because evaluating a circuit, like forward chaining, can be emulated by DPLL using the unit propagation rule.

other hand, forgets all previous percepts and remembers just the individual propositions stored in registers. Thus, $P_{1,2}$ and $P_{2,1}$ remain *individually* unknown after the first percept, so no conclusion will be drawn about $B_{2,2}$.

- *Ease of construction*: This is a very important issue about which it is hard to be precise. Certainly, this author found it much easier to state the "physics" declaratively, whereas devising small, acyclic, not-too-incomplete circuits for direct detection of pits seemed quite difficult.

In sum, it seems there are *tradeoffs* among computational efficiency, conciseness, completeness, and ease of construction. When the connection between percepts and actions is simple—as in the connection between $Glitter$ and $Grab$—a circuit seems optimal. For more complex connections, the declarative approach may be better. In a domain such as chess, for example, the declarative rules are concise and easily encoded (at least in first-order logic), but a circuit for computing moves directly from board states would be unimaginably vast.

We see different points on these tradeoffs in the animal kingdom. The lower animals with very simple nervous systems are probably circuit-based, whereas higher animals, including humans, seem to perform inference on explicit representations. This enables them to compute much more complex agent functions. Humans also have circuits to implement reflexes, and perhaps also **compile** declarative representations into circuits when certain inferences become routine. In this way, a **hybrid agent** design (see Chapter 2) can have the best of both worlds.

COMPILATION

7.8 Summary

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

- Intelligent agents need knowledge about the world in order to reach good decisions.
- Knowledge is contained in agents in the form of **sentences** in a **knowledge representation language** that are stored in a **knowledge base**.
- A knowledge-based agent is composed of a knowledge base and an inference mechanism. It operates by storing sentences about the world in its knowledge base, using the inference mechanism to infer new sentences, and using these sentences to decide what action to take.
- A representation language is defined by its **syntax**, which specifies the structure of sentences, and its **semantics**, which defines the **truth** of each sentence in each **possible world** or **model**.
- The relationship of **entailment** between sentences is crucial to our understanding of reasoning. A sentence α entails another sentence β if β is true in all worlds where α is true. Equivalent definitions include the **validity** of the sentence $\alpha \Rightarrow \beta$ and the **unsatisfiability** of the sentence $\alpha \wedge \neg\beta$.

- Inference is the process of deriving new sentences from old ones. **Sound** inference algorithms derive *only* sentences that are entailed; **complete** algorithms derive *all* sentences that are entailed.
- **Propositional logic** is a very simple language consisting of **proposition symbols** and **logical connectives**. It can handle propositions that are known true, known false, or completely unknown.
- The set of possible models, given a fixed propositional vocabulary, is finite, so entailment can be checked by enumerating models. Efficient **model-checking** inference algorithms for propositional logic include backtracking and local-search methods and can often solve large problems very quickly.
- **Inference rules** are patterns of sound inference that can be used to find proofs. The **resolution** rule yields a complete inference algorithm for knowledge bases that are expressed in **conjunctive normal form**. **Forward chaining** and **backward chaining** are very natural reasoning algorithms for knowledge bases in **Horn form**.
- Two kinds of agents can be built on the basis of propositional logic: **inference-based agents** use inference algorithms to keep track of the world and deduce hidden properties, whereas **circuit-based agents** represent propositions as bits in registers and update them using signal propagation in logical circuits.
- Propositional logic is reasonably effective for certain tasks within an agent, but does not scale to environments of unbounded size because it lacks the expressive power to deal concisely with time, space, and universal patterns of relationships among objects.

Bibliographical and Historical Notes

John McCarthy's paper "Programs with Common Sense" (McCarthy, 1958, 1968) promulgated the notion of agents that use logical reasoning to mediate between percepts and actions. It also raised the flag of declarativism, pointing out that telling an agent what it needs to know is a very elegant way to build software. Allen Newell's (1982) article "The Knowledge Level" makes the case that rational agents can be described and analyzed at an abstract level defined by the knowledge they possess rather than the programs they run. The declarative and procedural approaches to AI are compared in Boden (1977). The debate was revived by, among others, Brooks (1991) and Nilsson (1991).

Logic itself had its origins in ancient Greek philosophy and mathematics. Various logical principles—principles connecting the syntactic structure of sentences with their truth and falsity, with their meaning, or with the validity of arguments in which they figure—are scattered in the works of Plato. The first known systematic study of logic was carried out by Aristotle, whose work was assembled by his students after his death in 322 B.C. as a

SYLLOGISMS

treatise called the *Organon*. Aristotle's **syllogisms** were what we would now call inference rules. Although the syllogisms included elements of both propositional and first-order logic, the system as a whole was very weak by modern standards. It did not allow for patterns of inference that apply to sentences of arbitrary complexity, as in modern propositional logic.

The closely related Megarian and Stoic schools (originating in the fifth century B.C. and continuing for several centuries thereafter) introduced the systematic study of implication and other basic constructs still used in modern propositional logic. The use of truth tables for defining logical connectives is due to Philo of Megara. The Stoics took five basic inference rules as valid without proof, including the rule we now call Modus Ponens. They derived a number of other rules from these five, using among other principles the deduction theorem (page 210) and were much clearer about the notion of proof than Aristotle was. The Stoics claimed that their logic was complete in the sense of capturing all valid inferences, but what remains is too fragmentary to tell. A good account of the history of Megarian and Stoic logic, as far as it is known, is given by Benson Mates (1953).

The idea of reducing logical inference to a purely mechanical process applied to a formal language is due to Wilhelm Leibniz (1646–1716). Leibniz's own mathematical logic, however, was severely defective, and he is better remembered simply for introducing these ideas as goals to be attained than for his attempts at realizing them.

George Boole (1847) introduced the first comprehensive and workable system of formal logic in his book *The Mathematical Analysis of Logic*. Boole's logic was closely modeled on the ordinary algebra of real numbers and used substitution of logically equivalent expressions as its primary inference method. Although Boole's system still fell short of full propositional logic, it was close enough that other mathematicians could quickly fill in the gaps. Schröder (1877) described conjunctive normal form, while Horn form was introduced much later by Alfred Horn (1951). The first comprehensive exposition of modern propositional logic (and first-order logic) is found in Gottlob Frege's (1879) *Begriffschrift* ("Concept Writing" or "Conceptual Notation").

The first mechanical device to carry out logical inferences was constructed by the third Earl of Stanhope (1753–1816). The Stanhope Demonstrator could handle syllogisms and certain inferences in the theory of probability. William Stanley Jevons, one of those who improved upon and extended Boole's work, constructed his "logical piano" in 1869 to perform inferences in Boolean logic. An entertaining and instructive history of these and other early mechanical devices for reasoning is given by Martin Gardner (1968). The first published computer program for logical inference was the Logic Theorist of Newell, Shaw, and Simon (1957). This program was intended to model human thought processes. Martin Davis (1957) had actually designed a program that came up with a proof in 1954, but the Logic Theorist's results were published slightly earlier. Both Davis's 1954 program and the Logic Theorist were based on somewhat ad hoc methods that did not strongly influence later automated deduction.

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

Stephen Cook (1971) showed that deciding satisfiability of a sentence in propositional logic is NP-complete. Since deciding entailment is equivalent to deciding unsatisfiability, it is co-NP-complete. Many subsets of propositional logic are known for which the satisfiability problem is polynomially solvable; Horn clauses are one such subset. The linear-time forward-chaining algorithm for Horn clauses is due to Dowling and Gallier (1984), who describe their algorithm as a dataflow process similar to the propagation of signals in a circuit. Satisfiability has become one of the canonical examples for NP reductions; for example Kaye (2000) showed that the Minesweeper game (see Exercise 7.11) is NP-complete.

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

The "phase transition" in satisfiability of random k-SAT problems was first observed by Simon and Dubois (1989). Empirical results due to Crawford and Auton (1993) suggest that it lies at a clause/variable ratio of around 4.24 for large random 3-SAT problems; this paper also describes a very efficient implementation of DPLL. (Bayardo and Schrag, 1997) describe another efficient DPLL implementation using techniques from constraint satisfaction, and (Moskewicz *et al.*, 2001) describe CHAFF, which solves million-variable hardware verification problems and was the winner of the SAT 2002 Competition. Li and Anbulagan (1997) discuss heuristics based on unit propagation that allow for fast solvers. Cheeseman *et al.* (1991) provide data on a number of related problems and conjecture that all NP hard problems have a phase transition. Kirkpatrick and Selman (1994) describe ways in which techniques from statistical physics might provide insight into the precise "shape" of the phase transition. Theoretical analysis of its *location* is still rather weak: all that can be proved is that it lies in the range [3.003,4.598] for random 3-SAT. Cook and Mitchell (1997) give an excellent survey of results on this and several other satisfiability-related topics.

Early theoretical investigations showed that DPLL has polynomial average-case complexity for certain natural distributions of problems. This potentially exciting fact became less exciting when Franco and Paull (1983) showed that the same problems could be solved in constant time simply by guessing random assignments. The random-generation method described in the chapter produces much harder problems. Motivated by the empirical success of local search on these problems, Koutsoupias and Papadimitriou (1992) showed that a simple hill-climbing algorithm can solve *almost all* satisfiability problem instances very quickly, suggesting that hard problems are rare. Moreover, Schöning (1999) exhibited a randomized variant of GSAT whose *worst-case* expected runtime on 3-SAT problems is 1.333^n—still exponential, but substantially faster than previous worst-case bounds. Satisfiability algorithms are still a very active area of research; the collection of articles in Du *et al.* (1999) provides a good starting point.

Circuit-based agents can be traced back to the seminal paper of McCulloch and Pitts (1943), which initiated the field of neural networks. Contrary to popular supposition, the paper was concerned with the implementation of a Boolean circuit-based agent design in the brain. Circuit-based agents have received little attention in AI, however. The most notable exception is the work of Stan Rosenschein (Rosenschein, 1985; Kaelbling and Rosenschein, 1990), who developed ways to compile circuit-based agents from declarative descriptions of the task environment. The circuits for updating propositions stored in registers are closely related to the **successor-state axiom** developed for first-order logic by Reiter (1991). The work of Rod Brooks (1986, 1989) demonstrates the effectiveness of circuit-based designs for controlling robots—a topic we take up in Chapter 25. Brooks (1991) argues that circuit-based designs are *all* that is needed for AI—that representation and reasoning are cumbersome, expensive, and unnecessary. In our view, neither approach is sufficient by itself.

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

Exercises

7.1 Describe the wumpus world according to the properties of task environments listed in Chapter 2.

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

$\alpha_2 =$ "There is no pit in [2,2]."
$\alpha_3 =$ "There is a wumpus in [1,3]."

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

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

a. Write a recursive algorithm PL-TRUE?(s, m) that returns $true$ if and only if the sentence s is true in the model m (where m assigns a truth value for every symbol in s). The algorithm should run in time linear in the size of the sentence. (Alternatively, use a version of this function from the online code repository.)

b. Give three examples of sentences that can be determined to be true or false in a *partial* model that does not specify a truth value for some of the symbols.

c. Show that the truth value (if any) of a sentence in a partial model cannot be determined efficiently in general.

d. Modify your PL-TRUE? algorithm so that it can sometimes judge truth from partial models, while retaining its recursive structure and linear runtime. Give three examples of sentences whose truth in a partial model is *not* detected by your algorithm.

e. Investigate whether the modified algorithm makes TT-ENTAILS? more efficient.

7.4 Prove each of the following assertions:

a. α is valid if and only if $\mathit{True} \models \alpha$.

b. For any α, $\mathit{False} \models \alpha$.

c. $\alpha \models \beta$ if and only if the sentence $(\alpha \;\Rightarrow\; \beta)$ is valid.

d. $\alpha \equiv \beta$ if and only if the sentence $(\alpha \;\Leftrightarrow\; \beta)$ is valid.

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

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

a. $(A \land B) \lor (B \land C)$

b. $A \lor B$

c. $A \;\Leftrightarrow\; B \;\Leftrightarrow\; C$

7.6 We have defined four different binary logical connectives.

a. Are there any others that might be useful?

b. How many binary connectives can there be?

c. Why are some of them not very useful?

7.7 Using a method of your choice, verify each of the equivalences in Figure 7.11.

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

a. $\mathit{Smoke} \;\Rightarrow\; \mathit{Smoke}$

b. $\mathit{Smoke} \;\Rightarrow\; \mathit{Fire}$

c. $(\mathit{Smoke} \;\Rightarrow\; \mathit{Fire}) \;\Rightarrow\; (\neg\mathit{Smoke} \;\Rightarrow\; \neg\mathit{Fire})$

d. $\mathit{Smoke} \lor \mathit{Fire} \lor \neg\mathit{Fire}$

e. $((\mathit{Smoke} \land \mathit{Heat}) \;\Rightarrow\; \mathit{Fire}) \;\Leftrightarrow\; ((\mathit{Smoke} \;\Rightarrow\; \mathit{Fire}) \lor (\mathit{Heat} \;\Rightarrow\; \mathit{Fire}))$

f. $(\mathit{Smoke} \;\Rightarrow\; \mathit{Fire}) \;\Rightarrow\; ((\mathit{Smoke} \land \mathit{Heat}) \;\Rightarrow\; \mathit{Fire})$

g. $\mathit{Big} \lor \mathit{Dumb} \lor (\mathit{Big} \;\Rightarrow\; \mathit{Dumb})$

h. $(\mathit{Big} \land \mathit{Dumb}) \lor \neg\mathit{Dumb}$

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

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

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

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

a. Let $X_{i,j}$ be true iff square $[i, j]$ contains a mine. Write down the assertion that there are exactly two mines adjacent to [1,1] as a sentence involving some logical combination of $X_{i,j}$ propositions.

b. Generalize your assertion from (a) by explaining how to construct a CNF sentence asserting that k of n neighbors contain mines.

c. Explain precisely how an agent can use DPLL to prove that a given square does (or does not) contain a mine, ignoring the global constraint that there are exactly M mines in all.

d. Suppose that the global constraint is constructed via your method from part (b). How does the number of clauses depend on M and N? Suggest a way to modify DPLL so that the global constraint does not need to be represented explicitly.

e. Are any conclusions derived by the method in part (c) invalidated when the global constraint is taken into account?

f. Give examples of configurations of probe values that induce *long-range dependencies* such that the contents of a given unprobed square would give information about the contents of a far-distant square. [*Hint*: consider an $N \times 1$ board.]

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

a. Show that the clause $(\neg P_1 \vee \cdots \vee \neg P_m \vee Q)$ is logically equivalent to the implication sentence $(P_1 \wedge \cdots \wedge P_m) \Rightarrow Q$.

b. Show that every clause (regardless of the number of positive literals) can be written in the form $(P_1 \wedge \cdots \wedge P_m) \Rightarrow (Q_1 \vee \cdots \vee Q_n)$, where the Ps and Qs are proposition symbols. A knowledge base consisting of such sentences is in **implicative normal form** or **Kowalski form**.

IMPLICATIVE NORMAL FORM

c. Write down the full resolution rule for sentences in implicative normal form.

7.13 In this exercise, you will design more of the circuit-based wumpus agent.

a. Write an equation, similar to Equation (7.4), for the $Arrow$ proposition, which should be true when the agent still has an arrow. Draw the corresponding circuit.

b. Repeat part (a) for $FacingRight$, using Equation (7.5) as a model.

c. Create versions of Equations 7.7 and 7.8 for finding the wumpus, and draw the circuit.

7.14 Discuss what is meant by *optimal* behavior in the wumpus world. Show that our definition of the PL-WUMPUS-AGENT is not optimal, and suggest ways to improve it.

7.15 Extend PL-WUMPUS-AGENT so that it keeps keeps track of all relevant facts *within* the knowledge base.

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

7.17 Trace the behavior of DPLL on the knowledge base in Figure 7.15 when trying to prove Q, and compare this behavior with that of the forward chaining algorithm.