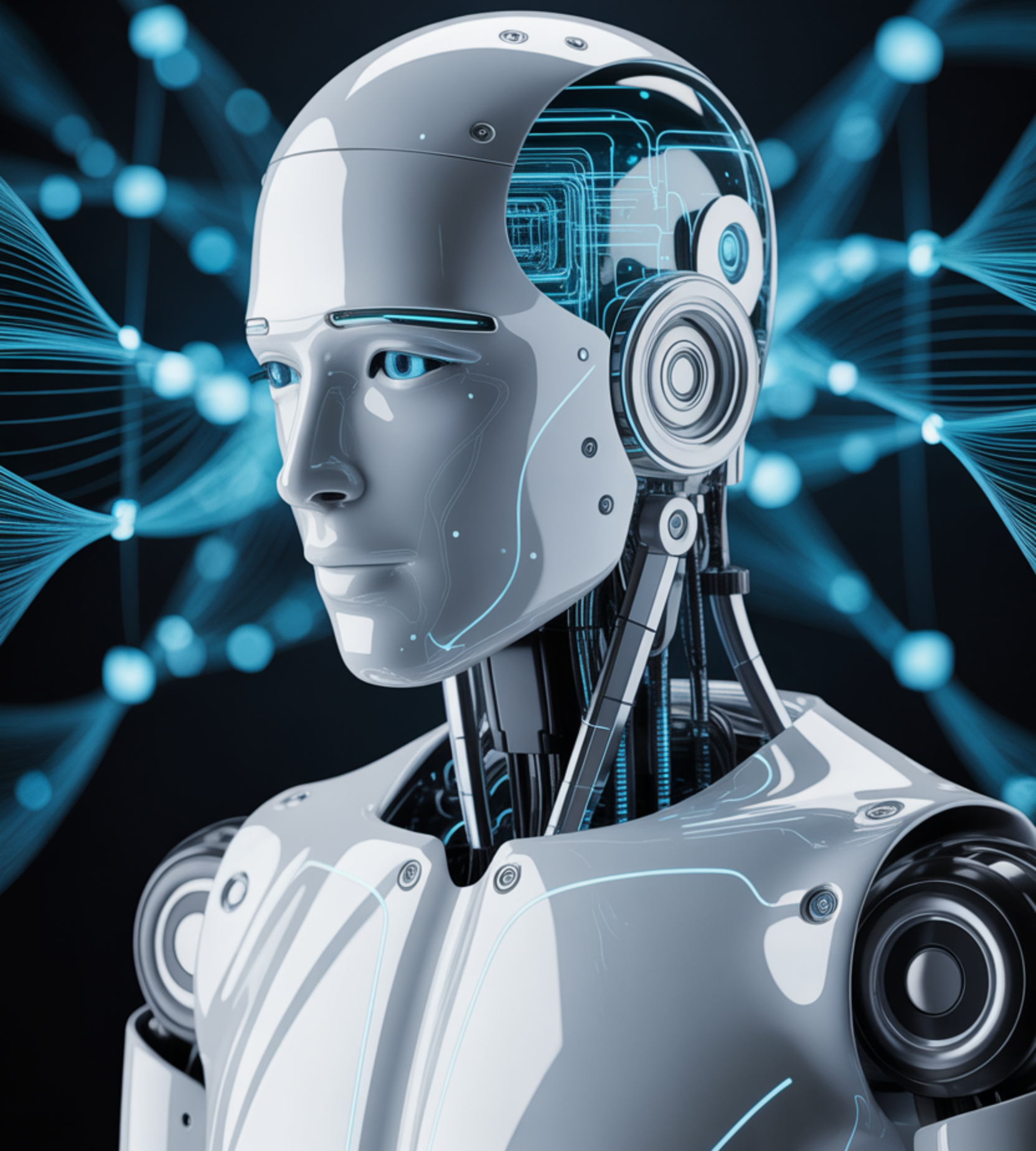


ARTIFICIAL INTELLIGENCE



INTRODUCTION

In which we try to explain why we consider artificial intelligence to be a subject most worthy of study, and in which we try to decide what exactly it is, this being a good thing to decide before embarking.

We call ourselves *Homo sapiens*—man the wise—because our **intelligence** is so important to us. For thousands of years, we have tried to understand *how we think and act*—that is, how our brain, a mere handful of matter, can perceive, understand, predict, and manipulate a world far larger and more complicated than itself. The field of **artificial intelligence**, or AI, is concerned with not just understanding but also *building* intelligent entities—machines that can compute how to act effectively and safely in a wide variety of novel situations.

Intelligence

Artificial intelligence

Surveys regularly rank AI as one of the most interesting and fastest-growing fields, and it is already generating over a trillion dollars a year in revenue. AI expert Kai-Fu Lee predicts that its impact will be “more than anything in the history of mankind.” Moreover, the intellectual frontiers of AI are wide open. Whereas a student of an older science such as physics might feel that the best ideas have already been discovered by Galileo, Newton, Curie, Einstein, and the rest, AI still has many openings for full-time masterminds.

AI currently encompasses a huge variety of subfields, ranging from the general (learning, reasoning, perception, and so on) to the specific, such as playing chess, proving mathematical theorems, writing poetry, driving a car, or diagnosing diseases. AI is relevant to any intellectual task; it is truly a universal field.

1.1 What Is AI?

We have claimed that AI is interesting, but we have not said what it *is*. Historically, researchers have pursued several different versions of AI. Some have defined intelligence in terms of fidelity to *human* performance, while others prefer an abstract, formal definition of intelligence called **rationality**—loosely speaking, doing the “right thing.” The subject matter itself also varies: some consider intelligence to be a property of internal *thought processes* and *reasoning*, while others focus on intelligent *behavior*, an external characterization.¹

Rationality

From these two dimensions—human vs. rational² and thought vs. behavior—there are four possible combinations, and there have been adherents and research programs for all

¹ In the public eye, there is sometimes confusion between the terms “artificial intelligence” and “machine learning.” Machine learning is a subfield of AI that studies the ability to improve performance based on experience. Some AI systems use machine learning methods to achieve competence, but some do not.

² We are not suggesting that humans are “irrational” in the dictionary sense of “deprived of normal mental clarity.” We are merely conceding that human decisions are not always mathematically perfect.

processing. The problem of understanding language turned out to be considerably more complex than it seemed in 1957. Understanding language requires an understanding of the subject matter and context, not just an understanding of the structure of sentences. This might seem obvious, but it was not widely appreciated until the 1960s. Much of the early work in **knowledge representation** (the study of how to put knowledge into a form that a computer can reason with) was tied to language and informed by research in linguistics, which was connected in turn to decades of work on the philosophical analysis of language.

1.3 The History of Artificial Intelligence

One quick way to summarize the milestones in AI history is to list the Turing Award winners: Marvin Minsky (1969) and John McCarthy (1971) for defining the foundations of the field based on representation and reasoning; Allen Newell and Herbert Simon (1975) for symbolic models of problem solving and human cognition; Ed Feigenbaum and Raj Reddy (1994) for developing expert systems that encode human knowledge to solve real-world problems; Judea Pearl (2011) for developing probabilistic reasoning techniques that deal with uncertainty in a principled manner; and finally Yoshua Bengio, Geoffrey Hinton, and Yann LeCun (2019) for making “deep learning” (multilayer neural networks) a critical part of modern computing. The rest of this section goes into more detail on each phase of AI history.

1.3.1 The inception of artificial intelligence (1943–1956)

The first work that is now generally recognized as AI was done by Warren McCulloch and Walter Pitts (1943). Inspired by the mathematical modeling work of Pitts’s advisor Nicolas Rashevsky (1936, 1938), they drew on three sources: knowledge of the basic physiology and function of neurons in the brain; a formal analysis of propositional logic due to Russell and Whitehead; and Turing’s theory of computation. They proposed a model of artificial neurons in which each neuron is characterized as being “on” or “off,” with a switch to “on” occurring in response to stimulation by a sufficient number of neighboring neurons. The state of a neuron was conceived of as “factually equivalent to a proposition which proposed its adequate stimulus.” They showed, for example, that any computable function could be computed by some network of connected neurons, and that all the logical connectives (AND, OR, NOT, etc.) could be implemented by simple network structures. McCulloch and Pitts also suggested that suitably defined networks could learn. Donald Hebb (1949) demonstrated a simple updating rule for modifying the connection strengths between neurons. His rule, now called **Hebbian learning**, remains an influential model to this day.

Hebbian learning

Two undergraduate students at Harvard, Marvin Minsky (1927–2016) and Dean Edmonds, built the first neural network computer in 1950. The SNARC, as it was called, used 3000 vacuum tubes and a surplus automatic pilot mechanism from a B-24 bomber to simulate a network of 40 neurons. Later, at Princeton, Minsky studied universal computation in neural networks. His Ph.D. committee was skeptical about whether this kind of work should be considered mathematics, but von Neumann reportedly said, “If it isn’t now, it will be someday.”

There were a number of other examples of early work that can be characterized as AI, including two checkers-playing programs developed independently in 1952 by Christopher Strachey at the University of Manchester and by Arthur Samuel at IBM. However, Alan Turing’s vision was the most influential. He gave lectures on the topic as early as 1947 at the London Mathematical Society and articulated a persuasive agenda in his 1950 article “Com-

1.5 Risks and Benefits of AI

Francis Bacon, a philosopher credited with creating the scientific method, noted in *The Wisdom of the Ancients* (1609) that the “mechanical arts are of ambiguous use, serving as well for hurt as for remedy.” As AI plays an increasingly important role in the economic, social, scientific, medical, financial, and military spheres, we would do well to consider the hurts and remedies—in modern parlance, the risks and benefits—that it can bring. The topics summarized here are covered in greater depth in Chapters 28 and 29.

To begin with the benefits: put simply, our entire civilization is the product of our human intelligence. If we have access to substantially greater machine intelligence, the ceiling on our ambitions is raised substantially. The potential for AI and robotics to free humanity from menial repetitive work and to dramatically increase the production of goods and services could presage an era of peace and plenty. The capacity to accelerate scientific research could result in cures for disease and solutions for climate change and resource shortages. As Demis Hassabis, CEO of Google DeepMind, has suggested: “First solve AI, then use AI to solve everything else.”

Long before we have an opportunity to “solve AI,” however, we will incur risks from the misuse of AI, inadvertent or otherwise. Some of these are already apparent, while others seem likely based on current trends:

- *Lethal autonomous weapons*: These are defined by the United Nations as weapons that can locate, select, and eliminate human targets without human intervention. A primary concern with such weapons is their *scalability*: the absence of a requirement for human supervision means that a small group can deploy an arbitrarily large number of weapons against human targets defined by any feasible recognition criterion. The technologies needed for autonomous weapons are similar to those needed for self-driving cars. Informal expert discussions on the potential risks of lethal autonomous weapons began at the UN in 2014, moving to the formal pre-treaty stage of a Group of Governmental Experts in 2017.
- *Surveillance and persuasion*: While it is expensive, tedious, and sometimes legally questionable for security personnel to monitor phone lines, video camera feeds, emails, and other messaging channels, AI (speech recognition, computer vision, and natural language understanding) can be used in a scalable fashion to perform mass surveillance of individuals and detect activities of interest. By tailoring information flows to individuals through social media, based on machine learning techniques, political behavior can be modified and controlled to some extent—a concern that became apparent in elections beginning in 2016.
- *Biased decision making*: Careless or deliberate misuse of machine learning algorithms for tasks such as evaluating parole and loan applications can result in decisions that are biased by race, gender, or other protected categories. Often, the data themselves reflect pervasive bias in society.
- *Impact on employment*: Concerns about machines eliminating jobs are centuries old. The story is never simple: machines do some of the tasks that humans might otherwise do, but they also make humans more productive and therefore more employable, and make companies more profitable and therefore able to pay higher wages. They may render some activities economically viable that would otherwise be impractical. Their

CHAPTER 2

INTELLIGENT AGENTS

In which we discuss the nature of agents, perfect or otherwise, the diversity of environments, and the resulting menagerie of agent types.

Chapter 1 identified the concept of **rational agents** as central to our approach to artificial intelligence. In this chapter, we make this notion more concrete. We will see that the concept of rationality can be applied to a wide variety of agents operating in any imaginable environment. Our plan in this book is to use this concept to develop a small set of design principles for building successful agents—systems that can reasonably be called **intelligent**.

We begin by examining agents, environments, and the coupling between them. The observation that some agents behave better than others leads naturally to the idea of a rational agent—one that behaves as well as possible. How well an agent can behave depends on the nature of the environment; some environments are more difficult than others. We give a crude categorization of environments and show how properties of an environment influence the design of suitable agents for that environment. We describe a number of basic “skeleton” agent designs, which we flesh out in the rest of the book.

2.1 Agents and Environments

Environment
Sensor
Actuator

An **agent** is anything that can be viewed as perceiving its **environment** through **sensors** and acting upon that environment through **actuators**. This simple idea is illustrated in Figure 2.1. A human agent has eyes, ears, and other organs for sensors and hands, legs, vocal tract, and so on for actuators. A robotic agent might have cameras and infrared range finders for sensors and various motors for actuators. A software agent receives file contents, network packets, and human input (keyboard/mouse/touchscreen/voice) as sensory inputs and acts on the environment by writing files, sending network packets, and displaying information or generating sounds. The environment could be everything—the entire universe! In practice it is just that part of the universe whose state we care about when designing this agent—the part that affects what the agent perceives and that is affected by the agent’s actions.

Percept
Percept sequence



Agent function

We use the term **percept** to refer to the content an agent’s sensors are perceiving. An agent’s **percept sequence** is the complete history of everything the agent has ever perceived. In general, *an agent’s choice of action at any given instant can depend on its built-in knowledge and on the entire percept sequence observed to date, but not on anything it hasn’t perceived*. By specifying the agent’s choice of action for every possible percept sequence, we have said more or less everything there is to say about the agent. Mathematically speaking, we say that an agent’s behavior is described by the **agent function** that maps any given percept sequence to an action.

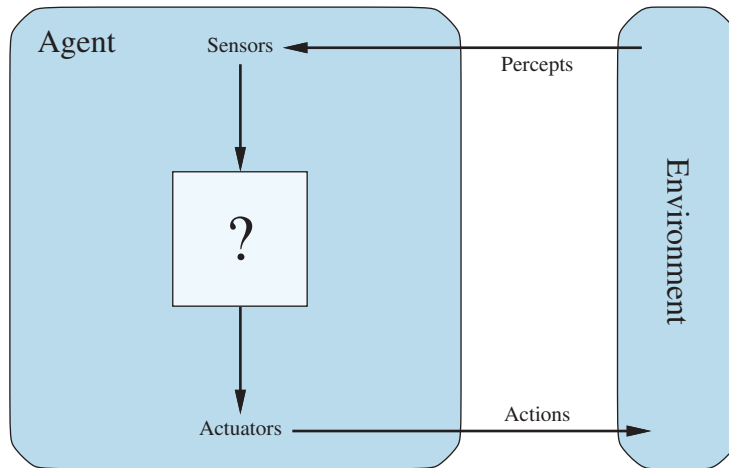


Figure 2.1 Agents interact with environments through sensors and actuators.

We can imagine *tabulating* the agent function that describes any given agent; for most agents, this would be a very large table—infinite, in fact, unless we place a bound on the length of percept sequences we want to consider. Given an agent to experiment with, we can, in principle, construct this table by trying out all possible percept sequences and recording which actions the agent does in response.¹ The table is, of course, an *external* characterization of the agent. *Internally*, the agent function for an artificial agent will be implemented by an **agent program**. It is important to keep these two ideas distinct. The agent function is an abstract mathematical description; the agent program is a concrete implementation, running within some physical system.

Agent program

To illustrate these ideas, we use a simple example—the vacuum-cleaner world, which consists of a robotic vacuum-cleaning agent in a world consisting of squares that can be either dirty or clean. Figure 2.2 shows a configuration with just two squares, *A* and *B*. The vacuum agent perceives which square it is in and whether there is dirt in the square. The agent starts in square *A*. The available actions are to move to the right, move to the left, suck up the dirt, or do nothing.² One very simple agent function is the following: if the current square is dirty, then suck; otherwise, move to the other square. A partial tabulation of this agent function is shown in Figure 2.3 and an agent program that implements it appears in Figure 2.8 on page 67.

Looking at Figure 2.3, we see that various vacuum-world agents can be defined simply by filling in the right-hand column in various ways. The obvious question, then, is this: *What is the right way to fill out the table?* In other words, what makes an agent good or bad, intelligent or stupid? We answer these questions in the next section. ◀

¹ If the agent uses some randomization to choose its actions, then we would have to try each sequence many times to identify the probability of each action. One might imagine that acting randomly is rather silly, but we show later in this chapter that it can be very intelligent.

² In a real robot, it would be unlikely to have an actions like “move right” and “move left.” Instead the actions would be “spin wheels forward” and “spin wheels backward.” We have chosen the actions to be easier to follow on the page, not for ease of implementation in an actual robot.

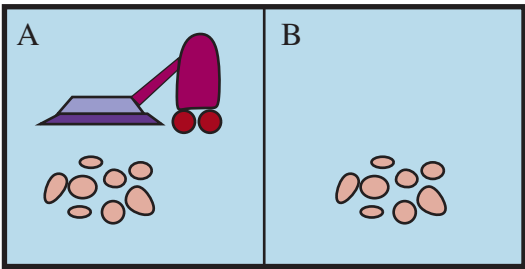


Figure 2.2 A vacuum-cleaner world with just two locations. Each location can be clean or dirty, and the agent can move left or right and can clean the square that it occupies. Different versions of the vacuum world allow for different rules about what the agent can perceive, whether its actions always succeed, and so on.

Percept sequence	Action
[A, Clean]	Right
[A, Dirty]	Suck
[B, Clean]	Left
[B, Dirty]	Suck
[A, Clean], [A, Clean]	Right
[A, Clean], [A, Dirty]	Suck
⋮	⋮
[A, Clean], [A, Clean], [A, Clean]	Right
[A, Clean], [A, Clean], [A, Dirty]	Suck
⋮	⋮

Figure 2.3 Partial tabulation of a simple agent function for the vacuum-cleaner world shown in Figure 2.2. The agent cleans the current square if it is dirty, otherwise it moves to the other square. Note that the table is of unbounded size unless there is a restriction on the length of possible percept sequences.

Before closing this section, we should emphasize that the notion of an agent is meant to be a tool for analyzing systems, not an absolute characterization that divides the world into agents and non-agents. One could view a hand-held calculator as an agent that chooses the action of displaying “4” when given the percept sequence “2 + 2 =,” but such an analysis would hardly aid our understanding of the calculator. In a sense, all areas of engineering can be seen as designing artifacts that interact with the world; AI operates at (what the authors consider to be) the most interesting end of the spectrum, where the artifacts have significant computational resources and the task environment requires nontrivial decision making.

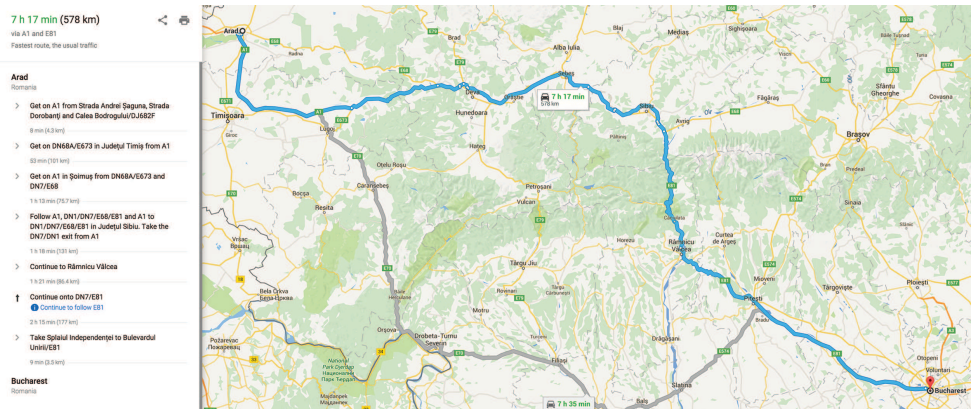


Figure 3.28 A Web service providing driving directions, computed by a search algorithm.

milliseconds—the number of nodes generated is reduced by a factor of 10,000 compared with the use of Manhattan distance. For 24-puzzles, a speedup of roughly a factor of a million can be obtained. Disjoint pattern databases work for sliding-tile puzzles because the problem can be divided up in such a way that each move affects only one subproblem—because only one tile is moved at a time.

3.6.4 Generating heuristics with landmarks

There are online services that host maps with tens of millions of vertices and find cost-optimal driving directions in milliseconds (Figure 3.28). How can they do that, when the best search algorithms we have considered so far are about a million times slower? There are many tricks, but the most important one is **precomputation** of some optimal path costs. Although the precomputation can be time-consuming, it need only be done once, and then can be amortized over billions of user search requests.

We could generate a perfect heuristic by precomputing and storing the cost of the optimal path between every pair of vertices. That would take $O(|V|^2)$ space, and $O(|E|^3)$ time—practical for graphs with 10 thousand vertices, but not 10 million.

A better approach is to choose a few (perhaps 10 or 20) **landmark points**¹⁶ from the vertices. Then for each landmark L and for each other vertex v in the graph, we compute and store $C^*(v, L)$, the exact cost of the optimal path from v to L . (We also need $C^*(L, v)$; on an undirected graph this is the same as $C^*(v, L)$; on a directed graph—e.g., with one-way streets—we need to compute this separately.) Given the stored C^* tables, we can easily create an efficient (although inadmissible) heuristic: the minimum, over all landmarks, of the cost of getting from the current node to the landmark, and then to the goal:

$$h_L(n) = \min_{L \in \text{Landmarks}} C^*(n, L) + C^*(L, \text{goal})$$

If the optimal path happens to go through a landmark, this heuristic will be exact; if not it is inadmissible—it overestimates the cost to the goal. In an A^* search, if you have exact heuristics, then once you reach a node that is on an optimal path, every node you expand

¹⁶ Landmark points are sometimes called “pivots” or “anchors.”

4.4.3 Solving partially observable problems

The preceding section showed how to derive the RESULTS function for a nondeterministic belief-state problem from an underlying physical problem, given the PERCEPT function. With this formulation, the AND–OR search algorithm of Figure 4.11 can be applied directly to derive a solution. Figure 4.16 shows part of the search tree for the local-sensing vacuum world, assuming an initial percept $[L, Dirty]$. The solution is the conditional plan

$[Suck, Right, \text{if } Rstate = \{6\} \text{ then } Suck \text{ else } []]$.

Notice that, because we supplied a belief-state problem to the AND–OR search algorithm, it returned a conditional plan that tests the belief state rather than the actual state. This is as it should be: in a partially observable environment the agent won't know the actual state.

As in the case of standard search algorithms applied to sensorless problems, the AND–OR search algorithm treats belief states as black boxes, just like any other states. One can improve on this by checking for previously generated belief states that are subsets or supersets of the current state, just as for sensorless problems. One can also derive incremental search algorithms, analogous to those described for sensorless problems, that provide substantial speedups over the black-box approach.

4.4.4 An agent for partially observable environments

An agent for partially observable environments formulates a problem, calls a search algorithm (such as AND–OR–SEARCH) to solve it, and executes the solution. There are two main differences between this agent and the one for fully observable deterministic environments. First, the solution will be a conditional plan rather than a sequence; to execute an if–then–else expression, the agent will need to test the condition and execute the appropriate branch of the conditional. Second, the agent will need to maintain its belief state as it performs actions and receives percepts. This process resembles the prediction–observation–update process in Equation (4.5) but is actually simpler because the percept is given by the environment rather than calculated by the agent. Given an initial belief state b , an action a , and a percept o , the new belief state is:

$$b' = \text{UPDATE}(\text{PREDICT}(b, a), o). \quad (4.6)$$

Consider a *kindergarten* vacuum world wherein agents sense only the state of their current square, and any square may become dirty at any time unless the agent is actively cleaning it at that moment.⁷ Figure 4.17 shows the belief state being maintained in this environment.

In partially observable environments—which include the vast majority of real-world environments—maintaining one's belief state is a core function of any intelligent system. This function goes under various names, including **monitoring**, **filtering**, and **state estimation**. Equation (4.6) is called a recursive state estimator because it computes the new belief state from the previous one rather than by examining the entire percept sequence. If the agent is not to “fall behind,” the computation has to happen as fast as percepts are coming in. As the environment becomes more complex, the agent will only have time to compute an approximate belief state, perhaps focusing on the implications of the percept for the aspects of the environment that are of current interest. Most work on this problem has been done for

Monitoring
Filtering
State estimation

⁷ The usual apologies to those who are unfamiliar with the effect of small children on the environment.

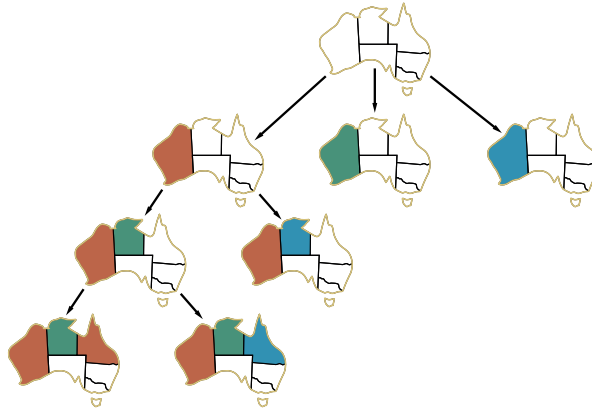


Figure 5.6 Part of the search tree for the map-coloring problem in Figure 5.1.

5.3.1 Variable and value ordering

The backtracking algorithm contains the line

```
var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment).
```

The simplest strategy for SELECT-UNASSIGNED-VARIABLE is static ordering: choose the variables in order, $\{X_1, X_2, \dots\}$. The next simplest is to choose randomly. Neither strategy is optimal. For example, after the assignments for $WA = \text{red}$ and $NT = \text{green}$ in Figure 5.6, there is only one possible value for SA , so it makes sense to assign $SA = \text{blue}$ next rather than assigning Q . In fact, after SA is assigned, the choices for Q , NSW , and V are all forced.

This intuitive idea—choosing the variable with the fewest “legal” values—is called the **minimum-remaining-values** (MRV) heuristic. It also has been called the “most constrained variable” or “fail-first” heuristic, the latter because it picks a variable that is most likely to cause a failure soon, thereby pruning the search tree. If some variable X has no legal values left, the MRV heuristic will select X and failure will be detected immediately—avoiding pointless searches through other variables. The MRV heuristic usually performs better than a random or static ordering, sometimes by orders of magnitude, although the results vary depending on the problem.

The MRV heuristic doesn’t help at all in choosing the first region to color in Australia, because initially every region has three legal colors. In this case, the **degree heuristic** comes in handy. It attempts to reduce the branching factor on future choices by selecting the variable that is involved in the largest number of constraints on other unassigned variables. In Figure 5.1, SA is the variable with highest degree, 5; the other variables have degree 2 or 3, except for T , which has degree 0. If we assign the SA first, we can then go around the five mainland regions in clockwise or counterclockwise order and assign each one a color that is different than SA and different than the previous region. The minimum-remaining-values heuristic is usually a more powerful guide, but the degree heuristic can be useful as a tie-breaker.

Once a variable has been selected, the algorithm must decide on the order in which to examine its values. The **least-constraining-value** heuristic is effective for this. It prefers the value that rules out the fewest choices for the neighboring variables in the constraint

Minimum-remaining-values

Degree heuristic

Least-constraining-value

CHAPTER 6

ADVERSARIAL SEARCH AND GAMES

In which we explore environments where other agents are plotting against us.

Adversarial search

In this chapter we cover **competitive environments**, in which two or more agents have conflicting goals, giving rise to **adversarial search** problems. Rather than deal with the chaos of real-world skirmishes, we will concentrate on games, such as chess, Go, and poker. For AI researchers, the simplified nature of these games is a plus: the state of a game is easy to represent, and agents are usually restricted to a small number of actions whose effects are defined by precise rules. Physical games, such as croquet and ice hockey, have more complicated descriptions, a larger range of possible actions, and rather imprecise rules defining the legality of actions. With the exception of robot soccer, these physical games have not attracted much interest in the AI community.

6.1 Game Theory

Economy

There are at least three stances we can take towards multi-agent environments. The first stance, appropriate when there are a very large number of agents, is to consider them in the aggregate as an **economy**, allowing us to do things like predict that increasing demand will cause prices to rise, without having to predict the action of any individual agent.

Second, we could consider adversarial agents as just a part of the environment—a part that makes the environment nondeterministic. But if we model the adversaries in the same way that, say, rain sometimes falls and sometimes doesn't, we miss the idea that our adversaries are actively trying to defeat us, whereas the rain supposedly has no such intention.

Pruning

The third stance is to explicitly model the adversarial agents with the techniques of adversarial game-tree search. That is what this chapter covers. We begin with a restricted class of games and define the optimal move and an algorithm for finding it: minimax search, a generalization of AND–OR search (from Figure 4.11). We show that **pruning** makes the search more efficient by ignoring portions of the search tree that make no difference to the optimal move. For nontrivial games, we will usually not have enough time to be sure of finding the optimal move (even with pruning); we will have to cut off the search at some point.

For each state where we choose to stop searching, we ask who is winning. To answer this question we have a choice: we can apply a heuristic **evaluation function** to estimate who is winning based on features of the state (Section 6.3), or we can average the outcomes of many fast simulations of the game from that state all the way to the end (Section 6.4).

Imperfect information

Section 6.5 discusses games that include an element of chance (through rolling dice or shuffling cards) and Section 6.6 covers games of **imperfect information** (such as poker and bridge, where not all cards are visible to all players).

arrive at bounds for the average without looking at every number. For example, say that all utility values are between -2 and $+2$; then the value of leaf nodes is bounded, and in turn we *can* place an upper bound on the value of a chance node without looking at all its children.

In games where the branching factor for chance nodes is high—consider a game like Yahtzee where you roll 5 dice on every turn—you may want to consider forward pruning that samples a smaller number of the possible chance branches. Or you may want to avoid using an evaluation function altogether, and opt for Monte Carlo tree search instead, where each playout includes random dice rolls.

6.6 Partially Observable Games

Bobby Fischer declared that “chess is war,” but chess lacks at least one major characteristic of real wars, namely, **partial observability**. In the “fog of war,” the whereabouts of enemy units is often unknown until revealed by direct contact. As a result, warfare includes the use of scouts and spies to gather information and the use of concealment and bluff to confuse the enemy.

Partially observable games share these characteristics and are thus qualitatively different from the games in the preceding sections. Video games such as StarCraft are particularly challenging, being partially observable, multi-agent, nondeterministic, dynamic, and unknown.

In *deterministic* partially observable games, uncertainty about the state of the board arises entirely from lack of access to the choices made by the opponent. This class includes children’s games such as Battleship (where each player’s ships are placed in locations hidden from the opponent) and Stratego (where piece locations are known but piece types are hidden). We will examine the game of **Kriegspiel**, a partially observable variant of chess in which pieces are completely invisible to the opponent. Other games also have partially observable versions: Phantom Go, Phantom tic-tac-toe, and Screen Shogi.

Kriegspiel

6.6.1 Kriegspiel: Partially observable chess

The rules of Kriegspiel are as follows: White and Black each see a board containing only their own pieces. A referee, who can see all the pieces, adjudicates the game and periodically makes announcements that are heard by both players. First, White proposes to the referee a move that would be legal if there were no black pieces. If the black pieces prevent the move, the referee announces “illegal,” and White keeps proposing moves until a legal one is found—learning more about the location of Black’s pieces in the process.

Once a legal move is proposed, the referee announces one or more of the following: “Capture on square X ” if there is a capture, and “Check by D ” if the black king is in check, where D is the direction of the check, and can be one of “Knight,” “Rank,” “File,” “Long diagonal,” or “Short diagonal.” If Black is checkmated or stalemated, the referee says so; otherwise, it is Black’s turn to move.

Kriegspiel may seem terrifyingly impossible, but humans manage it quite well and computer programs are beginning to catch up. It helps to recall the notion of a **belief state** as defined in Section 4.4 and illustrated in Figure 4.14—the set of all *logically possible* board states given the complete history of percepts to date. Initially, White’s belief state is a singleton because Black’s pieces haven’t moved yet. After White makes a move and Black responds, White’s belief state contains 20 positions, because Black has 20 replies to any

$(\alpha \wedge \beta) \equiv (\beta \wedge \alpha)$	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.

in propositional logic—the **SAT** problem—was the first problem proved to be NP-complete. Many problems in computer science are really satisfiability problems. For example, all the constraint satisfaction problems in Chapter 5 ask whether the constraints are satisfiable by some assignment.

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.

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

7.5.1 Inference and proofs

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

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

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

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

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

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

SAT

Reductio ad
absurdum
Refutation
Contradiction

Inference rules
Proof
Modus Ponens

And-Elimination

either the action at time t causes F to be true at $t + 1$, or F was already true at time t and the action at time t does not cause it to be false. An axiom of this form is called a **successor-state axiom** and has this form:

$$F^{t+1} \Leftrightarrow \text{ActionCauses}F^t \vee (F^t \wedge \neg \text{ActionCausesNot}F^t).$$

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

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

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

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

Exercise 7.SSAX asks you to write out axioms for the remaining wumpus world fluents.

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

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

At this point, we have $\text{ASK}(KB, L_{1,2}^6) = \text{true}$, so the agent knows where it is. Moreover, $\text{ASK}(KB, W_{1,3}) = \text{true}$ and $\text{ASK}(KB, P_{3,1}) = \text{true}$, so the agent has found the wumpus and one of the pits. The most important question for the agent is whether a square is OK to move into—that is, whether the square is free of a pit or live wumpus. It's convenient to add axioms for this, having the form

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

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

Solving the representational and inferential frame problems is a big step forward, but a pernicious problem remains: we need to confirm that *all* the necessary preconditions of an action hold for it to have its intended effect. We said that the *Forward* action moves the agent

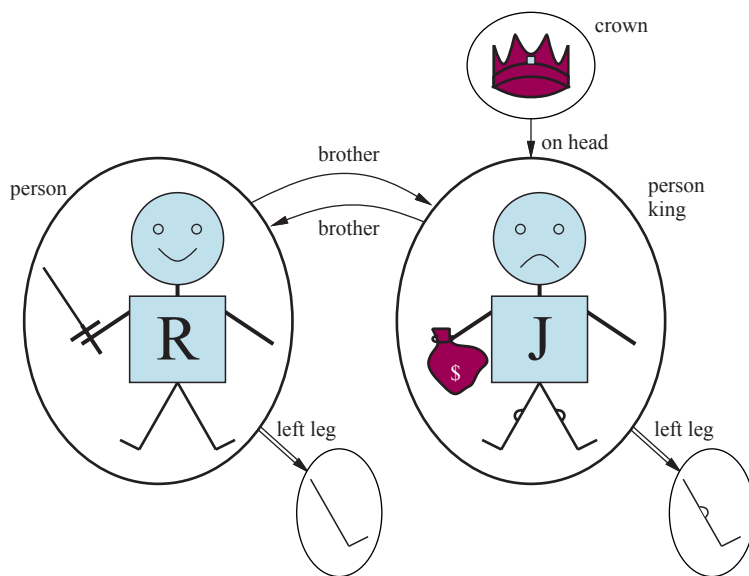


Figure 8.2 A model containing five objects, two binary relations (brother and on-head), three unary relations (person, king, and crown), and one unary function (left-leg).

includes the following mappings:

$$\begin{aligned} \langle \text{Richard the Lionheart} \rangle &\rightarrow \text{Richard's left leg} \\ \langle \text{King John} \rangle &\rightarrow \text{John's left leg} . \end{aligned} \tag{8.2}$$

Strictly speaking, models in first-order logic require **total functions**, that is, there must be a value for every input tuple. Thus the crown must have a left leg and so must each of the left legs. There is a technical solution to this awkward problem involving an additional “invisible” object that is the left leg of everything that has no left leg, including itself. Fortunately, as long as one makes no assertions about the left legs of things that have no left legs, these technicalities are of no import.

Total functions

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

8.2.2 Symbols and interpretations

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

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

Constant symbol

Predicate symbol

Function symbol

Arity

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

9.4 Backward Chaining

The second major family of logical inference algorithms uses **backward chaining** over definite clauses. These algorithms work backward from the goal, chaining through rules to find known facts that support the proof.

9.4.1 A backward-chaining algorithm

Figure 9.6 shows a backward-chaining algorithm for definite clauses. FOL-BC-ASK(*KB*, *goal*) will be proved if the knowledge base contains a rule of the form $lhs \Rightarrow goal$, where *lhs* (left-hand side) is a list of conjuncts. An atomic fact like *American(West)* is considered as a clause whose *lhs* is the empty list. Now a query that contains variables might be proved in multiple ways. For example, the query *Person(x)* could be proved with the substitution $\{x/John\}$ as well as with $\{x/Richard\}$. So we implement FOL-BC-ASK as a generator—a function that returns multiple times, each time giving one possible result (see Appendix B).

Backward chaining is a kind of AND/OR search—the OR part because the goal query can be proved by any rule in the knowledge base, and the AND part because all the conjuncts in the *lhs* of a clause must be proved. FOL-BC-OR works by fetching all clauses that might

```
function FOL-BC-ASK(KB, query) returns a generator of substitutions
  return FOL-BC-OR(KB, query, { })
```

```
function FOL-BC-OR(KB, goal,  $\theta$ ) returns a substitution
  for each rule in FETCH-RULES-FOR-GOAL(KB, goal) do
    ( $lhs \Rightarrow rhs$ )  $\leftarrow$  STANDARDIZE-VARIABLES(rule)
    for each  $\theta'$  in FOL-BC-AND(KB, lhs, UNIFY(rhs, goal,  $\theta$ )) do
      yield  $\theta'$ 
```

```
function FOL-BC-AND(KB, goals,  $\theta$ ) returns a substitution
  if  $\theta = failure$  then return
  else if LENGTH(goals) = 0 then yield  $\theta$ 
  else
     $first, rest \leftarrow$  FIRST(goals), REST(goals)
    for each  $\theta'$  in FOL-BC-OR(KB, SUBST( $\theta$ , first),  $\theta$ ) do
      for each  $\theta''$  in FOL-BC-AND(KB, rest,  $\theta'$ ) do
        yield  $\theta''$ 
```

Figure 9.6 A simple backward-chaining algorithm for first-order knowledge bases.

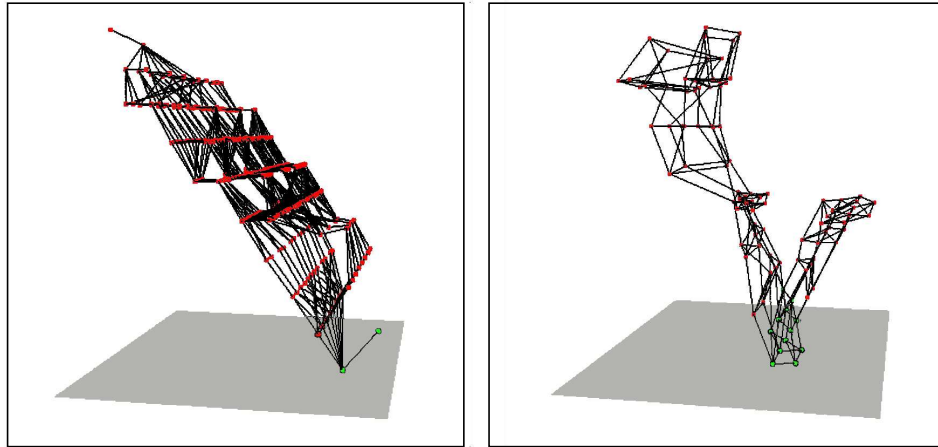


Figure 11.6 Two state spaces from planning problems with the ignore-delete-lists heuristic. The height above the bottom plane is the heuristic score of a state; states on the bottom plane are goals. There are no local minima, so search for the goal is straightforward. From Hoffmann (2005).

Ignore-delete-lists
heuristic

Another possibility is the **ignore-delete-lists heuristic**. Assume for a moment that all goals and preconditions contain only positive literals.² We want to create a relaxed version of the original problem that will be easier to solve, and where the length of the solution will serve as a good heuristic. We can do that by removing the delete lists from all actions (i.e., removing all negative literals from effects). That makes it possible to make monotonic progress towards the goal—no action will ever undo progress made by another action. It turns out it is still NP-hard to find the optimal solution to this relaxed problem, but an approximate solution can be found in polynomial time by hill climbing.

Figure 11.6 diagrams part of the state space for two planning problems using the ignore-delete-lists heuristic. The dots represent states and the edges actions, and the height of each dot above the bottom plane represents the heuristic value. States on the bottom plane are solutions. In both of these problems, there is a wide path to the goal. There are no dead ends, so no need for backtracking; a simple hill-climbing search will easily find a solution to these problems (although it may not be an optimal solution).

11.3.1 Domain-independent pruning

Factored representations make it obvious that many states are just variants of other states. For example, suppose we have a dozen blocks on a table, and the goal is to have block *A* on top of a three-block tower. The first step in a solution is to place some block *x* on top of block *y* (where *x*, *y*, and *A* are all different). After that, place *A* on top of *x* and we're done. There are 11 choices for *x*, and given *x*, 10 choices for *y*, and thus 110 states to consider. But all these states are symmetric: choosing one over another makes no difference, and thus a planner should only consider one of them. This is the process of **symmetry reduction**: we prune out

Symmetry reduction

² Many problems are written with this convention. For problems that aren't, replace every negative literal $\neg P$ in a goal or precondition with a new positive literal, P' , and modify the initial state and the action effects accordingly.

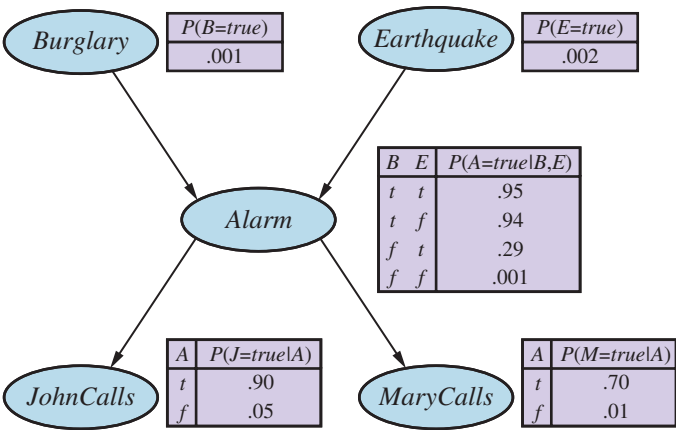


Figure 13.2 A typical Bayesian network, showing both the topology and the conditional probability tables (CPTs). In the CPTs, the letters B , E , A , J , and M stand for *Burglary*, *Earthquake*, *Alarm*, *JohnCalls*, and *MaryCalls*, respectively.

Conditioning case

tion 13.2.) Each row in a CPT contains the conditional probability of each node value for a **conditioning case**. A conditioning case is just a possible combination of values for the parent nodes—a miniature possible world, if you like. Each row must sum to 1, because the entries represent an exhaustive set of cases for the variable. For Boolean variables, once you know that the probability of a true value is p , the probability of false must be $1 - p$, so we often omit the second number, as in Figure 13.2. In general, a table for a Boolean variable with k Boolean parents contains 2^k independently specifiable probabilities. A node with no parents has only one row, representing the prior probabilities of each possible value of the variable.

Notice that the network does not have nodes corresponding to Mary’s currently listening to loud music or to the telephone ringing and confusing John. These factors are summarized in the uncertainty associated with the links from *Alarm* to *JohnCalls* and *MaryCalls*. This shows both laziness and ignorance in operation, as explained on page 404: it would be a lot of work to find out why those factors would be more or less likely in any particular case, and we have no reasonable way to obtain the relevant information anyway.

The probabilities actually summarize a *potentially infinite* set of circumstances in which the alarm might fail to go off (high humidity, power failure, dead battery, cut wires, a dead mouse stuck inside the bell, etc.) or John or Mary might fail to call and report it (out to lunch, on vacation, temporarily deaf, passing helicopter, etc.). In this way, a small agent can cope with a very large world, at least approximately.

13.2 The Semantics of Bayesian Networks

The *syntax* of a Bayes net consists of a directed acyclic graph with some local probability information attached to each node. The *semantics* defines how the syntax corresponds to a joint distribution over the variables of the network.

Assume that the Bayes net contains n variables, $X_1, \dots, X_n$. A generic entry in the joint distribution is then $P(X_1 = x_1 \wedge \dots \wedge X_n = x_n)$, or $P(x_1, \dots, x_n)$ for short. The semantics of

Compactness and node ordering

As well as being a complete and nonredundant representation of the domain, a Bayes net can often be far more *compact* than the full joint distribution. This property is what makes it feasible to handle domains with many variables. The compactness of Bayesian networks is an example of a general property of **locally structured** (also called **sparse**) systems. In a locally structured system, each subcomponent interacts directly with only a bounded number of other components, regardless of the total number of components. Local structure is usually associated with linear rather than exponential growth in complexity.

Locally structured

Sparse

In the case of Bayes nets, it is reasonable to suppose that in most domains each random variable is directly influenced by at most k others, for some constant k . If we assume n Boolean variables for simplicity, then the amount of information needed to specify each conditional probability table will be at most 2^k numbers, and the complete network can be specified by $2^k \cdot n$ numbers. In contrast, the joint distribution contains 2^n numbers. To make this concrete, suppose we have $n = 30$ nodes, each with five parents ($k = 5$). Then the Bayesian network requires 960 numbers, but the full joint distribution requires over a billion.

Specifying the conditional probability tables for a fully connected network, in which each variable has all of its predecessors as parents, requires the same amount of information as specifying the joint distribution in tabular form. For this reason, we often leave out links even though a slight dependency exists, because the slight gain in accuracy is not worth the the additional complexity in the network. For example, one might object to our burglary network on the grounds that if there is a large earthquake, then John and Mary would not call even if they heard the alarm, because they assume that the earthquake is the cause. Whether to add the link from *Earthquake* to *JohnCalls* and *MaryCalls* (and thus enlarge the tables) depends on the importance of getting more accurate probabilities compared with the cost of specifying the extra information.

Even in a locally structured domain, we will get a compact Bayes net only if we choose the node ordering well. What happens if we happen to choose the wrong order? Consider the burglary example again. Suppose we decide to add the nodes in the order *MaryCalls*, *JohnCalls*, *Alarm*, *Burglary*, *Earthquake*. We then get the somewhat more complicated network shown in Figure 13.3(a). The process goes as follows:

- Adding *MaryCalls*: No parents.
- Adding *JohnCalls*: If Mary calls, that probably means the alarm has gone off, which makes it more likely that John calls. Therefore, *JohnCalls* needs *MaryCalls* as a parent.
- Adding *Alarm*: Clearly, if both call, it is more likely that the alarm has gone off than if just one or neither calls, so we need both *MaryCalls* and *JohnCalls* as parents.
- Adding *Burglary*: If we know the alarm state, then the call from John or Mary might give us information about our phone ringing or Mary's music, but not about burglary:

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

Hence we need just *Alarm* as parent.

- Adding *Earthquake*: If the alarm is on, it is more likely that there has been an earthquake. (The alarm is an earthquake detector of sorts.) But if we know that there has been a burglary, then that explains the alarm, and the probability of an earthquake would be only slightly above normal. Hence, we need both *Alarm* and *Burglary* as parents.

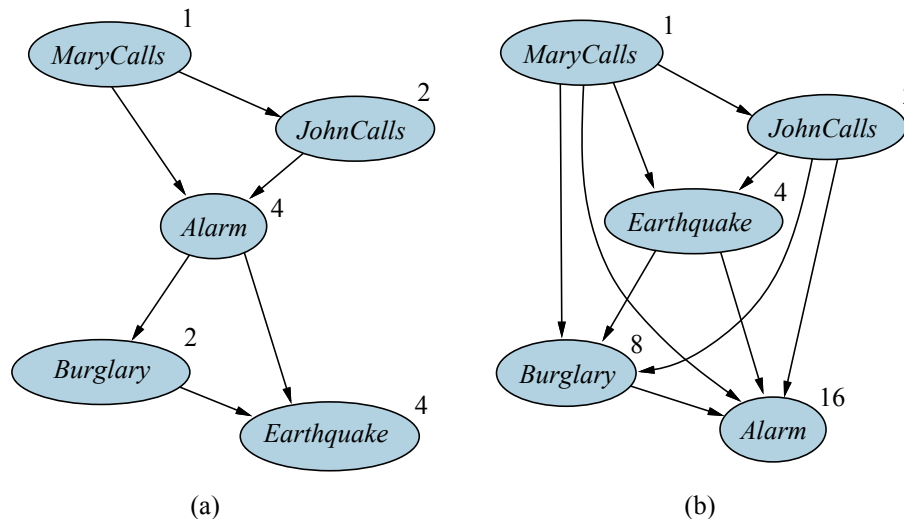


Figure 13.3 Network structure and number of parameters depends on order of introduction. (a) The structure obtained with ordering M, J, A, B, E . (b) The structure obtained with M, J, E, B, A . Each node is annotated with the number of parameters required; 13 in all for (a) and 31 for (b). In Figure 13.2, only 10 parameters were required.

The resulting network has two more links than the original network in Figure 13.2 and requires 13 conditional probabilities rather than 10. What's worse, some of the links represent tenuous relationships that require difficult and unnatural probability judgments, such as assessing the probability of *Earthquake*, given *Burglary* and *Alarm*. This phenomenon is quite general and is related to the distinction between **causal** and **diagnostic** models introduced in Section 12.5.1 (see also Exercise 13.WUMD). *If we stick to a causal model, we end up having to specify fewer numbers, and the numbers will often be easier to come up with.* For example, in the domain of medicine, it has been shown by Tversky and Kahneman (1982) that expert physicians prefer to give probability judgments for causal rules rather than for diagnostic ones. Section 13.5 explores the idea of causal models in more depth.

Figure 13.3(b) shows a very bad node ordering: *MaryCalls*, *JohnCalls*, *Earthquake*, *Burglary*, *Alarm*. This network requires 31 distinct probabilities to be specified—exactly the same number as the full joint distribution. It is important to realize, however, that any of the three networks can represent *exactly the same joint distribution*. The two versions in Figure 13.3 simply fail to represent all the conditional independence relationships and hence end up specifying a lot of unnecessary numbers instead.

13.2.1 Conditional independence relations in Bayesian networks

From the semantics of Bayes nets as defined in Equation (13.2), we can derive a number of conditional independence properties. We have already seen the property that a variable is conditionally independent of its other predecessors, given its parents. It is also possible to prove the more general “non-descendants” property that:

Each variable is conditionally independent of its non-descendants, given its parents.