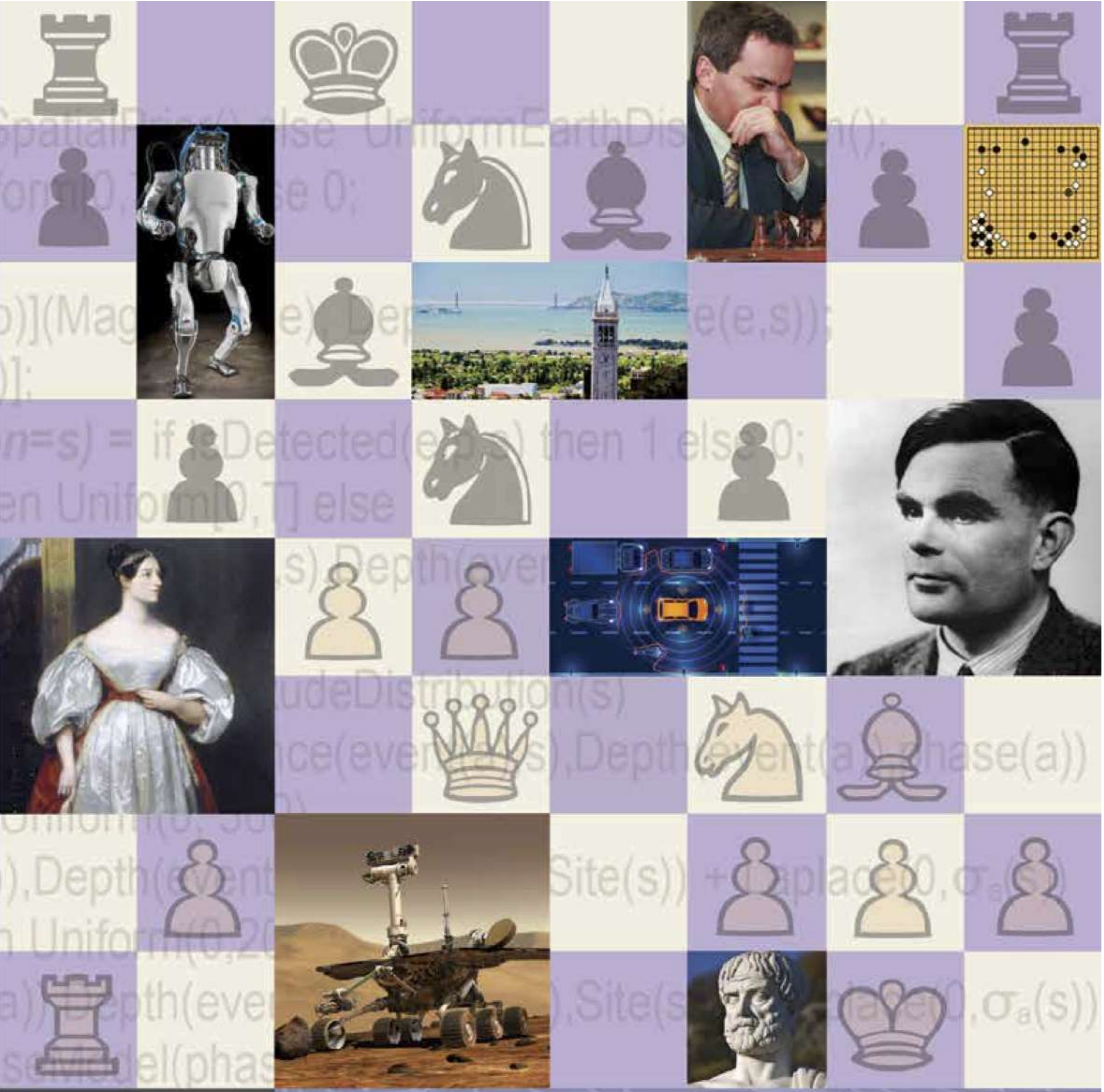


Stuart
Russell
Peter
Norvig

Artificial Intelligence
A Modern Approach

Fourth Edition

For Loy, Gordon, Lucy, George, and Isaac — S.J.R.

For Kris, Isabella, and Juliet — P.N.

Preface

Artificial Intelligence (AI) is a big field, and this is a big book. We have tried to explore the full breadth of the field, which encompasses logic, probability, and continuous mathematics; perception, reasoning, learning, and action; fairness, trust, social good, and safety; and applications that range from microelectronic devices to robotic planetary explorers to online services with billions of users.

The subtitle of this book is "A Modern Approach." That means we have chosen to tell the story from a current perspective. We synthesize what is now known into a common framework, recasting early work using the ideas and terminology that are prevalent today. We apologize to those whose subfields are, as a result, less recognizable.

New to this edition

This edition reflects the changes in AI since the last edition in 2010:

- We focus more on machine learning rather than hand-crafted knowledge engineering, due to the increased availability of data, computing resources, and new algorithms.
- Deep learning, probabilistic programming, and multiagent systems receive expanded coverage, each with their own chapter.
- The coverage of natural language understanding, robotics, and computer vision has been revised to reflect the impact of deep learning.
- The robotics chapter now includes robots that interact with humans and the application of reinforcement learning to robotics.
- Previously we defined the goal of AI as creating systems that try to maximize expected utility, where the specific utility information—the objective—is supplied by the human designers of the system. Now we no longer assume that the objective is fixed and known by the AI system; instead, the system may be uncertain about the true objectives of the humans on whose behalf it operates. It must learn what to maximize and must function appropriately even while uncertain about the objective.
- We increase coverage of the impact of AI on society, including the vital issues of ethics, fairness, trust, and safety.
- We have moved the exercises from the end of each chapter to an online site. This allows us to continuously add to, update, and improve the exercises, to meet the needs of instructors and to reflect advances in the field and in AI-related software tools.
- Overall, about 25% of the material in the book is brand new. The remaining 75% has been largely rewritten to present a more unified picture of the field. 22% of the citations in this edition are to works published after 2010.

Overview of the book

The main unifying theme is the idea of an **intelligent agent**. We define AI as the study of agents that receive percepts from the environment and perform actions. Each such agent implements a function that maps percept sequences to actions, and we cover different ways to represent these functions, such as reactive agents, real-time planners, decision-theoretic systems, and deep learning systems. We emphasize learning both as a construction method for competent systems and as a way of extending the reach of the designer into unknown environments. We treat robotics and vision not as independently defined problems, but as occurring in the service of achieving goals. We stress the importance of the task environment in determining the appropriate agent design.

Our primary aim is to convey the *ideas* that have emerged over the past seventy years of AI research and the past two millennia of related work. We have tried to avoid excessive formality in the presentation of these ideas, while retaining precision. We have included mathematical formulas and pseudocode algorithms to make the key ideas concrete; mathematical concepts and notation are described in Appendix A and our pseudocode is described in Appendix B.

This book is primarily intended for use in an undergraduate course or course sequence. The book has 28 chapters, each requiring about a week's worth of lectures, so working through the whole book requires a two-semester sequence. A one-semester course can use selected chapters to suit the interests of the instructor and students. The book can also be used in a graduate-level course (perhaps with the addition of some of the primary sources suggested in the bibliographical notes), or for self-study or as a reference.

▶ Throughout the book, *important points* are marked with a triangle icon in the margin. Wherever a new **term** is defined, it is also noted in the margin. Subsequent significant uses of the **term** are in bold, but not in the margin. We have included a comprehensive index and an extensive bibliography.

Term

The only prerequisite is familiarity with basic concepts of computer science (algorithms, data structures, complexity) at a sophomore level. Freshman calculus and linear algebra are useful for some of the topics.

Online resources

Online resources are available through `pearsonhighered.com/cs-resources` or at the book's Web site, `aima.cs.berkeley.edu`. There you will find:

- Exercises, programming projects, and research projects. These are no longer at the end of each chapter; they are online only. Within the book, we refer to an online exercise with a name like "Exercise 6.NARY." Instructions on the Web site allow you to find exercises by name or by topic.
- Implementations of the algorithms in the book in Python, Java, and other programming languages (currently hosted at `github.com/aimacode`).
- A list of over 1400 schools that have used the book, many with links to online course materials and syllabi.
- Supplementary material and links for students and instructors.
- Instructions on how to report errors in the book, in the likely event that some exist.

Book cover

The cover depicts the final position from the decisive game 6 of the 1997 chess match in which the program Deep Blue defeated Garry Kasparov (playing Black), making this the first time a computer had beaten a world champion in a chess match. Kasparov is shown at the top. To his right is a pivotal position from the second game of the historic Go match between former world champion Lee Sedol and DeepMind's ALPHAGO program. Move 37 by ALPHAGO violated centuries of Go orthodoxy and was immediately seen by human experts as an embarrassing mistake, but it turned out to be a winning move. At top left is an Atlas humanoid robot built by Boston Dynamics. A depiction of a self-driving car sensing its environment appears between Ada Lovelace, the world's first computer programmer, and Alan Turing, whose fundamental work defined artificial intelligence. At the bottom of the chess board are a Mars Exploration Rover robot and a statue of Aristotle, who pioneered the study of logic; his planning algorithm from *De Motu Animalium* appears behind the authors' names. Behind the chess board is a probabilistic programming model used by the UN Comprehensive Nuclear-Test-Ban Treaty Organization for detecting nuclear explosions from seismic signals.

Acknowledgments

It takes a global village to make a book. Over 600 people read parts of the book and made suggestions for improvement. The complete list is at `aima.cs.berkeley.edu/ack.html`; we are grateful to all of them. We have space here to mention only a few especially important contributors. First the contributing writers:

- Judea Pearl (Section 13.5, Causal Networks);
- Vikash Mansinghka (Section 15.3, Programs as Probability Models);
- Michael Wooldridge (Chapter 18, Multiagent Decision Making);
- Ian Goodfellow (Chapter 21, Deep Learning);
- Jacob Devlin and Mei-Wing Chang (Chapter 24, Deep Learning for Natural Language);
- Jitendra Malik and David Forsyth (Chapter 25, Computer Vision);
- Anca Dragan (Chapter 26, Robotics).

Then some key roles:

- Cynthia Yeung and Malika Cantor (project management);
- Julie Sussman and Tom Galloway (copyediting and writing suggestions);
- Omari Stephens (illustrations);
- Tracy Johnson (editor);
- Erin Ault and Rose Kernan (cover and color conversion);
- Nalin Chhibber, Sam Goto, Raymond de Lacaze, Ravi Mohan, Ciaran O'Reilly, Amit Patel, Dragomir Radiv, and Samagra Sharma (online code development and mentoring);
- Google Summer of Code students (online code development).

Stuart would like to thank his wife, Loy Sheflott, for her endless patience and boundless wisdom. He hopes that Gordon, Lucy, George, and Isaac will soon be reading this book after they have forgiven him for working so long on it. RUGS (Russell's Unusual Group of Students) have been unusually helpful, as always.

Peter would like to thank his parents (Torsten and Gerda) for getting him started, and his wife (Kris), children (Bella and Juliet), colleagues, boss, and friends for encouraging and tolerating him through the long hours of writing and rewriting.

About the Authors

Stuart Russell was born in 1962 in Portsmouth, England. He received his B.A. with first-class honours in physics from Oxford University in 1982, and his Ph.D. in computer science from Stanford in 1986. He then joined the faculty of the University of California at Berkeley, where he is a professor and former chair of computer science, director of the Center for Human-Compatible AI, and holder of the Smith–Zadeh Chair in Engineering. In 1990, he received the Presidential Young Investigator Award of the National Science Foundation, and in 1995 he was cowinner of the Computers and Thought Award. He is a Fellow of the American Association for Artificial Intelligence, the Association for Computing Machinery, and the American Association for the Advancement of Science, an Honorary Fellow of Wadham College, Oxford, and an Andrew Carnegie Fellow. He held the Chaire Blaise Pascal in Paris from 2012 to 2014. He has published over 300 papers on a wide range of topics in artificial intelligence. His other books include *The Use of Knowledge in Analogy and Induction*, *Do the Right Thing: Studies in Limited Rationality* (with Eric Wefald), and *Human Compatible: Artificial Intelligence and the Problem of Control*.

Peter Norvig is currently a Director of Research at Google, Inc., and was previously the director responsible for the core Web search algorithms. He co-taught an online AI class that signed up 160,000 students, helping to kick off the current round of massive open online classes. He was head of the Computational Sciences Division at NASA Ames Research Center, overseeing research and development in artificial intelligence and robotics. He received a B.S. in applied mathematics from Brown University and a Ph.D. in computer science from Berkeley. He has been a professor at the University of Southern California and a faculty member at Berkeley and Stanford. He is a Fellow of the American Association for Artificial Intelligence, the Association for Computing Machinery, the American Academy of Arts and Sciences, and the California Academy of Science. His other books are *Paradigms of AI Programming: Case Studies in Common Lisp*, *Verbmobil: A Translation System for Face-to-Face Dialog*, and *Intelligent Help Systems for UNIX*.

The two authors shared the inaugural AAAI/EAAI Outstanding Educator award in 2016.

Contents

CHAPTER 1

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.

Intelligence

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.

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. rational[2] 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.

four. The methods used are necessarily different: the pursuit of human-like intelligence must be in part an empirical science related to psychology, involving observations and hypotheses about actual human behavior and thought processes; a rationalist approach, on the other hand, involves a combination of mathematics and engineering, and connects to statistics, control theory, and economics. The various groups have both disparaged and helped each other. Let us look at the four approaches in more detail.

1.1.1 Acting humanly: The Turing test approach

Turing test

The **Turing test**, proposed by Alan Turing (1950), was designed as a thought experiment that would sidestep the philosophical vagueness of the question "Can a machine think?" A computer passes the test if a human interrogator, after posing some written questions, cannot tell whether the written responses come from a person or from a computer. Chapter 27 discusses the details of the test and whether a computer would really be intelligent if it passed. For now, we note that programming a computer to pass a rigorously applied test provides plenty to work on. The computer would need the following capabilities:

Natural language processing
Knowledge representation
Automated reasoning
Machine learning

- **natural language processing** to communicate successfully in a human language;
- **knowledge representation** to store what it knows or hears;
- **automated reasoning** to answer questions and to draw new conclusions;
- **machine learning** to adapt to new circumstances and to detect and extrapolate patterns.

Total Turing test

Turing viewed the *physical* simulation of a person as unnecessary to demonstrate intelligence. However, other researchers have proposed a **total Turing test**, which requires interaction with objects and people in the real world. To pass the total Turing test, a robot will need

Computer vision
Robotics

- **computer vision** and speech recognition to perceive the world;
- **robotics** to manipulate objects and move about.

These six disciplines compose most of AI. Yet AI researchers have devoted little effort to passing the Turing test, believing that it is more important to study the underlying principles of intelligence. The quest for "artificial flight" succeeded when engineers and inventors stopped imitating birds and started using wind tunnels and learning about aerodynamics. Aeronautical engineering texts do not define the goal of their field as making "machines that fly so exactly like pigeons that they can fool even other pigeons."

1.1.2 Thinking humanly: The cognitive modeling approach

To say that a program thinks like a human, we must know how humans think. We can learn about human thought in three ways:

Introspection
Psychological experiment
Brain imaging

- **introspection**—trying to catch our own thoughts as they go by;
- **psychological experiments**—observing a person in action;
- **brain imaging**—observing the brain in action.

Once we have a sufficiently precise theory of the mind, it becomes possible to express the theory as a computer program. If the program's input–output behavior matches corresponding human behavior, that is evidence that some of the program's mechanisms could also be operating in humans.

For example, Allen Newell and Herbert Simon, who developed GPS, the "General Problem Solver" (Newell and Simon, 1961), were not content merely to have their program solve problems correctly. They were more concerned with comparing the sequence and timing of its reasoning steps to those of human subjects solving the same problems. The interdisciplinary field of **cognitive science** brings together computer models from AI and experimental techniques from psychology to construct precise and testable theories of the human mind.

Cognitive science

Cognitive science is a fascinating field in itself, worthy of several textbooks and at least one encyclopedia (Wilson and Keil, 1999). We will occasionally comment on similarities or differences between AI techniques and human cognition. Real cognitive science, however, is necessarily based on experimental investigation of actual humans or animals. We will leave that for other books, as we assume the reader has only a computer for experimentation.

In the early days of AI there was often confusion between the approaches. An author would argue that an algorithm performs well on a task and that it is *therefore* a good model of human performance, or vice versa. Modern authors separate the two kinds of claims; this distinction has allowed both AI and cognitive science to develop more rapidly. The two fields fertilize each other, most notably in computer vision, which incorporates neurophysiological evidence into computational models. Recently, the combination of neuroimaging methods combined with machine learning techniques for analyzing such data has led to the beginnings of a capability to "read minds"—that is, to ascertain the semantic content of a person's inner thoughts. This capability could, in turn, shed further light on how human cognition works.

1.1.3 Thinking rationally: The "laws of thought" approach

The Greek philosopher Aristotle was one of the first to attempt to codify "right thinking"—that is, irrefutable reasoning processes. His **syllogisms** provided patterns for argument structures that always yielded correct conclusions when given correct premises. The canonical example starts with *Socrates is a man* and *all men are mortal* and concludes that *Socrates is mortal*. (This example is probably due to Sextus Empiricus rather than Aristotle.) These laws of thought were supposed to govern the operation of the mind; their study initiated the field called **logic**.

Syllogism

Logicians in the 19th century developed a precise notation for statements about objects in the world and the relations among them. (Contrast this with ordinary arithmetic notation, which provides only for statements about *numbers*.) By 1965, programs could, in principle, solve *any* solvable problem described in logical notation. The so-called **logicist** tradition within artificial intelligence hopes to build on such programs to create intelligent systems.

Logicist

Logic as conventionally understood requires knowledge of the world that is *certain*—a condition that, in reality, is seldom achieved. We simply don't know the rules of, say, politics or warfare in the same way that we know the rules of chess or arithmetic. The theory of **probability** fills this gap, allowing rigorous reasoning with uncertain information. In principle, it allows the construction of a comprehensive model of rational thought, leading from raw perceptual information to an understanding of how the world works to predictions about the future. What it does not do, is generate intelligent *behavior*. For that, we need a theory of rational action. Rational thought, by itself, is not enough.

Probability

1.1.4 Acting rationally: The rational agent approach

An **agent** is just something that acts (*agent* comes from the Latin *agere*, to do). Of course, all computer programs do something, but computer agents are expected to do more: operate autonomously, perceive their environment, persist over a prolonged time period, adapt to

Agent

Rational agent

change, and create and pursue goals. A **rational agent** is one that acts so as to achieve the best outcome or, when there is uncertainty, the best expected outcome.

In the "laws of thought" approach to AI, the emphasis was on correct inferences. Making correct inferences is sometimes *part* of being a rational agent, because one way to act rationally is to deduce that a given action is best and then to act on that conclusion. On the other hand, there are ways of acting rationally that cannot be said to involve inference. For example, recoiling from a hot stove is a reflex action that is usually more successful than a slower action taken after careful deliberation.

All the skills needed for the Turing test also allow an agent to act rationally. Knowledge representation and reasoning enable agents to reach good decisions. We need to be able to generate comprehensible sentences in natural language to get by in a complex society. We need learning not only for erudition, but also because it improves our ability to generate effective behavior, especially in circumstances that are new.

The rational-agent approach to AI has two advantages over the other approaches. First, it is more general than the "laws of thought" approach because correct inference is just one of several possible mechanisms for achieving rationality. Second, it is more amenable to scientific development. The standard of rationality is mathematically well defined and completely general. We can often work back from this specification to derive agent designs that provably achieve it—something that is largely impossible if the goal is to imitate human behavior or thought processes.

Do the right thing

Standard model

For these reasons, the rational-agent approach to AI has prevailed throughout most of the field's history. In the early decades, rational agents were built on logical foundations and formed definite plans to achieve specific goals. Later, methods based on probability theory and machine learning allowed the creation of agents that could make decisions under uncertainty to attain the best expected outcome. In a nutshell, *AI has focused on the study and construction of agents that* ***do the right thing***. What counts as the right thing is defined by the objective that we provide to the agent. This general paradigm is so pervasive that we might call it the **standard model**. It prevails not only in AI, but also in control theory, where a controller minimizes a cost function; in operations research, where a policy maximizes a sum of rewards; in statistics, where a decision rule minimizes a loss function; and in economics, where a decision maker maximizes utility or some measure of social welfare.

Limited rationality

We need to make one important refinement to the standard model to account for the fact that perfect rationality—always taking the exactly optimal action—is not feasible in complex environments. The computational demands are just too high. Chapters 5 and 17 deal with the issue of **limited rationality**—acting appropriately when there is not enough time to do all the computations one might like. However, perfect rationality often remains a good starting point for theoretical analysis.

1.1.5 Beneficial machines

The standard model has been a useful guide for AI research since its inception, but it is probably not the right model in the long run. The reason is that the standard model assumes that we will supply a fully specified objective to the machine.

For an artificially defined task such as chess or shortest-path computation, the task comes with an objective built in—so the standard model is applicable. As we move into the real world, however, it becomes more and more difficult to specify the objective completely and correctly. For example, in designing a self-driving car, one might think that the objective is to reach the destination safely. But driving along any road incurs a risk of injury due to other errant drivers, equipment failure, and so on; thus, a strict goal of safety requires staying in the garage. There is a tradeoff between making progress towards the destination and incurring a risk of injury. How should this tradeoff be made? Furthermore, to what extent can we allow the car to take actions that would annoy other drivers? How much should the car moderate its acceleration, steering, and braking to avoid shaking up the passenger? These kinds of questions are difficult to answer a priori. They are particularly problematic in the general area of human–robot interaction, of which the self-driving car is one example.

The problem of achieving agreement between our true preferences and the objective we put into the machine is called the **value alignment problem**: the values or objectives put into the machine must be aligned with those of the human. If we are developing an AI system in the lab or in a simulator—as has been the case for most of the field's history—there is an easy fix for an incorrectly specified objective: reset the system, fix the objective, and try again. As the field progresses towards increasingly capable intelligent systems that are deployed in the real world, this approach is no longer viable. A system deployed with an incorrect objective will have negative consequences. Moreover, the more intelligent the system, the more negative the consequences.

Value alignment problem

Returning to the apparently unproblematic example of chess, consider what happens if the machine is intelligent enough to reason and act beyond the confines of the chessboard. In that case, it might attempt to increase its chances of winning by such ruses as hypnotizing or blackmailing its opponent or bribing the audience to make rustling noises during its opponent's thinking time.[3] It might also attempt to hijack additional computing power for itself. *These behaviors are not "unintelligent" or "insane"; they are a logical consequence of defining winning as the* sole *objective for the machine.*

It is impossible to anticipate all the ways in which a machine pursuing a fixed objective might misbehave. There is good reason, then, to think that the standard model is inadequate. We don't want machines that are intelligent in the sense of pursuing *their* objectives; we want them to pursue *our* objectives. If we cannot transfer those objectives perfectly to the machine, then we need a new formulation—one in which the machine is pursuing our objectives, but is necessarily *uncertain* as to what they are. When a machine knows that it doesn't know the complete objective, it has an incentive to act cautiously, to ask permission, to learn more about our preferences through observation, and to defer to human control. Ultimately, we want agents that are **provably beneficial** to humans. We will return to this topic in Section 1.5.

Provably beneficial

1.2 The Foundations of Artificial Intelligence

In this section, we provide a brief history of the disciplines that contributed ideas, viewpoints, and techniques to AI. Like any history, this one concentrates on a small number of people, events, and ideas and ignores others that also were important. We organize the history around a series of questions. We certainly would not wish to give the impression that these questions are the only ones the disciplines address or that the disciplines have all been working toward AI as their ultimate fruition.

[3] In one of the first books on chess, Ruy Lopez (1561) wrote, "Always place the board so the sun is in your opponent's eyes."

1.2.1 Philosophy

- Can formal rules be used to draw valid conclusions?
- How does the mind arise from a physical brain?
- Where does knowledge come from?
- How does knowledge lead to action?

Aristotle (384–322 BCE) was the first to formulate a precise set of laws governing the rational part of the mind. He developed an informal system of syllogisms for proper reasoning, which in principle allowed one to generate conclusions mechanically, given initial premises.

Ramon Llull (c. 1232–1315) devised a system of reasoning published as *Ars Magna* or *The Great Art* (1305). Llull tried to implement his system using an actual mechanical device: a set of paper wheels that could be rotated into different permutations.

Around 1500, Leonardo da Vinci (1452–1519) designed but did not build a mechanical calculator; recent reconstructions have shown the design to be functional. The first known calculating machine was constructed around 1623 by the German scientist Wilhelm Schickard (1592–1635). Blaise Pascal (1623–1662) built the Pascaline in 1642 and wrote that it "produces effects which appear nearer to thought than all the actions of animals." Gottfried Wilhelm Leibniz (1646–1716) built a mechanical device intended to carry out operations on concepts rather than numbers, but its scope was rather limited. In his 1651 book *Leviathan*, Thomas Hobbes (1588–1679) suggested the idea of a thinking machine, an "artificial animal" in his words, arguing "For what is the heart but a spring; and the nerves, but so many strings; and the joints, but so many wheels." He also suggested that reasoning was like numerical computation: "For 'reason' ... is nothing but 'reckoning,' that is adding and subtracting."

It's one thing to say that the mind operates, at least in part, according to logical or numerical rules, and to build physical systems that emulate some of those rules. It's another to say that the mind itself *is* such a physical system. René Descartes (1596–1650) gave the first clear discussion of the distinction between mind and matter. He noted that a purely physical conception of the mind seems to leave little room for free will. If the mind is governed entirely by physical laws, then it has no more free will than a rock "deciding" to fall downward.

Dualism

Descartes was a proponent of **dualism**. He held that there is a part of the human mind (or soul or spirit) that is outside of nature, exempt from physical laws. Animals, on the other hand, did not possess this dual quality; they could be treated as machines.

An alternative to dualism is **materialism**, which holds that the brain's operation according to the laws of physics *constitutes* the mind. Free will is simply the way that the perception of available choices appears to the choosing entity. The terms **physicalism** and **naturalism** are also used to describe this view that stands in contrast to the supernatural.

Empiricism

Given a physical mind that manipulates knowledge, the next problem is to establish the source of knowledge. The **empiricism** movement, starting with Francis Bacon's (1561–1626) *Novum Organum*,[4] is characterized by a dictum of John Locke (1632–1704): "Nothing is in the understanding, which was not first in the senses."

Induction

David Hume's (1711–1776) *A Treatise of Human Nature* (Hume, 1739) proposed what is now known as the principle of **induction**: that general rules are acquired by exposure to repeated associations between their elements.

[4] The *Novum Organum* is an update of Aristotle's *Organon*, or instrument of thought.

Building on the work of Ludwig Wittgenstein (1889–1951) and Bertrand Russell (1872–1970), the famous Vienna Circle (Sigmund, 2017), a group of philosophers and mathematicians meeting in Vienna in the 1920s and 1930s, developed the doctrine of **logical positivism**. This doctrine holds that all knowledge can be characterized by logical theories connected, ultimately, to **observation sentences** that correspond to sensory inputs; thus logical positivism combines rationalism and empiricism.

Logical positivism

Observation sentence

The **confirmation theory** of Rudolf Carnap (1891–1970) and Carl Hempel (1905–1997) attempted to analyze the acquisition of knowledge from experience by quantifying the degree of belief that should be assigned to logical sentences based on their connection to observations that confirm or disconfirm them. Carnap's book *The Logical Structure of the World* (1928) was perhaps the first theory of mind as a computational process.

Confirmation theory

The final element in the philosophical picture of the mind is the connection between knowledge and action. This question is vital to AI because intelligence requires action as well as reasoning. Moreover, only by understanding how actions are justified can we understand how to build an agent whose actions are justifiable (or rational).

Aristotle argued (in *De Motu Animalium*) that actions are justified by a logical connection between goals and knowledge of the action's outcome:

> But how does it happen that thinking is sometimes accompanied by action and sometimes not, sometimes by motion, and sometimes not? It looks as if almost the same thing happens as in the case of reasoning and making inferences about unchanging objects. But in that case the end is a speculative proposition … whereas here the conclusion which results from the two premises is an action. … I need covering; a cloak is a covering. I need a cloak. What I need, I have to make; I need a cloak. I have to make a cloak. And the conclusion, the "I have to make a cloak," is an action.

In the *Nicomachean Ethics* (Book III. 3, 1112b), Aristotle further elaborates on this topic, suggesting an algorithm:

> We deliberate not about ends, but about means. For a doctor does not deliberate whether he shall heal, nor an orator whether he shall persuade, … They assume the end and consider how and by what means it is attained, and if it seems easily and best produced thereby; while if it is achieved by one means only they consider *how* it will be achieved by this and by what means *this* will be achieved, till they come to the first cause, … and what is last in the order of analysis seems to be first in the order of becoming. And if we come on an impossibility, we give up the search, e.g., if we need money and this cannot be got; but if a thing appears possible we try to do it.

Aristotle's algorithm was implemented 2300 years later by Newell and Simon in their **General Problem Solver** program. We would now call it a greedy regression planning system (see Chapter 11). Methods based on logical planning to achieve definite goals dominated the first few decades of theoretical research in AI.

Thinking purely in terms of actions achieving goals is often useful but sometimes inapplicable. For example, if there are several different ways to achieve a goal, there needs to be some way to choose among them. More importantly, it may not be possible to achieve a goal with certainty, but some action must still be taken. How then should one decide? Antoine Arnauld (1662), analyzing the notion of rational decisions in gambling, proposed a quantitative formula for maximizing the expected monetary value of the outcome. Later, Daniel Bernoulli (1738) introduced the more general notion of **utility** to capture the internal, subjective value

Utility

of an outcome. The modern notion of rational decision making under uncertainty involves maximizing expected utility, as explained in Chapter 16.

Utilitarianism

In matters of ethics and public policy, a decision maker must consider the interests of multiple individuals. Jeremy Bentham (1823) and John Stuart Mill (1863) promoted the idea of **utilitarianism**: that rational decision making based on maximizing utility should apply to all spheres of human activity, including public policy decisions made on behalf of many individuals. Utilitarianism is a specific kind of **consequentialism**: the idea that what is right and wrong is determined by the expected outcomes of an action.

Deontological ethics

In contrast, Immanuel Kant, in 1875 proposed a theory of rule-based or **deontological ethics**, in which "doing the right thing" is determined not by outcomes but by universal social laws that govern allowable actions, such as "don't lie" or "don't kill." Thus, a utilitarian could tell a white lie if the expected good outweighs the bad, but a Kantian would be bound not to, because lying is inherently wrong. Mill acknowledged the value of rules, but understood them as efficient decision procedures compiled from first-principles reasoning about consequences. Many modern AI systems adopt exactly this approach.

1.2.2 Mathematics

- What are the formal rules to draw valid conclusions?
- What can be computed?
- How do we reason with uncertain information?

Philosophers staked out some of the fundamental ideas of AI, but the leap to a formal science required the mathematization of logic and probability and the introduction of a new branch of mathematics: computation.

Formal logic

The idea of **formal logic** can be traced back to the philosophers of ancient Greece, India, and China, but its mathematical development really began with the work of George Boole (1815–1864), who worked out the details of propositional, or Boolean, logic (Boole, 1847). In 1879, Gottlob Frege (1848–1925) extended Boole's logic to include objects and relations, creating the first-order logic that is used today.[5] In addition to its central role in the early period of AI research, first-order logic motivated the work of Gödel and Turing that underpinned computation itself, as we explain below.

Probability

The theory of **probability** can be seen as generalizing logic to situations with uncertain information—a consideration of great importance for AI. Gerolamo Cardano (1501–1576) first framed the idea of probability, describing it in terms of the possible outcomes of gambling events. In 1654, Blaise Pascal (1623–1662), in a letter to Pierre Fermat (1601–1665), showed how to predict the future of an unfinished gambling game and assign average payoffs to the gamblers. Probability quickly became an invaluable part of the quantitative sciences, helping to deal with uncertain measurements and incomplete theories. Jacob Bernoulli (1654–1705, uncle of Daniel), Pierre Laplace (1749–1827), and others advanced the theory and introduced new statistical methods. Thomas Bayes (1702–1761) proposed a rule for updating probabilities in the light of new evidence; Bayes' rule is a crucial tool for AI systems.

Statistics

The formalization of probability, combined with the availability of data, led to the emergence of **statistics** as a field. One of the first uses was John Graunt's analysis of Lon-

[5] Frege's proposed notation for first-order logic—an arcane combination of textual and geometric features—never became popular.

don census data in 1662. Ronald Fisher is considered the first modern statistician (Fisher, 1922). He brought together the ideas of probability, experiment design, analysis of data, and computing—in 1919, he insisted that he couldn't do his work without a mechanical calculator called the MILLIONAIRE (the first calculator that could do multiplication), even though the cost of the calculator was more than his annual salary (Ross, 2012).

The history of computation is as old as the history of numbers, but the first nontrivial **algorithm** is thought to be Euclid's algorithm for computing greatest common divisors. The word *algorithm* comes from Muhammad ibn Musa al-Khwarizmi, a 9th century mathematician, whose writings also introduced Arabic numerals and algebra to Europe. Boole and others discussed algorithms for logical deduction, and, by the late 19th century, efforts were under way to formalize general mathematical reasoning as logical deduction.

Algorithm

Kurt Gödel (1906–1978) showed that there exists an effective procedure to prove any true statement in the first-order logic of Frege and Russell, but that first-order logic could not capture the principle of mathematical induction needed to characterize the natural numbers. In 1931, Gödel showed that limits on deduction do exist. His **incompleteness theorem** showed that in any formal theory as strong as Peano arithmetic (the elementary theory of natural numbers), there are necessarily true statements that have no proof within the theory.

Incompleteness theorem

This fundamental result can also be interpreted as showing that some functions on the integers cannot be represented by an algorithm—that is, they cannot be computed. This motivated Alan Turing (1912–1954) to try to characterize exactly which functions *are* **computable**—capable of being computed by an effective procedure. The Church–Turing thesis proposes to identify the general notion of computability with functions computed by a Turing machine (Turing, 1936). Turing also showed that there were some functions that no Turing machine can compute. For example, no machine can tell *in general* whether a given program will return an answer on a given input or run forever.

Computability

Although computability is important to an understanding of computation, the notion of **tractability** has had an even greater impact on AI. Roughly speaking, a problem is called intractable if the time required to solve instances of the problem grows exponentially with the size of the instances. The distinction between polynomial and exponential growth in complexity was first emphasized in the mid-1960s (Cobham, 1964; Edmonds, 1965). It is important because exponential growth means that even moderately large instances cannot be solved in any reasonable time.

Tractability

The theory of **NP-completeness**, pioneered by Cook (1971) and Karp (1972), provides a basis for analyzing the tractability of problems: any problem class to which the class of NP-complete problems can be reduced is likely to be intractable. (Although it has not been proved that NP-complete problems are necessarily intractable, most theoreticians believe it.) These results contrast with the optimism with which the popular press greeted the first computers—"Electronic Super-Brains" that were "Faster than Einstein!" Despite the increasing speed of computers, careful use of resources and necessary imperfection will characterize intelligent systems. Put crudely, the world is an *extremely* large problem instance!

NP-completeness

1.2.3 Economics

- How should we make decisions in accordance with our preferences?
- How should we do this when others may not go along?
- How should we do this when the payoff may be far in the future?

The science of economics originated in 1776, when Adam Smith (1723–1790) published *An Inquiry into the Nature and Causes of the Wealth of Nations*. Smith proposed to analyze economies as consisting of many individual agents attending to their own interests. Smith was not, however, advocating financial greed as a moral position: his earlier (1759) book *The Theory of Moral Sentiments* begins by pointing out that concern for the well-being of others is an essential component of the interests of every individual.

Most people think of economics as being about money, and indeed the first mathematical analysis of decisions under uncertainty, the maximum-expected-value formula of Arnauld (1662), dealt with the monetary value of bets. Daniel Bernoulli (1738) noticed that this formula didn't seem to work well for larger amounts of money, such as investments in maritime trading expeditions. He proposed instead a principle based on maximization of expected utility, and explained human investment choices by proposing that the marginal utility of an additional quantity of money diminished as one acquired more money.

Léon Walras (pronounced "Valrasse") (1834–1910) gave utility theory a more general foundation in terms of preferences between gambles on any outcomes (not just monetary outcomes). The theory was improved by Ramsey (1931) and later by John von Neumann and Oskar Morgenstern in their book *The Theory of Games and Economic Behavior* (1944). Economics is no longer the study of money; rather it is the study of desires and preferences.

Decision theory

Decision theory, which combines probability theory with utility theory, provides a formal and complete framework for individual decisions (economic or otherwise) made under uncertainty—that is, in cases where probabilistic descriptions appropriately capture the decision maker's environment. This is suitable for "large" economies where each agent need pay no attention to the actions of other agents as individuals. For "small" economies, the situation is much more like a **game**: the actions of one player can significantly affect the utility of another (either positively or negatively). Von Neumann and Morgenstern's development of **game theory** (see also Luce and Raiffa, 1957) included the surprising result that, for some games, a rational agent should adopt policies that are (or least appear to be) randomized. Unlike decision theory, game theory does not offer an unambiguous prescription for selecting actions. In AI, decisions involving multiple agents are studied under the heading of **multiagent systems** (Chapter 18).

Operations research

Economists, with some exceptions, did not address the third question listed above: how to make rational decisions when payoffs from actions are not immediate but instead result from several actions taken *in sequence*. This topic was pursued in the field of **operations research**, which emerged in World War II from efforts in Britain to optimize radar installations, and later found innumerable civilian applications. The work of Richard Bellman (1957) formalized a class of sequential decision problems called **Markov decision processes**, which we study in Chapter 17 and, under the heading of **reinforcement learning**, in Chapter 22.

Satisficing

Work in economics and operations research has contributed much to our notion of rational agents, yet for many years AI research developed along entirely separate paths. One reason was the apparent complexity of making rational decisions. The pioneering AI researcher Herbert Simon (1916–2001) won the Nobel Prize in economics in 1978 for his early work showing that models based on **satisficing**—making decisions that are "good enough," rather than laboriously calculating an optimal decision—gave a better description of actual human behavior (Simon, 1947). Since the 1990s, there has been a resurgence of interest in decision-theoretic techniques for AI.

1.2.4 Neuroscience

- How do brains process information?

Neuroscience

Neuroscience is the study of the nervous system, particularly the brain. Although the exact way in which the brain enables thought is one of the great mysteries of science, the fact that it *does* enable thought has been appreciated for thousands of years because of the evidence that strong blows to the head can lead to mental incapacitation. It has also long been known that human brains are somehow different; in about 335 BCE Aristotle wrote, “Of all the animals, man has the largest brain in proportion to his size.”[6] Still, it was not until the middle of the 18th century that the brain was widely recognized as the seat of consciousness. Before then, candidate locations included the heart and the spleen.

Neuron

Paul Broca’s (1824–1880) investigation of aphasia (speech deficit) in brain-damaged patients in 1861 initiated the study of the brain’s functional organization by identifying a localized area in the left hemisphere—now called Broca’s area—that is responsible for speech production.[7] By that time, it was known that the brain consisted largely of nerve cells, or **neurons**, but it was not until 1873 that Camillo Golgi (1843–1926) developed a staining technique allowing the observation of individual neurons (see Figure 1.1). This technique was used by Santiago Ramon y Cajal (1852–1934) in his pioneering studies of neuronal organization.[8] It is now widely accepted that cognitive functions result from the electrochemical operation of these structures. That is, *a collection of simple cells can lead to thought, action, and consciousness.* In the pithy words of John Searle (1992), *brains cause minds*.

We now have some data on the mapping between areas of the brain and the parts of the body that they control or from which they receive sensory input. Such mappings are able to change radically over the course of a few weeks, and some animals seem to have multiple maps. Moreover, we do not fully understand how other areas can take over functions when one area is damaged. There is almost no theory on how an individual memory is stored or on how higher-level cognitive functions operate.

Optogenetics

The measurement of intact brain activity began in 1929 with the invention by Hans Berger of the electroencephalograph (EEG). The development of functional magnetic resonance imaging (fMRI) (Ogawa *et al.*, 1990; Cabeza and Nyberg, 2001) is giving neuroscientists unprecedentedly detailed images of brain activity, enabling measurements that correspond in interesting ways to ongoing cognitive processes. These are augmented by advances in single-cell electrical recording of neuron activity and by the methods of **optogenetics** (Crick, 1999; Zemelman *et al.*, 2002; Han and Boyden, 2007), which allow both measurement and control of individual neurons modified to be light-sensitive.

Brain–machine interface

The development of **brain–machine interfaces** (Lebedev and Nicolelis, 2006) for both sensing and motor control not only promises to restore function to disabled individuals, but also sheds light on many aspects of neural systems. A remarkable finding from this work is that the brain is able to adjust itself to interface successfully with an external device, treating it in effect like another sensory organ or limb.

[6] It has since been discovered that the tree shrew and some bird species exceed the human brain/body ratio.

[7] Many cite Alexander Hood (1824) as a possible prior source.

[8] Golgi persisted in his belief that the brain’s functions were carried out primarily in a continuous medium in which neurons were embedded, whereas Cajal propounded the “neuronal doctrine.” The two shared the Nobel Prize in 1906 but gave mutually antagonistic acceptance speeches.

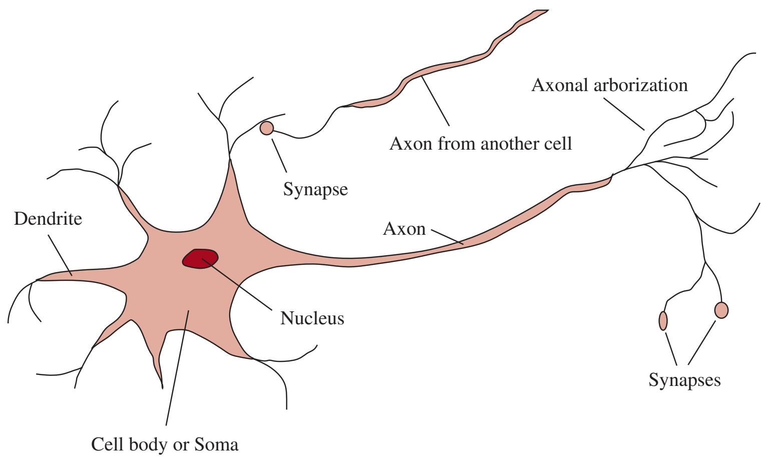


Figure 1.1 The parts of a nerve cell or neuron. Each neuron consists of a cell body, or soma, that contains a cell nucleus. Branching out from the cell body are a number of fibers called dendrites and a single long fiber called the axon. The axon stretches out for a long distance, much longer than the scale in this diagram indicates. Typically, an axon is 1 cm long (100 times the diameter of the cell body), but can reach up to 1 meter. A neuron makes connections with 10 to 100,000 other neurons at junctions called synapses. Signals are propagated from neuron to neuron by a complicated electrochemical reaction. The signals control brain activity in the short term and also enable long-term changes in the connectivity of neurons. These mechanisms are thought to form the basis for learning in the brain. Most information processing goes on in the cerebral cortex, the outer layer of the brain. The basic organizational unit appears to be a column of tissue about 0.5 mm in diameter, containing about 20,000 neurons and extending the full depth of the cortex (about 4 mm in humans).

Brains and digital computers have somewhat different properties. Figure 1.2 shows that computers have a cycle time that is a million times faster than a brain. The brain makes up for that with far more storage and interconnection than even a high-end personal computer, although the largest supercomputers match the brain on some metrics. Futurists make much of these numbers, pointing to an approaching **singularity** at which computers reach a superhuman level of performance (Vinge, 1993; Kurzweil, 2005; Doctorow and Stross, 2012), and then rapidly improve themselves even further. But the comparisons of raw numbers are not especially informative. Even with a computer of virtually unlimited capacity, we still require further conceptual breakthroughs in our understanding of intelligence (see Chapter 28). Crudely put, without the right theory, faster machines just give you the wrong answer faster.

Singularity

1.2.5 Psychology

- How do humans and animals think and act?

The origins of scientific psychology are usually traced to the work of the German physicist Hermann von Helmholtz (1821–1894) and his student Wilhelm Wundt (1832–1920). Helmholtz applied the scientific method to the study of human vision, and his *Handbook of Physiological Optics* has been described as “the single most important treatise on the physics and physiology of human vision” (Nalwa, 1993, p.15). In 1879, Wundt opened the first laboratory of experimental psychology, at the University of Leipzig. Wundt insisted on carefully

	Supercomputer	Personal Computer	Human Brain
Computational units	10^6 GPUs + CPUs	8 CPU cores	10^6 columns
	10^{15} transistors	10^{10} transistors	10^{11} neurons
Storage units	10^{16} bytes RAM	10^{10} bytes RAM	10^{11} neurons
	10^{17} bytes disk	10^{12} bytes disk	10^{14} synapses
Cycle time	10^{-9} sec	10^{-9} sec	10^{-3} sec
Operations/sec	10^{18}	10^{10}	10^{17}

Figure 1.2 A crude comparison of a leading supercomputer, Summit (Feldman, 2017); a typical personal computer of 2019; and the human brain. Human brain power has not changed much in thousands of years, whereas supercomputers have improved from megaFLOPs in the 1960s to gigaFLOPs in the 1980s, teraFLOPs in the 1990s, petaFLOPs in 2008, and exaFLOPs in 2018 (1 exaFLOP = 10^{18} floating point operations per second).

controlled experiments in which his workers would perform a perceptual or associative task while introspecting on their thought processes. The careful controls went a long toward making psychology a science, but the subjective nature of the data made it unlikely that experimenters would ever disconfirm their own theories.

Biologists studying animal behavior, on the other hand, lacked introspective data and developed an objective methodology, as described by H. S. Jennings (1906) in his influential work *Behavior of the Lower Organisms*. Applying this viewpoint to humans, the **behaviorism** movement, led by John Watson (1878–1958), rejected *any* theory involving mental processes on the grounds that introspection could not provide reliable evidence. Behaviorists insisted on studying only objective measures of the percepts (or *stimulus*) given to an animal and its resulting actions (or *response*). Behaviorism discovered a lot about rats and pigeons but had less success at understanding humans.

Behaviorism

Cognitive psychology, which views the brain as an information-processing device, can be traced back at least to the works of William James (1842–1910). Helmholtz also insisted that perception involved a form of unconscious logical inference. The cognitive viewpoint was largely eclipsed by behaviorism in the United States, but at Cambridge's Applied Psychology Unit, directed by Frederic Bartlett (1886–1969), cognitive modeling was able to flourish. *The Nature of Explanation*, by Bartlett's student and successor Kenneth Craik (1943), forcefully reestablished the legitimacy of such "mental" terms as beliefs and goals, arguing that they are just as scientific as, say, using pressure and temperature to talk about gases, despite gasses being made of molecules that have neither.

Cognitive psychology

Craik specified the three key steps of a knowledge-based agent: (1) the stimulus must be translated into an internal representation, (2) the representation is manipulated by cognitive processes to derive new internal representations, and (3) these are in turn retranslated back into action. He clearly explained why this was a good design for an agent:

> If the organism carries a "small-scale model" of external reality and of its own possible actions within its head, it is able to try out various alternatives, conclude which is the best of them, react to future situations before they arise, utilize the knowledge of past events in dealing with the present and future, and in every way to react in a much fuller, safer, and more competent manner to the emergencies which face it. (Craik, 1943)

After Craik's death in a bicycle accident in 1945, his work was continued by Donald Broadbent, whose book *Perception and Communication* (1958) was one of the first works to model psychological phenomena as information processing. Meanwhile, in the United States, the development of computer modeling led to the creation of the field of **cognitive science**. The field can be said to have started at a workshop in September 1956 at MIT—just two months after the conference at which AI itself was "born."

At the workshop, George Miller presented *The Magic Number Seven*, Noam Chomsky presented *Three Models of Language*, and Allen Newell and Herbert Simon presented *The Logic Theory Machine*. These three influential papers showed how computer models could be used to address the psychology of memory, language, and logical thinking, respectively. It is now a common (although far from universal) view among psychologists that "a cognitive theory should be like a computer program" (Anderson, 1980); that is, it should describe the operation of a cognitive function in terms of the processing of information.

For purposes of this review, we will count the field of **human–computer interaction** (HCI) under psychology. Doug Engelbart, one of the pioneers of HCI, championed the idea of **intelligence augmentation**—IA rather than AI. He believed that computers should augment human abilities rather than automate away human tasks. In 1968, Engelbart's "mother of all demos" showed off for the first time the computer mouse, a windowing system, hypertext, and video conferencing—all in an effort to demonstrate what human knowledge workers could collectively accomplish with some intelligence augmentation.

Intelligence augmentation

Today we are more likely to see IA and AI as two sides of the same coin, with the former emphasizing human control and the latter emphasizing intelligent behavior on the part of the machine. Both are needed for machines to be useful to humans.

1.2.6 Computer engineering

- How can we build an efficient computer?

The modern digital electronic computer was invented independently and almost simultaneously by scientists in three countries embattled in World War II. The first *operational* computer was the electromechanical Heath Robinson,[9] built in 1943 by Alan Turing's team for a single purpose: deciphering German messages. In 1943, the same group developed the Colossus, a powerful general-purpose machine based on vacuum tubes.[10] The first operational *programmable* computer was the Z-3, the invention of Konrad Zuse in Germany in 1941. Zuse also invented floating-point numbers and the first high-level programming language, Plankalkül. The first *electronic* computer, the ABC, was assembled by John Atanasoff and his student Clifford Berry between 1940 and 1942 at Iowa State University. Atanasoff's research received little support or recognition; it was the ENIAC, developed as part of a secret military project at the University of Pennsylvania by a team including John Mauchly and J. Presper Eckert, that proved to be the most influential forerunner of modern computers.

Since that time, each generation of computer hardware has brought an increase in speed and capacity and a decrease in price—a trend captured in **Moore's law**. Performance doubled every 18 months or so until around 2005, when power dissipation problems led manufacturers

Moore's law

[9] A complex machine named after a British cartoonist who depicted whimsical and absurdly complicated contraptions for everyday tasks such as buttering toast.

[10] In the postwar period, Turing wanted to use these computers for AI research—for example, he created an outline of the first chess program (Turing *et al.*, 1953)—but the British government blocked this research.

to start multiplying the number of CPU cores rather than the clock speed. Current expectations are that future increases in functionality will come from massive parallelism—a curious convergence with the properties of the brain. We also see new hardware designs based on the idea that in dealing with an uncertain world, we don't need 64 bits of precision in our numbers; just 16 bits (as in the `bfloat16` format) or even 8 bits will be enough, and will enable faster processing.

We are just beginning to see hardware tuned for AI applications, such as the graphics processing unit (GPU), tensor processing unit (TPU), and wafer scale engine (WSE). From the 1960s to about 2012, the amount of computing power used to train top machine learning applications followed Moore's law. Beginning in 2012, things changed: from 2012 to 2018 there was a 300,000-fold increase, which works out to a doubling every 100 days or so (Amodei and Hernandez, 2018). A machine learning model that took a full day to train in 2014 takes only two minutes in 2018 (Ying *et al.*, 2018). Although it is not yet practical, **quantum computing** holds out the promise of far greater accelerations for some important subclasses of AI algorithms.

Quantum computing

Of course, there were calculating devices before the electronic computer. The earliest automated machines, dating from the 17th century, were discussed on page 6. The first *programmable* machine was a loom, devised in 1805 by Joseph Marie Jacquard (1752–1834), that used punched cards to store instructions for the pattern to be woven.

In the mid-19th century, Charles Babbage (1792–1871) designed two computing machines, neither of which he completed. The Difference Engine was intended to compute mathematical tables for engineering and scientific projects. It was finally built and shown to work in 1991 (Swade, 2000). Babbage's Analytical Engine was far more ambitious: it included addressable memory, stored programs based on Jacquard's punched cards, and conditional jumps. It was the first machine capable of universal computation.

Babbage's colleague Ada Lovelace, daughter of the poet Lord Byron, understood its potential, describing it as "a thinking or . . . a reasoning machine," one capable of reasoning about "all subjects in the universe" (Lovelace, 1843). She also anticipated AI's hype cycles, writing, "It is desirable to guard against the possibility of exaggerated ideas that might arise as to the powers of the Analytical Engine." Unfortunately, Babbage's machines and Lovelace's ideas were largely forgotten.

AI also owes a debt to the software side of computer science, which has supplied the operating systems, programming languages, and tools needed to write modern programs (and papers about them). But this is one area where the debt has been repaid: work in AI has pioneered many ideas that have made their way back to mainstream computer science, including time sharing, interactive interpreters, personal computers with windows and mice, rapid development environments, the linked-list data type, automatic storage management, and key concepts of symbolic, functional, declarative, and object-oriented programming.

1.2.7 Control theory and cybernetics

- How can artifacts operate under their own control?

Ktesibios of Alexandria (c. 250 BCE) built the first self-controlling machine: a water clock with a regulator that maintained a constant flow rate. This invention changed the definition of what an artifact could do. Previously, only living things could modify their behavior in response to changes in the environment. Other examples of self-regulating feedback control systems include the steam engine governor, created by James Watt (1736–1819), and the thermostat, invented by Cornelis Drebbel (1572–1633), who also invented the submarine. James Clerk Maxwell (1868) initiated the mathematical theory of control systems.

Control theory

A central figure in the post-war development of **control theory** was Norbert Wiener (1894–1964). Wiener was a brilliant mathematician who worked with Bertrand Russell, among others, before developing an interest in biological and mechanical control systems and their connection to cognition. Like Craik (who also used control systems as psychological models), Wiener and his colleagues Arturo Rosenblueth and Julian Bigelow challenged the behaviorist orthodoxy (Rosenblueth *et al.*, 1943). They viewed purposive behavior as arising from a regulatory mechanism trying to minimize "error"—the difference between current state and goal state. In the late 1940s, Wiener, along with Warren McCulloch, Walter Pitts, and John von Neumann, organized a series of influential conferences that explored the new mathematical and computational models of cognition. Wiener's book *Cybernetics* (1948) became a bestseller and awoke the public to the possibility of artificially intelligent machines.

Cybernetics

Meanwhile, in Britain, W. Ross Ashby pioneered similar ideas (Ashby, 1940). Ashby, Alan Turing, Grey Walter, and others formed the Ratio Club for "those who had Wiener's ideas before Wiener's book appeared." Ashby's *Design for a Brain* (1948, 1952) elaborated on his idea that intelligence could be created by the use of **homeostatic** devices containing appropriate feedback loops to achieve stable adaptive behavior.

Homeostatic

Modern control theory, especially the branch known as stochastic optimal control, has as its goal the design of systems that maximize a **cost function** over time. This roughly matches the standard model of AI: designing systems that behave optimally. Why, then, are AI and control theory two different fields, despite the close connections among their founders? The answer lies in the close coupling between the mathematical techniques that were familiar to the participants and the corresponding sets of problems that were encompassed in each world view. Calculus and matrix algebra, the tools of control theory, lend themselves to systems that are describable by fixed sets of continuous variables, whereas AI was founded in part as a way to escape from these perceived limitations. The tools of logical inference and computation allowed AI researchers to consider problems such as language, vision, and symbolic planning that fell completely outside the control theorist's purview.

Cost function

1.2.8 Linguistics

- How does language relate to thought?

In 1957, B. F. Skinner published *Verbal Behavior*. This was a comprehensive, detailed account of the behaviorist approach to language learning, written by the foremost expert in the field. But curiously, a review of the book became as well known as the book itself, and served to almost kill off interest in behaviorism. The author of the review was the linguist Noam Chomsky, who had just published a book on his own theory, *Syntactic Structures*. Chomsky pointed out that the behaviorist theory did not address the notion of creativity in language—it did not explain how children could understand and make up sentences that they had never heard before. Chomsky's theory—based on syntactic models going back to the Indian linguist Panini (c. 350 BCE)—could explain this, and unlike previous theories, it was formal enough that it could in principle be programmed.

Computational linguistics

Modern linguistics and AI, then, were "born" at about the same time, and grew up together, intersecting in a hybrid field called **computational linguistics** or **natural language**

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; 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 "Computing Machinery and Intelligence." Therein, he introduced the Turing test, machine learning, genetic algorithms, and reinforcement learning. He dealt with many of the objections raised to the possibility of AI, as described in Chapter 27. He also suggested that it would be easier to create human-level AI by developing learning algorithms and then teaching the machine rather than by programming its intelligence by hand. In subsequent lectures he warned that achieving this goal might not be the best thing for the human race.

In 1955, John McCarthy of Dartmouth College convinced Minsky, Claude Shannon, and Nathaniel Rochester to help him bring together U.S. researchers interested in automata theory, neural nets, and the study of intelligence. They organized a two-month workshop at Dartmouth in the summer of 1956. There were 10 attendees in all, including Allen Newell and Herbert Simon from Carnegie Tech,[11] Trenchard More from Princeton, Arthur Samuel from IBM, and Ray Solomonoff and Oliver Selfridge from MIT. The proposal states:[12]

> We propose that a 2 month, 10 man study of artificial intelligence be carried out during the summer of 1956 at Dartmouth College in Hanover, New Hampshire. The study is to proceed on the basis of the conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it. An attempt will be made to find how to make machines use language, form abstractions and concepts, solve kinds of problems now reserved for humans, and improve themselves. We think that a significant advance can be made in one or more of these problems if a carefully selected group of scientists work on it together for a summer.

Despite this optimistic prediction, the Dartmouth workshop did not lead to any breakthroughs. Newell and Simon presented perhaps the most mature work, a mathematical theorem-proving system called the Logic Theorist (LT). Simon claimed, "We have invented a computer program capable of thinking non-numerically, and thereby solved the venerable mind–body problem."[13] Soon after the workshop, the program was able to prove most of the theorems in Chapter 2 of Russell and Whitehead's *Principia Mathematica*. Russell was reportedly delighted when told that LT had come up with a proof for one theorem that was shorter than the one in *Principia*. The editors of the *Journal of Symbolic Logic* were less impressed; they rejected a paper coauthored by Newell, Simon, and Logic Theorist.

1.3.2 Early enthusiasm, great expectations (1952–1969)

The intellectual establishment of the 1950s, by and large, preferred to believe that "a machine can never do X." (See Chapter 27 for a long list of X's gathered by Turing.) AI researchers naturally responded by demonstrating one X after another. They focused in particular on tasks considered indicative of intelligence in humans, including games, puzzles, mathematics, and IQ tests. John McCarthy referred to this period as the "Look, Ma, no hands!" era.

---

[11] Now Carnegie Mellon University (CMU).

[12] This was the first official usage of McCarthy's term *artificial intelligence*. Perhaps "computational rationality" would have been more precise and less threatening, but "AI" has stuck. At the 50th anniversary of the Dartmouth conference, McCarthy stated that he resisted the terms "computer" or "computational" in deference to Norbert Wiener, who was promoting analog cybernetic devices rather than digital computers.

[13] Newell and Simon also invented a list-processing language, IPL, to write LT. They had no compiler and translated it into machine code by hand. To avoid errors, they worked in parallel, calling out binary numbers to each other as they wrote each instruction to make sure they agreed.

Newell and Simon followed up their success with LT with the General Problem Solver, or GPS. Unlike LT, this program was designed from the start to imitate human problem-solving protocols. Within the limited class of puzzles it could handle, it turned out that the order in which the program considered subgoals and possible actions was similar to that in which humans approached the same problems. Thus, GPS was probably the first program to embody the "thinking humanly" approach. The success of GPS and subsequent programs as models of cognition led Newell and Simon (1976) to formulate the famous **physical symbol system** hypothesis, which states that "a physical symbol system has the necessary and sufficient means for general intelligent action." What they meant is that any system (human or machine) exhibiting intelligence must operate by manipulating data structures composed of symbols. We will see later that this hypothesis has been challenged from many directions.

Physical symbol system

At IBM, Nathaniel Rochester and his colleagues produced some of the first AI programs. Herbert Gelernter (1959) constructed the Geometry Theorem Prover, which was able to prove theorems that many students of mathematics would find quite tricky. This work was a precursor of modern mathematical theorem provers.

Of all the exploratory work done during this period, perhaps the most influential in the long run was that of Arthur Samuel on checkers (draughts). Using methods that we now call reinforcement learning (see Chapter 22), Samuel's programs learned to play at a strong amateur level. He thereby disproved the idea that computers can do only what they are told to: his program quickly learned to play a better game than its creator. The program was demonstrated on television in 1956, creating a strong impression. Like Turing, Samuel had trouble finding computer time. Working at night, he used machines that were still on the testing floor at IBM's manufacturing plant. Samuel's program was the precursor of later systems such as TD-GAMMON (Tesauro, 1992), which was among the world's best backgammon players, and ALPHAGO (Silver *et al.*, 2016), which shocked the world by defeating the human world champion at Go (see Chapter 5).

In 1958, John McCarthy made two important contributions to AI. In MIT AI Lab Memo No. 1, he defined the high-level language **Lisp**, which was to become the dominant AI programming language for the next 30 years. In a paper entitled *Programs with Common Sense*, he advanced a conceptual proposal for AI systems based on knowledge and reasoning. The paper describes the Advice Taker, a hypothetical program that would embody general knowledge of the world and could use it to derive plans of action. The concept was illustrated with simple logical axioms that suffice to generate a plan to drive to the airport. The program was also designed to accept new axioms in the normal course of operation, thereby allowing it to achieve competence in new areas *without being reprogrammed*. The Advice Taker thus embodied the central principles of knowledge representation and reasoning: that it is useful to have a formal, explicit representation of the world and its workings and to be able to manipulate that representation with deductive processes. The paper influenced the course of AI and remains relevant today.

Lisp

1958 also marked the year that Marvin Minsky moved to MIT. His initial collaboration with McCarthy did not last, however. McCarthy stressed representation and reasoning in formal logic, whereas Minsky was more interested in getting programs to work and eventually developed an anti-logic outlook. In 1963, McCarthy started the AI lab at Stanford. His plan to use logic to build the ultimate Advice Taker was advanced by J. A. Robinson's discovery in 1965 of the resolution method (a complete theorem-proving algorithm for first-order

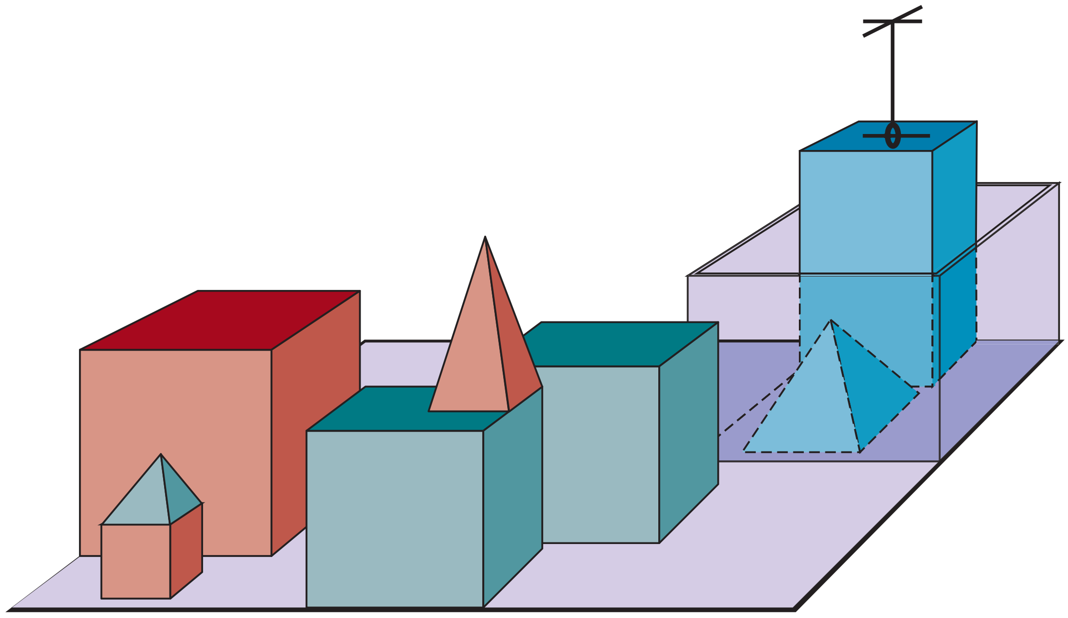


Figure 1.3 A scene from the blocks world. SHRDLU (Winograd, 1972) has just completed the command "Find a block which is taller than the one you are holding and put it in the box."

logic; see Chapter 9). Work at Stanford emphasized general-purpose methods for logical reasoning. Applications of logic included Cordell Green's question-answering and planning systems (Green, 1969b) and the Shakey robotics project at the Stanford Research Institute (SRI). The latter project, discussed further in Chapter 26, was the first to demonstrate the complete integration of logical reasoning and physical activity.

Microworld

At MIT, Minsky supervised a series of students who chose limited problems that appeared to require intelligence to solve. These limited domains became known as **microworlds**. James Slagle's SAINT program (1963) was able to solve closed-form calculus integration problems typical of first-year college courses. Tom Evans's ANALOGY program (1968) solved geometric analogy problems that appear in IQ tests. Daniel Bobrow's STUDENT program (1967) solved algebra story problems, such as the following:

> If the number of customers Tom gets is twice the square of 20 percent of the number of advertisements he runs, and the number of advertisements he runs is 45, what is the number of customers Tom gets?

Blocks world

The most famous microworld is the **blocks world**, which consists of a set of solid blocks placed on a tabletop (or more often, a simulation of a tabletop), as shown in Figure 1.3. A typical task in this world is to rearrange the blocks in a certain way, using a robot hand that can pick up one block at a time. The blocks world was home to the vision project of David Huffman (1971), the vision and constraint-propagation work of David Waltz (1975), the learning theory of Patrick Winston (1970), the natural-language-understanding program of Terry Winograd (1972), and the planner of Scott Fahlman (1974).

Early work building on the neural networks of McCulloch and Pitts also flourished. The work of Shmuel Winograd and Jack Cowan (1963) showed how a large number of elements could collectively represent an individual concept, with a corresponding increase in robustness and parallelism. Hebb's learning methods were enhanced by Bernie Widrow (Widrow and Hoff, 1960; Widrow, 1962), who called his networks **adalines**, and by Frank Rosenblatt (1962) with his **perceptrons**. The **perceptron convergence theorem** (Block *et al.*, 1962) says that the learning algorithm can adjust the connection strengths of a perceptron to match any input data, provided such a match exists.

1.3.3 A dose of reality (1966–1973)

From the beginning, AI researchers were not shy about making predictions of their coming successes. The following statement by Herbert Simon in 1957 is often quoted:

> It is not my aim to surprise or shock you—but the simplest way I can summarize is to say that there are now in the world machines that think, that learn and that create. Moreover, their ability to do these things is going to increase rapidly until—in a visible future—the range of problems they can handle will be coextensive with the range to which the human mind has been applied.

The term "visible future" is vague, but Simon also made more concrete predictions: that within 10 years a computer would be chess champion and a significant mathematical theorem would be proved by machine. These predictions came true (or approximately true) within 40 years rather than 10. Simon's overconfidence was due to the promising performance of early AI systems on simple examples. In almost all cases, however, these early systems failed on more difficult problems.

There were two main reasons for this failure. The first was that many early AI systems were based primarily on "informed introspection" as to how humans perform a task, rather than on a careful analysis of the task, what it means to be a solution, and what an algorithm would need to do to reliably produce such solutions.

The second reason for failure was a lack of appreciation of the intractability of many of the problems that AI was attempting to solve. Most of the early problem-solving systems worked by trying out different combinations of steps until the solution was found. This strategy worked initially because microworlds contained very few objects and hence very few possible actions and very short solution sequences. Before the theory of computational complexity was developed, it was widely thought that "scaling up" to larger problems was simply a matter of faster hardware and larger memories. The optimism that accompanied the development of resolution theorem proving, for example, was soon dampened when researchers failed to prove theorems involving more than a few dozen facts. *The fact that a program can find a solution in principle does not mean that the program contains any of the mechanisms needed to find it in practice.* ◀

The illusion of unlimited computational power was not confined to problem-solving programs. Early experiments in **machine evolution** (now called **genetic programming**) (Friedberg, 1958; Friedberg *et al.*, 1959) were based on the undoubtedly correct belief that by making an appropriate series of small mutations to a machine-code program, one can generate a program with good performance for any particular task. The idea, then, was to try random mutations with a selection process to preserve mutations that seemed useful. Despite thousands of hours of CPU time, almost no progress was demonstrated.

Machine evolution

Failure to come to grips with the "combinatorial explosion" was one of the main criticisms of AI contained in the Lighthill report (Lighthill, 1973), which formed the basis for the decision by the British government to end support for AI research in all but two universities. (Oral tradition paints a somewhat different and more colorful picture, with political ambitions and personal animosities whose description is beside the point.)

A third difficulty arose because of some fundamental limitations on the basic structures being used to generate intelligent behavior. For example, Minsky and Papert's book *Perceptrons* (1969) proved that, although perceptrons (a simple form of neural network) could be shown to learn anything they were capable of representing, they could represent very little. In particular, a two-input perceptron could not be trained to recognize when its two inputs were different. Although their results did not apply to more complex, multilayer networks, research funding for neural-net research soon dwindled to almost nothing. Ironically, the new back-propagation learning algorithms that were to cause an enormous resurgence in neural-net research in the late 1980s and again in the 2010s had already been developed in other contexts in the early 1960s (Kelley, 1960; Bryson, 1962).

1.3.4 Expert systems (1969–1986)

The picture of problem solving that had arisen during the first decade of AI research was of a general-purpose search mechanism trying to string together elementary reasoning steps to find complete solutions. Such approaches have been called **weak methods** because, although general, they do not scale up to large or difficult problem instances. The alternative to weak methods is to use more powerful, domain-specific knowledge that allows larger reasoning steps and can more easily handle typically occurring cases in narrow areas of expertise. One might say that to solve a hard problem, you have to almost know the answer already.

Weak method

The DENDRAL program (Buchanan *et al.*, 1969) was an early example of this approach. It was developed at Stanford, where Ed Feigenbaum (a former student of Herbert Simon), Bruce Buchanan (a philosopher turned computer scientist), and Joshua Lederberg (a Nobel laureate geneticist) teamed up to solve the problem of inferring molecular structure from the information provided by a mass spectrometer. The input to the program consists of the elementary formula of the molecule (e.g., $C_6H_{13}NO_2$) and the mass spectrum giving the masses of the various fragments of the molecule generated when it is bombarded by an electron beam. For example, the mass spectrum might contain a peak at $m = 15$, corresponding to the mass of a methyl (CH_3) fragment.

The naive version of the program generated all possible structures consistent with the formula, and then predicted what mass spectrum would be observed for each, comparing this with the actual spectrum. As one might expect, this is intractable for even moderate-sized molecules. The DENDRAL researchers consulted analytical chemists and found that they worked by looking for well-known patterns of peaks in the spectrum that suggested common substructures in the molecule. For example, the following rule is used to recognize a ketone (C=O) subgroup (which weighs 28):

if M is the mass of the whole molecule and there are two peaks at x_1 and x_2 such that
(a) $x_1 + x_2 = M + 28$; (b) $x_1 - 28$ is a high peak; (c) $x_2 - 28$ is a high peak; and
(d) At least one of x_1 and x_2 is high
then there is a ketone subgroup.

Recognizing that the molecule contains a particular substructure reduces the number of possible candidates enormously. According to its authors, DENDRAL was powerful because it embodied the relevant knowledge of mass spectroscopy not in the form of first principles but in efficient "cookbook recipes" (Feigenbaum *et al.*, 1971). The significance of DENDRAL was that it was the first successful *knowledge-intensive* system: its expertise derived from large numbers of special-purpose rules. In 1971, Feigenbaum and others at Stanford began the Heuristic Programming Project (HPP) to investigate the extent to which the new methodology of **expert systems** could be applied to other areas.

Expert systems

The next major effort was the MYCIN system for diagnosing blood infections. With about 450 rules, MYCIN was able to perform as well as some experts, and considerably better than junior doctors. It also contained two major differences from DENDRAL. First, unlike the DENDRAL rules, no general theoretical model existed from which the MYCIN rules could be deduced. They had to be acquired from extensive interviewing of experts. Second, the rules had to reflect the uncertainty associated with medical knowledge. MYCIN incorporated a calculus of uncertainty called **certainty factors** (see Chapter 13), which seemed (at the time) to fit well with how doctors assessed the impact of evidence on the diagnosis.

Certainty factor

The first successful commercial expert system, R1, began operation at the Digital Equipment Corporation (McDermott, 1982). The program helped configure orders for new computer systems; by 1986, it was saving the company an estimated $40 million a year. By 1988, DEC's AI group had 40 expert systems deployed, with more on the way. DuPont had 100 in use and 500 in development. Nearly every major U.S. corporation had its own AI group and was either using or investigating expert systems.

The importance of domain knowledge was also apparent in the area of natural language understanding. Despite the success of Winograd's SHRDLU system, its methods did not extend to more general tasks: for problems such as ambiguity resolution it used simple rules that relied on the tiny scope of the blocks world.

Several researchers, including Eugene Charniak at MIT and Roger Schank at Yale, suggested that robust language understanding would require general knowledge about the world and a general method for using that knowledge. (Schank went further, claiming, "There is no such thing as syntax," which upset a lot of linguists but did serve to start a useful discussion.) Schank and his students built a series of programs (Schank and Abelson, 1977; Wilensky, 1978; Schank and Riesbeck, 1981) that all had the task of understanding natural language. The emphasis, however, was less on language *per se* and more on the problems of representing and reasoning with the knowledge required for language understanding.

The widespread growth of applications to real-world problems led to the development of a wide range of representation and reasoning tools. Some were based on logic—for example, the Prolog language became popular in Europe and Japan, and the PLANNER family in the United States. Others, following Minsky's idea of **frames** (1975), adopted a more structured approach, assembling facts about particular object and event types and arranging the types into a large taxonomic hierarchy analogous to a biological taxonomy.

Frames

In 1981, the Japanese government announced the "Fifth Generation" project, a 10-year plan to build massively parallel, intelligent computers running Prolog. The budget was to exceed a $1.3 billion in today's money. In response, the United States formed the Microelectronics and Computer Technology Corporation (MCC), a consortium designed to assure national competitiveness. In both cases, AI was part of a broad effort, including chip design and human-interface research. In Britain, the Alvey report reinstated the funding removed by the Lighthill report. However, none of these projects ever met its ambitious goals in terms of new AI capabilities or economic impact.

Overall, the AI industry boomed from a few million dollars in 1980 to billions of dollars in 1988, including hundreds of companies building expert systems, vision systems, robots, and software and hardware specialized for these purposes.

Soon after that came a period called the "AI winter," in which many companies fell by the wayside as they failed to deliver on extravagant promises. It turned out to be difficult to build and maintain expert systems for complex domains, in part because the reasoning methods used by the systems broke down in the face of uncertainty and in part because the systems could not learn from experience.

1.3.5 The return of neural networks (1986–present)

In the mid-1980s at least four different groups reinvented the **back-propagation** learning algorithm first developed in the early 1960s. The algorithm was applied to many learning problems in computer science and psychology, and the widespread dissemination of the results in the collection *Parallel Distributed Processing* (Rumelhart and McClelland, 1986) caused great excitement.

Connectionist

These so-called **connectionist** models were seen by some as direct competitors both to the symbolic models promoted by Newell and Simon and to the logicist approach of McCarthy and others. It might seem obvious that at some level humans manipulate symbols—in fact, the anthropologist Terrence Deacon's book *The Symbolic Species* (1997) suggests that this is the *defining characteristic* of humans. Against this, Geoff Hinton, a leading figure in the resurgence of neural networks in the 1980s and 2010s, has described symbols as the "luminiferous aether of AI"—a reference to the non-existent medium through which many 19th-century physicists believed that electromagnetic waves propagated. Certainly, many concepts that we name in language fail, on closer inspection, to have the kind of logically defined necessary and sufficient conditions that early AI researchers hoped to capture in axiomatic form. It may be that connectionist models form internal concepts in a more fluid and imprecise way that is better suited to the messiness of the real world. They also have the capability to learn from examples—they can compare their predicted output value to the true value on a problem and modify their parameters to decrease the difference, making them more likely to perform well on future examples.

1.3.6 Probabilistic reasoning and machine learning (1987–present)

The brittleness of expert systems led to a new, more scientific approach incorporating probability rather than Boolean logic, machine learning rather than hand-coding, and experimental results rather than philosophical claims.[14] It became more common to build on existing theories than to propose brand-new ones, to base claims on rigorous theorems or solid experimental methodology (Cohen, 1995) rather than on intuition, and to show relevance to real-world applications rather than toy examples.

Shared benchmark problem sets became the norm for demonstrating progress, including the UC Irvine repository for machine learning data sets, the International Planning Compe-

[14] Some have characterized this change as a victory of the **neats**—those who think that AI theories should be grounded in mathematical rigor—over the **scruffies**—those who would rather try out lots of ideas, write some programs, and then assess what seems to be working. Both approaches are important. A shift toward neatness implies that the field has reached a level of stability and maturity. The present emphasis on deep learning may represent a resurgence of the scruffies.

tition for planning algorithms, the LibriSpeech corpus for speech recognition, the MNIST data set for handwritten digit recognition, ImageNet and COCO for image object recognition, SQuAD for natural language question answering, the WMT competition for machine translation, and the International SAT Competitions for Boolean satisfiability solvers.

AI was founded in part as a rebellion against the limitations of existing fields like control theory and statistics, but in this period it embraced the positive results of those fields. As David McAllester (1998) put it:

> In the early period of AI it seemed plausible that new forms of symbolic computation, e.g., frames and semantic networks, made much of classical theory obsolete. This led to a form of isolationism in which AI became largely separated from the rest of computer science. This isolationism is currently being abandoned. There is a recognition that machine learning should not be isolated from information theory, that uncertain reasoning should not be isolated from stochastic modeling, that search should not be isolated from classical optimization and control, and that automated reasoning should not be isolated from formal methods and static analysis.

The field of speech recognition illustrates the pattern. In the 1970s, a wide variety of different architectures and approaches were tried. Many of these were rather ad hoc and fragile, and worked on only a few carefully selected examples. In the 1980s, approaches using **hidden Markov models** (HMMs) came to dominate the area. Two aspects of HMMs are relevant. First, they are based on a rigorous mathematical theory. This allowed speech researchers to build on several decades of mathematical results developed in other fields. Second, they are generated by a process of training on a large corpus of real speech data. This ensures that the performance is robust, and in rigorous blind tests HMMs improved their scores steadily. As a result, speech technology and the related field of handwritten character recognition made the transition to widespread industrial and consumer applications. Note that there was no scientific claim that humans use HMMs to recognize speech; rather, HMMs provided a mathematical framework for understanding and solving the problem. We will see in Section 1.3.8, however, that deep learning has rather upset this comfortable narrative.

Hidden Markov models

1988 was an important year for the connection between AI and other fields, including statistics, operations research, decision theory, and control theory. Judea Pearl's (1988) *Probabilistic Reasoning in Intelligent Systems* led to a new acceptance of probability and decision theory in AI. Pearl's development of **Bayesian networks** yielded a rigorous and efficient formalism for representing uncertain knowledge as well as practical algorithms for probabilistic reasoning. Chapters 12 to 16 cover this area, in addition to more recent developments that have greatly increased the expressive power of probabilistic formalisms; Chapter 20 describes methods for learning Bayesian networks and related models from data.

Bayesian network

A second major contribution in 1988 was Rich Sutton's work connecting reinforcement learning—which had been used in Arthur Samuel's checker-playing program in the 1950s—to the theory of Markov decision processes (MDPs) developed in the field of operations research. A flood of work followed connecting AI planning research to MDPs, and the field of reinforcement learning found applications in robotics and process control as well as acquiring deep theoretical foundations.

One consequence of AI's newfound appreciation for data, statistical modeling, optimization, and machine learning was the gradual reunification of subfields such as computer vision, robotics, speech recognition, multiagent systems, and natural language processing that had become somewhat separate from core AI. The process of reintegration has yielded significant benefits both in terms of applications—for example, the deployment of practical robots expanded greatly during this period—and in a better theoretical understanding of the core problems of AI.

1.3.7 Big data (2001–present)

Big data

Remarkable advances in computing power and the creation of the World Wide Web have facilitated the creation of very large data sets—a phenomenon sometimes known as **big data**. These data sets include trillions of words of text, billions of images, and billions of hours of speech and video, as well as vast amounts of genomic data, vehicle tracking data, clickstream data, social network data, and so on.

This has led to the development of learning algorithms specially designed to take advantage of very large data sets. Often, the vast majority of examples in such data sets are *unlabeled*; for example, in Yarowsky's (1995) influential work on word-sense disambiguation, occurrences of a word such as "plant" are not labeled in the data set to indicate whether they refer to flora or factory. With large enough data sets, however, suitable learning algorithms can achieve an accuracy of over 96% on the task of identifying which sense was intended in a sentence. Moreover, Banko and Brill (2001) argued that the improvement in performance obtained from increasing the size of the data set by two or three orders of magnitude outweighs any improvement that can be obtained from tweaking the algorithm.

A similar phenomenon seems to occur in computer vision tasks such as filling in holes in photographs—holes caused either by damage or by the removal of ex-friends. Hays and Efros (2007) developed a clever method for doing this by blending in pixels from similar images; they found that the technique worked poorly with a database of only thousands of images but crossed a threshold of quality with millions of images. Soon after, the availability of tens of millions of images in the ImageNet database (Deng *et al.*, 2009) sparked a revolution in the field of computer vision.

The availability of big data and the shift towards machine learning helped AI recover commercial attractiveness (Havenstein, 2005; Halevy *et al.*, 2009). Big data was a crucial factor in the 2011 victory of IBM's Watson system over human champions in the Jeopardy! quiz game, an event that had a major impact on the public's perception of AI.

1.3.8 Deep learning (2011–present)

Deep learning

The term **deep learning** refers to machine learning using multiple layers of simple, adjustable computing elements. Experiments were carried out with such networks as far back as the 1970s, and in the form of **convolutional neural networks** they found some success in handwritten digit recognition in the 1990s (LeCun *et al.*, 1995). It was not until 2011, however, that deep learning methods really took off. This occurred first in speech recognition and then in visual object recognition.

In the 2012 ImageNet competition, which required classifying images into one of a thousand categories (armadillo, bookshelf, corkscrew, etc.), a deep learning system created in Geoffrey Hinton's group at the University of Toronto (Krizhevsky *et al.*, 2013) demonstrated a dramatic improvement over previous systems, which were based largely on handcrafted features. Since then, deep learning systems have exceeded human performance on some vision tasks (and lag behind in some other tasks). Similar gains have been reported in speech recognition, machine translation, medical diagnosis, and game playing. The use of a deep network to represent the evaluation function contributed to ALPHAGO's victories over the leading human Go players (Silver *et al.*, 2016, 2017, 2018).

These remarkable successes have led to a resurgence of interest in AI among students, companies, investors, governments, the media, and the general public. It seems that every week there is news of a new AI application approaching or exceeding human performance, often accompanied by speculation of either accelerated success or a new AI winter.

Deep learning relies heavily on powerful hardware. Whereas a standard computer CPU can do 10^9 or 10^{10} operations per second. a deep learning algorithm running on specialized hardware (e.g., GPU, TPU, or FPGA) might consume between 10^{14} and 10^{17} operations per second, mostly in the form of highly parallelized matrix and vector operations. Of course, deep learning also depends on the availability of large amounts of training data, and on a few algorithmic tricks (see Chapter 21).

1.4 The State of the Art

Stanford University's One Hundred Year Study on AI (also known as AI100) convenes panels of experts to provide reports on the state of the art in AI. Their 2016 report (Stone *et al.*, 2016; Grosz and Stone, 2018) concludes that "Substantial increases in the future uses of AI applications, including more self-driving cars, healthcare diagnostics and targeted treatment, and physical assistance for elder care can be expected" and that "Society is now at a crucial juncture in determining how to deploy AI-based technologies in ways that promote rather than hinder democratic values such as freedom, equality, and transparency." AI100 also produces an **AI Index** at `aiindex.org` to help track progress. Some highlights from the 2018 and 2019 reports (comparing to a year 2000 baseline unless otherwise stated): AI Index

- Publications: AI papers increased 20-fold between 2010 and 2019 to about 20,000 a year. The most popular category was machine learning. (Machine learning papers in arXiv.org doubled every year from 2009 to 2017.) Computer vision and natural language processing were the next most popular.
- Sentiment: About 70% of news articles on AI are neutral, but articles with positive tone increased from 12% in 2016 to 30% in 2018. The most common issues are ethical: data privacy and algorithm bias.
- Students: Course enrollment increased 5-fold in the U.S. and 16-fold internationally from a 2010 baseline. AI is the most popular specialization in Computer Science.
- Diversity: AI Professors worldwide are about 80% male, 20% female. Similar numbers hold for Ph.D. students and industry hires.
- Conferences: Attendance at NeurIPS increased 800% since 2012 to 13,500 attendees. Other conferences are seeing annual growth of about 30%.
- Industry: AI startups in the U.S. increased 20-fold to over 800.
- Internationalization: China publishes more papers per year than the U.S. and about as many as all of Europe. However, in citation-weighted impact, U.S. authors are 50% ahead of Chinese authors. Singapore, Brazil, Australia, Canada, and India are the fastest growing countries in terms of the number of AI hires.

- Vision: Error rates for object detection (as achieved in LSVRC, the Large-Scale Visual Recognition Challenge) improved from 28% in 2010 to 2% in 2017, exceeding human performance. Accuracy on open-ended visual question answering (VQA) improved from 55% to 68% since 2015, but lags behind human performance at 83%.
- Speed: Training time for the image recognition task dropped by a factor of 100 in just the past two years. The amount of computing power used in top AI applications is doubling every 3.4 months.
- Language: Accuracy on question answering, as measured by F1 score on the Stanford Question Answering Dataset (SQUAD), increased from 60 to 95 from 2015 to 2019; on the SQUAD 2 variant, progress was faster, going from 62 to 90 in just one year. Both scores exceed human-level performance.
- Human benchmarks: By 2019, AI systems had reportedly met or exceeded human-level performance in chess, Go, poker, Pac-Man, Jeopardy!, ImageNet object detection, speech recognition in a limited domain, Chinese-to-English translation in a restricted domain, Quake III, Dota 2, StarCraft II, various Atari games, skin cancer detection, prostate cancer detection, protein folding, and diabetic retinopathy diagnosis.

When (if ever) will AI systems achieve human-level performance across a broad variety of tasks? Ford (2018) interviews AI experts and finds a wide range of target years, from 2029 to 2200, with a mean of 2099. In a similar survey (Grace *et al.*, 2017) 50% of respondents thought this could happen by 2066, although 10% thought it could happen as early as 2025, and a few said "never." The experts were also split on whether we need fundamental new breakthroughs or just refinements on current approaches. But don't take their predictions too seriously; as Philip Tetlock (2017) demonstrates in the area of predicting world events, experts are no better than amateurs.

How will future AI systems operate? We can't yet say. As detailed in this section, the field has adopted several stories about itself—first the bold idea that intelligence by a machine was even possible, then that it could be achieved by encoding expert knowledge into logic, then that probabilistic models of the world would be the main tool, and most recently that machine learning would induce models that might not be based on any well-understood theory at all. The future will reveal what model comes next.

What can AI do today? Perhaps not as much as some of the more optimistic media articles might lead one to believe, but still a great deal. Here are some examples:

Robotic vehicles: The history of robotic vehicles stretches back to radio-controlled cars of the 1920s, but the first demonstrations of autonomous road driving without special guides occurred in the 1980s (Kanade *et al.*, 1986; Dickmanns and Zapp, 1987). After successful demonstrations of driving on dirt roads in the 132-mile DARPA Grand Challenge in 2005 (Thrun, 2006) and on streets with traffic in the 2007 Urban Challenge, the race to develop self-driving cars began in earnest. In 2018, Waymo test vehicles passed the landmark of 10 million miles driven on public roads without a serious accident, with the human driver stepping in to take over control only once every 6,000 miles. Soon after, the company began offering a commercial robotic taxi service.

In the air, autonomous fixed-wing drones have been providing cross-country blood deliveries in Rwanda since 2016. Quadcopters perform remarkable aerobatic maneuvers, explore buildings while constructing 3-D maps, and self-assemble into autonomous formations.

Legged locomotion: BigDog, a quadruped robot by Raibert *et al.* (2008), upended our notions of how robots move—no longer the slow, stiff-legged, side-to-side gait of Hollywood movie robots, but something closely resembling an animal and able to recover when shoved or when slipping on an icy puddle. Atlas, a humanoid robot, not only walks on uneven terrain but jumps onto boxes and does backflips (Ackerman and Guizzo, 2016).

Autonomous planning and scheduling: A hundred million miles from Earth, NASA's Remote Agent program became the first on-board autonomous planning program to control the scheduling of operations for a spacecraft (Jonsson *et al.*, 2000). Remote Agent generated plans from high-level goals specified from the ground and monitored the execution of those plans—detecting, diagnosing, and recovering from problems as they occurred. Today, the EUROPA planning toolkit (Barreiro *et al.*, 2012) is used for daily operations of NASA's Mars rovers and the SEXTANT system (Winternitz, 2017) allows autonomous navigation in deep space, beyond the global GPS system.

During the Persian Gulf crisis of 1991, U.S. forces deployed a Dynamic Analysis and Replanning Tool, DART (Cross and Walker, 1994), to do automated logistics planning and scheduling for transportation. This involved up to 50,000 vehicles, cargo, and people at a time, and had to account for starting points, destinations, routes, transport capacities, port and airfield capacities, and conflict resolution among all parameters. The Defense Advanced Research Project Agency (DARPA) stated that this single application more than paid back DARPA's 30-year investment in AI.

Every day, ride hailing companies such as Uber and mapping services such as Google Maps provide driving directions for hundreds of millions of users, quickly plotting an optimal route taking into account current and predicted future traffic conditions.

Machine translation: Online machine translation systems now enable the reading of documents in over 100 languages, including the native languages of over 99% of humans, and render hundreds of billions of words per day for hundreds of millions of users. While not perfect, they are generally adequate for understanding. For closely related languages with a great deal of training data (such as French and English) translations within a narrow domain are close to the level of a human (Wu *et al.*, 2016b).

Speech recognition: In 2017, Microsoft showed that its Conversational Speech Recognition System had reached a word error rate of 5.1%, matching human performance on the Switchboard task, which involves transcribing telephone conversations (Xiong *et al.*, 2017). About a third of computer interaction worldwide is now done by voice rather than keyboard; Skype provides real-time speech-to-speech translation in ten languages. Alexa, Siri, Cortana, and Google offer assistants that can answer questions and carry out tasks for the user; for example the Google Duplex service uses speech recognition and speech synthesis to make restaurant reservations for users, carrying out a fluent conversation on their behalf.

Recommendations: Companies such as Amazon, Facebook, Netflix, Spotify, YouTube, Walmart, and others use machine learning to recommend what you might like based on your past experiences and those of others like you. The field of recommender systems has a long history (Resnick and Varian, 1997) but is changing rapidly due to new deep learning methods that analyze content (text, music, video) as well as history and metadata (van den Oord *et al.*, 2014; Zhang *et al.*, 2017). Spam filtering can also be considered a form of recommendation (or dis-recommendation); current AI techniques filter out over 99.9% of spam, and email services can also recommend potential recipients, as well as possible response text.

Game playing: When Deep Blue defeated world chess champion Garry Kasparov in 1997, defenders of human supremacy placed their hopes on Go. Piet Hut, an astrophysicist and Go enthusiast, predicted that it would take “a hundred years before a computer beats humans at Go—maybe even longer.” But just 20 years later, ALPHAGO surpassed all human players (Silver *et al.*, 2017). Ke Jie, the world champion, said, “Last year, it was still quite human-like when it played. But this year, it became like a god of Go.” ALPHAGO benefited from studying hundreds of thousands of past games by human Go players, and from the distilled knowledge of expert Go players that worked on the team.

A followup program, ALPHAZERO, used no input from humans (except for the rules of the game), and was able to learn through self-play alone to defeat all opponents, human and machine, at Go, chess, and shogi (Silver *et al.*, 2018). Meanwhile, human champions have been beaten by AI systems at games as diverse as Jeopardy! (Ferrucci *et al.*, 2010), poker (Bowling *et al.*, 2015; Moravčík *et al.*, 2017; Brown and Sandholm, 2019), and the video games Dota 2 (Fernandez and Mahlmann, 2018), StarCraft II (Vinyals *et al.*, 2019), and Quake III (Jaderberg *et al.*, 2019).

Image understanding: Not content with exceeding human accuracy on the challenging ImageNet object recognition task, computer vision researchers have taken on the more difficult problem of image captioning. Some impressive examples include “A person riding a motorcycle on a dirt road,” “Two pizzas sitting on top of a stove top oven,” and “A group of young people playing a game of frisbee” (Vinyals *et al.*, 2017b). Current systems are far from perfect, however: a “refrigerator filled with lots of food and drinks” turns out to be a no-parking sign partially obscured by lots of small stickers.

Medicine: AI algorithms now equal or exceed expert doctors at diagnosing many conditions, particularly when the diagnosis is based on images. Examples include Alzheimer’s disease (Ding *et al.*, 2018), metastatic cancer (Liu *et al.*, 2017; Esteva *et al.*, 2017), ophthalmic disease (Gulshan *et al.*, 2016), and skin diseases (Liu *et al.*, 2019c). A systematic review and meta-analysis (Liu *et al.*, 2019a) found that the performance of AI programs, on average, was equivalent to health care professionals. One current emphasis in medical AI is in facilitating human–machine partnerships. For example, the LYNA system achieves 99.6% overall accuracy in diagnosing metastatic breast cancer—better than an unaided human expert—but the combination does better still (Liu *et al.*, 2018; Steiner *et al.*, 2018).

The widespread adoption of these techniques is now limited not by diagnostic accuracy but by the need to demonstrate improvement in clinical outcomes and to ensure transparency, lack of bias, and data privacy (Topol, 2019). In 2017, only two medical AI applications were approved by the FDA, but that increased to 12 in 2018, and continues to rise.

Climate science: A team of scientists won the 2018 Gordon Bell Prize for a deep learning model that discovers detailed information about extreme weather events that were previously buried in climate data. They used a supercomputer with specialized GPU hardware to exceed the exaop level (10^{18} operations per second), the first machine learning program to do so (Kurth *et al.*, 2018). Rolnick *et al.* (2019) present a 60-page catalog of ways in which machine learning can be used to tackle climate change.

These are just a few examples of artificial intelligence systems that exist today. Not magic or science fiction—but rather science, engineering, and mathematics, to which this book provides an introduction.

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 27 and 28.

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

use generally results in increasing wealth but tends to have the effect of shifting wealth from labor to capital, further exacerbating increases in inequality. Previous advances in technology—such as the invention of mechanical looms—have resulted in serious disruptions to employment, but eventually people find new kinds of work to do. On the other hand, it is possible that AI will be doing those new kinds of work too. This topic is rapidly becoming a major focus for economists and governments around the world.

- *Safety-critical applications*: As AI techniques advance, they are increasingly used in high-stakes, safety-critical applications such as driving cars and managing the water supplies of cities. Fatal accidents have already occurred and highlight the difficulty of formal verification and statistical risk analysis for systems developed using machine learning techniques. The field of AI will need to develop technical and ethical standards at least comparable to those prevalent in other engineering and healthcare disciplines where people's lives are at stake.
- *Cybersecurity*: AI techniques are useful in defending against cyberattack, for example by detecting unusual patterns of behavior, but they will also contribute to the potency, survivability, and proliferation capability of malware. For example, reinforcement learning methods have been used to create highly effective tools for automated, personalized blackmail and phishing attacks.

We will revisit these topics in more depth in Section 27.3. As AI systems become more capable, they will take on more of the societal roles previously played by humans. Just as humans have used these roles in the past to perpetrate mischief, we can expect that humans may misuse AI systems in these roles to perpetrate even more mischief. All of the examples given above point to the importance of governance and, eventually, regulation. At present, the research community and the major corporations involved in AI research have developed voluntary self-governance principles for AI-related activities (see Section 27.3). Governments and international organizations are setting up advisory bodies to devise appropriate regulations for each specific use case, to prepare for the economic and social impacts, and to take advantage of AI capabilities to address major societal problems.

What of the longer term? Will we achieve the long-standing goal: the creation of intelligence comparable to or more capable than human intelligence? And, if we do, what then?

For much of AI's history, these questions have been overshadowed by the daily grind of getting AI systems to do anything even remotely intelligent. As with any broad discipline, the great majority of AI researchers have specialized in a specific subfield such as game-playing, knowledge representation, vision, or natural language understanding—often on the assumption that progress in these subfields would contribute to the broader goals of AI. Nils Nilsson (1995), one of the original leaders of the Shakey project at SRI, reminded the field of those broader goals and warned that the subfields were in danger of becoming ends in themselves. Later, some influential founders of AI, including John McCarthy (2007), Marvin Minsky (2007), and Patrick Winston (Beal and Winston, 2009), concurred with Nilsson's warnings, suggesting that instead of focusing on measurable performance in specific applications, AI should return to its roots of striving for, in Herb Simon's words, "machines that think, that learn and that create." They called the effort **human-level AI** or HLAI—a machine should be able to learn to do anything a human can do. Their first symposium was in 2004 (Minsky *et al.*, 2004). Another effort with similar goals, the **artificial general intelligence (AGI)**

Human-level AI

Artificial general intelligence (AGI)

movement (Goertzel and Pennachin, 2007), held its first conference and organized the *Journal of Artificial General Intelligence* in 2008.

Artificial superintelligence (ASI)

At around the same time, concerns were raised that creating **artificial superintelligence** or **ASI**—intelligence that far surpasses human ability—might be a bad idea (Yudkowsky, 2008; Omohundro, 2008). Turing (1996) himself made the same point in a lecture given in Manchester in 1951, drawing on earlier ideas from Samuel Butler (1863):[15]

> It seems probable that once the machine thinking method had started, it would not take long to outstrip our feeble powers. . . . At some stage therefore we should have to expect the machines to take control, in the way that is mentioned in Samuel Butler's *Erewhon*.

These concerns have only become more widespread with recent advances in deep learning, the publication of books such as *Superintelligence* by Nick Bostrom (2014), and public pronouncements from Stephen Hawking, Bill Gates, Martin Rees, and Elon Musk.

Gorilla problem

Experiencing a general sense of unease with the idea of creating superintelligent machines is only natural. We might call this the **gorilla problem**: about seven million years ago, a now-extinct primate evolved, with one branch leading to gorillas and one to humans. Today, the gorillas are not too happy about the human branch; they have essentially no control over their future. If this is the result of success in creating superhuman AI—that humans cede control over their future—then perhaps we should stop work on AI, and, as a corollary, give up the benefits it might bring. This is the essence of Turing's warning: it is not obvious that we can control machines that are more intelligent than us.

If superhuman AI were a black box that arrived from outer space, then indeed it would be wise to exercise caution in opening the box. But it is not: *we* design the AI systems, so if they do end up "taking control," as Turing suggests, it would be the result of a design failure.

To avoid such an outcome, we need to understand the source of potential failure. Norbert Wiener (1960), who was motivated to consider the long-term future of AI after seeing Arthur Samuel's checker-playing program learn to beat its creator, had this to say:

> If we use, to achieve our purposes, a mechanical agency with whose operation we cannot interfere effectively . . . we had better be quite sure that the purpose put into the machine is the purpose which we really desire.

King Midas problem

Many cultures have myths of humans who ask gods, genies, magicians, or devils for something. Invariably, in these stories, they get what they literally ask for, and then regret it. The third wish, if there is one, is to undo the first two. We will call this the **King Midas problem**: Midas, a legendary King in Greek mythology, asked that everything he touched should turn to gold, but then regretted it after touching his food, drink, and family members.[16]

We touched on this issue in Section 1.1.5, where we pointed out the need for a significant modification to the standard model of putting fixed objectives into the machine. The solution to Wiener's predicament is not to have a definite "purpose put into the machine" at all. Instead, we want machines that strive to achieve human objectives but know that they don't know for certain exactly what those objectives are.

[15] Even earlier, in 1847, Richard Thornton, editor of the *Primitive Expounder*, railed against mechanical calculators: "Mind . . . outruns itself and does away with the necessity of its own existence by inventing machines to do its own thinking. . . . But who knows that such machines when brought to greater perfection, may not think of a plan to remedy all their own defects and then grind out ideas beyond the ken of mortal mind!"

[16] Midas would have done better if he had followed basic principles of safety and included an "undo" button and a "pause" button in his wish.

It is perhaps unfortunate that almost all AI research to date has been carried out within the standard model, which means that almost all of the technical material in this edition reflects that intellectual framework. There are, however, some early results within the new framework. In Chapter 16, we show that a machine has a positive incentive to allow itself to be switched off if and only if it is uncertain about the human objective. In Chapter 18, we formulate and study **assistance games**, which describe mathematically the situation in which a human has an objective and a machine tries to achieve it, but is initially uncertain about what it is. In Chapter 22, we explain the methods of **inverse reinforcement learning** that allow machines to learn more about human preferences from observations of the choices that humans make. In Chapter 27, we explore two of the principal difficulties: first, that our choices depend on our preferences through a very complex cognitive architecture that is hard to invert; and, second, that we humans may not have consistent preferences in the first place—either individually or as a group—so it may not be clear what AI systems *should* be doing for us.

Assistance game

Inverse reinforcement learning

Summary

This chapter defines AI and establishes the cultural background against which it has developed. Some of the important points are as follows:

- Different people approach AI with different goals in mind. Two important questions to ask are: Are you concerned with thinking, or behavior? Do you want to model humans, or try to achieve the optimal results?
- According to what we have called the standard model, AI is concerned mainly with **rational action**. An ideal **intelligent agent** takes the best possible action in a situation. We study the problem of building agents that are intelligent in this sense.
- Two refinements to this simple idea are needed: first, the ability of any agent, human or otherwise, to choose rational actions is limited by the computational intractability of doing so; second, the concept of a machine that pursues a definite objective needs to be replaced with that of a machine pursuing objectives to benefit humans, but uncertain as to what those objectives are.
- Philosophers (going back to 400 BCE) made AI conceivable by suggesting that the mind is in some ways like a machine, that it operates on knowledge encoded in some internal language, and that thought can be used to choose what actions to take.
- Mathematicians provided the tools to manipulate statements of logical certainty as well as uncertain, probabilistic statements. They also set the groundwork for understanding computation and reasoning about algorithms.
- Economists formalized the problem of making decisions that maximize the expected utility to the decision maker.
- Neuroscientists discovered some facts about how the brain works and the ways in which it is similar to and different from computers.
- Psychologists adopted the idea that humans and animals can be considered information-processing machines. Linguists showed that language use fits into this model.
- Computer engineers provided the ever-more-powerful machines that make AI applications possible, and software engineers made them more usable.

- Control theory deals with designing devices that act optimally on the basis of feedback from the environment. Initially, the mathematical tools of control theory were quite different from those used in AI, but the fields are coming closer together.
- The history of AI has had cycles of success, misplaced optimism, and resulting cutbacks in enthusiasm and funding. There have also been cycles of introducing new, creative approaches and systematically refining the best ones.
- AI has matured considerably compared to its early decades, both theoretically and methodologically. As the problems that AI deals with became more complex, the field moved from Boolean logic to probabilistic reasoning, and from hand-crafted knowledge to machine learning from data. This has led to improvements in the capabilities of real systems and greater integration with other disciplines.
- As AI systems find application in the real world, it has become necessary to consider a wide range of risks and ethical consequences.
- In the longer term, we face the difficult problem of controlling superintelligent AI systems that may evolve in unpredictable ways. Solving this problem seems to necessitate a change in our conception of AI.

Bibliographical and Historical Notes

A comprehensive history of AI is given by Nils Nilsson (2009), one of the early pioneers of the field. Pedro Domingos (2015) and Melanie Mitchell (2019) give overviews of machine learning for a general audience, and Kai-Fu Lee (2018) describes the race for international leadership in AI. Martin Ford (2018) interviews 23 leading AI researchers.

The main professional societies for AI are the Association for the Advancement of Artificial Intelligence (AAAI), the ACM Special Interest Group in Artificial Intelligence (SIGAI, formerly SIGART), the European Association for AI, and the Society for Artificial Intelligence and Simulation of Behaviour (AISB). The Partnership on AI brings together many commercial and nonprofit organizations concerned with the ethical and social impacts of AI. AAAI's *AI Magazine* contains many topical and tutorial articles, and its Web site, `aaai.org`, contains news, tutorials, and background information.

The most recent work appears in the proceedings of the major AI conferences: the International Joint Conference on AI (IJCAI), the annual European Conference on AI (ECAI), and the AAAI Conference. Machine learning is covered by the International Conference on Machine Learning and the Neural Information Processing Systems (NeurIPS) meeting. The major journals for general AI are *Artificial Intelligence*, *Computational Intelligence*, the *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *IEEE Intelligent Systems*, and the *Journal of Artificial Intelligence Research*. There are also many conferences and journals devoted to specific areas, which we cover in the appropriate chapters.

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

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.

Agent function

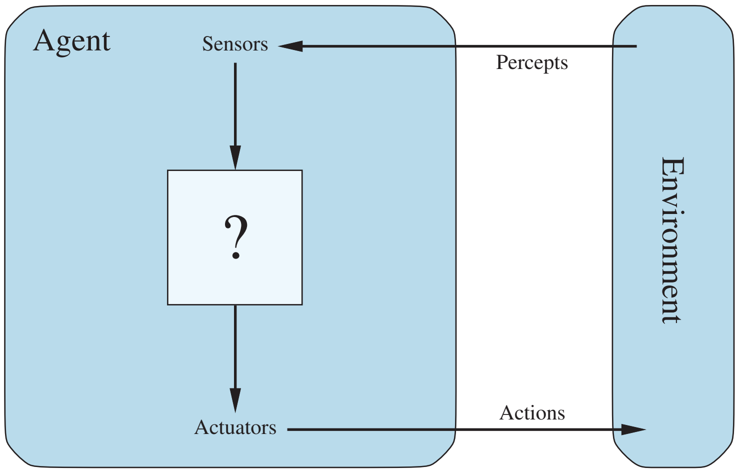


Figure 2.1 Agents interact with environments through sensors and actuators.

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 49.

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.

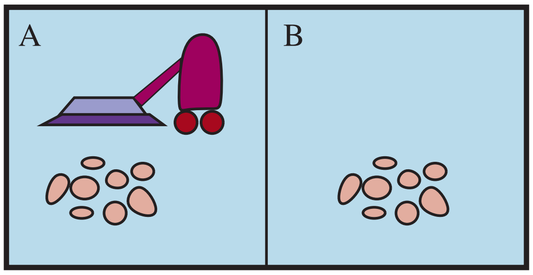


Figure 2.2 A vacuum-cleaner world with just two locations. Each location can be clean or dirty, and the agent can move left or right and can clean the square that it occupies. Different versions of the vacuum world allow for different rules about what the agent can perceive, whether its actions always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

2.2 Good Behavior: The Concept of Rationality

A **rational agent** is one that does the right thing. Obviously, doing the right thing is better than doing the wrong thing, but what does it mean to do the right thing?

Rational agent

2.2.1 Performance measures

Moral philosophy has developed several different notions of the "right thing," but AI has generally stuck to one notion called **consequentialism**: we evaluate an agent's behavior by its consequences. When an agent is plunked down in an environment, it generates a sequence of actions according to the percepts it receives. This sequence of actions causes the environment to go through a sequence of states. If the sequence is desirable, then the agent has performed well. This notion of desirability is captured by a **performance measure** that evaluates any given sequence of environment states.

Consequentialism

Performance measure

Humans have desires and preferences of their own, so the notion of rationality as applied to humans has to do with their success in choosing actions that produce sequences of environment states that are desirable *from their point of view*. Machines, on the other hand, do *not* have desires and preferences of their own; the performance measure is, initially at least, in the mind of the designer of the machine, or in the mind of the users the machine is designed for. We will see that some agent designs have an explicit representation of (a version of) the performance measure, while in other designs the performance measure is entirely implicit—the agent may do the right thing, but it doesn't know why.

Recalling Norbert Wiener's warning to ensure that "the purpose put into the machine is the purpose which we really desire" (page 33), notice that it can be quite hard to formulate a performance measure correctly. Consider, for example, the vacuum-cleaner agent from the preceding section. We might propose to measure performance by the amount of dirt cleaned up in a single eight-hour shift. With a rational agent, of course, what you ask for is what you get. A rational agent can maximize this performance measure by cleaning up the dirt, then dumping it all on the floor, then cleaning it up again, and so on. A more suitable performance measure would reward the agent for having a clean floor. For example, one point could be awarded for each clean square at each time step (perhaps with a penalty for electricity consumed and noise generated). *As a general rule, it is better to design performance measures according to what one actually wants to be achieved in the environment, rather than according to how one thinks the agent should behave.*

Even when the obvious pitfalls are avoided, some knotty problems remain. For example, the notion of "clean floor" in the preceding paragraph is based on average cleanliness over time. Yet the same average cleanliness can be achieved by two different agents, one of which does a mediocre job all the time while the other cleans energetically but takes long breaks. Which is preferable might seem to be a fine point of janitorial science, but in fact it is a deep philosophical question with far-reaching implications. Which is better—a reckless life of highs and lows, or a safe but humdrum existence? Which is better—an economy where everyone lives in moderate poverty, or one in which some live in plenty while others are very poor? We leave these questions as an exercise for the diligent reader.

For most of the book, we will assume that the performance measure can be specified correctly. For the reasons given above, however, we must accept the possibility that we might put the wrong purpose into the machine—precisely the King Midas problem described on page 33. Moreover, when designing one piece of software, copies of which will belong to different users, we cannot anticipate the exact preferences of each individual user. Thus, we may need to build agents that reflect initial uncertainty about the true performance measure and learn more about it as time goes by; such agents are described in Chapters 16, 18, and 22.

2.2.2 Rationality

What is rational at any given time depends on four things:

- The performance measure that defines the criterion of success.
- The agent's prior knowledge of the environment.
- The actions that the agent can perform.
- The agent's percept sequence to date.

Definition of a rational agent

This leads to a **definition of a rational agent**:

> *For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.*

Consider the simple vacuum-cleaner agent that cleans a square if it is dirty and moves to the other square if not; this is the agent function tabulated in Figure 2.3. Is this a rational agent? That depends! First, we need to say what the performance measure is, what is known about the environment, and what sensors and actuators the agent has. Let us assume the following:

- The performance measure awards one point for each clean square at each time step, over a "lifetime" of 1000 time steps.
- The "geography" of the environment is known *a priori* (Figure 2.2) but the dirt distribution and the initial location of the agent are not. Clean squares stay clean and sucking cleans the current square. The *Right* and *Left* actions move the agent one square except when this would take the agent outside the environment, in which case the agent remains where it is.
- The only available actions are *Right*, *Left*, and *Suck*.
- The agent correctly perceives its location and whether that location contains dirt.

Under these circumstances the agent is indeed rational; its expected performance is at least as good as any other agent's.

One can see easily that the same agent would be irrational under different circumstances. For example, once all the dirt is cleaned up, the agent will oscillate needlessly back and forth; if the performance measure includes a penalty of one point for each movement, the agent will fare poorly. A better agent for this case would do nothing once it is sure that all the squares are clean. If clean squares can become dirty again, the agent should occasionally check and re-clean them if needed. If the geography of the environment is unknown, the agent will need to **explore** it. Exercise 2.VACR asks you to design agents for these cases.

2.2.3 Omniscience, learning, and autonomy

Omniscience

We need to be careful to distinguish between rationality and **omniscience**. An omniscient agent knows the *actual* outcome of its actions and can act accordingly; but omniscience is impossible in reality. Consider the following example: I am walking along the Champs Elysées one day and I see an old friend across the street. There is no traffic nearby and I'm not otherwise engaged, so, being rational, I start to cross the street. Meanwhile, at 33,000 feet, a cargo door falls off a passing airliner,[3] and before I make it to the other side of the street I am flattened. Was I irrational to cross the street? It is unlikely that my obituary would read "Idiot attempts to cross street."

This example shows that rationality is not the same as perfection. Rationality maximizes *expected* performance, while perfection maximizes *actual* performance. Retreating from a requirement of perfection is not just a question of being fair to agents. The point is that if we expect an agent to do what turns out after the fact to be the best action, it will be impossible to design an agent to fulfill this specification—unless we improve the performance of crystal balls or time machines.

Our definition of rationality does not require omniscience, then, because the rational choice depends only on the percept sequence *to date*. We must also ensure that we haven't inadvertently allowed the agent to engage in decidedly underintelligent activities. For example, if an agent does not look both ways before crossing a busy road, then its percept sequence will not tell it that there is a large truck approaching at high speed. Does our definition of rationality say that it's now OK to cross the road? Far from it!

First, it would not be rational to cross the road given this uninformative percept sequence: the risk of accident from crossing without looking is too great. Second, a rational agent should choose the "looking" action before stepping into the street, because looking helps maximize the expected performance. Doing actions *in order to modify future percepts*—sometimes called **information gathering**—is an important part of rationality and is covered in depth in Chapter 16. A second example of information gathering is provided by the **exploration** that must be undertaken by a vacuum-cleaning agent in an initially unknown environment.

Information gathering

Our definition requires a rational agent not only to gather information but also to **learn** as much as possible from what it perceives. The agent's initial configuration could reflect some prior knowledge of the environment, but as the agent gains experience this may be modified and augmented. There are extreme cases in which the environment is completely known *a priori* and completely predictable. In such cases, the agent need not perceive or learn; it simply acts correctly.

Learning

Of course, such agents are fragile. Consider the lowly dung beetle. After digging its nest and laying its eggs, it fetches a ball of dung from a nearby heap to plug the entrance. If the ball of dung is removed from its grasp *en route*, the beetle continues its task and pantomimes plugging the nest with the nonexistent dung ball, never noticing that it is missing. Evolution has built an assumption into the beetle's behavior, and when it is violated, unsuccessful behavior results.

Slightly more intelligent is the sphex wasp. The female sphex will dig a burrow, go out and sting a caterpillar and drag it to the burrow, enter the burrow again to check all is well, drag the caterpillar inside, and lay its eggs. The caterpillar serves as a food source when the eggs hatch. So far so good, but if an entomologist moves the caterpillar a few inches away while the sphex is doing the check, it will revert to the "drag the caterpillar" step of its plan and will continue the plan without modification, re-checking the burrow, even after dozens of caterpillar-moving interventions. The sphex is unable to learn that its innate plan is failing, and thus will not change it.

[3] See N. Henderson, "New door latches urged for Boeing 747 jumbo jets," *Washington Post*, August 24, 1989.

Autonomy

To the extent that an agent relies on the prior knowledge of its designer rather than on its own percepts and learning processes, we say that the agent lacks **autonomy**. A rational agent should be autonomous—it should learn what it can to compensate for partial or incorrect prior knowledge. For example, a vacuum-cleaning agent that learns to predict where and when additional dirt will appear will do better than one that does not.

As a practical matter, one seldom requires complete autonomy from the start: when the agent has had little or no experience, it would have to act randomly unless the designer gave some assistance. Just as evolution provides animals with enough built-in reflexes to survive long enough to learn for themselves, it would be reasonable to provide an artificial intelligent agent with some initial knowledge as well as an ability to learn. After sufficient experience of its environment, the behavior of a rational agent can become effectively *independent* of its prior knowledge. Hence, the incorporation of learning allows one to design a single rational agent that will succeed in a vast variety of environments.

2.3 The Nature of Environments

Task environment

Now that we have a definition of rationality, we are almost ready to think about building rational agents. First, however, we must think about **task environments**, which are essentially the "problems" to which rational agents are the "solutions." We begin by showing how to specify a task environment, illustrating the process with a number of examples. We then show that task environments come in a variety of flavors. The nature of the task environment directly affects the appropriate design for the agent program.

2.3.1 Specifying the task environment

PEAS

In our discussion of the rationality of the simple vacuum-cleaner agent, we had to specify the performance measure, the environment, and the agent's actuators and sensors. We group all these under the heading of the **task environment**. For the acronymically minded, we call this the **PEAS** (**P**erformance, **E**nvironment, **A**ctuators, **S**ensors) description. In designing an agent, the first step must always be to specify the task environment as fully as possible.

The vacuum world was a simple example; let us consider a more complex problem: an automated taxi driver. Figure 2.4 summarizes the PEAS description for the taxi's task environment. We discuss each element in more detail in the following paragraphs.

First, what is the **performance measure** to which we would like our automated driver to aspire? Desirable qualities include getting to the correct destination; minimizing fuel consumption and wear and tear; minimizing the trip time or cost; minimizing violations of traffic laws and disturbances to other drivers; maximizing safety and passenger comfort; maximizing profits. Obviously, some of these goals conflict, so tradeoffs will be required.

Next, what is the driving **environment** that the taxi will face? Any taxi driver must deal with a variety of roads, ranging from rural lanes and urban alleys to 12-lane freeways. The roads contain other traffic, pedestrians, stray animals, road works, police cars, puddles, and potholes. The taxi must also interact with potential and actual passengers. There are also some optional choices. The taxi might need to operate in Southern California, where snow is seldom a problem, or in Alaska, where it seldom is not. It could always be driving on the right, or we might want it to be flexible enough to drive on the left when in Britain or Japan. Obviously, the more restricted the environment, the easier the design problem.

Agent Type	Performance Measure	Environment	Actuators	Sensors
Taxi driver	Safe, fast, legal, comfortable trip, maximize profits, minimize impact on other road users	Roads, other traffic, police, pedestrians, customers, weather	Steering, accelerator, brake, signal, horn, display, speech	Cameras, radar, speedometer, GPS, engine sensors, accelerometer, microphones, touchscreen

Figure 2.4 PEAS description of the task environment for an automated taxi driver.

The **actuators** for an automated taxi include those available to a human driver: control over the engine through the accelerator and control over steering and braking. In addition, it will need output to a display screen or voice synthesizer to talk back to the passengers, and perhaps some way to communicate with other vehicles, politely or otherwise.

The basic **sensors** for the taxi will include one or more video cameras so that it can see, as well as lidar and ultrasound sensors to detect distances to other cars and obstacles. To avoid speeding tickets, the taxi should have a speedometer, and to control the vehicle properly, especially on curves, it should have an accelerometer. To determine the mechanical state of the vehicle, it will need the usual array of engine, fuel, and electrical system sensors. Like many human drivers, it might want to access GPS signals so that it doesn't get lost. Finally, it will need touchscreen or voice input for the passenger to request a destination.

In Figure 2.5, we have sketched the basic PEAS elements for a number of additional agent types. Further examples appear in Exercise 2.PEAS. The examples include physical as well as virtual environments. Note that virtual task environments can be just as complex as the "real" world: for example, a **software agent** (or software robot or **softbot**) that trades on auction and reselling Web sites deals with millions of other users and billions of objects, many with real images.

Software agent

Softbot

2.3.2 Properties of task environments

The range of task environments that might arise in AI is obviously vast. We can, however, identify a fairly small number of dimensions along which task environments can be categorized. These dimensions determine, to a large extent, the appropriate agent design and the applicability of each of the principal families of techniques for agent implementation. First we list the dimensions, then we analyze several task environments to illustrate the ideas. The definitions here are informal; later chapters provide more precise statements and examples of each kind of environment.

Fully observable vs. **partially observable**: If an agent's sensors give it access to the complete state of the environment at each point in time, then we say that the task environment is fully observable. A task environment is effectively fully observable if the sensors detect all aspects that are *relevant* to the choice of action; relevance, in turn, depends on the

Fully observable

Partially observable

Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	Healthy patient, reduced costs	Patient, hospital, staff	Display of questions, tests, diagnoses, treatments	Touchscreen/voice entry of symptoms and findings
Satellite image analysis system	Correct categorization of objects, terrain	Orbiting satellite, downlink, weather	Display of scene categorization	High-resolution digital camera
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts; bins	Jointed arm and hand	Camera, tactile and joint angle sensors
Refinery controller	Purity, yield, safety	Refinery, raw materials, operators	Valves, pumps, heaters, stirrers, displays	Temperature, pressure, flow, chemical sensors
Interactive English tutor	Student's score on test	Set of students, testing agency	Display of exercises, feedback, speech	Keyboard entry, voice

Figure 2.5 Examples of agent types and their PEAS descriptions.

performance measure. Fully observable environments are convenient because the agent need not maintain any internal state to keep track of the world. An environment might be partially observable because of noisy and inaccurate sensors or because parts of the state are simply missing from the sensor data—for example, a vacuum agent with only a local dirt sensor cannot tell whether there is dirt in other squares, and an automated taxi cannot see what other drivers are thinking. If the agent has no sensors at all then the environment is **unobservable**. One might think that in such cases the agent's plight is hopeless, but, as we discuss in Chapter 4, the agent's goals may still be achievable, sometimes with certainty.

Unobservable

Single-agent

Multiagent

Single-agent vs. **multiagent**: The distinction between single-agent and multiagent environments may seem simple enough. For example, an agent solving a crossword puzzle by itself is clearly in a single-agent environment, whereas an agent playing chess is in a two-agent environment. However, there are some subtle issues. First, we have described how an entity *may* be viewed as an agent, but we have not explained which entities *must* be viewed as agents. Does an agent *A* (the taxi driver for example) have to treat an object *B* (another vehicle) as an agent, or can it be treated merely as an object behaving according to the laws of physics, analogous to waves at the beach or leaves blowing in the wind? The key distinction is whether *B*'s behavior is best described as maximizing a performance measure whose value depends on agent *A*'s behavior.

For example, in chess, the opponent entity B is trying to maximize its performance measure, which, by the rules of chess, minimizes agent A's performance measure. Thus, chess is a **competitive** multiagent environment. On the other hand, in the taxi-driving environment, avoiding collisions maximizes the performance measure of all agents, so it is a partially **cooperative** multiagent environment. It is also partially competitive because, for example, only one car can occupy a parking space.

Competitive

Cooperative

The agent-design problems in multiagent environments are often quite different from those in single-agent environments; for example, communication often emerges as a rational behavior in multiagent environments; in some competitive environments, randomized behavior is rational because it avoids the pitfalls of predictability.

Deterministic vs. **nondeterministic**. If the next state of the environment is completely determined by the current state and the action executed by the agent(s), then we say the environment is deterministic; otherwise, it is nondeterministic. In principle, an agent need not worry about uncertainty in a fully observable, deterministic environment. If the environment is partially observable, however, then it could *appear* to be nondeterministic.

Deterministic

Nondeterministic

Most real situations are so complex that it is impossible to keep track of all the unobserved aspects; for practical purposes, they must be treated as nondeterministic. Taxi driving is clearly nondeterministic in this sense, because one can never predict the behavior of traffic exactly; moreover, one's tires may blow out unexpectedly and one's engine may seize up without warning. The vacuum world as we described it is deterministic, but variations can include nondeterministic elements such as randomly appearing dirt and an unreliable suction mechanism (Exercise 2.VFIN).

One final note: the word **stochastic** is used by some as a synonym for "nondeterministic," but we make a distinction between the two terms; we say that a model of the environment is stochastic if it explicitly deals with probabilities (e.g., "there's a 25% chance of rain tomorrow") and "nondeterministic" if the possibilities are listed without being quantified (e.g., "there's a chance of rain tomorrow").

Stochastic

Episodic vs. **sequential**: In an episodic task environment, the agent's experience is divided into atomic episodes. In each episode the agent receives a percept and then performs a single action. Crucially, the next episode does not depend on the actions taken in previous episodes. Many classification tasks are episodic. For example, an agent that has to spot defective parts on an assembly line bases each decision on the current part, regardless of previous decisions; moreover, the current decision doesn't affect whether the next part is defective. In sequential environments, on the other hand, the current decision could affect all future decisions.[4] Chess and taxi driving are sequential: in both cases, short-term actions can have long-term consequences. Episodic environments are much simpler than sequential environments because the agent does not need to think ahead.

Episodic

Sequential

Static vs. **dynamic**: If the environment can change while an agent is deliberating, then we say the environment is dynamic for that agent; otherwise, it is static. Static environments are easy to deal with because the agent need not keep looking at the world while it is deciding on an action, nor need it worry about the passage of time. Dynamic environments, on the other hand, are continuously asking the agent what it wants to do; if it hasn't decided yet,

Static

Dynamic

[4] The word "sequential" is also used in computer science as the antonym of "parallel." The two meanings are largely unrelated.

that counts as deciding to do nothing. If the environment itself does not change with the passage of time but the agent's performance score does, then we say the environment is **semidynamic**. Taxi driving is clearly dynamic: the other cars and the taxi itself keep moving while the driving algorithm dithers about what to do next. Chess, when played with a clock, is semidynamic. Crossword puzzles are static.

Semidynamic

Discrete

Continuous

Discrete vs. **continuous**: The discrete/continuous distinction applies to the *state* of the environment, to the way *time* is handled, and to the *percepts* and *actions* of the agent. For example, the chess environment has a finite number of distinct states (excluding the clock). Chess also has a discrete set of percepts and actions. Taxi driving is a continuous-state and continuous-time problem: the speed and location of the taxi and of the other vehicles sweep through a range of continuous values and do so smoothly over time. Taxi-driving actions are also continuous (steering angles, etc.). Input from digital cameras is discrete, strictly speaking, but is typically treated as representing continuously varying intensities and locations.

Known

Unknown

Known vs. **unknown**: Strictly speaking, this distinction refers not to the environment itself but to the agent's (or designer's) state of knowledge about the "laws of physics" of the environment. In a known environment, the outcomes (or outcome probabilities if the environment is nondeterministic) for all actions are given. Obviously, if the environment is unknown, the agent will have to learn how it works in order to make good decisions.

The distinction between known and unknown environments is not the same as the one between fully and partially observable environments. It is quite possible for a *known* environment to be *partially* observable—for example, in solitaire card games, I know the rules but am still unable to see the cards that have not yet been turned over. Conversely, an *unknown* environment can be *fully* observable—in a new video game, the screen may show the entire game state but I still don't know what the buttons do until I try them.

As noted on page 39, the performance measure itself may be unknown, either because the designer is not sure how to write it down correctly or because the ultimate user—whose preferences matter—is not known. For example, a taxi driver usually won't know whether a new passenger prefers a leisurely or speedy journey, a cautious or aggressive driving style. A virtual personal assistant starts out knowing nothing about the personal preferences of its new owner. In such cases, the agent may learn more about the performance measure based on further interactions with the designer or user. This, in turn, suggests that the task environment is necessarily viewed as a multiagent environment.

The hardest case is *partially observable*, *multiagent*, *nondeterministic*, *sequential*, *dynamic*, *continuous*, and *unknown*. Taxi driving is hard in all these senses, except that the driver's environment is mostly known. Driving a rented car in a new country with unfamiliar geography, different traffic laws, and nervous passengers is a lot more exciting.

Figure 2.6 lists the properties of a number of familiar environments. Note that the properties are not always cut and dried. For example, we have listed the medical-diagnosis task as single-agent because the disease process in a patient is not profitably modeled as an agent; but a medical-diagnosis system might also have to deal with recalcitrant patients and skeptical staff, so the environment could have a multiagent aspect. Furthermore, medical diagnosis is episodic if one conceives of the task as selecting a diagnosis given a list of symptoms; the problem is sequential if the task can include proposing a series of tests, evaluating progress over the course of treatment, handling multiple patients, and so on.

Task Environment	Observable	Agents	Deterministic	Episodic	Static	Discrete
Crossword puzzle	Fully	Single	Deterministic	Sequential	Static	Discrete
Chess with a clock	Fully	Multi	Deterministic	Sequential	Semi	Discrete
Poker	Partially	Multi	Stochastic	Sequential	Static	Discrete
Backgammon	Fully	Multi	Stochastic	Sequential	Static	Discrete
Taxi driving	Partially	Multi	Stochastic	Sequential	Dynamic	Continuous
Medical diagnosis	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
Image analysis	Fully	Single	Deterministic	Episodic	Semi	Continuous
Part-picking robot	Partially	Single	Stochastic	Episodic	Dynamic	Continuous
Refinery controller	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
English tutor	Partially	Multi	Stochastic	Sequential	Dynamic	Discrete

Figure 2.6 Examples of task environments and their characteristics.

We have not included a "known/unknown" column because, as explained earlier, this is not strictly a property of the environment. For some environments, such as chess and poker, it is quite easy to supply the agent with full knowledge of the rules, but it is nonetheless interesting to consider how an agent might learn to play these games without such knowledge.

The code repository associated with this book (`aima.cs.berkeley.edu`) includes multiple environment implementations, together with a general-purpose environment simulator for evaluating an agent's performance. Experiments are often carried out not for a single environment but for many environments drawn from an **environment class**. For example, to evaluate a taxi driver in simulated traffic, we would want to run many simulations with different traffic, lighting, and weather conditions. We are then interested in the agent's average performance over the environment class.

Environment class

2.4 The Structure of Agents

So far we have talked about agents by describing *behavior*—the action that is performed after any given sequence of percepts. Now we must bite the bullet and talk about how the insides work. The job of AI is to design an **agent program** that implements the agent function—the mapping from percepts to actions. We assume this program will run on some sort of computing device with physical sensors and actuators—we call this the **agent architecture**:

Agent program

Agent architecture

$$agent = architecture + program\,.$$

Obviously, the program we choose has to be one that is appropriate for the architecture. If the program is going to recommend actions like *Walk*, the architecture had better have legs. The architecture might be just an ordinary PC, or it might be a robotic car with several onboard computers, cameras, and other sensors. In general, the architecture makes the percepts from the sensors available to the program, runs the program, and feeds the program's action choices to the actuators as they are generated. Most of this book is about designing agent programs, although Chapters 25 and 26 deal directly with the sensors and actuators.

```
function TABLE-DRIVEN-AGENT(percept) returns an action
  persistent: percepts, a sequence, initially empty
              table, a table of actions, indexed by percept sequences, initially fully specified

  append percept to the end of percepts
  action ← LOOKUP(percepts, table)
  return action
```

Figure 2.7 The TABLE-DRIVEN-AGENT program is invoked for each new percept and returns an action each time. It retains the complete percept sequence in memory.

2.4.1 Agent programs

The agent programs that we design in this book all have the same skeleton: they take the current percept as input from the sensors and return an action to the actuators.[5] Notice the difference between the agent program, which takes the current percept as input, and the agent function, which may depend on the entire percept history. The agent program has no choice but to take just the current percept as input because nothing more is available from the environment; if the agent's actions need to depend on the entire percept sequence, the agent will have to remember the percepts.

We describe the agent programs in the simple pseudocode language that is defined in Appendix B. (The online code repository contains implementations in real programming languages.) For example, Figure 2.7 shows a rather trivial agent program that keeps track of the percept sequence and then uses it to index into a table of actions to decide what to do. The table—an example of which is given for the vacuum world in Figure 2.3—represents explicitly the agent function that the agent program embodies. To build a rational agent in this way, we as designers must construct a table that contains the appropriate action for every possible percept sequence.

It is instructive to consider why the table-driven approach to agent construction is doomed to failure. Let $\mathcal{P}$ be the set of possible percepts and let T be the lifetime of the agent (the total number of percepts it will receive). The lookup table will contain $\sum_{t=1}^{T} |\mathcal{P}|^t$ entries. Consider the automated taxi: the visual input from a single camera (eight cameras is typical) comes in at the rate of roughly 70 megabytes per second (30 frames per second, 1080×720 pixels with 24 bits of color information). This gives a lookup table with over $10^{600{,}000{,}000{,}000}$ entries for an hour's driving. Even the lookup table for chess—a tiny, well-behaved fragment of the real world—has (it turns out) at least 10^{150} entries. In comparison, the number of atoms in the observable universe is less than 10^{80}. The daunting size of these tables means that (a) no physical agent in this universe will have the space to store the table; (b) the designer would not have time to create the table; and (c) no agent could ever learn all the right table entries from its experience.

Despite all this, TABLE-DRIVEN-AGENT *does* do what we want, assuming the table is filled in correctly: it implements the desired agent function.

[5] There are other choices for the agent program skeleton; for example, we could have the agent programs be **coroutines** that run asynchronously with the environment. Each such coroutine has an input and output port and consists of a loop that reads the input port for percepts and writes actions to the output port.

```
function REFLEX-VACUUM-AGENT([location,status]) returns an action
   if status = Dirty then return Suck
   else if location = A then return Right
   else if location = B then return Left
```

Figure 2.8 The agent program for a simple reflex agent in the two-location vacuum environment. This program implements the agent function tabulated in Figure 2.3.

The key challenge for AI is to find out how to write programs that, to the extent possible, produce rational behavior from a smallish program rather than from a vast table.

We have many examples showing that this can be done successfully in other areas: for example, the huge tables of square roots used by engineers and schoolchildren prior to the 1970s have now been replaced by a five-line program for Newton's method running on electronic calculators. The question is, can AI do for general intelligent behavior what Newton did for square roots? We believe the answer is yes.

In the remainder of this section, we outline four basic kinds of agent programs that embody the principles underlying almost all intelligent systems:

- Simple reflex agents;
- Model-based reflex agents;
- Goal-based agents; and
- Utility-based agents.

Each kind of agent program combines particular components in particular ways to generate actions. Section 2.4.6 explains in general terms how to convert all these agents into *learning agents* that can improve the performance of their components so as to generate better actions. Finally, Section 2.4.7 describes the variety of ways in which the components themselves can be represented within the agent. This variety provides a major organizing principle for the field and for the book itself.

2.4.2 Simple reflex agents

Simple reflex agent

The simplest kind of agent is the **simple reflex agent**. These agents select actions on the basis of the *current* percept, ignoring the rest of the percept history. For example, the vacuum agent whose agent function is tabulated in Figure 2.3 is a simple reflex agent, because its decision is based only on the current location and on whether that location contains dirt. An agent program for this agent is shown in Figure 2.8.

Notice that the vacuum agent program is very small indeed compared to the corresponding table. The most obvious reduction comes from ignoring the percept history, which cuts down the number of relevant percept sequences from 4^T to just 4. A further, small reduction comes from the fact that when the current square is dirty, the action does not depend on the location. Although we have written the agent program using if-then-else statements, it is simple enough that it can also be implemented as a Boolean circuit.

Simple reflex behaviors occur even in more complex environments. Imagine yourself as the driver of the automated taxi. If the car in front brakes and its brake lights come on, then you should notice this and initiate braking. In other words, some processing is done on the

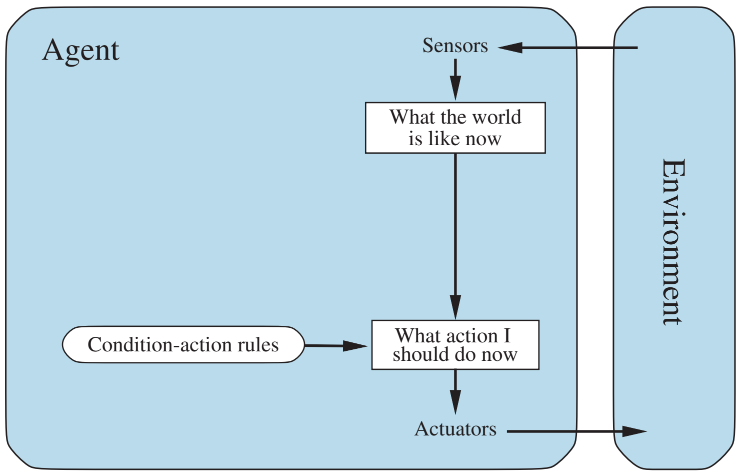


Figure 2.9 Schematic diagram of a simple reflex agent. We use rectangles to denote the current internal state of the agent's decision process, and ovals to represent the background information used in the process.

visual input to establish the condition we call "The car in front is braking." Then, this triggers some established connection in the agent program to the action "initiate braking." We call such a connection a **condition–action rule**,[6] written as

Condition–action rule

if *car-in-front-is-braking* **then** *initiate-braking*.

Humans also have many such connections, some of which are learned responses (as for driving) and some of which are innate reflexes (such as blinking when something approaches the eye). In the course of the book, we show several different ways in which such connections can be learned and implemented.

The program in Figure 2.8 is specific to one particular vacuum environment. A more general and flexible approach is first to build a general-purpose interpreter for condition–action rules and then to create rule sets for specific task environments. Figure 2.9 gives the structure of this general program in schematic form, showing how the condition–action rules allow the agent to make the connection from percept to action. Do not worry if this seems trivial; it gets more interesting shortly.

An agent program for Figure 2.9 is shown in Figure 2.10. The INTERPRET-INPUT function generates an abstracted description of the current state from the percept, and the RULE-MATCH function returns the first rule in the set of rules that matches the given state description. Note that the description in terms of "rules" and "matching" is purely conceptual; as noted above, actual implementations can be as simple as a collection of logic gates implementing a Boolean circuit. Alternatively, a "neural" circuit can be used, where the logic gates are replaced by the nonlinear units of artificial neural networks (see Chapter 21).

Simple reflex agents have the admirable property of being simple, but they are of limited intelligence. The agent in Figure 2.10 will work *only if the correct decision can be made on the basis of just the current percept—that is, only if the environment is fully observable.*

[6] Also called **situation–action rules**, **productions**, or **if–then rules**.

```
function SIMPLE-REFLEX-AGENT(percept) returns an action
  persistent: rules, a set of condition–action rules

  state ← INTERPRET-INPUT(percept)
  rule ← RULE-MATCH(state, rules)
  action ← rule.ACTION
  return action
```

Figure 2.10 A simple reflex agent. It acts according to a rule whose condition matches the current state, as defined by the percept.

Even a little bit of unobservability can cause serious trouble. For example, the braking rule given earlier assumes that the condition *car-in-front-is-braking* can be determined from the current percept—a single frame of video. This works if the car in front has a centrally mounted (and hence uniquely identifiable) brake light. Unfortunately, older models have different configurations of taillights, brake lights, and turn-signal lights, and it is not always possible to tell from a single image whether the car is braking or simply has its taillights on. A simple reflex agent driving behind such a car would either brake continuously and unnecessarily, or, worse, never brake at all.

We can see a similar problem arising in the vacuum world. Suppose that a simple reflex vacuum agent is deprived of its location sensor and has only a dirt sensor. Such an agent has just two possible percepts: [*Dirty*] and [*Clean*]. It can *Suck* in response to [*Dirty*]; what should it do in response to [*Clean*]? Moving *Left* fails (forever) if it happens to start in square *A*, and moving *Right* fails (forever) if it happens to start in square *B*. Infinite loops are often unavoidable for simple reflex agents operating in partially observable environments.

Randomization

Escape from infinite loops is possible if the agent can **randomize** its actions. For example, if the vacuum agent perceives [*Clean*], it might flip a coin to choose between *Right* and *Left*. It is easy to show that the agent will reach the other square in an average of two steps. Then, if that square is dirty, the agent will clean it and the task will be complete. Hence, a randomized simple reflex agent might outperform a deterministic simple reflex agent.

We mentioned in Section 2.3 that randomized behavior of the right kind can be rational in some multiagent environments. In single-agent environments, randomization is usually *not* rational. It is a useful trick that helps a simple reflex agent in some situations, but in most cases we can do much better with more sophisticated deterministic agents.

2.4.3 Model-based reflex agents

Internal state

The most effective way to handle partial observability is for the agent to *keep track of the part of the world it can't see now*. That is, the agent should maintain some sort of **internal state** that depends on the percept history and thereby reflects at least some of the unobserved aspects of the current state. For the braking problem, the internal state is not too extensive—just the previous frame from the camera, allowing the agent to detect when two red lights at the edge of the vehicle go on or off simultaneously. For other driving tasks such as changing lanes, the agent needs to keep track of where the other cars are if it can't see them all at once. And for any driving to be possible at all, the agent needs to keep track of where its keys are.

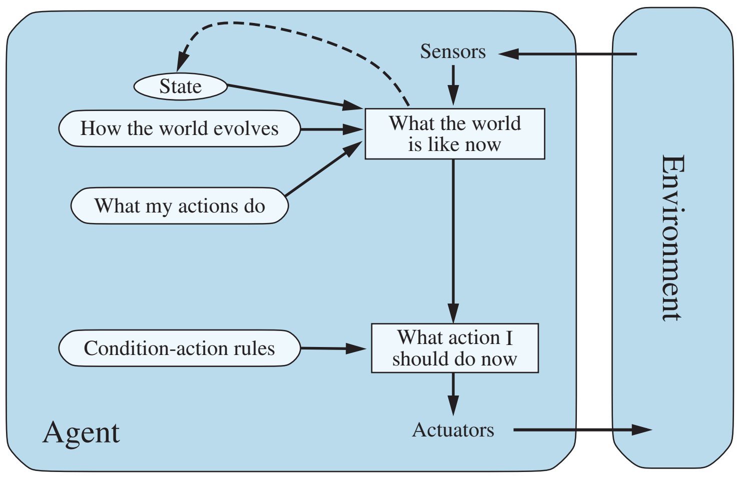


Figure 2.11 A model-based reflex agent.

Updating this internal state information as time goes by requires two kinds of knowledge to be encoded in the agent program in some form. First, we need some information about how the world changes over time, which can be divided roughly into two parts: the effects of the agent's actions and how the world evolves independently of the agent. For example, when the agent turns the steering wheel clockwise, the car turns to the right, and when it's raining the car's cameras can get wet. This knowledge about "how the world works"—whether implemented in simple Boolean circuits or in complete scientific theories—is called a **transition model** of the world.

Transition model

Second, we need some information about how the state of the world is reflected in the agent's percepts. For example, when the car in front initiates braking, one or more illuminated red regions appear in the forward-facing camera image, and, when the camera gets wet, droplet-shaped objects appear in the image partially obscuring the road. This kind of knowledge is called a **sensor model**.

Sensor model

Together, the transition model and sensor model allow an agent to keep track of the state of the world—to the extent possible given the limitations of the agent's sensors. An agent that uses such models is called a **model-based agent**.

Model-based agent

Figure 2.11 gives the structure of the model-based reflex agent with internal state, showing how the current percept is combined with the old internal state to generate the updated description of the current state, based on the agent's model of how the world works. The agent program is shown in Figure 2.12. The interesting part is the function UPDATE-STATE, which is responsible for creating the new internal state description. The details of how models and states are represented vary widely depending on the type of environment and the particular technology used in the agent design.

Regardless of the kind of representation used, it is seldom possible for the agent to determine the current state of a partially observable environment *exactly*. Instead, the box labeled "what the world is like now" (Figure 2.11) represents the agent's "best guess" (or sometimes best guesses, if the agent entertains multiple possibilities). For example, an automated taxi

```
function MODEL-BASED-REFLEX-AGENT(percept) returns an action
  persistent: state, the agent's current conception of the world state
              transition_model, a description of how the next state depends on
                  the current state and action
              sensor_model, a description of how the current world state is reflected
                  in the agent's percepts
              rules, a set of condition–action rules
              action, the most recent action, initially none

  state ← UPDATE-STATE(state, action, percept, transition_model, sensor_model)
  rule ← RULE-MATCH(state, rules)
  action ← rule.ACTION
  return action
```

Figure 2.12 A model-based reflex agent. It keeps track of the current state of the world, using an internal model. It then chooses an action in the same way as the reflex agent.

may not be able to see around the large truck that has stopped in front of it and can only guess about what may be causing the hold-up. Thus, uncertainty about the current state may be unavoidable, but the agent still has to make a decision.

2.4.4 Goal-based agents

Knowing something about the current state of the environment is not always enough to decide what to do. For example, at a road junction, the taxi can turn left, turn right, or go straight on. The correct decision depends on where the taxi is trying to get to. In other words, as well as a current state description, the agent needs some sort of **goal** information that describes situations that are desirable—for example, being at a particular destination. The agent program can combine this with the model (the same information as was used in the model-based reflex agent) to choose actions that achieve the goal. Figure 2.13 shows the goal-based agent's structure.

Goal

Sometimes goal-based action selection is straightforward—for example, when goal satisfaction results immediately from a single action. Sometimes it will be more tricky—for example, when the agent has to consider long sequences of twists and turns in order to find a way to achieve the goal. **Search** (Chapters 3 to 5) and **planning** (Chapter 11) are the subfields of AI devoted to finding action sequences that achieve the agent's goals.

Notice that decision making of this kind is fundamentally different from the condition–action rules described earlier, in that it involves consideration of the future—both "What will happen if I do such-and-such?" and "Will that make me happy?" In the reflex agent designs, this information is not explicitly represented, because the built-in rules map directly from percepts to actions. The reflex agent brakes when it sees brake lights, period. It has no idea why. A goal-based agent brakes when it sees brake lights because that's the only action that it predicts will achieve its goal of not hitting other cars.

Although the goal-based agent appears less efficient, it is more flexible because the knowledge that supports its decisions is represented explicitly and can be modified. For example, a goal-based agent's behavior can easily be changed to go to a different destination,

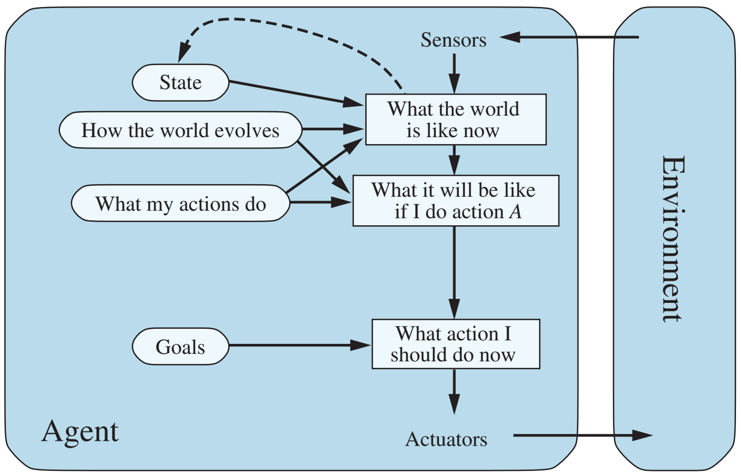

Sensors
State
How the world evolves
What my actions do
What the world is like now
What it will be like if I do action *A*
Goals
What action I should do now
Agent
Actuators
Environment

Figure 2.13 A model-based, goal-based agent. It keeps track of the world state as well as a set of goals it is trying to achieve, and chooses an action that will (eventually) lead to the achievement of its goals.

simply by specifying that destination as the goal. The reflex agent's rules for when to turn and when to go straight will work only for a single destination; they must all be replaced to go somewhere new.

2.4.5 Utility-based agents

Goals alone are not enough to generate high-quality behavior in most environments. For example, many action sequences will get the taxi to its destination (thereby achieving the goal), but some are quicker, safer, more reliable, or cheaper than others. Goals just provide a crude binary distinction between "happy" and "unhappy" states. A more general performance measure should allow a comparison of different world states according to exactly how happy they would make the agent. Because "happy" does not sound very scientific, economists and computer scientists use the term **utility** instead.[7]

Utility

We have already seen that a performance measure assigns a score to any given sequence of environment states, so it can easily distinguish between more and less desirable ways of getting to the taxi's destination. An agent's **utility function** is essentially an internalization of the performance measure. Provided that the internal utility function and the external performance measure are in agreement, an agent that chooses actions to maximize its utility will be rational according to the external performance measure.

Utility function

Let us emphasize again that this is not the *only* way to be rational—we have already seen a rational agent program for the vacuum world (Figure 2.8) that has no idea what its utility function is—but, like goal-based agents, a utility-based agent has many advantages in terms of flexibility and learning. Furthermore, in two kinds of cases, goals are inadequate but a utility-based agent can still make rational decisions. First, when there are conflicting goals, only some of which can be achieved (for example, speed and safety), the utility function specifies the appropriate tradeoff. Second, when there are several goals that the agent can

[7] The word "utility" here refers to "the quality of being useful," not to the electric company or waterworks.

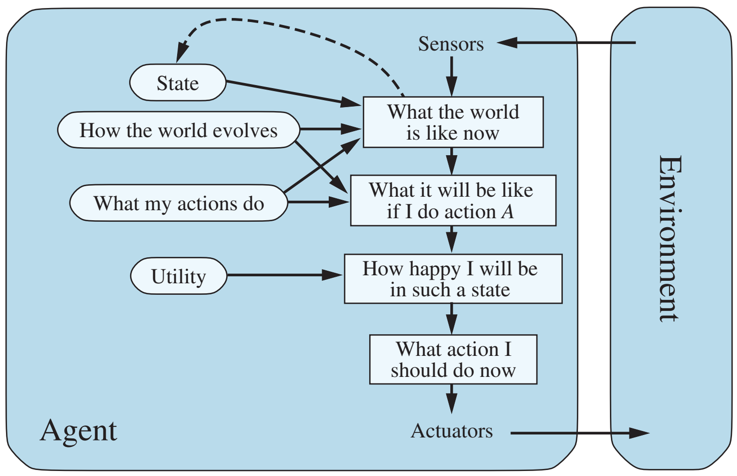

Sensors
State
How the world evolves
What my actions do
Utility
What the world is like now
What it will be like if I do action *A*
How happy I will be in such a state
What action I should do now
Actuators
Agent
Environment

Figure 2.14 A model-based, utility-based agent. It uses a model of the world, along with a utility function that measures its preferences among states of the world. Then it chooses the action that leads to the best expected utility, where expected utility is computed by averaging over all possible outcome states, weighted by the probability of the outcome.

aim for, none of which can be achieved with certainty, utility provides a way in which the likelihood of success can be weighed against the importance of the goals.

Partial observability and nondeterminism are ubiquitous in the real world, and so, therefore, is decision making under uncertainty. Technically speaking, a rational utility-based agent chooses the action that maximizes the **expected utility** of the action outcomes—that is, the utility the agent expects to derive, on average, given the probabilities and utilities of each outcome. (Appendix A defines expectation more precisely.) In Chapter 16, we show that any rational agent must behave *as if* it possesses a utility function whose expected value it tries to maximize. An agent that possesses an *explicit* utility function can make rational decisions with a general-purpose algorithm that does not depend on the specific utility function being maximized. In this way, the "global" definition of rationality—designating as rational those agent functions that have the highest performance—is turned into a "local" constraint on rational-agent designs that can be expressed in a simple program.

Expected utility

The utility-based agent structure appears in Figure 2.14. Utility-based agent programs appear in Chapters 16 and 17, where we design decision-making agents that must handle the uncertainty inherent in nondeterministic or partially observable environments. Decision making in multiagent environments is also studied in the framework of utility theory, as explained in Chapter 18.

At this point, the reader may be wondering, "Is it that simple? We just build agents that maximize expected utility, and we're done?" It's true that such agents would be intelligent, but it's not simple. A utility-based agent has to model and keep track of its environment, tasks that have involved a great deal of research on perception, representation, reasoning, and learning. The results of this research fill many of the chapters of this book. Choosing the utility-maximizing course of action is also a difficult task, requiring ingenious algorithms that fill several more chapters. Even with these algorithms, perfect rationality is usually

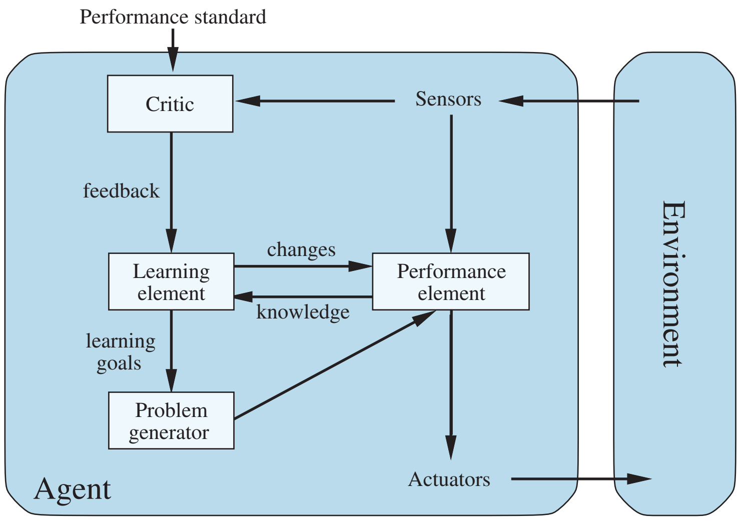


Figure 2.15 A general learning agent. The "performance element" box represents what we have previously considered to be the whole agent program. Now, the "learning element" box gets to modify that program to improve its performance.

unachievable in practice because of computational complexity, as we noted in Chapter 1. We also note that not all utility-based agents are model-based; we will see in Chapters 22 and 26 that a **model-free agent** can learn what action is best in a particular situation without ever learning exactly how that action changes the environment.

Model-free agent

Finally, all of this assumes that the designer can specify the utility function correctly; Chapters 17, 18, and 22 consider the issue of unknown utility functions in more depth.

2.4.6 Learning agents

We have described agent programs with various methods for selecting actions. We have not, so far, explained how the agent programs *come into being*. In his famous early paper, Turing (1950) considers the idea of actually programming his intelligent machines by hand. He estimates how much work this might take and concludes, "Some more expeditious method seems desirable." The method he proposes is to build learning machines and then to teach them. In many areas of AI, this is now the preferred method for creating state-of-the-art systems. Any type of agent (model-based, goal-based, utility-based, etc.) can be built as a learning agent (or not).

Learning has another advantage, as we noted earlier: it allows the agent to operate in initially unknown environments and to become more competent than its initial knowledge alone might allow. In this section, we briefly introduce the main ideas of learning agents. Throughout the book, we comment on opportunities and methods for learning in particular kinds of agents. Chapters 19–22 go into much more depth on the learning algorithms themselves.

A learning agent can be divided into four conceptual components, as shown in Figure 2.15. The most important distinction is between the **learning element**, which is responsible for making improvements, and the **performance element**, which is responsible for selecting external actions. The performance element is what we have previously considered

Learning element

Performance element

to be the entire agent: it takes in percepts and decides on actions. The learning element uses feedback from the **critic** on how the agent is doing and determines how the performance element should be modified to do better in the future.

Critic

The design of the learning element depends very much on the design of the performance element. When trying to design an agent that learns a certain capability, the first question is not "How am I going to get it to learn this?" but "What kind of performance element will my agent use to do this once it has learned how?" Given a design for the performance element, learning mechanisms can be constructed to improve every part of the agent.

The critic tells the learning element how well the agent is doing with respect to a fixed performance standard. The critic is necessary because the percepts themselves provide no indication of the agent's success. For example, a chess program could receive a percept indicating that it has checkmated its opponent, but it needs a performance standard to know that this is a good thing; the percept itself does not say so. It is important that the performance standard be fixed. Conceptually, one should think of it as being outside the agent altogether because the agent must not modify it to fit its own behavior.

The last component of the learning agent is the **problem generator**. It is responsible for suggesting actions that will lead to new and informative experiences. If the performance element had its way, it would keep doing the actions that are best, given what it knows, but if the agent is willing to explore a little and do some perhaps suboptimal actions in the short run, it might discover much better actions for the long run. The problem generator's job is to suggest these exploratory actions. This is what scientists do when they carry out experiments. Galileo did not think that dropping rocks from the top of a tower in Pisa was valuable in itself. He was not trying to break the rocks or to modify the brains of unfortunate pedestrians. His aim was to modify his own brain by identifying a better theory of the motion of objects.

Problem generator

The learning element can make changes to any of the "knowledge" components shown in the agent diagrams (Figures 2.9, 2.11, 2.13, and 2.14). The simplest cases involve learning directly from the percept sequence. Observation of pairs of successive states of the environment can allow the agent to learn "What my actions do" and "How the world evolves" in response to its actions. For example, if the automated taxi exerts a certain braking pressure when driving on a wet road, then it will soon find out how much deceleration is actually achieved, and whether it skids off the road. The problem generator might identify certain parts of the model that are in need of improvement and suggest experiments, such as trying out the brakes on different road surfaces under different conditions.

Improving the model components of a model-based agent so that they conform better with reality is almost always a good idea, regardless of the external performance standard. (In some cases, it is better from a computational point of view to have a simple but slightly inaccurate model rather than a perfect but fiendishly complex model.) Information from the external standard is needed when trying to learn a reflex component or a utility function.

For example, suppose the taxi-driving agent receives no tips from passengers who have been thoroughly shaken up during the trip. The external performance standard must inform the agent that the loss of tips is a negative contribution to its overall performance; then the agent might be able to learn that violent maneuvers do not contribute to its own utility. In a sense, the performance standard distinguishes part of the incoming percept as a **reward** (or **penalty**) that provides direct feedback on the quality of the agent's behavior. Hard-wired performance standards such as pain and hunger in animals can be understood in this way.

Reward

Penalty

More generally, *human choices* can provide information about human preferences. For example, suppose the taxi does not know that people generally don't like loud noises, and settles on the idea of blowing its horn continuously as a way of ensuring that pedestrians know it's coming. The consequent human behavior—covering ears, using bad language, and possibly cutting the wires to the horn—would provide evidence to the agent with which to update its utility function. This issue is discussed further in Chapter 22.

In summary, agents have a variety of components, and those components can be represented in many ways within the agent program, so there appears to be great variety among learning methods. There is, however, a single unifying theme. Learning in intelligent agents can be summarized as a process of modification of each component of the agent to bring the components into closer agreement with the available feedback information, thereby improving the overall performance of the agent.

2.4.7 How the components of agent programs work

We have described agent programs (in very high-level terms) as consisting of various components, whose function it is to answer questions such as: "What is the world like now?" "What action should I do now?" "What do my actions do?" The next question for a student of AI is, "How on Earth do these components work?" It takes about a thousand pages to begin to answer that question properly, but here we want to draw the reader's attention to some basic distinctions among the various ways that the components can represent the environment that the agent inhabits.

Roughly speaking, we can place the representations along an axis of increasing complexity and expressive power—atomic, factored, and structured. To illustrate these ideas, it helps to consider a particular agent component, such as the one that deals with "What my actions do." This component describes the changes that might occur in the environment as the result of taking an action, and Figure 2.16 provides schematic depictions of how those transitions might be represented.

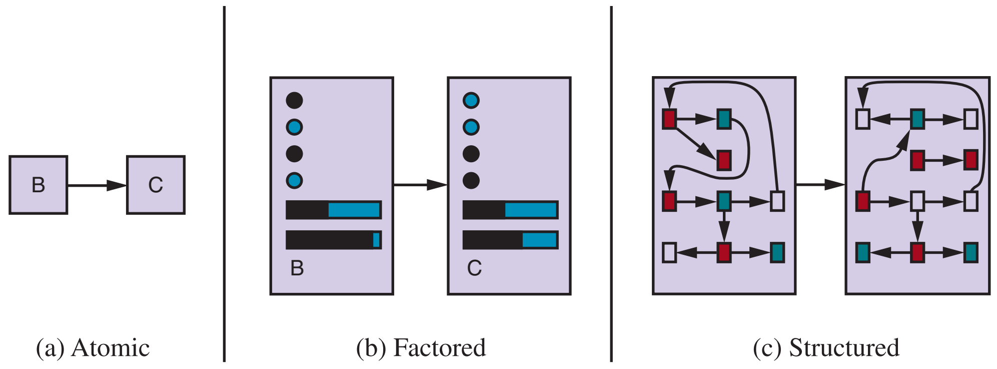


Figure 2.16 Three ways to represent states and the transitions between them. (a) Atomic representation: a state (such as B or C) is a black box with no internal structure; (b) Factored representation: a state consists of a vector of attribute values; values can be Boolean, real-valued, or one of a fixed set of symbols. (c) Structured representation: a state includes objects, each of which may have attributes of its own as well as relationships to other objects.

Atomic representation

In an **atomic representation** each state of the world is indivisible—it has no internal structure. Consider the task of finding a driving route from one end of a country to the other via some sequence of cities (we address this problem in Figure 3.1 on page 64). For the purposes of solving this problem, it may suffice to reduce the state of the world to just the name of the city we are in—a single atom of knowledge, a "black box" whose only discernible property is that of being identical to or different from another black box. The standard algorithms underlying search and game-playing (Chapters 3–5), hidden Markov models (Chapter 14), and Markov decision processes (Chapter 17) all work with atomic representations.

Factored representation

Variable

Attribute

Value

A **factored representation** splits up each state into a fixed set of **variables** or **attributes**, each of which can have a **value**. Consider a higher-fidelity description for the same driving problem, where we need to be concerned with more than just atomic location in one city or another; we might need to pay attention to how much gas is in the tank, our current GPS coordinates, whether or not the oil warning light is working, how much money we have for tolls, what station is on the radio, and so on. While two different atomic states have nothing in common—they are just different black boxes—two different factored states can share some attributes (such as being at some particular GPS location) and not others (such as having lots of gas or having no gas); this makes it much easier to work out how to turn one state into another. Many important areas of AI are based on factored representations, including constraint satisfaction algorithms (Chapter 6), propositional logic (Chapter 7), planning (Chapter 11), Bayesian networks (Chapters 12–16), and various machine learning algorithms.

Structured representation

For many purposes, we need to understand the world as having *things* in it that are *related* to each other, not just variables with values. For example, we might notice that a large truck ahead of us is reversing into the driveway of a dairy farm, but a loose cow is blocking the truck's path. A factored representation is unlikely to be pre-equipped with the attribute *TruckAheadBackingIntoDairyFarmDrivewayBlockedByLooseCow* with value *true* or *false*. Instead, we would need a **structured representation**, in which objects such as cows and trucks and their various and varying relationships can be described explicitly (see Figure 2.16(c)). Structured representations underlie relational databases and first-order logic (Chapters 8, 9, and 10), first-order probability models (Chapter 15), and much of natural language understanding (Chapters 23 and 24). In fact, much of what humans express in natural language concerns objects and their relationships.

Expressiveness

As we mentioned earlier, the axis along which atomic, factored, and structured representations lie is the axis of increasing **expressiveness**. Roughly speaking, a more expressive representation can capture, at least as concisely, everything a less expressive one can capture, plus some more. Often, the more expressive language is *much* more concise; for example, the rules of chess can be written in a page or two of a structured-representation language such as first-order logic but require thousands of pages when written in a factored-representation language such as propositional logic and around 10^{38} pages when written in an atomic language such as that of finite-state automata. On the other hand, reasoning and learning become more complex as the expressive power of the representation increases. To gain the benefits of expressive representations while avoiding their drawbacks, intelligent systems for the real world may need to operate at all points along the axis simultaneously.

Localist representation

Another axis for representation involves the mapping of concepts to locations in physical memory, whether in a computer or in a brain. If there is a one-to-one mapping between concepts and memory locations, we call that a **localist representation**. On the other hand, if the representation of a concept is spread over many memory locations, and each memory location is employed as part of the representation of multiple different concepts, we call that a **distributed representation**. Distributed representations are more robust against noise and information loss. With a localist representation, the mapping from concept to memory location is arbitrary, and if a transmission error garbles a few bits, we might confuse *Truck* with the unrelated concept *Truce*. But with a distributed representation, you can think of each concept representing a point in multidimensional space, and if you garble a few bits you move to a nearby point in that space, which will have similar meaning.

Distributed representation

Summary

This chapter has been something of a whirlwind tour of AI, which we have conceived of as the science of agent design. The major points to recall are as follows:

- An **agent** is something that perceives and acts in an environment. The **agent function** for an agent specifies the action taken by the agent in response to any percept sequence.
- The **performance measure** evaluates the behavior of the agent in an environment. A **rational agent** acts so as to maximize the expected value of the performance measure, given the percept sequence it has seen so far.
- A **task environment** specification includes the performance measure, the external environment, the actuators, and the sensors. In designing an agent, the first step must always be to specify the task environment as fully as possible.
- Task environments vary along several significant dimensions. They can be fully or partially observable, single-agent or multiagent, deterministic or nondeterministic, episodic or sequential, static or dynamic, discrete or continuous, and known or unknown.
- In cases where the performance measure is unknown or hard to specify correctly, there is a significant risk of the agent optimizing the wrong objective. In such cases the agent design should reflect uncertainty about the true objective.
- The **agent program** implements the agent function. There exists a variety of basic agent program designs reflecting the kind of information made explicit and used in the decision process. The designs vary in efficiency, compactness, and flexibility. The appropriate design of the agent program depends on the nature of the environment.
- **Simple reflex agents** respond directly to percepts, whereas **model-based reflex agents** maintain internal state to track aspects of the world that are not evident in the current percept. **Goal-based agents** act to achieve their goals, and **utility-based agents** try to maximize their own expected "happiness."
- All agents can improve their performance through **learning**.

Bibliographical and Historical Notes

The central role of action in intelligence—the notion of practical reasoning—goes back at least as far as Aristotle's *Nicomachean Ethics*. Practical reasoning was also the subject of McCarthy's influential paper "Programs with Common Sense" (1958). The fields of robotics and control theory are, by their very nature, concerned principally with physical agents. The concept of a **controller** in control theory is identical to that of an agent in AI. Perhaps surprisingly, AI has concentrated for most of its history on isolated components of agents—question-answering systems, theorem-provers, vision systems, and so on—rather than on whole agents. The discussion of agents in the text by Genesereth and Nilsson (1987) was an influential exception. The whole-agent view is now widely accepted and is a central theme in recent texts (Padgham and Winikoff, 2004; Jones, 2007; Poole and Mackworth, 2017).

Controller

Chapter 1 traced the roots of the concept of rationality in philosophy and economics. In AI, the concept was of peripheral interest until the mid-1980s, when it began to suffuse many discussions about the proper technical foundations of the field. A paper by Jon Doyle (1983) predicted that rational agent design would come to be seen as the core mission of AI, while other popular topics would spin off to form new disciplines.

Careful attention to the properties of the environment and their consequences for rational agent design is most apparent in the control theory tradition—for example, classical control systems (Dorf and Bishop, 2004; Kirk, 2004) handle fully observable, deterministic environments; stochastic optimal control (Kumar and Varaiya, 1986; Bertsekas and Shreve, 2007) handles partially observable, stochastic environments; and hybrid control (Henzinger and Sastry, 1998; Cassandras and Lygeros, 2006) deals with environments containing both discrete and continuous elements. The distinction between fully and partially observable environments is also central in the **dynamic programming** literature developed in the field of operations research (Puterman, 1994), which we discuss in Chapter 17.

Although simple reflex agents were central to behaviorist psychology (see Chapter 1), most AI researchers view them as too simple to provide much leverage. (Rosenschein (1985) and Brooks (1986) questioned this assumption; see Chapter 26.) A great deal of work has gone into finding efficient algorithms for keeping track of complex environments (Bar-Shalom *et al.*, 2001; Choset *et al.*, 2005; Simon, 2006), most of it in the probabilistic setting.

Goal-based agents are presupposed in everything from Aristotle's view of practical reasoning to McCarthy's early papers on logical AI. Shakey the Robot (Fikes and Nilsson, 1971; Nilsson, 1984) was the first robotic embodiment of a logical, goal-based agent. A full logical analysis of goal-based agents appeared in Genesereth and Nilsson (1987), and a goal-based programming methodology called agent-oriented programming was developed by Shoham (1993). The agent-based approach is now extremely popular in software engineering (Ciancarini and Wooldridge, 2001). It has also infiltrated the area of operating systems, where **autonomic computing** refers to computer systems and networks that monitor and control themselves with a perceive–act loop and machine learning methods (Kephart and Chess, 2003). Noting that a collection of agent programs designed to work well together in a true multiagent environment necessarily exhibits modularity—the programs share no internal state and communicate with each other only through the environment—it is common within the field of **multiagent systems** to design the agent program of a single agent as a collection of autonomous sub-agents. In some cases, one can even prove that the resulting system gives the same optimal solutions as a monolithic design.

Autonomic computing

The goal-based view of agents also dominates the cognitive psychology tradition in the area of problem solving, beginning with the enormously influential *Human Problem Solving* (Newell and Simon, 1972) and running through all of Newell's later work (Newell, 1990). Goals, further analyzed as *desires* (general) and *intentions* (currently pursued), are central to the influential theory of agents developed by Michael Bratman (1987).

As noted in Chapter 1, the development of utility theory as a basis for rational behavior goes back hundreds of years. In AI, early research eschewed utilities in favor of goals, with some exceptions (Feldman and Sproull, 1977). The resurgence of interest in probabilistic methods in the 1980s led to the acceptance of maximization of expected utility as the most general framework for decision making (Horvitz *et al.*, 1988). The text by Pearl (1988) was the first in AI to cover probability and utility theory in depth; its exposition of practical methods for reasoning and decision making under uncertainty was probably the single biggest factor in the rapid shift towards utility-based agents in the 1990s (see Chapter 16). The formalization of reinforcement learning within a decision-theoretic framework also contributed to this shift (Sutton, 1988). Somewhat remarkably, almost all AI research until very recently has assumed that the performance measure can be exactly and correctly specified in the form of a utility function or reward function (Hadfield-Menell *et al.*, 2017a; Russell, 2019).

The general design for learning agents portrayed in Figure 2.15 is classic in the machine learning literature (Buchanan *et al.*, 1978; Mitchell, 1997). Examples of the design, as embodied in programs, go back at least as far as Arthur Samuel's (1959, 1967) learning program for playing checkers. Learning agents are discussed in depth in Chapters 19–22.

Some early papers on agent-based approaches are collected by Huhns and Singh (1998) and Wooldridge and Rao (1999). Texts on multiagent systems provide a good introduction to many aspects of agent design (Weiss, 2000a; Wooldridge, 2009). Several conference series devoted to agents began in the 1990s, including the International Workshop on Agent Theories, Architectures, and Languages (ATAL), the International Conference on Autonomous Agents (AGENTS), and the International Conference on Multi-Agent Systems (ICMAS). In 2002, these three merged to form the International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS). From 2000 to 2012 there were annual workshops on Agent-Oriented Software Engineering (AOSE). The journal *Autonomous Agents and Multi-Agent Systems* was founded in 1998. Finally, *Dung Beetle Ecology* (Hanski and Cambefort, 1991) provides a wealth of interesting information on the behavior of dung beetles. YouTube has inspiring video recordings of their activities.

CHAPTER 3

SOLVING PROBLEMS BY SEARCHING

In which we see how an agent can look ahead to find a sequence of actions that will eventually achieve its goal.

When the correct action to take is not immediately obvious, an agent may need to to *plan ahead*: to consider a *sequence* of actions that form a path to a goal state. Such an agent is called a **problem-solving agent**, and the computational process it undertakes is called **search**.

Problem-solving agent

Search

Problem-solving agents use **atomic** representations, as described in Section 2.4.7—that is, states of the world are considered as wholes, with no internal structure visible to the problem-solving algorithms. Agents that use **factored** or **structured** representations of states are called **planning agents** and are discussed in Chapters 7 and 11.

We will cover several search algorithms. In this chapter, we consider only the simplest environments: episodic, single agent, fully observable, deterministic, static, discrete, and known. We distinguish between **informed** algorithms, in which the agent can estimate how far it is from the goal, and **uninformed** algorithms, where no such estimate is available. Chapter 4 relaxes the constraints on environments, and Chapter 5 considers multiple agents.

This chapter uses the concepts of asymptotic complexity (that is, $O(n)$ notation). Readers unfamiliar with these concepts should consult Appendix A.

3.1 Problem-Solving Agents

Imagine an agent enjoying a touring vacation in Romania. The agent wants to take in the sights, improve its Romanian, enjoy the nightlife, avoid hangovers, and so on. The decision problem is a complex one. Now, suppose the agent is currently in the city of Arad and has a nonrefundable ticket to fly out of Bucharest the following day. The agent observes street signs and sees that there are three roads leading out of Arad: one toward Sibiu, one to Timisoara, and one to Zerind. None of these are the goal, so unless the agent is familiar with the geography of Romania, it will not know which road to follow.[1]

If the agent has no additional information—that is, if the environment is **unknown**—then the agent can do no better than to execute one of the actions at random. This sad situation is discussed in Chapter 4. In this chapter, we will assume our agents always have access to information about the world, such as the map in Figure 3.1. With that information, the agent can follow this four-phase problem-solving process:

Goal formulation

- **Goal formulation**: The agent adopts the **goal** of reaching Bucharest. Goals organize behavior by limiting the objectives and hence the actions to be considered.

[1] We are assuming that most readers are in the same position and can easily imagine themselves to be as clueless as our agent. We apologize to Romanian readers who are unable to take advantage of this pedagogical device.

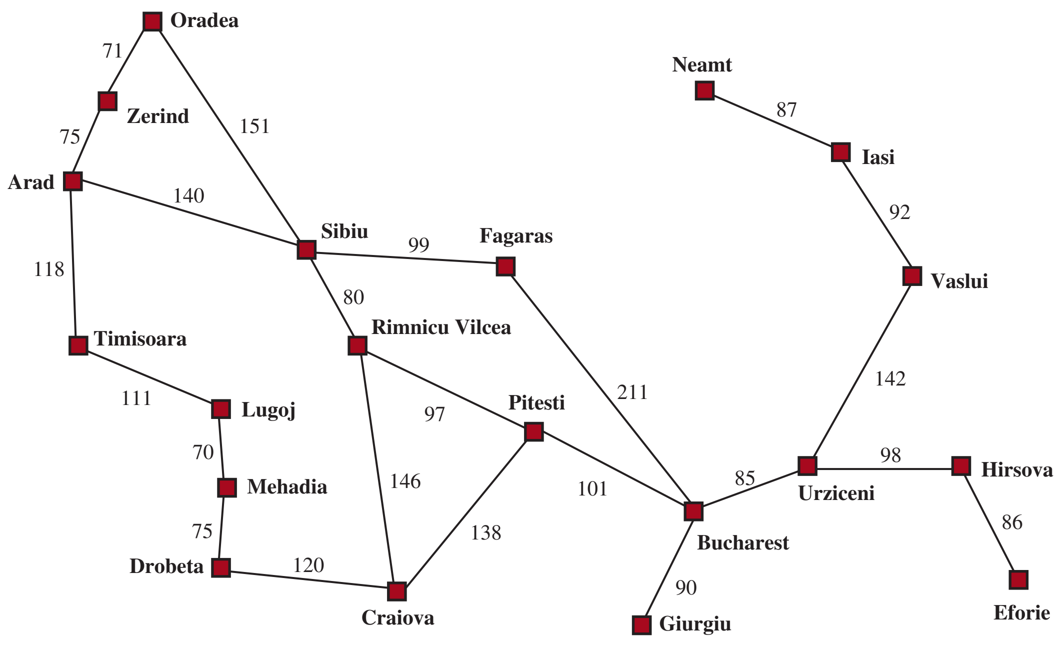


Figure 3.1 A simplified road map of part of Romania, with road distances in miles.

Problem formulation

- **Problem formulation**: The agent devises a description of the states and actions necessary to reach the goal—an abstract model of the relevant part of the world. For our agent, one good model is to consider the actions of traveling from one city to an adjacent city, and therefore the only fact about the state of the world that will change due to an action is the current city.

Search

Solution

- **Search**: Before taking any action in the real world, the agent simulates sequences of actions in its model, searching until it finds a sequence of actions that reaches the goal. Such a sequence is called a **solution**. The agent might have to simulate multiple sequences that do not reach the goal, but eventually it will find a solution (such as going from Arad to Sibiu to Fagaras to Bucharest), or it will find that no solution is possible.

Execution

- **Execution**: The agent can now execute the actions in the solution, one at a time.

It is an important property that in a fully observable, deterministic, known environment, *the solution to any problem is a fixed sequence of actions:* drive to Sibiu, then Fagaras, then Bucharest. If the model is correct, then once the agent has found a solution, it can ignore its percepts while it is executing the actions—closing its eyes, so to speak—because the solution is guaranteed to lead to the goal. Control theorists call this an **open-loop** system: ignoring the percepts breaks the loop between agent and environment. If there is a chance that the model is incorrect, or the environment is nondeterministic, then the agent would be safer using a **closed-loop** approach that monitors the percepts (see Section 4.4).

Open-loop

Closed-loop

In partially observable or nondeterministic environments, a solution would be a branching strategy that recommends different future actions depending on what percepts arrive. For example, the agent might plan to drive from Arad to Sibiu but might need a contingency plan in case it arrives in Zerind by accident or finds a sign saying "Drum Închis" (Road Closed).

3.1.1 Search problems and solutions

A search **problem** can be defined formally as follows: Problem

- A set of possible **states** that the environment can be in. We call this the **state space**. States State space
- The **initial state** that the agent starts in. For example: *Arad*. Initial state
- A set of one or more **goal states**. Sometimes there is one goal state (e.g., *Bucharest*), sometimes there is a small set of alternative goal states, and sometimes the goal is defined by a property that applies to many states (potentially an infinite number). For example, in a vacuum-cleaner world, the goal might be to have no dirt in any location, regardless of any other facts about the state. We can account for all three of these possibilities by specifying an IS-GOAL method for a problem. In this chapter we will sometimes say "the goal" for simplicity, but what we say also applies to "any one of the possible goal states." Goal states
- The **actions** available to the agent. Given a state s, ACTIONS(s) returns a finite[2] set of actions that can be executed in s. We say that each of these actions is **applicable** in s. An example: Action Applicable

$$\textsc{Actions}(\mathit{Arad}) = \{\mathit{ToSibiu}, \mathit{ToTimisoara}, \mathit{ToZerind}\}\,.$$

- A **transition model**, which describes what each action does. RESULT(s,a) returns the state that results from doing action a in state s. For example, Transition model

$$\textsc{Result}(\mathit{Arad}, \mathit{ToZerind}) = \mathit{Zerind}\,.$$

- An **action cost function**, denoted by ACTION-COST(s,a,s') when we are programming or $c(s,a,s')$ when we are doing math, that gives the numeric cost of applying action a in state s to reach state s'. A problem-solving agent should use a cost function that reflects its own performance measure; for example, for route-finding agents, the cost of an action might be the length in miles (as seen in Figure 3.1), or it might be the time it takes to complete the action. Action cost function

A sequence of actions forms a **path**, and a **solution** is a path from the initial state to a goal state. We assume that action costs are additive; that is, the total cost of a path is the sum of the individual action costs. An **optimal solution** has the lowest path cost among all solutions. In this chapter, we assume that all action costs will be positive, to avoid certain complications.[3] Path Optimal solution

The state space can be represented as a **graph** in which the vertices are states and the directed edges between them are actions. The map of Romania shown in Figure 3.1 is such a graph, where each road indicates two actions, one in each direction. Graph

[2] For problems with an infinite number of actions we would need techniques that go beyond this chapter.

[3] In any problem with a cycle of net negative cost, the cost-optimal solution is to go around that cycle an infinite number of times. The Bellman–Ford and Floyd–Warshall algorithms (not covered here) handle negative-cost actions, as long as there are no negative cycles. It is easy to accommodate zero-cost actions, as long as the number of consecutive zero-cost actions is bounded. For example, we might have a robot where there is a cost to move, but zero cost to rotate 90^o; the algorithms in this chapter can handle this as long as no more than three consecutive 90^o turns are allowed. There is also a complication with problems that have an infinite number of arbitrarily small action costs. Consider a version of Zeno's paradox where there is an action to move half way to the goal, at a cost of half of the previous move. This problem has no solution with a finite number of actions, but to prevent a search from taking an unbounded number of actions without quite reaching the goal, we can require that all action costs be at least ϵ, for some small positive value ϵ.

3.1.2 Formulating problems

Our formulation of the problem of getting to Bucharest is a **model**—an abstract mathematical description—and not the real thing. Compare the simple atomic state description *Arad* to an actual cross-country trip, where the state of the world includes so many things: the traveling companions, the current radio program, the scenery out of the window, the proximity of law enforcement officers, the distance to the next rest stop, the condition of the road, the weather, the traffic, and so on. All these considerations are left out of our model because they are irrelevant to the problem of finding a route to Bucharest.

Abstraction

The process of removing detail from a representation is called **abstraction**. A good problem formulation has the right level of detail. If the actions were at the level of "move the right foot forward a centimeter" or "turn the steering wheel one degree left," the agent would probably never find its way out of the parking lot, let alone to Bucharest.

Level of abstraction

Can we be more precise about the appropriate **level of abstraction**? Think of the abstract states and actions we have chosen as corresponding to large sets of detailed world states and detailed action sequences. Now consider a solution to the abstract problem: for example, the path from Arad to Sibiu to Rimnicu Vilcea to Pitesti to Bucharest. This abstract solution corresponds to a large number of more detailed paths. For example, we could drive with the radio on between Sibiu and Rimnicu Vilcea, and then switch it off for the rest of the trip.

The abstraction is *valid* if we can elaborate any abstract solution into a solution in the more detailed world; a sufficient condition is that for every detailed state that is "in Arad," there is a detailed path to some state that is "in Sibiu," and so on.[4] The abstraction is *useful* if carrying out each of the actions in the solution is easier than the original problem; in our case, the action "drive from Arad to Sibiu" can be carried out without further search or planning by a driver with average skill. The choice of a good abstraction thus involves removing as much detail as possible while retaining validity and ensuring that the abstract actions are easy to carry out. Were it not for the ability to construct useful abstractions, intelligent agents would be completely swamped by the real world.

3.2 Example Problems

The problem-solving approach has been applied to a vast array of task environments. We list some of the best known here, distinguishing between *standardized* and *real-world* problems.

Standardized problem

A **standardized problem** is intended to illustrate or exercise various problem-solving methods. It can be given a concise, exact description and hence is suitable as a benchmark for researchers to compare the performance of algorithms.

Real-world problem

A **real-world problem**, such as robot navigation, is one whose solutions people actually use, and whose formulation is idiosyncratic, not standardized, because, for example, each robot has different sensors that produce different data.

3.2.1 Standardized problems

Grid world

A **grid world** problem is a two-dimensional rectangular array of square cells in which agents can move from cell to cell. Typically the agent can move to any obstacle-free adjacent cell—horizontally or vertically and in some problems diagonally. Cells can contain objects, which

[4] See Section 11.4.

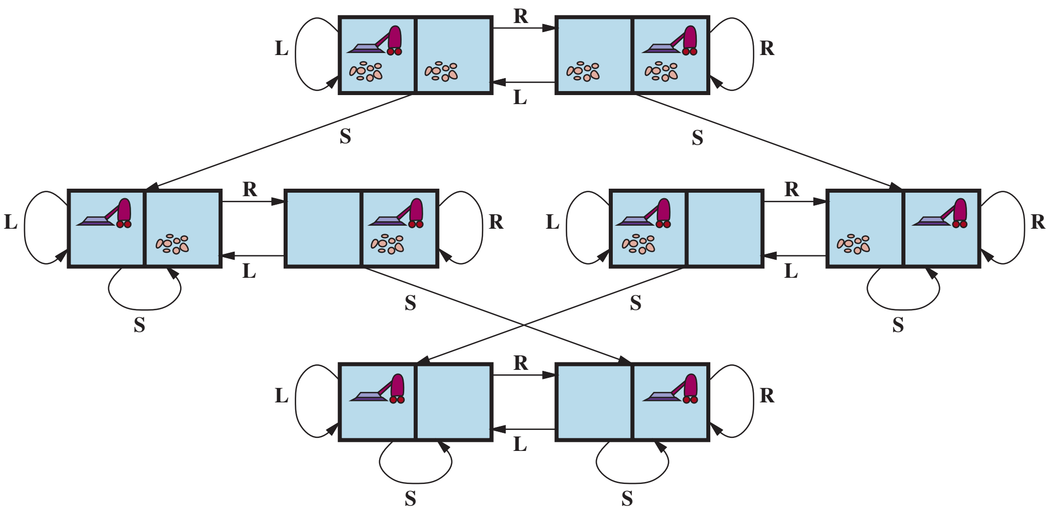


Figure 3.2 The state-space graph for the two-cell vacuum world. There are 8 states and three actions for each state: L = *Left*, R = *Right*, S = *Suck*.

the agent can pick up, push, or otherwise act upon; a wall or other impassible obstacle in a cell prevents an agent from moving into that cell. The **vacuum world** from Section 2.1 can be formulated as a grid world problem as follows:

- **States**: A state of the world says which objects are in which cells. For the vacuum world, the objects are the agent and any dirt. In the simple two-cell version, the agent can be in either of the two cells, and each call can either contain dirt or not, so there are $2 \cdot 2 \cdot 2 = 8$ states (see Figure 3.2). In general, a vacuum environment with n cells has $n \cdot 2^n$ states.
- **Initial state**: Any state can be designated as the initial state.
- **Actions**: In the two-cell world we defined three actions: *Suck*, move *Left*, and move *Right*. In a two-dimensional multi-cell world we need more movement actions. We could add *Upward* and *Downward*, giving us four **absolute** movement actions, or we could switch to **egocentric actions**, defined relative to the viewpoint of the agent—for example, *Forward*, *Backward*, *TurnRight*, and *TurnLeft*.
- **Transition model**: *Suck* removes any dirt from the agent's cell; *Forward* moves the agent ahead one cell in the direction it is facing, unless it hits a wall, in which case the action has no effect. *Backward* moves the agent in the opposite direction, while *TurnRight* and *TurnLeft* change the direction it is facing by 90°.
- **Goal states**: The states in which every cell is clean.
- **Action cost**: Each action costs 1.

Sokoban puzzle

Another type of grid world is the **sokoban puzzle**, in which the agent's goal is to push a number of boxes, scattered about the grid, to designated storage locations. There can be at most one box per cell. When an agent moves forward into a cell containing a box and there is an empty cell on the other side of the box, then both the box and the agent move forward.

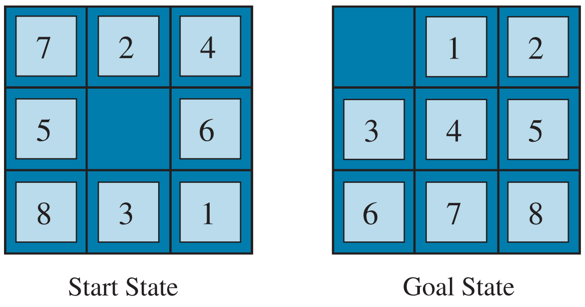


Figure 3.3 A typical instance of the 8-puzzle.

The agent can't push a box into another box or a wall. For a world with n non-obstacle cells and b boxes, there are $n \times n!/(b!(n-b)!)$ states; for example on an 8×8 grid with a dozen boxes, there are over 200 trillion states.

Sliding-tile puzzle

In a **sliding-tile puzzle**, a number of tiles (sometimes called blocks or pieces) are arranged in a grid with one or more blank spaces so that some of the tiles can slide into the blank space. One variant is the Rush Hour puzzle, in which cars and trucks slide around a 6×6 grid in an attempt to free a car from the traffic jam. Perhaps the best-known variant is the **8-puzzle** (see Figure 3.3), which consists of a 3×3 grid with eight numbered tiles and one blank space, and the **15-puzzle** on a 4×4 grid. The object is to reach a specified goal state, such as the one shown on the right of the figure. The standard formulation of the 8 puzzle is as follows:

8-puzzle

15-puzzle

- **States**: A state description specifies the location of each of the tiles.
- **Initial state**: Any state can be designated as the initial state. Note that a parity property partitions the state space—any given goal can be reached from exactly half of the possible initial states (see Exercise 3.PART).
- **Actions**: While in the physical world it is a tile that slides, the simplest way of describing an action is to think of the blank space moving *Left*, *Right*, *Up*, or *Down*. If the blank is at an edge or corner then not all actions will be applicable.
- **Transition model**: Maps a state and action to a resulting state; for example, if we apply *Left* to the start state in Figure 3.3, the resulting state has the 5 and the blank switched.
- **Goal state**: Although any state could be the goal, we typically specify a state with the numbers in order, as in Figure 3.3.
- **Action cost**: Each action costs 1.

Note that every problem formulation involves abstractions. The 8-puzzle actions are abstracted to their beginning and final states, ignoring the intermediate locations where the tile is sliding. We have abstracted away actions such as shaking the board when tiles get stuck and ruled out extracting the tiles with a knife and putting them back again. We are left with a description of the rules, avoiding all the details of physical manipulations.

Our final standardized problem was devised by Donald Knuth (1964) and illustrates how infinite state spaces can arise. Knuth conjectured that starting with the number 4, a sequence of square root, floor, and factorial operations can reach any desired positive integer. For example, we can reach 5 from 4 as follows:

$$\left\lfloor \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} \right\rfloor = 5\,.$$

The problem definition is simple:

- **States**: Positive real numbers.
- **Initial state**: 4.
- **Actions**: Apply square root, floor, or factorial operation (factorial for integers only).
- **Transition model**: As given by the mathematical definitions of the operations.
- **Goal state**: The desired positive integer.
- **Action cost**: Each action costs 1.

The state space for this problem is infinite: for any integer greater than 2 the factorial operator will always yield a larger integer. The problem is interesting because it explores very large numbers: the shortest path to 5 goes through (4!)! = 620,448,401,733,239,439,360,000. Infinite state spaces arise frequently in tasks involving the generation of mathematical expressions, circuits, proofs, programs, and other recursively defined objects.

3.2.2 Real-world problems

We have already seen how the **route-finding problem** is defined in terms of specified locations and transitions along edges between them. Route-finding algorithms are used in a variety of applications. Some, such as Web sites and in-car systems that provide driving directions, are relatively straightforward extensions of the Romania example. (The main complications are varying costs due to traffic-dependent delays, and rerouting due to road closures.) Others, such as routing video streams in computer networks, military operations planning, and airline travel-planning systems, involve much more complex specifications. Consider the airline travel problems that must be solved by a travel-planning Web site:

- **States**: Each state obviously includes a location (e.g., an airport) and the current time. Furthermore, because the cost of an action (a flight segment) may depend on previous segments, their fare bases, and their status as domestic or international, the state must record extra information about these “historical” aspects.
- **Initial state**: The user’s home airport.
- **Actions**: Take any flight from the current location, in any seat class, leaving after the current time, leaving enough time for within-airport transfer if needed.
- **Transition model**: The state resulting from taking a flight will have the flight’s destination as the new location and the flight’s arrival time as the new time.
- **Goal state**: A destination city. Sometimes the goal can be more complex, such as “arrive at the destination on a nonstop flight.”
- **Action cost**: A combination of monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer reward points, and so on.

Commercial travel advice systems use a problem formulation of this kind, with many additional complications to handle the airlines' byzantine fare structures. Any seasoned traveler knows, however, that not all air travel goes according to plan. A really good system should include contingency plans—what happens if this flight is delayed and the connection is missed?

Touring problem

Touring problems describe a set of locations that must be visited, rather than a single goal destination. The **traveling salesperson problem (TSP)** is a touring problem in which every city on a map must be visited. The aim is to find a tour with cost $< C$ (or in the optimization version, to find a tour with the lowest cost possible). An enormous amount of effort has been expended to improve the capabilities of TSP algorithms. The algorithms can also be extended to handle fleets of vehicles. For example, a search and optimization algorithm for routing school buses in Boston saved $5 million, cut traffic and air pollution, and saved time for drivers and students (Bertsimas *et al.*, 2019). In addition to planning trips, search algorithms have been used for tasks such as planning the movements of automatic circuit-board drills and of stocking machines on shop floors.

Traveling salesperson problem (TSP)

VLSI layout

A **VLSI layout** problem requires positioning millions of components and connections on a chip to minimize area, minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield. The layout problem comes after the logical design phase and is usually split into two parts: **cell layout** and **channel routing**. In cell layout, the primitive components of the circuit are grouped into cells, each of which performs some recognized function. Each cell has a fixed footprint (size and shape) and requires a certain number of connections to each of the other cells. The aim is to place the cells on the chip so that they do not overlap and so that there is room for the connecting wires to be placed between the cells. Channel routing finds a specific route for each wire through the gaps between the cells. These search problems are extremely complex, but definitely worth solving.

Robot navigation

Robot navigation is a generalization of the route-finding problem described earlier. Rather than following distinct paths (such as the roads in Romania), a robot can roam around, in effect making its own paths. For a circular robot moving on a flat surface, the space is essentially two-dimensional. When the robot has arms and legs that must also be controlled, the search space becomes many-dimensional—one dimension for each joint angle. Advanced techniques are required just to make the essentially continuous search space finite (see Chapter 26). In addition to the complexity of the problem, real robots must also deal with errors in their sensor readings and motor controls, with partial observability, and with other agents that might alter the environment.

Automatic assembly sequencing

Automatic assembly sequencing of complex objects (such as electric motors) by a robot has been standard industry practice since the 1970s. Algorithms first find a feasible assembly sequence and then work to optimize the process. Minimizing the amount of manual human labor on the assembly line can produce significant savings in time and cost. In assembly problems, the aim is to find an order in which to assemble the parts of some object. If the wrong order is chosen, there will be no way to add some part later in the sequence without undoing some of the work already done. Checking an action in the sequence for feasibility is a difficult geometrical search problem closely related to robot navigation. Thus, the generation of legal actions is the expensive part of assembly sequencing. Any practical algorithm must avoid exploring all but a tiny fraction of the state space. One important assembly problem is **protein design**, in which the goal is to find a sequence of amino acids that will fold into a three-dimensional protein with the right properties to cure some disease.

Protein design

3.3 Search Algorithms

A **search algorithm** takes a search problem as input and returns a solution, or an indication of failure. In this chapter we consider algorithms that superimpose a **search tree** over the state-space graph, forming various paths from the initial state, trying to find a path that reaches a goal state. Each **node** in the search tree corresponds to a state in the state space and the edges in the search tree correspond to actions. The root of the tree corresponds to the initial state of the problem.

Search algorithm

Node

It is important to understand the distinction between the state space and the search tree. The state space describes the (possibly infinite) set of states in the world, and the actions that allow transitions from one state to another. The search tree describes paths between these states, reaching towards the goal. The search tree may have multiple paths to (and thus multiple nodes for) any given state, but each node in the tree has a unique path back to the root (as in all trees).

Figure 3.4 shows the first few steps in finding a path from Arad to Bucharest. The root node of the search tree is at the initial state, *Arad*. We can **expand** the node, by considering

Expand

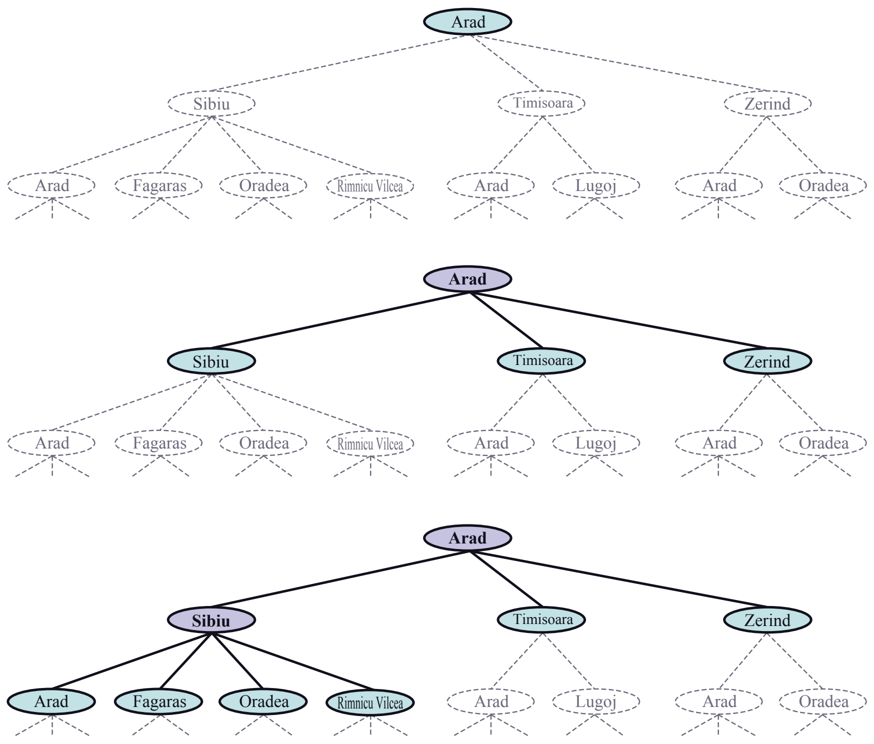


Figure 3.4 Three partial search trees for finding a route from Arad to Bucharest. Nodes that have been *expanded* are lavender with bold letters; nodes on the frontier that have been *generated* but not yet expanded are in green; the set of states corresponding to these two types of nodes are said to have been *reached*. Nodes that could be generated next are shown in faint dashed lines. Notice in the bottom tree there is a cycle from Arad to Sibiu to Arad; that can't be an optimal path, so search should not continue from there.

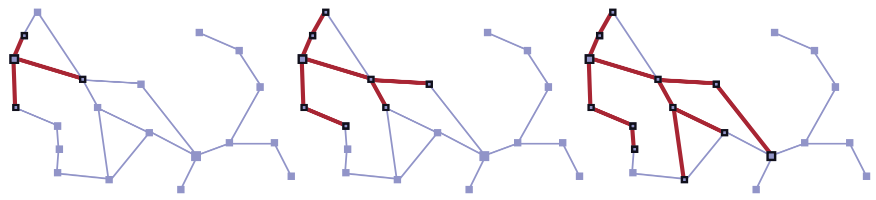


Figure 3.5 A sequence of search trees generated by a graph search on the Romania problem of Figure 3.1. At each stage, we have expanded every node on the frontier, extending every path with all applicable actions that don't result in a state that has already been reached. Notice that at the third stage, the topmost city (Oradea) has two successors, both of which have already been reached by other paths, so no paths are extended from Oradea.

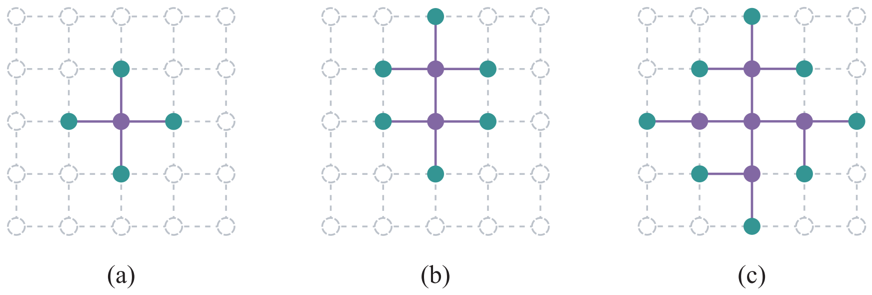


Figure 3.6 The separation property of graph search, illustrated on a rectangular-grid problem. The frontier (green) separates the interior (lavender) from the exterior (faint dashed). The frontier is the set of nodes (and corresponding states) that have been reached but not yet expanded; the interior is the set of nodes (and corresponding states) that have been expanded; and the exterior is the set of states that have not been reached. In (a), just the root has been expanded. In (b), the top frontier node is expanded. In (c), the remaining successors of the root are expanded in clockwise order.

the available ACTIONS for that state, using the RESULT function to see where those actions lead to, and **generating** a new node (called a **child node** or **successor node**) for each of the resulting states. Each child node has *Arad* as its **parent node**.

Generating

Child node

Successor node

Parent node

Now we must choose which of these three child nodes to consider next. This is the essence of search—following up one option now and putting the others aside for later. Suppose we choose to expand Sibiu first. Figure 3.4 (bottom) shows the result: a set of 6 unexpanded nodes (outlined in bold). We call this the **frontier** of the search tree. We say that any state that has had a node generated for it has been **reached** (whether or not that node has been expanded).[5] Figure 3.5 shows the search tree superimposed on the state-space graph.

Frontier

Reached

Note that the frontier **separates** two regions of the state-space graph: an interior region where every state has been expanded, and an exterior region of states that have not yet been reached. This property is illustrated in Figure 3.6.

Separator

[5] Some authors call the frontier the **open list**, which is both geographically less evocative and computationally less appropriate, because a queue is more efficient than a list here. Those authors use the term **closed list** to refer to the set of previously expanded nodes, which in our terminology would be the *reached* nodes minus the *frontier*.

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s′ ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s′)
    yield NODE(STATE=s′, PARENT=node, ACTION=action, PATH-COST=cost)
```

Figure 3.7 The best-first search algorithm, and the function for expanding a node. The data structures used here are described in Section 3.3.2. See Appendix B for **yield**.

3.3.1 Best-first search

How do we decide which node from the frontier to expand next? A very general approach is called **best-first search**, in which we choose a node, n, with minimum value of some **evaluation function**, $f(n)$. Figure 3.7 shows the algorithm. On each iteration we choose a node on the frontier with minimum $f(n)$ value, return it if its state is a goal state, and otherwise apply EXPAND to generate child nodes. Each child node is added to the frontier if it has not been reached before, or is re-added if it is now being reached with a path that has a lower path cost than any previous path. The algorithm returns either an indication of failure, or a node that represents a path to a goal. By employing different $f(n)$ functions, we get different specific algorithms, which this chapter will cover.

Best-first search

Evaluation function

3.3.2 Search data structures

Search algorithms require a data structure to keep track of the search tree. A **node** in the tree is represented by a data structure with four components:

- *node*.STATE: the state to which the node corresponds;
- *node*.PARENT: the node in the tree that generated this node;
- *node*.ACTION: the action that was applied to the parent's state to generate this node;
- *node*.PATH-COST: the total cost of the path from the initial state to this node. In mathematical formulas, we use $g(node)$ as a synonym for PATH-COST.

Following the PARENT pointers back from a node allows us to recover the states and actions along the path to that node. Doing this from a goal node gives us the solution.

Queue

We need a data structure to store the **frontier**. The appropriate choice is a **queue** of some kind, because the operations on a frontier are:

- Is-Empty(*frontier*) returns true only if there are no nodes in the frontier.
- Pop(*frontier*) removes the top node from the frontier and returns it.
- Top(*frontier*) returns (but does not remove) the top node of the frontier.
- Add(*node*, *frontier*) inserts node into its proper place in the queue.

Three kinds of queues are used in search algorithms:

Priority queue

- A **priority queue** first pops the node with the minimum cost according to some evaluation function, f. It is used in best-first search.

FIFO queue

- A **FIFO queue** or first-in-first-out queue first pops the node that was added to the queue first; we shall see it is used in breadth-first search.

LIFO queue

Stack

- A **LIFO queue** or last-in-first-out queue (also known as a **stack**) pops first the most recently added node; we shall see it is used in depth-first search.

The reached states can be stored as a lookup table (e.g. a hash table) where each key is a state and each value is the node for that state.

3.3.3 Redundant paths

Repeated state

Cycle

Loopy path

The search tree shown in Figure 3.4 (bottom) includes a path from Arad to Sibiu and back to Arad again. We say that *Arad* is a **repeated state** in the search tree, generated in this case by a **cycle** (also known as a **loopy path**). So even though the state space has only 20 states, the complete search tree is *infinite* because there is no limit to how often one can traverse a loop.

Redundant path

A cycle is a special case of a **redundant path**. For example, we can get to Sibiu via the path Arad–Sibiu (140 miles long) or the path Arad–Zerind–Oradea–Sibiu (297 miles long). This second path is redundant—it's just a worse way to get to the same state—and need not be considered in our quest for optimal paths.

Consider an agent in a 10×10 grid world, with the ability to move to any of 8 adjacent squares. If there are no obstacles, the agent can reach any of the 100 squares in 9 moves or fewer. But the number of paths of length 9 is almost 8^9 (a bit less because of the edges of the grid), or more than 100 million. In other words, the average cell can be reached by over a million redundant paths of length 9, and if we eliminate redundant paths, we can complete a search roughly a million times faster. As the saying goes, *algorithms that cannot remember the past are doomed to repeat it.* There are three approaches to this issue.

First, we can remember all previously reached states (as best-first search does), allowing us to detect all redundant paths, and keep only the best path to each state. This is appropriate for state spaces where there are many redundant paths, and is the preferred choice when the table of reached states will fit in memory.

Graph search

Tree-like search

Second, we can not worry about repeating the past. There are some problem formulations where it is rare or impossible for two paths to reach the same state. An example would be an assembly problem where each action adds a part to an evolving assemblage, and there is an ordering of parts so that it is possible to add A and then B, but not B and then A. For those problems, we could save memory space if we *don't* track reached states and we don't check for redundant paths. We call a search algorithm a **graph search** if it checks for redundant paths and a **tree-like search**[6] if it does not check. The Best-First-Search algorithm in

Figure 3.7 is a graph search algorithm; if we remove all references to *reached* we get a tree-like search that uses less memory but will examine redundant paths to the same state, and thus will run slower.

Third, we can compromise and check for cycles, but not for redundant paths in general. Since each node has a chain of parent pointers, we can check for cycles with no need for additional memory by following up the chain of parents to see if the state at the end of the path has appeared earlier in the path. Some implementations follow this chain all the way up, and thus eliminate all cycles; other implementations follow only a few links (e.g., to the parent, grandparent, and great-grandparent), and thus take only a constant amount of time, while eliminating all short cycles (and relying on other mechanisms to deal with long cycles).

3.3.4 Measuring problem-solving performance

Before we get into the design of various search algorithms, we will consider the criteria used to choose among them. We can evaluate an algorithm's performance in four ways:

- **Completeness**: Is the algorithm guaranteed to find a solution when there is one, and to correctly report failure when there is not? Completeness
- **Cost optimality**: Does it find a solution with the lowest path cost of all solutions?[7] Cost optimality
- **Time complexity**: How long does it take to find a solution? This can be measured in seconds, or more abstractly by the number of states and actions considered. Time complexity
- **Space complexity**: How much memory is needed to perform the search? Space complexity

To understand completeness, consider a search problem with a single goal. That goal could be anywhere in the state space; therefore a complete algorithm must be capable of systematically exploring every state that is reachable from the initial state. In finite state spaces that is straightforward to achieve: as long as we keep track of paths and cut off ones that are cycles (e.g. Arad to Sibiu to Arad), eventually we will reach every reachable state.

In infinite state spaces, more care is necessary. For example, an algorithm that repeatedly applied the "factorial" operator in Knuth's "4" problem would follow an infinite path from 4 to 4! to (4!)!, and so on. Similarly, on an infinite grid with no obstacles, repeatedly moving forward in a straight line also follows an infinite path of new states. In both cases the algorithm never returns to a state it has reached before, but is incomplete because wide expanses of the state space are never reached.

Systematic

To be complete, a search algorithm must be **systematic** in the way it explores an infinite state space, making sure it can eventually reach any state that is connected to the initial state. For example, on the infinite grid, one kind of systematic search is a spiral path that covers all the cells that are s steps from the origin before moving out to cells that are $s+1$ steps away. Unfortunately, in an infinite state space with no solution, a sound algorithm needs to keep searching forever; it can't terminate because it can't know if the next state will be a goal.

Time and space complexity are considered with respect to some measure of the problem difficulty. In theoretical computer science, the typical measure is the size of the state-space graph, $|V|+|E|$, where $|V|$ is the number of vertices (state nodes) of the graph and $|E|$ is

[6] We say "tree-like search" because the state space is still the same graph no matter how we search it; we are just choosing to treat it *as if* it were a tree, with only one path from each node back to the root.

[7] Some authors use the term "admissibility" for the property of finding the lowest-cost solution, and some use just "optimality," but that can be confused with other types of optimality.

the number of edges (distinct state/action pairs). This is appropriate when the graph is an explicit data structure, such as the map of Romania. But in many AI problems, the graph is represented only *implicitly* by the initial state, actions, and transition model. For an implicit state space, complexity can be measured in terms of d, the **depth** or number of actions in an optimal solution; m, the maximum number of actions in any path; and b, the **branching factor** or number of successors of a node that need to be considered.

Depth

Branching factor

3.4 Uninformed Search Strategies

An uninformed search algorithm is given no clue about how close a state is to the goal(s). For example, consider our agent in Arad with the goal of reaching Bucharest. An uninformed agent with no knowledge of Romanian geography has no clue whether going to Zerind or Sibiu is a better first step. In contrast, an informed agent (Section 3.5) who knows the location of each city knows that Sibiu is much closer to Bucharest and thus more likely to be on the shortest path.

3.4.1 Breadth-first search

Breadth-first search

When all actions have the same cost, an appropriate strategy is **breadth-first search**, in which the root node is expanded first, then all the successors of the root node are expanded next, then *their* successors, and so on. This is a systematic search strategy that is therefore complete even on infinite state spaces. We could implement breadth-first search as a call to BEST-FIRST-SEARCH where the evaluation function $f(n)$ is the depth of the node—that is, the number of actions it takes to reach the node.

However, we can get additional efficiency with a couple of tricks. A first-in-first-out queue will be faster than a priority queue, and will give us the correct order of nodes: new nodes (which are always deeper than their parents) go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first. In addition, *reached* can be a set of states rather than a mapping from states to nodes, because once we've reached a state, we can never find a better path to the state. That also means we can do an **early goal test**, checking whether a node is a solution as soon as it is *generated*, rather than the **late goal test** that best-first search uses, waiting until a node is popped off the queue. Figure 3.8 shows the progress of a breadth-first search on a binary tree, and Figure 3.9 shows the algorithm with the early-goal efficiency enhancements.

Early goal test

Late goal test

Breadth-first search always finds a solution with a minimal number of actions, because when it is generating nodes at depth d, it has already generated all the nodes at depth $d-1$, so if one of them were a solution, it would have been found. That means it is cost-optimal

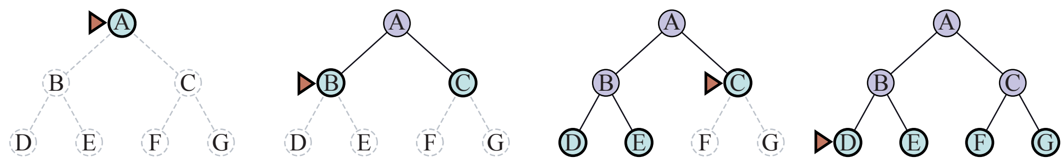


Figure 3.8 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by the triangular marker.

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure

function UNIFORM-COST-SEARCH(problem) returns a solution node, or failure
  return BEST-FIRST-SEARCH(problem, PATH-COST)
```

Figure 3.9 Breadth-first search and uniform-cost search algorithms.

for problems where all actions have the same cost, but not for problems that don't have that property. It is complete in either case. In terms of time and space, imagine searching a uniform tree where every state has b successors. The root of the search tree generates b nodes, each of which generates b more nodes, for a total of b^2 at the second level. Each of these generates b more nodes, yielding b^3 nodes at the third level, and so on. Now suppose that the solution is at depth d. Then the total number of nodes generated is

$$1+b+b^2+b^3+\cdots+b^d = O(b^d)$$

All the nodes remain in memory, so both time and space complexity are $O(b^d)$. Exponential bounds like that are scary. As a typical real-world example, consider a problem with branching factor $b = 10$, processing speed 1 million nodes/second, and memory requirements of 1 Kbyte/node. A search to depth $d = 10$ would take less than 3 hours, but would require 10 terabytes of memory. *The memory requirements are a bigger problem for breadth-first search than the execution time.* But time is still an important factor. At depth $d = 14$, even with infinite memory, the search would take 3.5 years. In general, *exponential-complexity search problems cannot be solved by uninformed search for any but the smallest instances.*

3.4.2 Dijkstra's algorithm or uniform-cost search

When actions have different costs, an obvious choice is to use best-first search where the evaluation function is the cost of the path from the root to the current node. This is called Dijkstra's algorithm by the theoretical computer science community, and **uniform-cost search** by the AI community. The idea is that while breadth-first search spreads out in waves of uniform depth—first depth 1, then depth 2, and so on—uniform-cost search spreads out in waves of uniform path-cost. The algorithm can be implemented as a call to BEST-FIRST-SEARCH with PATH-COST as the evaluation function, as shown in Figure 3.9.

Uniform-cost search

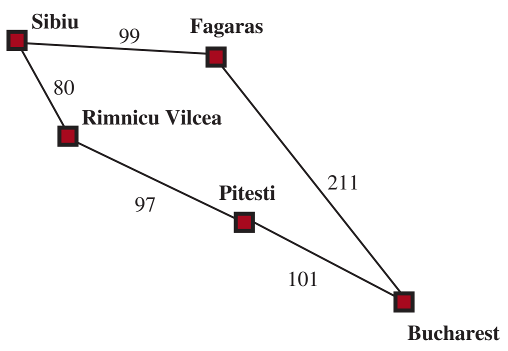


Figure 3.10 Part of the Romania state space, selected to illustrate uniform-cost search.

Consider Figure 3.10, where the problem is to get from Sibiu to Bucharest. The successors of Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 99, respectively. The least-cost node, Rimnicu Vilcea, is expanded next, adding Pitesti with cost $80+97=177$. The least-cost node is now Fagaras, so it is expanded, adding Bucharest with cost $99+211=310$. Bucharest is the goal, but the algorithm tests for goals only when it expands a node, not when it generates a node, so it has not yet detected that this is a path to the goal.

The algorithm continues on, choosing Pitesti for expansion next and adding a second path to Bucharest with cost $80+97+101=278$. It has a lower cost, so it replaces the previous path in *reached* and is added to the *frontier*. It turns out this node now has the lowest cost, so it is considered next, found to be a goal, and returned. Note that if we had checked for a goal upon generating a node rather than when expanding the lowest-cost node, then we would have returned a higher-cost path (the one through Fagaras).

The complexity of uniform-cost search is characterized in terms of C^*, the cost of the optimal solution,[8] and ϵ, a lower bound on the cost of each action, with $\epsilon > 0$. Then the algorithm's worst-case time and space complexity is $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, which can be much greater than b^d. This is because uniform-cost search can explore large trees of actions with low costs before exploring paths involving a high-cost and perhaps useful action. When all action costs are equal, $b^{1+\lfloor C^*/\epsilon \rfloor}$ is just b^{d+1}, and uniform-cost search is similar to breadth-first search.

Uniform-cost search is complete and is cost-optimal, because the first solution it finds will have a cost that is at least as low as the cost of any other node in the frontier. Uniform-cost search considers all paths systematically in order of increasing cost, never getting caught going down a single infinite path (assuming that all action costs are $> \epsilon > 0$).

3.4.3 Depth-first search and the problem of memory

Depth-first search

Depth-first search always expands the *deepest* node in the frontier first. It could be implemented as a call to BEST-FIRST-SEARCH where the evaluation function f is the negative of the depth. However, it is usually implemented not as a graph search but as a tree-like search that does not keep a table of reached states. The progress of the search is illustrated in Figure 3.11; search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. The search then "backs up" to the next deepest node that still has

[8] Here, and throughout the book, the "star" in C^* means an optimal value for C.

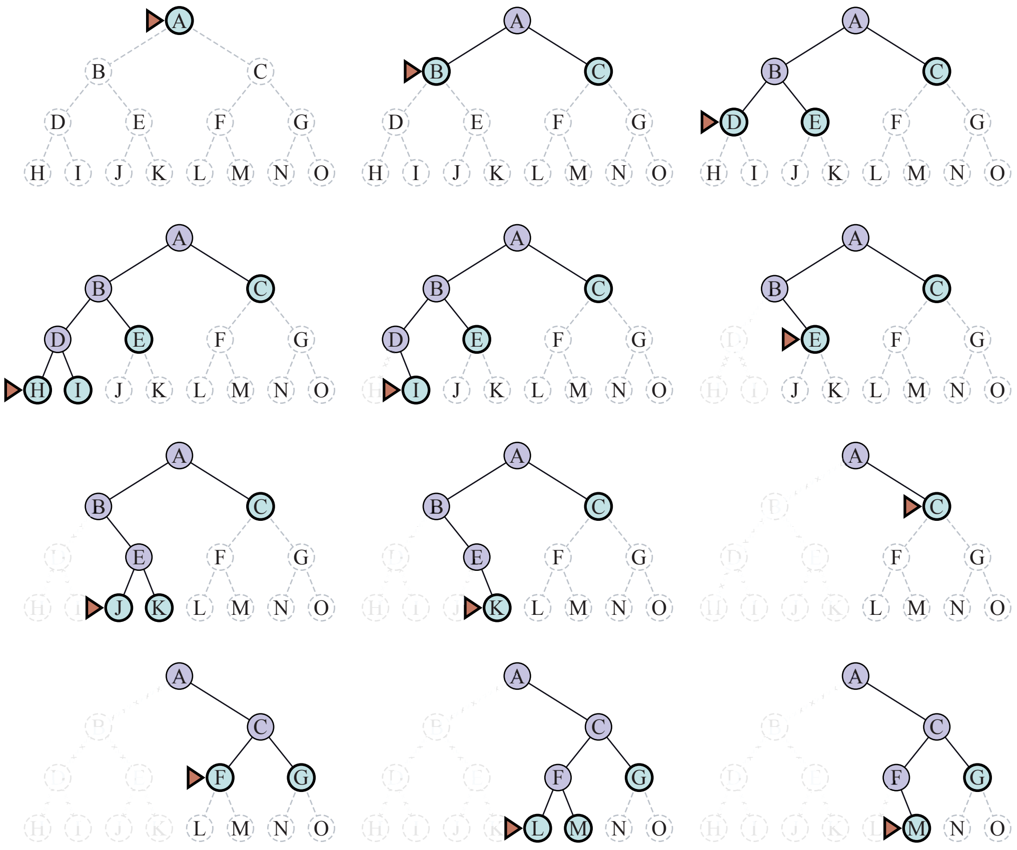


Figure 3.11 A dozen steps (left to right, top to bottom) in the progress of a depth-first search on a binary tree from start state A to goal M. The frontier is in green, with a triangle marking the node to be expanded next. Previously expanded nodes are lavender, and potential future nodes have faint dashed lines. Expanded nodes with no descendants in the frontier (very faint lines) can be discarded.

unexpanded successors. Depth-first search is not cost-optimal; it returns the first solution it finds, even if it is not cheapest.

For finite state spaces that are trees it is efficient and complete; for acyclic state spaces it may end up expanding the same state many times via different paths, but will (eventually) systematically explore the entire space.

In cyclic state spaces it can get stuck in an infinite loop; therefore some implementations of depth-first search check each new node for cycles. Finally, in infinite state spaces, depth-first search is not systematic: it can get stuck going down an infinite path, even if there are no cycles. Thus, depth-first search is incomplete.

With all this bad news, why would anyone consider using depth-first search rather than breadth-first or best-first? The answer is that for problems where a tree-like search is feasible, depth-first search has much smaller needs for memory. We don't keep a *reached* table at all, and the frontier is very small: think of the frontier in breadth-first search as the surface of an ever-expanding sphere, while the frontier in depth-first search is just a radius of the sphere.

For a finite tree-shaped state-space like the one in Figure 3.11, a depth-first tree-like search takes time proportional to the number of states, and has memory complexity of only $O(bm)$, where b is the branching factor and m is the maximum depth of the tree. Some problems that would require exabytes of memory with breadth-first search can be handled with only kilobytes using depth-first search. Because of its parsimonious use of memory, depth-first tree-like search has been adopted as the basic workhorse of many areas of AI, including constraint satisfaction (Chapter 6), propositional satisfiability (Chapter 7), and logic programming (Chapter 9).

Backtracking search

A variant of depth-first search called **backtracking search** uses even less memory. (See Chapter 6 for more details.) In backtracking, only one successor is generated at a time rather than all successors; each partially expanded node remembers which successor to generate next. In addition, successors are generated by *modifying* the current state description directly rather than allocating memory for a brand-new state. This reduces the memory requirements to just one state description and a path of $O(m)$ actions; a significant savings over $O(bm)$ states for depth-first search. With backtracking we also have the option of maintaining an efficient set data structure for the states on the current path, allowing us to check for a cyclic path in $O(1)$ time rather than $O(m)$. For backtracking to work, we must be able to *undo* each action when we backtrack. Backtracking is critical to the success of many problems with large state descriptions, such as robotic assembly.

3.4.4 Depth-limited and iterative deepening search

Depth-limited search

To keep depth-first search from wandering down an infinite path, we can use **depth-limited search**, a version of depth-first search in which we supply a depth limit, ℓ, and treat all nodes at depth ℓ as if they had no successors (see Figure 3.12). The time complexity is $O(b^{\ell})$ and the space complexity is $O(b\ell)$. Unfortunately, if we make a poor choice for ℓ the algorithm will fail to reach the solution, making it incomplete again.

Since depth-first search is a tree-like search, we can't keep it from wasting time on redundant paths in general, but we can eliminate cycles at the cost of some computation time. If we look only a few links up in the parent chain we can catch most cycles; longer cycles are handled by the depth limit.

Sometimes a good depth limit can be chosen based on knowledge of the problem. For example, on the map of Romania there are 20 cities. Therefore, $\ell = 19$ is a valid limit. But if we studied the map carefully, we would discover that any city can be reached from any other city in at most 9 actions. This number, known as the **diameter** of the state-space graph, gives us a better depth limit, which leads to a more efficient depth-limited search. However, for most problems we will not know a good depth limit until we have solved the problem.

Diameter

Iterative deepening search

Iterative deepening search solves the problem of picking a good value for ℓ by trying all values: first 0, then 1, then 2, and so on—until either a solution is found, or the depth-limited search returns the *failure* value rather than the *cutoff* value. The algorithm is shown in Figure 3.12. Iterative deepening combines many of the benefits of depth-first and breadth-first search. Like depth-first search, its memory requirements are modest: $O(bd)$ when there is a solution, or $O(bm)$ on finite state spaces with no solution. Like breadth-first search, iterative deepening is optimal for problems where all actions have the same cost, and is complete on finite acyclic state spaces, or on any finite state space when we check nodes for cycles all the way up the path.

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution node or failure
  for depth = 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result

function DEPTH-LIMITED-SEARCH(problem, ℓ) returns a node or failure or cutoff
  frontier ← a LIFO queue (stack) with NODE(problem.INITIAL) as an element
  result ← failure
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    if DEPTH(node) > ℓ then
      result ← cutoff
    else if not IS-CYCLE(node) do
      for each child in EXPAND(problem, node) do
        add child to frontier
  return result
```

Figure 3.12 Iterative deepening and depth-limited tree-like search. Iterative deepening repeatedly applies depth-limited search with increasing limits. It returns one of three different types of values: either a solution node; or *failure*, when it has exhausted all nodes and proved there is no solution at any depth; or *cutoff*, to mean there might be a solution at a deeper depth than ℓ. This is a tree-like search algorithm that does not keep track of *reached* states, and thus uses much less memory than best-first search, but runs the risk of visiting the same state multiple times on different paths. Also, if the IS-CYCLE check does not check *all* cycles, then the algorithm may get caught in a loop.

The time complexity is $O(b^d)$ when there is a solution, or $O(b^m)$ when there is none. Each iteration of iterative deepening search generates a new level, in the same way that breadth-first search does, but breadth-first does this by storing all nodes in memory, while iterative-deepening does it by repeating the previous levels, thereby saving memory at the cost of more time. Figure 3.13 shows four iterations of iterative-deepening search on a binary search tree, where the solution is found on the fourth iteration.

Iterative deepening search may seem wasteful because states near the top of the search tree are re-generated multiple times. But for many state spaces, most of the nodes are in the bottom level, so it does not matter much that the upper levels are repeated. In an iterative deepening search, the nodes on the bottom level (depth d) are generated once, those on the next-to-bottom level are generated twice, and so on, up to the children of the root, which are generated d times. So the total number of nodes generated in the worst case is

$$N(\text{IDS}) = (d)b^1 + (d-1)b^2 + (d-2)b^3 \cdots + b^d ,$$

which gives a time complexity of $O(b^d)$—asymptotically the same as breadth-first search. For example, if $b = 10$ and $d = 5$, the numbers are

$$\begin{aligned} N(\text{IDS}) &= 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}450 \\ N(\text{BFS}) &= 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}110 . \end{aligned}$$

If you are really concerned about the repetition, you can use a hybrid approach that runs

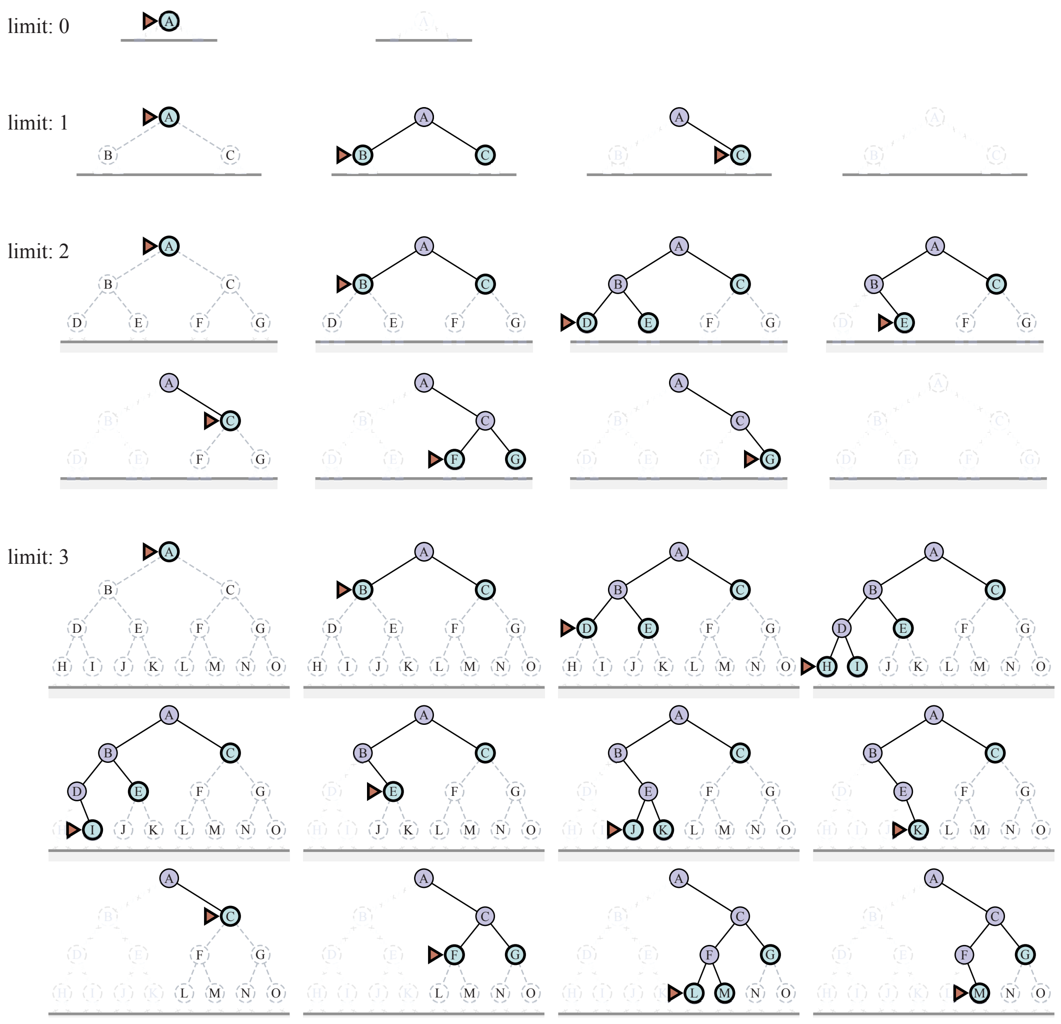


Figure 3.13 Four iterations of iterative deepening search for goal M on a binary tree, with the depth limit varying from 0 to 3. Note the interior nodes form a single path. The triangle marks the node to expand next; green nodes with dark outlines are on the frontier; the very faint nodes provably can't be part of a solution with this depth limit.

breadth-first search until almost all the available memory is consumed, and then runs iterative deepening from all the nodes in the frontier. *In general, iterative deepening is the preferred uninformed search method when the search state space is larger than can fit in memory and the depth of the solution is not known.*

3.4.5 Bidirectional search

Bidirectional search

The algorithms we have covered so far start at an initial state and can reach any one of multiple possible goal states. An alternative approach called **bidirectional search** simultaneously searches forward from the initial state and backwards from the goal state(s), hoping that the two searches will meet. The motivation is that $b^{d/2} + b^{d/2}$ is much less than b^d (e.g., 50,000 times less when $b = d = 10$).

```
function BIBF-SEARCH($problem_F$, $f_F$, $problem_B$, $f_B$) returns a solution node, or failure
  $node_F$ ← NODE($problem_F$.INITIAL)                    // Node for a start state
  $node_B$ ← NODE($problem_B$.INITIAL)                    // Node for a goal state
  $frontier_F$ ← a priority queue ordered by $f_F$, with $node_F$ as an element
  $frontier_B$ ← a priority queue ordered by $f_B$, with $node_B$ as an element
  $reached_F$ ← a lookup table, with one key $node_F$.STATE and value $node_F$
  $reached_B$ ← a lookup table, with one key $node_B$.STATE and value $node_B$
  solution ← failure
  while not TERMINATED(solution, $frontier_F$, $frontier_B$) do
    if $f_F$(TOP($frontier_F$)) $<$ $f_B$(TOP($frontier_B$)) then
      solution ← PROCEED(F, $problem_F$ $frontier_F$, $reached_F$, $reached_B$, solution)
    else solution ← PROCEED(B, $problem_B$, $frontier_B$, $reached_B$, $reached_F$, solution)
  return solution

function PROCEED(dir, problem, frontier, reached, $reached_2$, solution) returns a solution
      // Expand node on frontier; check against the other frontier in $reached_2$.
      // The variable "dir" is the direction: either F for forward or B for backward.
  node ← POP(frontier)
  for each child in EXPAND(problem, node) do
    s ← child.STATE
    if s not in reached or PATH-COST(child) $<$ PATH-COST(reached[s]) then
      reached[s] ← child
      add child to frontier
      if s is in $reached_2$ then
        $solution_2$ ← JOIN-NODES(dir, child, $reached_2$[s]))
        if PATH-COST($solution_2$) $<$ PATH-COST(solution) then
          solution ← $solution_2$
  return solution
```

Figure 3.14 Bidirectional best-first search keeps two frontiers and two tables of reached states. When a path in one frontier reaches a state that was also reached in the other half of the search, the two paths are joined (by the function JOIN-NODES) to form a solution. The first solution we get is not guaranteed to be the best; the function TERMINATED determines when to stop looking for new solutions.

For this to work, we need to keep track of two frontiers and two tables of reached states, and we need to be able to reason backwards: if state s' is a successor of s in the forward direction, then we need to know that s is a successor of s' in the backward direction. We have a solution when the two frontiers collide.[9]

There are many different versions of bidirectional search, just as there are many different unidirectional search algorithms. In this section, we describe bidirectional best-first search. Although there are two separate frontiers, the node to be expanded next is always one with a minimum value of the evaluation function, across either frontier. When the evaluation

[9] In our implementation, the *reached* data structure supports a query asking whether a given state is a member, and the frontier data structure (a priority queue) does not, so we check for a collision using *reached*; but conceptually we are asking if the two frontiers have met up. The implementation can be extended to handle multiple goal states by loading the node for each goal state into the backwards frontier and backwards reached table.

function is the path cost, we get bidirectional uniform-cost search, and if the cost of the optimal path is C^*, then no node with cost $> \frac{C^*}{2}$ will be expanded. This can result in a considerable speedup.

The general best-first bidirectional search algorithm is shown in Figure 3.14. We pass in two versions of the problem and the evaluation function, one in the forward direction (subscript F) and one in the backward direction (subscript B). When the evaluation function is the path cost, we know that the first solution found will be an optimal solution, but with different evaluation functions that is not necessarily true. Therefore, we keep track of the best solution found so far, and might have to update that several times before the TERMINATED test proves that there is no possible better solution remaining.

3.4.6 Comparing uninformed search algorithms

Figure 3.15 compares uninformed search algorithms in terms of the four evaluation criteria set forth in Section 3.3.4. This comparison is for tree-like search versions which don't check for repeated states. For graph searches which do check, the main differences are that depth-first search is complete for finite state spaces, and the space and time complexities are bounded by the size of the state space (the number of vertices and edges, $|V|+|E|$).

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Optimal cost?	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

Figure 3.15 Evaluation of search algorithms. b is the branching factor; m is the maximum depth of the search tree; d is the depth of the shallowest solution, or is m when there is no solution; ℓ is the depth limit. Superscript caveats are as follows: [1] complete if b is finite, and the state space either has a solution or is finite. [2] complete if all action costs are $\geq \epsilon > 0$; [3] cost-optimal if action costs are all identical; [4] if both directions are breadth-first or uniform-cost.

3.5 Informed (Heuristic) Search Strategies

Informed search

Heuristic function

This section shows how an **informed search** strategy—one that uses domain-specific hints about the location of goals—can find solutions more efficiently than an uninformed strategy. The hints come in the form of a **heuristic function**, denoted $h(n)$:[10]

$$h(n) = \text{ estimated cost of the cheapest path from the state at node } n \text{ to a goal state.}$$

For example, in route-finding problems, we can estimate the distance from the current state to a goal by computing the straight-line distance on the map between the two points. We study heuristics and where they come from in more detail in Section 3.6.

[10] It may seem odd that the heuristic function operates on a node, when all it really needs is the node's state. It is traditional to use $h(n)$ rather than $h(s)$ to be consistent with the evaluation function $f(n)$ and the path cost $g(n)$.

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Figure 3.16 Values of h_{SLD}—straight-line distances to Bucharest.

3.5.1 Greedy best-first search

Greedy best-first search

Greedy best-first search is a form of best-first search that expands first the node with the lowest $h(n)$ value—the node that appears to be closest to the goal—on the grounds that this is likely to lead to a solution quickly. So the evaluation function $f(n) = h(n)$.

Straight-line distance

Let us see how this works for route-finding problems in Romania; we use the **straight-line distance** heuristic, which we will call h_{SLD}. If the goal is Bucharest, we need to know the straight-line distances to Bucharest, which are shown in Figure 3.16. For example, $h_{SLD}(Arad) = 366$. Notice that the values of h_{SLD} cannot be computed from the problem description itself (that is, the ACTIONS and RESULT functions). Moreover, it takes a certain amount of world knowledge to know that h_{SLD} is correlated with actual road distances and is, therefore, a useful heuristic.

Figure 3.17 shows the progress of a greedy best-first search using h_{SLD} to find a path from Arad to Bucharest. The first node to be expanded from Arad will be Sibiu because the heuristic says it is closer to Bucharest than is either Zerind or Timisoara. The next node to be expanded will be Fagaras because it is now closest according to the heuristic. Fagaras in turn generates Bucharest, which is the goal. For this particular problem, greedy best-first search using h_{SLD} finds a solution without ever expanding a node that is not on the solution path. The solution it found does not have optimal cost, however: the path via Sibiu and Fagaras to Bucharest is 32 miles longer than the path through Rimnicu Vilcea and Pitesti. This is why the algorithm is called "greedy"—on each iteration it tries to get as close to a goal as it can, but greediness can lead to worse results than being careful.

Greedy best-first graph search is complete in finite state spaces, but not in infinite ones. The worst-case time and space complexity is $O(|V|)$. With a good heuristic function, however, the complexity can be reduced substantially, on certain problems reaching $O(bm)$.

3.5.2 A* search

A* search

The most common informed search algorithm is **A* search** (pronounced "A-star search"), a best-first search that uses the evaluation function

$$f(n) = g(n) + h(n)$$

where $g(n)$ is the path cost from the initial state to node n, and $h(n)$ is the *estimated* cost of the shortest path from n to a goal state, so we have

$$f(n) = \text{ estimated cost of the best path that continues from } n \text{ to a goal.}$$

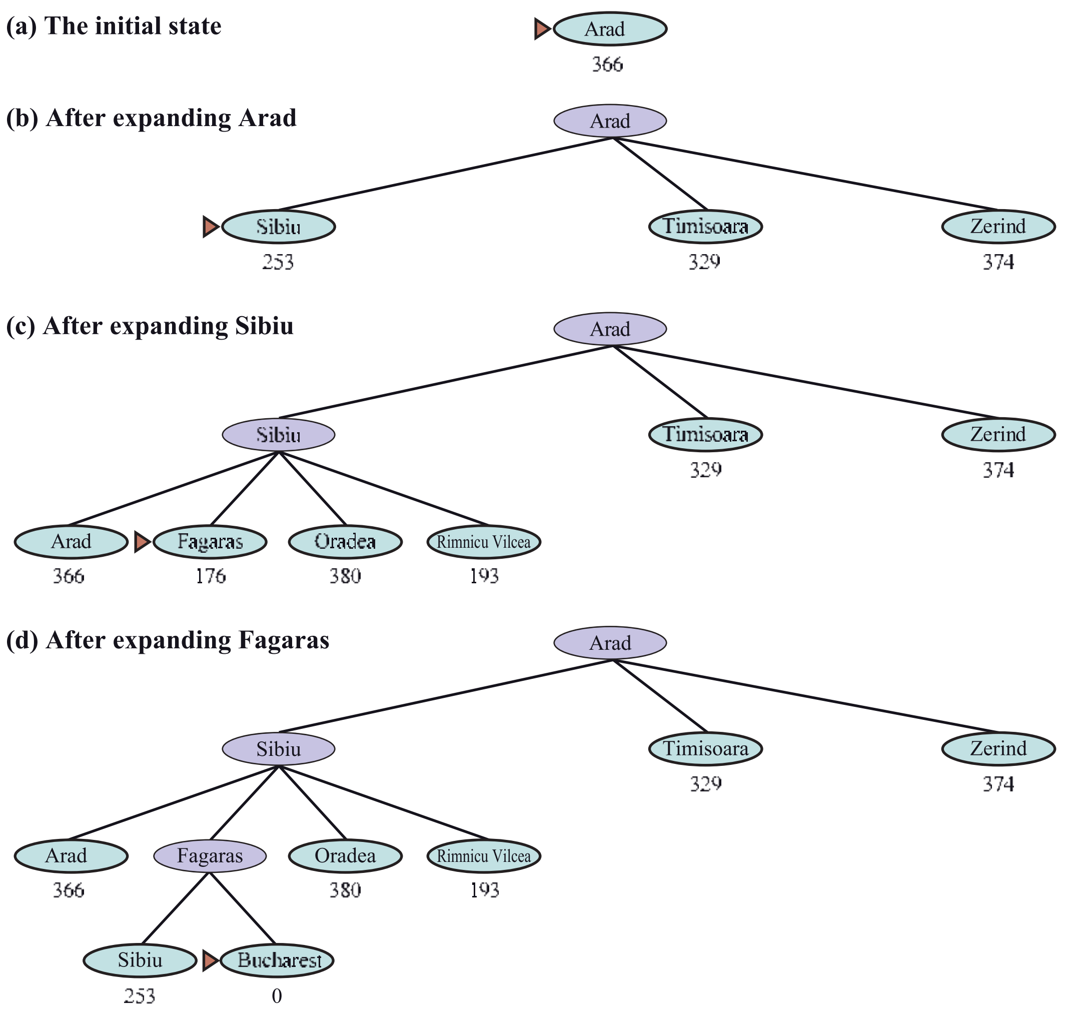


Figure 3.17 Stages in a greedy best-first tree-like search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

In Figure 3.18, we show the progress of an A* search with the goal of reaching Bucharest. The values of g are computed from the action costs in Figure 3.1, and the values of h_{SLD} are given in Figure 3.16. Notice that Bucharest first appears on the frontier at step (e), but it is not selected for expansion (and thus not detected as a solution) because at $f=450$ it is not the lowest-cost node on the frontier—that would be Pitesti, at $f=417$. Another way to say this is that there *might* be a solution through Pitesti whose cost is as low as 417, so the algorithm will not settle for a solution that costs 450. At step (f), a different path to Bucharest is now the lowest-cost node, at $f=418$, so it is selected and detected as the optimal solution.

Admissible heuristic

A* search is complete.[11] Whether A* is cost-optimal depends on certain properties of the heuristic. A key property is **admissibility**: an **admissible heuristic** is one that *never overestimates* the cost to reach a goal. (An admissible heuristic is therefore *optimistic*.) With

[11] Again, assuming all action costs are $> \epsilon > 0$, and the state space either has a solution or is finite.

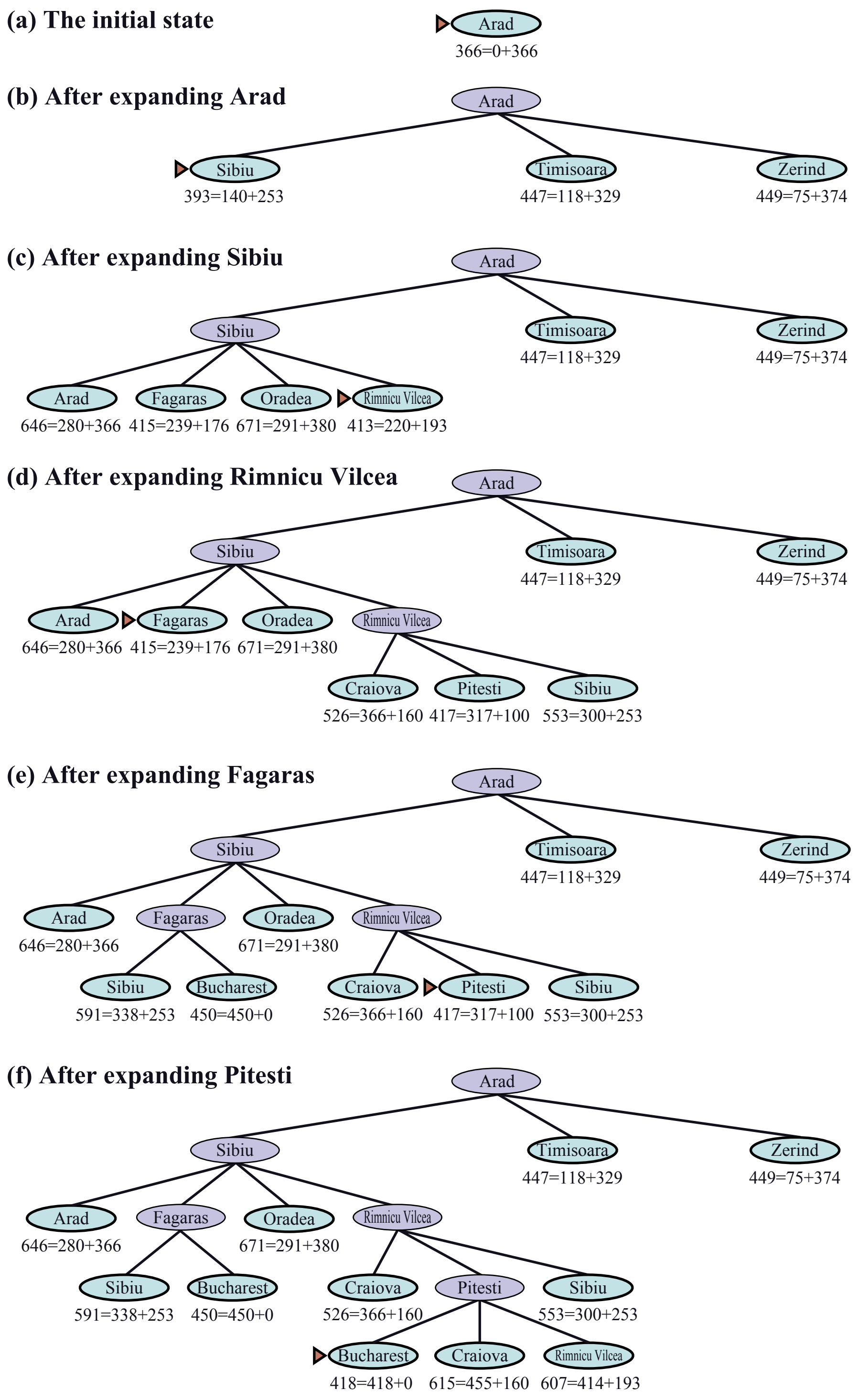


Figure 3.18 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 3.16.

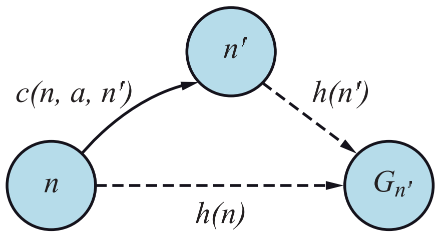


Figure 3.19 Triangle inequality: If the heuristic h is **consistent**, then the single number $h(n)$ will be less than the sum of the cost $c(n,a,a')$ of the action from n to n' plus the heuristic estimate $h(n')$.

an admissible heuristic, A* is cost-optimal, which we can show with a proof by contradiction. Suppose the optimal path has cost C^*, but the algorithm returns a path with cost $C > C^*$. Then there must be some node n which is on the optimal path and is unexpanded (because if all the nodes on the optimal path had been expanded, then we would have returned that optimal solution). So then, using the notation $g^*(n)$ to mean the cost of the optimal path from the start to n, and $h^*(n)$ to mean the cost of the optimal path from n to the nearest goal, we have:

$$\begin{aligned}
f(n) &> C^* \quad \text{(otherwise } n \text{ would have been expanded)}\\
f(n) &= g(n)+h(n) \quad \text{(by definition)}\\
f(n) &= g^*(n)+h(n) \quad \text{(because } n \text{ is on an optimal path)}\\
f(n) &\le g^*(n)+h^*(n) \quad \text{(because of admissibility, } h(n) \le h^*(n))\\
f(n) &\le C^* \quad \text{(by definition, } C^* = g^*(n)+h^*(n))
\end{aligned}$$

The first and last lines form a contradiction, so the supposition that the algorithm could return a suboptimal path must be wrong—it must be that A* returns only cost-optimal paths.

Consistency

A slightly stronger property is called **consistency**. A heuristic $h(n)$ is consistent if, for every node n and every successor n' of n generated by an action a, we have:

$$h(n) \le c(n,a,n') + h(n') .$$

Triangle inequality

This is a form of the **triangle inequality**, which stipulates that a side of a triangle cannot be longer than the sum of the other two sides (see Figure 3.19). An example of a consistent heuristic is the straight-line distance h_{SLD} that we used in getting to Bucharest.

Every consistent heuristic is admissible (but not vice versa), so with a consistent heuristic, A* is cost-optimal. In addition, with a consistent heuristic, the first time we reach a state it will be on an optimal path, so we never have to re-add a state to the frontier, and never have to change an entry in *reached*. But with an inconsistent heuristic, we may end up with multiple paths reaching the same state, and if each new path has a lower path cost than the previous one, then we will end up with multiple nodes for that state in the frontier, costing us both time and space. Because of that, some implementations of A* take care to only enter a state into the frontier once, and if a better path to the state is found, all the successors of the state are updated (which requires that nodes have child pointers as well as parent pointers). These complications have led many implementers to avoid inconsistent heuristics, but Felner *et al.* (2011) argues that the worst effects rarely happen in practice, and one shouldn't be afraid of inconsistent heuristics.

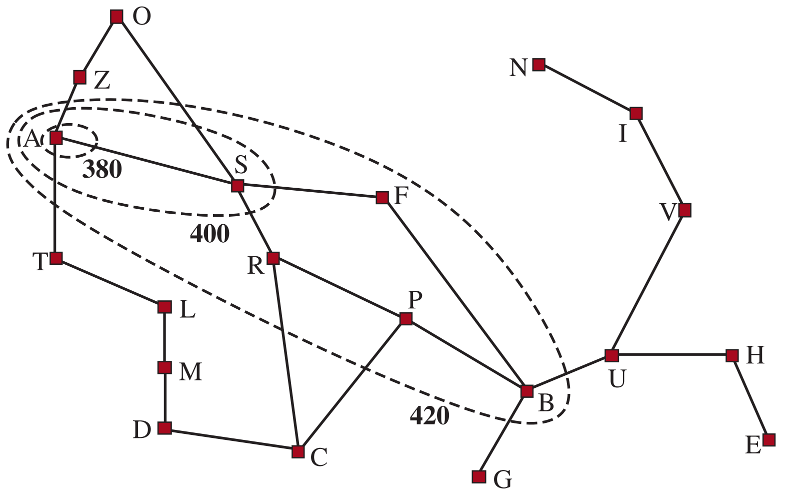


Figure 3.20 Map of Romania showing contours at $f = 380$, $f = 400$, and $f = 420$, with Arad as the start state. Nodes inside a given contour have $f = g + h$ costs less than or equal to the contour value.

With an inadmissible heuristic, A* may or may not be cost-optimal. Here are two cases where it is: First, if there is even one cost-optimal path on which $h(n)$ is admissible for all nodes n on the path, then that path will be found, no matter what the heuristic says for states off the path. Second, if the optimal solution has cost C^*, and the second-best has cost C_2, and if $h(n)$ overestimates some costs, but never by more than $C_2 - C^*$, then A* is guaranteed to return cost-optimal solutions.

3.5.3 Search contours

A useful way to visualize a search is to draw **contours** in the state space, just like the contours in a topographic map. Figure 3.20 shows an example. Inside the contour labeled 400, all nodes have $f(n) = g(n) + h(n) \leq 400$, and so on. Then, because A* expands the frontier node of lowest f-cost, we can see that an A* search fans out from the start node, adding nodes in concentric bands of increasing f-cost. Contour

With uniform-cost search, we also have contours, but of g-cost, not $g + h$. The contours with uniform-cost search will be "circular" around the start state, spreading out equally in all directions with no preference towards the goal. With A* search using a good heuristic, the $g + h$ bands will stretch toward a goal state (as in Figure 3.20) and become more narrowly focused around an optimal path.

It should be clear that as you extend a path, the g costs are **monotonic**: the path cost always increases as you go along a path, because action costs are always positive.[12] Therefore you get concentric contour lines that don't cross each other, and if you choose to draw the lines fine enough, you can put a line between any two nodes on any path. Monotonic

[12] Technically, we say "strictly monotonic" for costs that always increase, and "monotonic" for costs that never decrease, but might remain the same.

But it is not obvious whether the $f = g + h$ cost will monotonically increase. As you extend a path from n to n', the cost goes from $g(n)+h(n)$ to $g(n)+c(n,a,n')+h(n')$. Canceling out the $g(n)$ term, we see that the path's cost will be monotonically increasing if and only if $h(n) \leq c(n,a,n')+h(n')$; in other words if and only if the heuristic is consistent.[13] But note that a path might contribute several nodes in a row with the same $g(n)+h(n)$ score; this will happen whenever the decrease in h is exactly equal to the action cost just taken (for example, in a grid problem, when n is in the same row as the goal and you take a step towards the goal, g is increased by 1 and h is decreased by 1). If C^* is the cost of the optimal solution path, then we can say the following:

Surely expanded nodes

- A* expands all nodes that can be reached from the initial state on a path where every node on the path has $f(n) < C^*$. We say these are **surely expanded nodes**.
- A* might then expand some of the nodes right on the "goal contour" (where $f(n) = C^*$) before selecting a goal node.
- A* expands no nodes with $f(n) > C^*$.

Optimally efficient

We say that A* with a consistent heuristic is **optimally efficient** in the sense that any algorithm that extends search paths from the initial state, and uses the same heuristic information, must expand all nodes that are surely expanded by A* (because any one of them could have been part of an optimal solution). Among the nodes with $f(n)=C^*$, one algorithm could get lucky and choose the optimal one first while another algorithm is unlucky; we don't consider this difference in defining optimal efficiency.

Pruning

A* is efficient because it **prunes** away search tree nodes that are not necessary for finding an optimal solution. In Figure 3.18(b) we see that Timisoara has $f = 447$ and Zerind has $f = 449$. Even though they are children of the root and would be among the first nodes expanded by uniform-cost or breadth-first search, they are never expanded by A* search because the solution with $f = 418$ is found first. The concept of pruning—eliminating possibilities from consideration without having to examine them—is important for many areas of AI.

That A* search is complete, cost-optimal, and optimally efficient among all such algorithms is rather satisfying. Unfortunately, it does not mean that A* is the answer to all our searching needs. The catch is that for many problems, the number of nodes expanded can be exponential in the length of the solution. For example, consider a version of the vacuum world with a super-powerful vacuum that can clean up any one square at a cost of 1 unit, without even having to visit the square; in that scenario, squares can be cleaned in any order. With N initially dirty squares, there are 2^N states where some subset has been cleaned; all of those states are on an optimal solution path, and hence satisfy $f(n) < C^*$, so all of them would be visited by A*.

3.5.4 Satisficing search: Inadmissible heuristics and weighted A*

A* search has many good qualities, but it expands a lot of nodes. We can explore fewer nodes (taking less time and space) if we are willing to accept solutions that are suboptimal, but are "good enough"—what we call **satisficing** solutions. If we allow A* search to use an **inadmissible heuristic**—one that may overestimate—then we risk missing the optimal solution, but the heuristic can potentially be more accurate, thereby reducing the number of

Inadmissible heuristic

[13] In fact, the term "monotonic heuristic" is a synonym for "consistent heuristic." The two ideas were developed independently, and then it was proved that they are equivalent (Pearl, 1984).

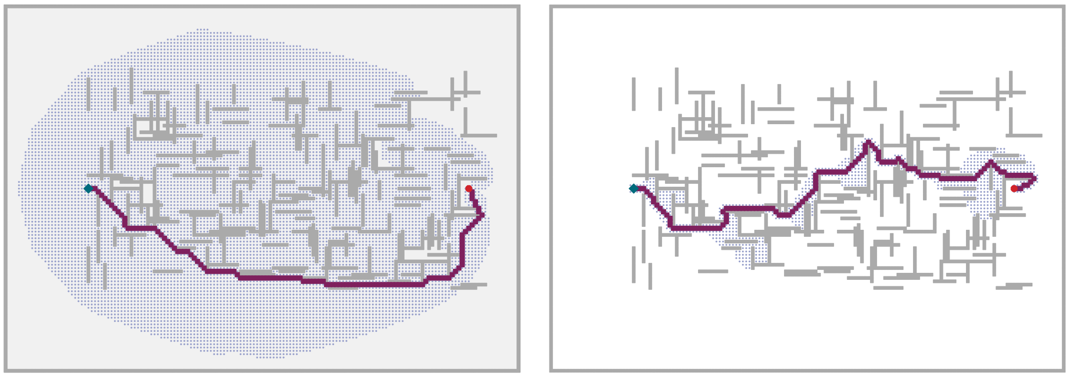

(a) (b)

Figure 3.21 Two searches on the same grid: (a) an A* search and (b) a weighted A* search with weight $W = 2$. The gray bars are obstacles, the purple line is the path from the green start to red goal, and the small dots are states that were reached by each search. On this particular problem, weighted A* explores 7 times fewer states and finds a path that is 5% more costly.

nodes expanded. For example, road engineers know the concept of a **detour index**, which is a multiplier applied to the straight-line distance to account for the typical curvature of roads. A detour index of 1.3 means that if two cities are 10 miles apart in straight-line distance, a good estimate of the best path between them is 13 miles. For most localities, the detour index ranges between 1.2 and 1.6.

Detour index

We can apply this idea to any problem, not just ones involving roads, with an approach called **weighted A* search** where we weight the heuristic value more heavily, giving us the evaluation function $f(n) = g(n) + W \times h(n)$, for some $W > 1$.

Weighted A* search

Figure 3.21 shows a search problem on a grid world. In (a), an A* search finds the optimal solution, but has to explore a large portion of the state space to find it. In (b), a weighted A* search finds a solution that is slightly costlier, but the search time is much faster. We see that the weighted search focuses the contour of reached states towards a goal. That means that fewer states are explored, but if the optimal path ever strays outside of the weighted search's contour (as it does in this case), then the optimal path will not be found. In general, if the optimal solution costs C^*, a weighted A* search will find a solution that costs somewhere between C^* and $W \times C^*$; but in practice we usually get results much closer to C^* than $W \times C^*$.

We have considered searches that evaluate states by combining g and h in various ways; weighted A* can be seen as a generalization of the others:

A* search:	$g(n) + h(n)$	$(W = 1)$
Uniform-cost search:	$g(n)$	$(W = 0)$
Greedy best-first search:	$h(n)$	$(W = \infty)$
Weighted A* search:	$g(n) + W \times h(n)$	$(1 < W < \infty)$

You could call weighted A* "somewhat-greedy search": like greedy best-first search, it focuses the search towards a goal; on the other hand, it won't ignore the path cost completely, and will suspend a path that is making little progress at great cost.

Bounded suboptimal search

There are a variety of suboptimal search algorithms, which can be characterized by the criteria for what counts as "good enough." In **bounded suboptimal search**, we look for a solution that is guaranteed to be within a constant factor W of the optimal cost. Weighted A* provides this guarantee. In **bounded-cost search**, we look for a solution whose cost is less than some constant C. And in **unbounded-cost search**, we accept a solution of any cost, as long as we can find it quickly.

Bounded-cost search

Unbounded-cost search

Speedy search

An example of an unbounded-cost search algorithm is **speedy search**, which is a version of greedy best-first search that uses as a heuristic the estimated number of actions required to reach a goal, regardless of the cost of those actions. Thus, for problems where all actions have the same cost it is the same as greedy best-first search, but when actions have different costs, it tends to lead the search to find a solution quickly, even if it might have a high cost.

3.5.5 Memory-bounded search

The main issue with A* is its use of memory. In this section we'll cover some implementation tricks that save space, and then some entirely new algorithms that take better advantage of the available space.

Memory is split between the *frontier* and the *reached* states. In our implementation of best-first search, a state that is on the frontier is stored in two places: as a node in the frontier (so we can decide what to expand next) and as an entry in the table of reached states (so we know if we have visited the state before). For many problems (such as exploring a grid), this duplication is not a concern, because the size of *frontier* is much smaller than *reached*, so duplicating the states in the frontier requires a comparatively trivial amount of memory. But some implementations keep a state in only one of the two places, saving a bit of space at the cost of complicating (and perhaps slowing down) the algorithm.

Another possibility is to remove states from *reached* when we can prove that they are no longer needed. For some problems, we can use the separation property (Figure 3.6 on page 72), along with the prohibition of U-turn actions, to ensure that all actions either move outwards from the frontier or onto another frontier state. In that case, we need only check the frontier for redundant paths, and we can eliminate the *reached* table.

Reference count

For other problems, we can keep **reference counts** of the number of times a state has been reached, and remove it from the *reached* table when there are no more ways to reach the state. For example, on a grid world where each state can be reached only from its four neighbors, once we have reached a state four times, we can remove it from the table.

Now let's consider new algorithms that are designed to conserve memory usage.

Beam search

Beam search limits the size of the frontier. The easiest approach is to keep only the k nodes with the best f-scores, discarding any other expanded nodes. This of course makes the search incomplete and suboptimal, but we can choose k to make good use of available memory, and the algorithm executes fast because it expands fewer nodes. For many problems it can find good near-optimal solutions. You can think of uniform-cost or A* search as spreading out everywhere in concentric contours, and think of beam search as exploring only a focused portion of those contours, the portion that contains the k best candidates.

An alternative version of beam search doesn't keep a strict limit on the size of the frontier but instead keeps every node whose f-score is within δ of the best f-score. That way, when there are a few strong-scoring nodes only a few will be kept, but if there are no strong nodes then more will be kept until a strong one emerges.

Iterative-deepening A* search (IDA*) is to A* what iterative-deepening search is to depth-first: IDA* gives us the benefits of A* without the requirement to keep all reached states in memory, at a cost of visiting some states multiple times. It is a very important and commonly used algorithm for problems that do not fit in memory.

Iterative-deepening A* search

In standard iterative deepening the cutoff is the depth, which is increased by one each iteration. In IDA* the cutoff is the f-cost ($g+h$); at each iteration, the cutoff value is the smallest f-cost of any node that exceeded the cutoff on the previous iteration. In other words, each iteration exhaustively searches an f-contour, finds a node just beyond that contour, and uses that node's f-cost as the next contour. For problems like the 8-puzzle where each path's f-cost is an integer, this works very well, resulting in steady progress towards the goal each iteration. If the optimal solution has cost C^*, then there can be no more than C^* iterations (for example, no more than 31 iterations on the hardest 8-puzzle problems). But for a problem where every node has a different f-cost, each new contour might contain only one new node, and the number of iterations could be equal to the number of states.

Recursive best-first search (RBFS) (Figure 3.22) attempts to mimic the operation of standard best-first search, but using only linear space. RBFS resembles a recursive depth-first search, but rather than continuing indefinitely down the current path, it uses the *f_limit* variable to keep track of the f-value of the best *alternative* path available from any ancestor of the current node. If the current node exceeds this limit, the recursion unwinds back to the alternative path. As the recursion unwinds, RBFS replaces the f-value of each node along the path with a **backed-up value**—the best f-value of its children. In this way, RBFS remembers the f-value of the best leaf in the forgotten subtree and can therefore decide whether it's worth reexpanding the subtree at some later time. Figure 3.23 shows how RBFS reaches Bucharest.

Recursive best-first search

Backed-up value

RBFS is somewhat more efficient than IDA*, but still suffers from excessive node re-generation. In the example in Figure 3.23, RBFS follows the path via Rimnicu Vilcea, then

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution or failure
   solution, fvalue ← RBFS(problem, NODE(problem.INITIAL), ∞)
  return solution

function RBFS(problem, node, f_limit) returns a solution or failure, and a new f-cost limit
  if problem.IS-GOAL(node.STATE) then return node
  successors ← LIST(EXPAND(node))
  if successors is empty then return failure, ∞
  for each s in  successors do        // update f with value from previous search
     s.f ← max(s.PATH-COST + h(s), node.f))
  while true do
     best ← the node in successors with lowest f-value
     if best.f > f_limit then return failure, best.f
     alternative ← the second-lowest f-value among successors
     result, best.f ← RBFS(problem, best, min(f_limit, alternative))
     if result ≠ failure then return result, best.f
```

Figure 3.22 The algorithm for recursive best-first search.

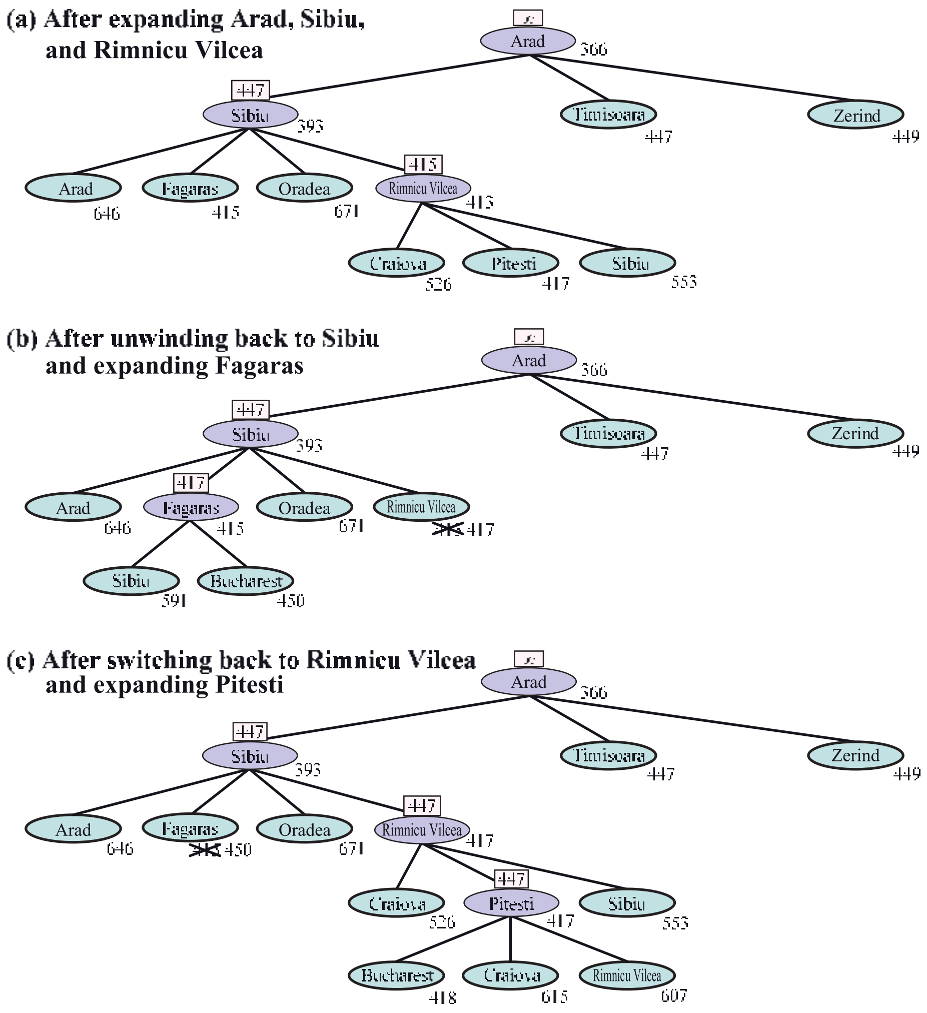


Figure 3.23 Stages in an RBFS search for the shortest route to Bucharest. The f-*limit* value for each recursive call is shown on top of each current node, and every node is labeled with its f-cost. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras). (b) The recursion unwinds and the best leaf value of the forgotten subtree (417) is backed up to Rimnicu Vilcea; then Fagaras is expanded, revealing a best leaf value of 450. (c) The recursion unwinds and the best leaf value of the forgotten subtree (450) is backed up to Fagaras; then Rimnicu Vilcea is expanded. This time, because the best alternative path (through Timisoara) costs at least 447, the expansion continues to Bucharest.

"changes its mind" and tries Fagaras, and then changes its mind back again. These mind changes occur because every time the current best path is extended, its f-value is likely to increase—h is usually less optimistic for nodes closer to a goal. When this happens, the second-best path might become the best path, so the search has to backtrack to follow it. Each mind change corresponds to an iteration of IDA* and could require many reexpansions of forgotten nodes to recreate the best path and extend it one more node.

RBFS is optimal if the heuristic function $h(n)$ is admissible. Its space complexity is linear in the depth of the deepest optimal solution, but its time complexity is rather difficult to characterize: it depends both on the accuracy of the heuristic function and on how often the best path changes as nodes are expanded. It expands nodes in order of increasing f-score, even if f is nonmonotonic.

IDA* and RBFS suffer from using *too little* memory. Between iterations, IDA* retains only a single number: the current f-cost limit. RBFS retains more information in memory, but it uses only linear space: even if more memory were available, RBFS has no way to make use of it. Because they forget most of what they have done, both algorithms may end up reexploring the same states many times over.

It seems sensible, therefore, to determine how much memory we have available, and allow an algorithm to use all of it. Two algorithms that do this are **MA*** (memory-bounded A*) and **SMA*** (simplified MA*). SMA* is—well—simpler, so we will describe it. SMA* proceeds just like A*, expanding the best leaf until memory is full. At this point, it cannot add a new node to the search tree without dropping an old one. SMA* always drops the *worst* leaf node—the one with the highest f-value. Like RBFS, SMA* then backs up the value of the forgotten node to its parent. In this way, the ancestor of a forgotten subtree knows the quality of the best path in that subtree. With this information, SMA* regenerates the subtree only when all other paths have been shown to look worse than the path it has forgotten. Another way of saying this is that if all the descendants of a node n are forgotten, then we will not know which way to go from n, but we will still have an idea of how worthwhile it is to go anywhere from n.

MA*

SMA*

The complete algorithm is described in the online code repository accompanying this book. There is one subtlety worth mentioning. We said that SMA* expands the best leaf and deletes the worst leaf. What if *all* the leaf nodes have the same f-value? To avoid selecting the same node for deletion and expansion, SMA* expands the *newest* best leaf and deletes the *oldest* worst leaf. These coincide when there is only one leaf, but in that case, the current search tree must be a single path from root to leaf that fills all of memory. If the leaf is not a goal node, then *even if it is on an optimal solution path*, that solution is not reachable with the available memory. Therefore, the node can be discarded exactly as if it had no successors.

SMA* is complete if there is any reachable solution—that is, if d, the depth of the shallowest goal node, is less than the memory size (expressed in nodes). It is optimal if any optimal solution is reachable; otherwise, it returns the best reachable solution. In practical terms, SMA* is a fairly robust choice for finding optimal solutions, particularly when the state space is a graph, action costs are not uniform, and node generation is expensive compared to the overhead of maintaining the frontier and the reached set.

On very hard problems, however, it will often be the case that SMA* is forced to switch back and forth continually among many candidate solution paths, only a small subset of which can fit in memory. (This resembles the problem of **thrashing** in disk paging systems.) Then the extra time required for repeated regeneration of the same nodes means that problems that would be practically solvable by A*, given unlimited memory, become intractable for SMA*. That is to say, *memory limitations can make a problem intractable from the point of view of computation time.* Although no current theory explains the tradeoff between time and memory, it seems that this is an inescapable problem. The only way out is to drop the optimality requirement.

Thrashing

3.5.6 Bidirectional heuristic search

With unidirectional best-first search, we saw that using $f(n) = g(n) + h(n)$ as the evaluation function gives us an A* search that is guaranteed to find optimal-cost solutions (assuming an admissible h) while being optimally efficient in the number of nodes expanded.

With bidirectional best-first search we could also try using $f(n) = g(n) + h(n)$, but unfortunately there is no guarantee that this would lead to an optimal-cost solution, nor that it would be optimally efficient, even with an admissible heuristic. With bidirectional search, it turns out that it is not individual nodes but rather *pairs* of nodes (one from each frontier) that can be proved to be surely expanded, so any proof of efficiency will have to consider pairs of nodes (Eckerle *et al.*, 2017).

We'll start with some new notation. We use $f_F(n) = g_F(n) + h_F(n)$ for nodes going in the forward direction (with the initial state as root) and $f_B(n) = g_B(n) + h_B(n)$ for nodes in the backward direction (with a goal state as root). Although both forward and backward searches are solving the same problem, they have different evaluation functions because, for example, the heuristics are different depending on whether you are striving for the goal or for the initial state. We'll assume admissible heuristics.

Consider a forward path from the initial state to a node m and a backward path from the goal to a node n. We can define a lower bound on the cost of a solution that follows the path from the initial state to m, then somehow gets to n, then follows the path to the goal as

$$lb(m,n) = \max(g_F(m) + g_B(n), f_F(m), f_B(n))$$

In other words, the cost of such a path must be at least as large as the sum of the path costs of the two parts (because the remaining connection between them must have nonnegative cost), and the cost must also be at least as much as the estimated f cost of either part (because the heuristic estimates are optimistic). Given that, the theorem is that for any pair of nodes m, n with $lb(m,n)$ less than the optimal cost C^*, we must expand either m or n, because the path that goes through both of them is a potential optimal solution. The difficulty is that we don't know for sure which node is best to expand, and therefore no bidirectional search algorithm can be guaranteed to be optimally efficient—any algorithm might expand up to twice the minimum number of nodes if it always chooses the wrong member of a pair to expand first. Some bidirectional heuristic search algorithms explicitly manage a queue of (m,n) pairs, but we will stick with bidirectional best-first search (Figure 3.14), which has two frontier priority queues, and give it an evaluation function that mimics the lb criteria:

$$f_2(n) = \max(2g(n), g(n) + h(n))$$

The node to expand next will be the one that minimizes this f_2 value; the node can come from either frontier. This f_2 function guarantees that we will never expand a node (from either frontier) with $g(n) > \frac{C^*}{2}$. We say the two halves of the search "meet in the middle" in the sense that when the two frontiers touch, no node inside of either frontier has a path cost greater than the bound $\frac{C^*}{2}$. Figure 3.24 works through an example bidirectional search.

We have described an approach where the h_F heuristic estimates the distance to the goal (or, when the problem has multiple goal states, the distance to the closest goal) and h_B estimates the distance to the start. This is called a **front-to-end** search. An alternative, called **front-to-front** search, attempts to estimate the distance to the other frontier. Clearly, if a frontier has millions of nodes, it would be inefficient to apply the heuristic function to every

Front-to-end

Front-to-front

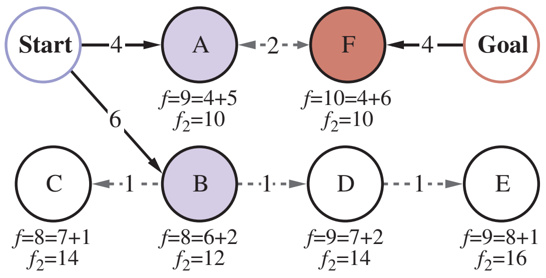


Figure 3.24 Bidirectional search maintains two frontiers: on the left, nodes A and B are successors of Start; on the right, node F is an inverse successor of Goal. Each node is labeled with $f = g + h$ values and the $f_2 = \max(2g, g+h)$ value. (The g values are the sum of the action costs as shown on each arrow; the h values are arbitrary and cannot be derived from anything in the figure.) The optimal solution, Start-A-F-Goal, has cost $C^* = 4+2+4 = 10$, so that means that a meet-in-the-middle bidirectional algorithm should not expand any node with $g > \frac{C^*}{2} = 5$; and indeed the next node to be expanded would be A or F (each with $g = 4$), leading us to an optimal solution. If we expanded the node with lowest f cost first, then B and C would come next, and D and E would be tied with A, but they all have $g > \frac{C^*}{2}$ and thus are never expanded when f_2 is the evaluation function.

one of them and take the minimum. But it can work to sample a few nodes from the frontier. In certain specific problem domains it is possible to *summarize* the frontier—for example, in a grid search problem, we can incrementally compute a bounding box of the frontier, and use as a heuristic the distance to the bounding box.

Bidirectional search is sometimes more efficient than unidirectional search, sometimes not. In general, if we have a very good heuristic, then A* search produces search contours that are focused on the goal, and adding bidirectional search does not help much. With an average heuristic, bidirectional search that meets in the middle tends to expand fewer nodes and is preferred. In the worst case of a poor heuristic, the search is no longer focused on the goal, and bidirectional search has the same asymptotic complexity as A*. Bidirectional search with the f_2 evaluation function and an admissible heuristic h is complete and optimal.

3.6 Heuristic Functions

In this section, we look at how the accuracy of a heuristic affects search performance, and also consider how heuristics can be invented. As our main example we'll return to the 8-puzzle. As mentioned in Section 3.2, the object of the puzzle is to slide the tiles horizontally or vertically into the empty space until the configuration matches the goal configuration (Figure 3.25).

There are $9!/2 = 181{,}400$ reachable states in an 8-puzzle, so a search could easily keep them all in memory. But for the 15-puzzle, there are $16!/2$ states—over 10 trillion—so to search that space we will need the help of a good admissible heuristic function. There is a long history of such heuristics for the 15-puzzle; here are two commonly used candidates:

- h_1 = the number of misplaced tiles (blank not included). For Figure 3.25, all eight tiles are out of position, so the start state has $h_1 = 8$. h_1 is an admissible heuristic because any tile that is out of place will require at least one move to get it to the right place.

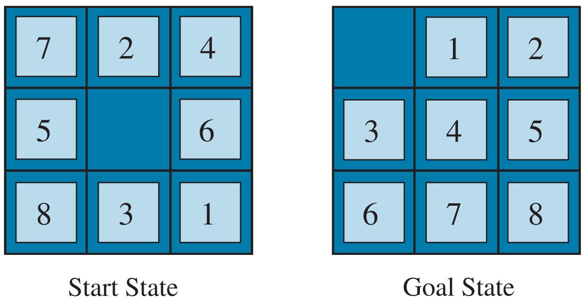


Figure 3.25 A typical instance of the 8-puzzle. The shortest solution is 26 actions long.

- h_2 = the sum of the distances of the tiles from their goal positions. Because tiles cannot move along diagonals, the distance is the sum of the horizontal and vertical distances—sometimes called the **city-block distance** or **Manhattan distance**. h_2 is also admissible because all any move can do is move one tile one step closer to the goal. Tiles 1 to 8 in the start state of Figure 3.25 give a Manhattan distance of

Manhattan distance

$$h_2 = 3+1+2+2+2+3+3+2 = 18\,.$$

As expected, neither of these overestimates the true solution cost, which is 26.

3.6.1 The effect of heuristic accuracy on performance

Effective branching factor

One way to characterize the quality of a heuristic is the **effective branching factor** b^*. If the total number of nodes generated by A* for a particular problem is N and the solution depth is d, then b^* is the branching factor that a uniform tree of depth d would have to have in order to contain $N+1$ nodes. Thus,

$$N+1 = 1+b^*+(b^*)^2+\cdots+(b^*)^d\,.$$

For example, if A* finds a solution at depth 5 using 52 nodes, then the effective branching factor is 1.92. The effective branching factor can vary across problem instances, but usually for a specific domain (such as 8-puzzles) it is fairly constant across all nontrivial problem instances. Therefore, experimental measurements of b^* on a small set of problems can provide a good guide to the heuristic's overall usefulness. A well-designed heuristic would have a value of b^* close to 1, allowing fairly large problems to be solved at reasonable computational cost.

Effective depth

Korf and Reid (1998) argue that a better way to characterize the effect of A* pruning with a given heuristic h is that it reduces the **effective depth** by a constant k_h compared to the true depth. This means that the total search cost is $O(b^{d-k_h})$ compared to $O(b^d)$ for an uninformed search. Their experiments on Rubik's Cube and n-puzzle problems show that this formula gives accurate predictions for total search cost for sampled problem instances across a wide range of solution lengths—at least for solution lengths larger than k_h.

For Figure 3.26 we generated random 8-puzzle problems and solved them with an uninformed breadth-first search and with A* search using both h_1 and h_2, reporting the average number of nodes generated and the corresponding effective branching factor for each search strategy and for each solution length. The results suggest that h_2 is better than h_1, and both are better than no heuristic at all.

	Search Cost (nodes generated)			Effective Branching Factor		
d	BFS	$A^*(h_1)$	$A^*(h_2)$	BFS	$A^*(h_1)$	$A^*(h_2)$
6	128	24	19	2.01	1.42	1.34
8	368	48	31	1.91	1.40	1.30
10	1033	116	48	1.85	1.43	1.27
12	2672	279	84	1.80	1.45	1.28
14	6783	678	174	1.77	1.47	1.31
16	17270	1683	364	1.74	1.48	1.32
18	41558	4102	751	1.72	1.49	1.34
20	91493	9905	1318	1.69	1.50	1.34
22	175921	22955	2548	1.66	1.50	1.34
24	290082	53039	5733	1.62	1.50	1.36
26	395355	110372	10080	1.58	1.50	1.35
28	463234	202565	22055	1.53	1.49	1.36

Figure 3.26 Comparison of the search costs and effective branching factors for 8-puzzle problems using breadth-first search, A* with h_1 (misplaced tiles), and A* with h_2 (Manhattan distance). Data are averaged over 100 puzzles for each solution length d from 6 to 28.

One might ask whether h_2 is *always* better than h_1. The answer is "Essentially, yes." It is easy to see from the definitions of the two heuristics that for any node n, $h_2(n) \geq h_1(n)$. We thus say that h_2 **dominates** h_1. Domination translates directly into efficiency: A* using h_2 will never expand more nodes than A* using h_1 (except in the case of breaking ties unluckily). The argument is simple. Recall the observation on page 90 that every node with $f(n) < C^*$ will surely be expanded. This is the same as saying that every node with $h(n) < C^* - g(n)$ is surely expanded when h is consistent. But because h_2 is at least as big as h_1 for all nodes, every node that is surely expanded by A* search with h_2 is also surely expanded with h_1, and h_1 might cause other nodes to be expanded as well. Hence, it is generally better to use a heuristic function with higher values, provided it is consistent and that the computation time for the heuristic is not too long.

Domination

3.6.2 Generating heuristics from relaxed problems

We have seen that both h_1 (misplaced tiles) and h_2 (Manhattan distance) are fairly good heuristics for the 8-puzzle and that h_2 is better. How might one have come up with h_2? Is it possible for a computer to invent such a heuristic mechanically?

h_1 and h_2 are estimates of the remaining path length for the 8-puzzle, but they are also perfectly accurate path lengths for *simplified* versions of the puzzle. If the rules of the puzzle were changed so that a tile could move anywhere instead of just to the adjacent empty square, then h_1 would give the exact length of the shortest solution. Similarly, if a tile could move one square in any direction, even onto an occupied square, then h_2 would give the exact length of the shortest solution. A problem with fewer restrictions on the actions is called a **relaxed problem**. The state-space graph of the relaxed problem is a *supergraph* of the original state space because the removal of restrictions creates added edges in the graph.

Relaxed problem

Because the relaxed problem adds edges to the state-space graph, any optimal solution in the original problem is, by definition, also a solution in the relaxed problem; but the relaxed problem may have *better* solutions if the added edges provide shortcuts. Hence, *the cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem.* Furthermore, because the derived heuristic is an exact cost for the relaxed problem, it must obey the triangle inequality and is therefore consistent (see page 88).

If a problem definition is written down in a formal language, it is possible to construct relaxed problems automatically.[14] For example, if the 8-puzzle actions are described as

A tile can move from square X to square Y if
X is adjacent to Y **and** Y is blank,

we can generate three relaxed problems by removing one or both of the conditions:

(a) A tile can move from square X to square Y if X is adjacent to Y.
(b) A tile can move from square X to square Y if Y is blank.
(c) A tile can move from square X to square Y.

From (a), we can derive h_2 (Manhattan distance). The reasoning is that h_2 would be the proper score if we moved each tile in turn to its destination. The heuristic derived from (b) is discussed in Exercise 3.GASC. From (c), we can derive h_1 (misplaced tiles) because it would be the proper score if tiles could move to their intended destination in one action. Notice that it is crucial that the relaxed problems generated by this technique can be solved essentially *without search*, because the relaxed rules allow the problem to be decomposed into eight independent subproblems. If the relaxed problem is hard to solve, then the values of the corresponding heuristic will be expensive to obtain.

A program called ABSOLVER can generate heuristics automatically from problem definitions, using the "relaxed problem" method and various other techniques (Prieditis, 1993). ABSOLVER generated a new heuristic for the 8-puzzle that was better than any preexisting heuristic and found the first useful heuristic for the famous Rubik's Cube puzzle.

If a collection of admissible heuristics $h_1 \ldots h_m$ is available for a problem and none of them is clearly better than the others, which should we choose? As it turns out, we can have the best of all worlds, by defining

$$h(n) = \max\{h_1(n), \ldots, h_k(n)\}\,.$$

This composite heuristic picks whichever function is most accurate on the node in question. Because the h_i components are admissible, h is admissible (and if h_i are all consistent, h is consistent). Furthermore, h dominates all of its component heuristics. The only drawback is that $h(n)$ takes longer to compute. If that is an issue, an alternative is to randomly select one of the heuristics at each evaluation, or use a machine learning algorithm to predict which heuristic will be best. Doing this can result in a heuristic that is inconsistent (even if every h_i is consistent), but in practice it usually leads to faster problem solving.

3.6.3 Generating heuristics from subproblems: Pattern databases

Subproblem

Admissible heuristics can also be derived from the solution cost of a **subproblem** of a given problem. For example, Figure 3.27 shows a subproblem of the 8-puzzle instance in Figure 3.25. The subproblem involves getting tiles 1, 2, 3, 4, and the blank into their correct positions. Clearly, the cost of the optimal solution of this subproblem is a lower bound on

[14] In Chapters 8 and 11, we describe formal languages suitable for this task; with formal descriptions that can be manipulated, the construction of relaxed problems can be automated. For now, we use English.

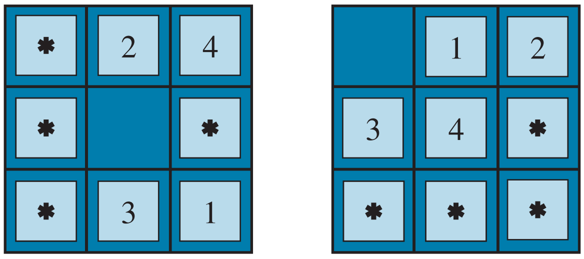


Figure 3.27 A subproblem of the 8-puzzle instance given in Figure 3.25. The task is to get tiles 1, 2, 3, 4, and the blank into their correct positions, without worrying about what happens to the other tiles.

the cost of the complete problem. It turns out to be more accurate than Manhattan distance in some cases.

Pattern database

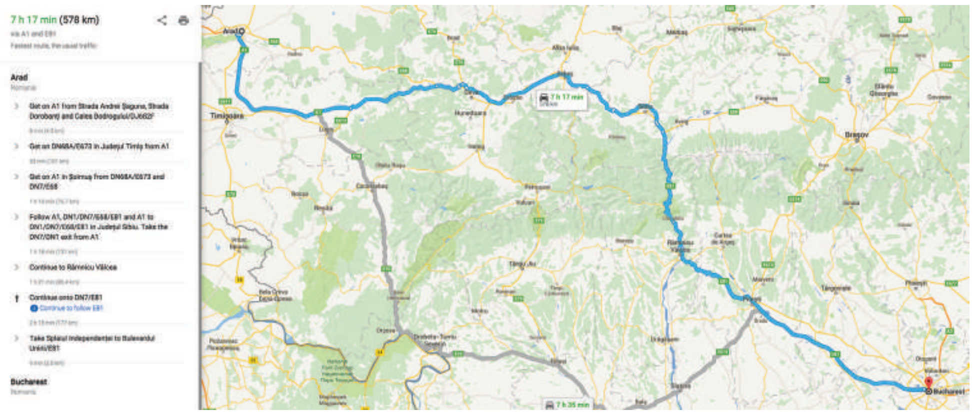
The idea behind **pattern databases** is to store these exact solution costs for every possible subproblem instance—in our example, every possible configuration of the four tiles and the blank. (There will be $9 \times 8 \times 7 \times 6 \times 5 = 15,120$ patterns in the database. The identities of the other four tiles are irrelevant for the purposes of solving the subproblem, but moves of those tiles do count toward the solution cost of the subproblem.) Then we compute an admissible heuristic h_{DB} for each state encountered during a search simply by looking up the corresponding subproblem configuration in the database. The database itself is constructed by searching back from the goal and recording the cost of each new pattern encountered;[15] the expense of this search is amortized over subsequent problem instances, and so makes sense if we expect to be asked to solve many problems.

The choice of tiles 1-2-3-4 to go with the blank is fairly arbitrary; we could also construct databases for 5-6-7-8, for 2-4-6-8, and so on. Each database yields an admissible heuristic, and these heuristics can be combined, as explained earlier, by taking the maximum value. A combined heuristic of this kind is much more accurate than the Manhattan distance; the number of nodes generated when solving random 15-puzzles can be reduced by a factor of 1000. However, with each additional database there are diminishing returns and increased memory and computation costs.

One might wonder whether the heuristics obtained from the 1-2-3-4 database and the 5-6-7-8 could be *added*, since the two subproblems seem not to overlap. Would this still give an admissible heuristic? The answer is no, because the solutions of the 1-2-3-4 subproblem and the 5-6-7-8 subproblem for a given state will almost certainly share some moves—it is unlikely that 1-2-3-4 can be moved into place without touching 5-6-7-8, and vice versa. But what if we don't count those moves—what if we don't abstract the other tiles to stars, but rather make them disappear? That is, we record not the total cost of solving the 1-2-3-4 subproblem, but just the number of moves involving 1-2-3-4. Then the sum of the two costs is still a lower bound on the cost of solving the entire problem. This is the idea behind **disjoint pattern databases**. With such databases, it is possible to solve random 15-puzzles in a few

Disjoint pattern databases

[15] By working backward from the goal, the exact solution cost of every instance encountered is immediately available. This is an example of **dynamic programming**, which we discuss further in Chapter 17.



7 h 17 min (578 km)
via A1 and E81
Fastest route, the usual traffic

Arad
Romania

Get on A1 from Strada Andrei Șaguna, Strada Dorobanți and Calea Bodrogului/DJ682F
8 min (4.3 km)

Get on DN68A/E673 in Județul Timiș from A1
33 min (37 km)

Get on A1 in Șoimuș from DN68A/E673 and DN7/E68
1 h 14 min (76.7 km)

Follow A1, DN1/DN7/E68/E81 and A1 to DN1/DN7/E68/E81 in Județul Sibiu. Take the DN7/DN1 exit from A1
1 h 18 min (130 km)

Continue to Râmnicu Vâlcea
1 h 21 min (85.4 km)

Continue onto DN7/E81
Continue to follow E81
2 h 13 min (177 km)

Take Splaiul Independenței to Bulevardul Unirii/E81
6 min (2.5 km)

Bucharest
Romania

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.

Precomputation

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.

Landmark point

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:

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

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."

from then on will be on an optimal path. Think of the contour lines as following along this optimal path. The search will trace along the optimal path, on each iteration adding an action with cost c to get to a result state whose h-value will be c less, meaning that the total $f = g + h$ score will remain constant at C^* all along the path.

Shortcuts

Some route-finding algorithms save even more time by adding **shortcuts**—artificial edges in the graph that define an optimal multi-action path. For example, if there were shortcuts predefined between all the 100 biggest cities in the U.S., and we were trying to navigate from the Berkeley campus in California to NYU in New York, we could take the shortcut between Sacramento and Manhattan and cover 90% of the path in one action.

$h_L(n)$ is efficient but not admissible. But with a bit more care, we can come up with a heuristic that is both efficient and admissible:

$$h_{DH}(n) = \max_{L \in Landmarks} |C^*(n,L) - C^*(goal,L)|$$

Differential heuristic

This is called a **differential heuristic** (because of the subtraction). Think of this with a landmark that is somewhere out beyond the goal. If the goal happens to be on the optimal path from n to the landmark, then this is saying "consider the entire path from n to L, then subtract off the last part of that path, from *goal* to L, giving us the exact cost of the path from n to *goal*." To the extent that the goal is a bit off of the optimal path to the landmark, the heuristic will be inexact, but still admissible. Landmarks that are not out beyond the goal will not be useful; a landmark that is exactly halfway between n and *goal* will give $h_{DH} = 0$, which is not helpful.

There are several ways to pick landmark points. Selecting points at random is fast, but we get better results if we take care to spread the landmarks out so they are not too close to each other. A greedy approach is to pick a first landmark at random, then find the point that is furthest from that, and add it to the set of landmarks, and continue, at each iteration adding the point that maximizes the distance to the nearest landmark. If you have logs of past search requests by your users, then you can pick landmarks that are frequently requested in searches. For the differential heuristic it is good if the landmarks are spread around the perimeter of the graph. Thus, a good technique is to find the centroid of the graph, arrange k pie-shaped wedges around the centroid, and in each wedge select the vertex that is farthest from the center.

Landmarks work especially well in route-finding problems because of the way roads are laid out in the world: a lot of traffic actually wants to travel between landmarks, so civil engineers build the widest and fastest roads along these routes; landmark search makes it easier to recover these routes.

3.6.5 Learning to search better

We have presented several fixed search strategies—breadth-first, A*, and so on—that have been carefully designed and programmed by computer scientists. Could an agent *learn* how to search better? The answer is yes, and the method rests on an important concept called the **metalevel state space**. Each state in a metalevel state space captures the internal (computational) state of a program that is searching in an ordinary state space such as the map of Romania. (To keep the two concepts separate, we call the map of Romania an **object-level state space**.) For example, the internal state of the A* algorithm consists of the current search tree. Each action in the metalevel state space is a computation step that alters the internal

Metalevel state space

Object-level state space

state; for example, each computation step in A* expands a leaf node and adds its successors to the tree. Thus, Figure 3.18, which shows a sequence of larger and larger search trees, can be seen as depicting a path in the metalevel state space where each state on the path is an object-level search tree.

Now, the path in Figure 3.18 has five steps, including one step, the expansion of Fagaras, that is not especially helpful. For harder problems, there will be many such missteps, and a **metalevel learning** algorithm can learn from these experiences to avoid exploring unpromising subtrees. The techniques used for this kind of learning are described in Chapter 22. The goal of learning is to minimize the **total cost** of problem solving, trading off computational expense and path cost.

Metalevel learning

3.6.6 Learning heuristics from experience

We have seen that one way to invent a heuristic is to devise a relaxed problem for which an optimal solution can be found easily. An alternative is to learn from experience. “Experience” here means solving lots of 8-puzzles, for instance. Each optimal solution to an 8-puzzle problem provides an example (goal, path) pair. From these examples, a learning algorithm can be used to construct a function h that can (with luck) approximate the true path cost for other states that arise during search. Most of these approaches learn an imperfect approximation to the heuristic function, and thus risk inadmissibility. This leads to an inevitable tradeoff between learning time, search run time, and solution cost. Techniques for machine learning are demonstrated in Chapter 19. The reinforcement learning methods described in Chapter 22 are also applicable to search.

Feature

Some machine learning techniques work better when supplied with **features** of a state that are relevant to predicting the state’s heuristic value, rather than with just the raw state description. For example, the feature “number of misplaced tiles” might be helpful in predicting the actual distance of an 8-puzzle state from the goal. Let’s call this feature $x_1(n)$. We could take 100 randomly generated 8-puzzle configurations and gather statistics on their actual solution costs. We might find that when $x_1(n)$ is 5, the average solution cost is around 14, and so on. Of course, we can use multiple features. A second feature $x_2(n)$ might be “number of pairs of adjacent tiles that are not adjacent in the goal state.” How should $x_1(n)$ and $x_2(n)$ be combined to predict $h(n)$? A common approach is to use a linear combination:

$$h(n) = c_1 x_1(n) + c_2 x_2(n) .$$

The constants c_1 and c_2 are adjusted to give the best fit to the actual data across the randomly generated configurations. One expects both c_1 and c_2 to be positive because misplaced tiles and incorrect adjacent pairs make the problem harder to solve. Notice that this heuristic satisfies the condition $h(n) = 0$ for goal states, but it is not necessarily admissible or consistent.

Summary

This chapter has introduced search algorithms that an agent can use to select action sequences in a wide variety of environments—as long as they are episodic, single-agent, fully observable, deterministic, static, discrete, and completely known. There are tradeoffs to be made between the amount of time the search takes, the amount of memory available, and the quality of the solution. We can be more efficient if we have domain-dependent knowledge in the form of a heuristic function that estimates how far a given state is from the goal, or if we precompute partial solutions involving patterns or landmarks.

- Before an agent can start searching, a well-defined **problem** must be formulated.
- A problem consists of five parts: the **initial state**, a set of **actions**, a **transition model** describing the results of those actions, a set of **goal states**, and an **action cost function**.
- The environment of the problem is represented by a **state space graph**. A **path** through the state space (a sequence of actions) from the initial state to a goal state is a **solution**.
- Search algorithms generally treat states and actions as **atomic**, without any internal structure (although we introduced features of states when it came time to do learning).
- Search algorithms are judged on the basis of **completeness**, **cost optimality**, **time complexity**, and **space complexity**.
- **Uninformed search** methods have access only to the problem definition. Algorithms build a search tree in an attempt to find a solution. Algorithms differ based on which node they expand first:
  - **Best-first search** selects nodes for expansion using to an **evaluation function**.
  - **Breadth-first search** expands the shallowest nodes first; it is complete, optimal for unit action costs, but has exponential space complexity.
  - **Uniform-cost search** expands the node with lowest path cost, $g(n)$, and is optimal for general action costs.
  - **Depth-first search** expands the deepest unexpanded node first. It is neither complete nor optimal, but has linear space complexity. **Depth-limited search** adds a depth bound.
  - **Iterative deepening search** calls depth-first search with increasing depth limits until a goal is found. It is complete when full cycle checking is done, optimal for unit action costs, has time complexity comparable to breadth-first search, and has linear space complexity.
  - **Bidirectional search** expands two frontiers, one around the initial state and one around the goal, stopping when the two frontiers meet.
- **Informed search** methods have access to a **heuristic** function $h(n)$ that estimates the cost of a solution from n. They may have access to additional information such as pattern databases with solution costs.
  - **Greedy best-first search** expands nodes with minimal $h(n)$. It is not optimal but is often efficient.
  - **A* search** expands nodes with minimal $f(n) = g(n) + h(n)$. A* is complete and optimal, provided that $h(n)$ is admissible. The space complexity of A* is still an issue for many problems.
  - **Bidirectional A* search** is sometimes more efficient than A* itself.
  - **IDA*** (iterative deepening A* search) is an iterative deepening version of A*, and thus adresses the space complexity issue.
  - **RBFS** (recursive best-first search) and **SMA*** (simplified memory-bounded A*) are robust, optimal search algorithms that use limited amounts of memory; given enough time, they can solve problems for which A* runs out of memory.

  - **Beam search** puts a limit on the size of the frontier; that makes it incomplete and suboptimal, but it often finds reasonably good solutions and runs faster than complete searches.
  - **Weighted A*** search focuses the search towards a goal, expanding fewer nodes, but sacrificing optimality.

- The performance of heuristic search algorithms depends on the quality of the heuristic function. One can sometimes construct good heuristics by relaxing the problem definition, by storing precomputed solution costs for subproblems in a pattern database, by defining landmarks, or by learning from experience with the problem class.

Bibliographical and Historical Notes

The topic of state-space search originated in the early years of AI. Newell and Simon's work on the Logic Theorist (1957) and GPS (1961) led to the establishment of search algorithms as the primary tool for 1960s AI researchers and to the establishment of problem solving as the canonical AI task. Work in operations research by Richard Bellman (1957) showed the importance of additive path costs in simplifying optimization algorithms. The text by Nils Nilsson (1971) established the area on a solid theoretical footing.

The 8-puzzle is a smaller cousin of the 15-puzzle, whose history is recounted at length by Slocum and Sonneveld (2006). In 1880, the 15-puzzle attracted broad attention from the public and mathematicians (Johnson and Story, 1879; Tait, 1880). The editors of the *American Journal of Mathematics* stated, "The '15' puzzle for the last few weeks has been prominently before the American public, and may safely be said to have engaged the attention of nine out of ten persons of both sexes and all ages and conditions of the community," while the *Weekly News-Democrat* of Emporia, Kansas wrote on March 12, 1880 that "It has become literally an epidemic all over the country."

The famous American game designer Sam Loyd falsely claimed to have invented the 15 puzzle (Loyd, 1959); actually it was invented by Noyes Chapman, a postmaster in Canastota, New York, in the mid-1870s (although a generic patent covering sliding blocks was granted to Ernest Kinsey in 1878). Ratner and Warmuth (1986) showed that the general $n \times n$ version of the 15-puzzle belongs to the class of NP-complete problems.

Rubik's Cube was of course invented in 1974 by Ernő Rubik, who also discovered an algorithm for finding good, but not optimal solutions. Korf (1997) found optimal solutions for some random problem instances using pattern databases and IDA* search. Rokicki *et al.* (2014) proved that any instance can be solved in 26 moves (if you consider a 180° twist to be two moves; 20 if it counts as one). The proof consumed 35 CPU years of computation; it does not lead immediately to an efficient algorithm. Agostinelli *et al.* (2019) used reinforcement learning, deep learning networks, and Monte Carlo tree search to learn a much more efficient solver for Rubik's cube. It is not guaranteed to find a cost-optimal solution, but does so about 60% of the time, and typical solutions times are less than a second.

Each of the real-world search problems listed in the chapter has been the subject of a good deal of research effort. Methods for selecting optimal airline flights remain proprietary for the most part, but Carl de Marcken has shown by a reduction to Diophantine decision problems that airline ticket pricing and restrictions have become so convoluted that the problem of selecting an optimal flight is formally *undecidable* (Robinson, 2002). The traveling salesperson problem (TSP) is a standard combinatorial problem in theoretical computer science (Lawler *et al.*, 1992). Karp (1972) proved the TSP decision problem to be NP-hard, but effective heuristic approximation methods were developed (Lin and Kernighan, 1973). Arora (1998) devised a fully polynomial approximation scheme for Euclidean TSPs. VLSI layout methods are surveyed by LaPaugh (2010), and many layout optimization papers appear in VLSI journals. Robotic navigation is discussed in Chapter 26. Automatic assembly sequencing was first demonstrated by FREDDY (Michie, 1972); a comprehensive review is given by (Bahubalendruni and Biswal, 2016).

Uninformed search algorithms are a central topic of computer science (Cormen *et al.*, 2009) and operations research (Dreyfus, 1969). Breadth-first search was formulated for solving mazes by Moore (1959). The method of dynamic programming (Bellman, 1957; Bellman and Dreyfus, 1962), which systematically records solutions for all subproblems of increasing lengths, can be seen as a form of breadth-first search.

Dijkstra's algorithm in the form it is usually presented in (Dijkstra, 1959) is applicable to explicit finite graphs. Nilsson (1971) introduced a version of Dijkstra's algorithm that he called uniform-cost search (because the algorithm "spreads out along contours of equal path cost") that allows for implicitly defined, infinite graphs. Nilsson's work also introduced the idea of closed and open lists, and the term "graph search." The name BEST-FIRST-SEARCH was introduced in the *Handbook of AI* (Barr and Feigenbaum, 1981). The Floyd–Warshall (Floyd, 1962) and Bellman-Ford (Bellman, 1958; Ford, 1956) algorithms allow negative step costs (as long as there are no negative cycles).

A version of iterative deepening designed to make efficient use of the chess clock was first used by Slate and Atkin (1977) in the CHESS 4.5 game-playing program. Martelli's algorithm B (1977) also includes an iterative deepening aspect. The iterative deepening technique was introduced by Bertram Raphael (1976) and came to the fore in work by Korf (1985a).

The use of heuristic information in problem solving appears in an early paper by Simon and Newell (1958), but the phrase "heuristic search" and the use of heuristic functions that estimate the distance to the goal came somewhat later (Newell and Ernst, 1965; Lin, 1965). Doran and Michie (1966) conducted extensive experimental studies of heuristic search. Although they analyzed path length and "penetrance" (the ratio of path length to the total number of nodes examined so far), they appear to have ignored the information provided by the path cost $g(n)$. The A^* algorithm, incorporating the current path cost into heuristic search, was developed by Hart, Nilsson, and Raphael (1968). Dechter and Pearl (1985) studied the conditions under which A^* is optimally efficient (in number of nodes expanded).

The original A^* paper (Hart *et al.*, 1968) introduced the consistency condition on heuristic functions. The monotone condition was introduced by Pohl (1977) as a simpler replacement, but Pearl (1984) showed that the two were equivalent.

Pohl (1977) pioneered the study of the relationship between the error in heuristic functions and the time complexity of A^*. Basic results were obtained for tree-like search with unit action costs and a single goal state (Pohl, 1977; Gaschnig, 1979; Huyn *et al.*, 1980; Pearl, 1984) and with multiple goal states (Dinh *et al.*, 2007). Korf and Reid (1998) showed how to predict the exact number of nodes expanded (not just an asymptotic approximation) on a variety of actual problem domains. The "effective branching factor" was proposed by Nilsson (1971) as an empirical measure of efficiency. For graph search, Helmert and Röger (2008) noted that several well-known problems contained exponentially many nodes on optimal-cost solution paths, implying exponential time complexity for A*.

There are many variations on the A* algorithm. Pohl (1970) introduced weighted A* search, and later a dynamic version (1973), where the weight changes over the depth of the tree. Ebendt and Drechsler (2009) synthesize the results and examine some applications. Hatem and Ruml (2014) show a simplified and improved version of weighted A* that is easier to implement. Wilt and Ruml (2014) introduce speedy search as an alternative to greedy search that focuses on minimizing search time, and Wilt and Ruml (2016) show that the best heuristics for satisficing search are different from the ones for optimal search. Burns *et al.* (2012) give some implementation tricks for writing fast search code, and Felner (2018) considers how the implementation changes when using an early goal test.

Pohl (1971) introduced bidirectional search. Holte *et al.* (2016) describe the version of bidirectional search that is guaranteed to meet in the middle, making it more widely applicable. Eckerle *et al.* (2017) describe the set of surely expanded pairs of nodes, and show that no bidirectional search can be optimally efficient. The NBS algorithm (Chen *et al.*, 2017) makes explicit use of a queue of pairs of nodes.

A combination of bidirectional A* and known landmarks was used to efficiently find driving routes for Microsoft's online map service (Goldberg *et al.*, 2006). After caching a set of paths between landmarks, the algorithm can find an optimal-cost path between any pair of points in a 24-million-point graph of the United States, searching less than 0.1% of the graph. Korf (1987) shows how to use subgoals, macro-operators, and abstraction to achieve remarkable speedups over previous techniques. Delling *et al.* (2009) describe how to use bidirectional search, landmarks, hierarchical structure, and other tricks to find driving routes. Anderson *et al.* (2008) describe a related technique, called **coarse-to-fine search**, which can be thought of as defining landmarks at various hierarchical levels of abstraction. Korf (1987) describes conditions under which coarse-to-fine search provides an exponential speedup. Knoblock (1991) provides experimental results and analysis to quantify the advantages of hierarchical search.

Coarse-to-fine search

A* and other state-space search algorithms are closely related to the **branch-and-bound** techniques that are widely used in operations research (Lawler and Wood, 1966; Rayward-Smith *et al.*, 1996). Kumar and Kanal (1988) attempt a "grand unification" of heuristic search, dynamic programming, and branch-and-bound techniques under the name of CDP—the "composite decision process."

Branch-and-bound

Because most computers in the 1960s had only a few thousand words of main memory, memory-bounded heuristic search was an early research topic. The Graph Traverser (Doran and Michie, 1966), one of the earliest search programs, commits to an action after searching best-first up to the memory limit. IDA* (Korf, 1985b) was the first widely used length-optimal, memory-bounded heuristic search algorithm, and a large number of variants have been developed. An analysis of the efficiency of IDA* and of its difficulties with real-valued heuristics appears in Patrick *et al.* (1992).

The original version of RBFS (Korf, 1993) is actually somewhat more complicated than the algorithm shown in Figure 3.22, which is actually closer to an independently developed algorithm called **iterative expansion** or IE (Russell, 1992). RBFS uses a lower bound as well as the upper bound; the two algorithms behave identically with admissible heuristics, but RBFS expands nodes in best-first order even with an inadmissible heuristic. The idea of

Iterative expansion

keeping track of the best alternative path appeared earlier in Bratko's (2009) elegant Prolog implementation of A* and in the DTA* algorithm (Russell and Wefald, 1991). The latter work also discusses metalevel state spaces and metalevel learning.

The MA* algorithm appeared in Chakrabarti *et al.* (1989). SMA*, or Simplified MA*, emerged from an attempt to implement MA* (Russell, 1992). Kaindl and Khorsand (1994) applied SMA* to produce a bidirectional search algorithm that was substantially faster than previous algorithms. Korf and Zhang (2000) describe a divide-and-conquer approach, and Zhou and Hansen (2002) introduce memory-bounded A* graph search and a strategy for switching to breadth-first search to increase memory-efficiency (Zhou and Hansen, 2006).

The idea that admissible heuristics can be derived by problem relaxation appears in the seminal paper by Held and Karp (1970), who used the minimum-spanning-tree heuristic to solve the TSP. (See Exercise 3.MSTR.) The automation of the relaxation process was implemented successfully by Prieditis (1993). There is a growing literature on the application of machine learning to discover heuristic functions (Samadi *et al.*, 2008; Arfaee *et al.*, 2010; Thayer *et al.*, 2011; Lelis *et al.*, 2012).

The use of pattern databases to derive admissible heuristics is due to Gasser (1995) and Culberson and Schaeffer (1996, 1998); disjoint pattern databases are described by Korf and Felner (2002); a similar method using symbolic patterns is due to Edelkamp (2009). Felner *et al.* (2007) show how to compress pattern databases to save space. The probabilistic interpretation of heuristics was investigated by Pearl (1984) and Hansson and Mayer (1989).

Pearl's (1984) *Heuristics* and Edelkamp and Schrödl's (2012) *Heuristic Search* are influential textbooks on search. Papers about new search algorithms appear at the International Symposium on Combinatorial Search (SoCS) and the International Conference on Automated Planning and Scheduling (ICAPS), as well as in general AI conferences such as AAAI and IJCAI, and journals such as *Artificial Intelligence* and *Journal of the ACM*.

CHAPTER 4

SEARCH IN COMPLEX ENVIRONMENTS

In which we relax the simplifying assumptions of the previous chapter, to get closer to the real world.

Chapter 3 addressed problems in fully observable, deterministic, static, known environments where the solution is a sequence of actions. In this chapter, we relax those constraints. We begin with the problem of finding a good state without worrying about the path to get there, covering both discrete (Section 4.1) and continuous (Section 4.2) states. Then we relax the assumptions of determinism (Section 4.3) and observability (Section 4.4). In a nondeterministic world, the agent will need a conditional plan and carry out different actions depending on what it observes—for example, stopping if the light is red and going if it is green. With partial observability, the agent will also need to keep track of the possible states it might be in. Finally, Section 4.5 guides the agent through an unknown space that it must learn as it goes, using **online search**.

4.1 Local Search and Optimization Problems

In the search problems of Chapter 3 we wanted to find paths through the search space, such as a path from Arad to Bucharest. But sometimes we care only about the final state, not the path to get there. For example, in the 8-queens problem (Figure 4.3), we care only about finding a valid final configuration of 8 queens (because if you know the configuration, it is trivial to reconstruct the steps that created it). This is also true for many important applications such as integrated-circuit design, factory floor layout, job shop scheduling, automatic programming, telecommunications network optimization, crop planning, and portfolio management.

Local search

Local search algorithms operate by searching from a start state to neighboring states, without keeping track of the paths, nor the set of states that have been reached. That means they are not systematic—they might never explore a portion of the search space where a solution actually resides. However, they have two key advantages: (1) they use very little memory; and (2) they can often find reasonable solutions in large or infinite state spaces for which systematic algorithms are unsuitable.

Optimization problem

Objective function

Local search algorithms can also solve **optimization problems**, in which the aim is to find the best state according to an **objective function**.

State-space landscape

To understand local search, consider the states of a problem laid out in a **state-space landscape**, as shown in Figure 4.1. Each point (state) in the landscape has an "elevation," defined by the value of the objective function. If elevation corresponds to an objective function,

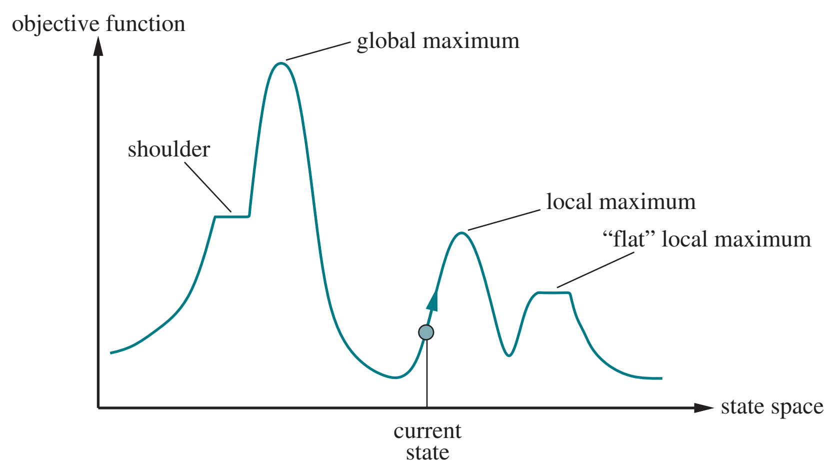


Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum.

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  current ← problem.INITIAL
  while true do
     neighbor ← a highest-valued successor state of current
     if VALUE(neighbor) ≤ VALUE(current) then return current
     current ← neighbor
```

Figure 4.2 The hill-climbing search algorithm, which is the most basic local search technique. At each step the current node is replaced by the best neighbor.

then the aim is to find the highest peak—a **global maximum**—and we call the process **hill climbing**. If elevation corresponds to cost, then the aim is to find the lowest valley—a **global minimum**—and we call it **gradient descent**.

Global maximum

Global minimum

4.1.1 Hill-climbing search

Hill climbing

Steepest ascent

The **hill-climbing** search algorithm is shown in Figure 4.2. It keeps track of one current state and on each iteration moves to the neighboring state with highest value—that is, it heads in the direction that provides the **steepest ascent**. It terminates when it reaches a "peak" where no neighbor has a higher value. Hill climbing does not look ahead beyond the immediate neighbors of the current state. This resembles trying to find the top of Mount Everest in a thick fog while suffering from amnesia. Note that one way to use hill-climbing search is to use the negative of a heuristic cost function as the objective function; that will climb locally to the state with smallest heuristic distance to the goal.

Complete-state formulation

To illustrate hill climbing, we will use the **8-queens problem** (Figure 4.3). We will use a **complete-state formulation**, which means that every state has all the components of a solution, but they might not all be in the right place. In this case every state has 8 queens

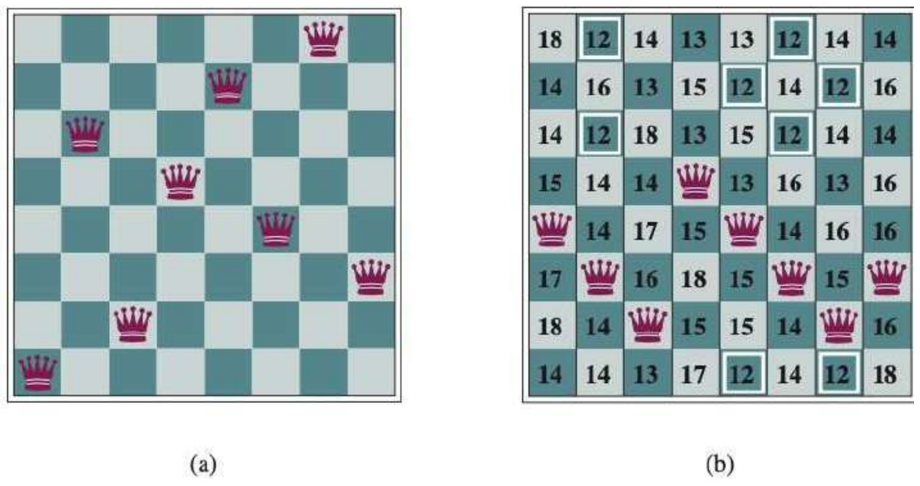


Figure 4.3 (a) The 8-queens problem: place 8 queens on a chess board so that no queen attacks another. (A queen attacks any piece in the same row, column, or diagonal.) This position is almost a solution, except for the two queens in the fourth and seventh columns that attack each other along the diagonal. (b) An 8-queens state with heuristic cost estimate $h = 17$. The board shows the value of h for each possible successor obtained by moving a queen within its column. There are 8 moves that are tied for best, with $h = 12$. The hill-climbing algorithm will pick one of these.

on the board, one per column. The initial state is chosen at random, and the successors of a state are all possible states generated by moving a single queen to another square in the same column (so each state has $8 \times 7 = 56$ successors). The heuristic cost function h is the number of pairs of queens that are attacking each other; this will be zero only for solutions. (It counts as an attack if two pieces are in the same line, even if there is an intervening piece between them.) Figure 4.3(b) shows a state that has $h = 17$. The figure also shows the h values of all its successors.

Greedy local search

Hill climbing is sometimes called **greedy local search** because it grabs a good neighbor state without thinking ahead about where to go next. Although greed is considered one of the seven deadly sins, it turns out that greedy algorithms often perform quite well. Hill climbing can make rapid progress toward a solution because it is usually quite easy to improve a bad state. For example, from the state in Figure 4.3(b), it takes just five steps to reach the state in Figure 4.3(a), which has $h = 1$ and is very nearly a solution. Unfortunately, hill climbing can get stuck for any of the following reasons:

Local maximum

- **Local maxima**: A local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upward toward the peak but will then be stuck with nowhere else to go. Figure 4.1 illustrates the problem schematically. More concretely, the state in Figure 4.3(a) is a local maximum (i.e., a local minimum for the cost h); every move of a single queen makes the situation worse.

Ridge

- **Ridges**: A ridge is shown in Figure 4.4. Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.

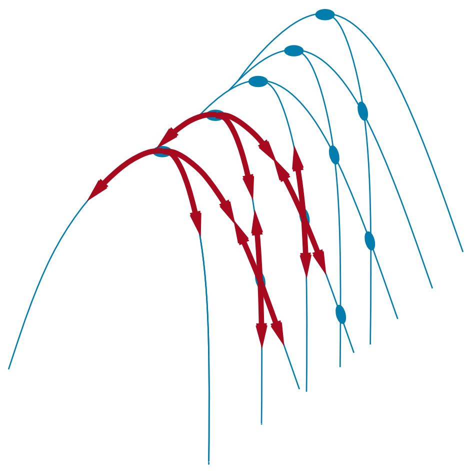


Figure 4.4 Illustration of why ridges cause difficulties for hill climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point downhill. Topologies like this are common in low-dimensional state spaces, such as points in a two-dimensional plane. But in state spaces with hundreds or thousands of dimensions, this intuitive picture does not hold, and there are usually at least a few dimensions that make it possible to escape from ridges and plateaus.

- **Plateaus**: A plateau is a flat area of the state-space landscape. It can be a flat local maximum, from which no uphill exit exists, or a **shoulder**, from which progress is possible. (See Figure 4.1.) A hill-climbing search can get lost wandering on the plateau.

Plateau

Shoulder

In each case, the algorithm reaches a point at which no progress is being made. Starting from a randomly generated 8-queens state, steepest-ascent hill climbing gets stuck 86% of the time, solving only 14% of problem instances. On the other hand, it works quickly, taking just 4 steps on average when it succeeds and 3 when it gets stuck—not bad for a state space with $8^8 \approx 17$ million states.

How could we solve more problems? One answer is to keep going when we reach a plateau—to allow a **sideways move** in the hope that the plateau is really a shoulder, as shown in Figure 4.1. But if we are actually on a flat local maximum, then this approach will wander on the plateau forever. Therefore, we can limit the number of consecutive sideways moves, stopping after, say, 100 consecutive sideways moves. This raises the percentage of problem instances solved by hill climbing from 14% to 94%. Success comes at a cost: the algorithm averages roughly 21 steps for each successful instance and 64 for each failure.

Sideways move

Many variants of hill climbing have been invented. **Stochastic hill climbing** chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move. This usually converges more slowly than steepest ascent, but in some state landscapes, it finds better solutions. **First-choice hill climbing** implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state. This is a good strategy when a state has many (e.g., thousands) of successors.

Stochastic hill climbing

First-choice hill climbing

Another variant is **random-restart hill climbing**, which adopts the adage, "If at first you don't succeed, try, try again." It conducts a series of hill-climbing searches from randomly

Random-restart hill climbing

generated initial states, until a goal is found. It is complete with probability 1, because it will eventually generate a goal state as the initial state. If each hill-climbing search has a probability p of success, then the expected number of restarts required is $1/p$. For 8-queens instances with no sideways moves allowed, $p \approx 0.14$, so we need roughly 7 iterations to find a goal (6 failures and 1 success). The expected number of steps is the cost of one successful iteration plus $(1-p)/p$ times the cost of failure, or roughly 22 steps in all. When we allow sideways moves, $1/0.94 \approx 1.06$ iterations are needed on average and $(1\times 21)+(0.06/0.94)\times 64 \approx 25$ steps. For 8-queens, then, random-restart hill climbing is very effective indeed. Even for three million queens, the approach can find solutions in seconds.[1]

The success of hill climbing depends very much on the shape of the state-space landscape: if there are few local maxima and plateaus, random-restart hill climbing will find a good solution very quickly. On the other hand, many real problems have a landscape that looks more like a widely scattered family of balding porcupines on a flat floor, with miniature porcupines living on the tip of each porcupine needle. NP-hard problems (see Appendix A) typically have an exponential number of local maxima to get stuck on. Despite this, a reasonably good local maximum can often be found after a small number of restarts.

4.1.2 Simulated annealing

A hill-climbing algorithm that never makes "downhill" moves toward states with lower value (or higher cost) is always vulnerable to getting stuck in a local maximum. In contrast, a purely random walk that moves to a successor state without concern for the value will eventually stumble upon the global maximum, but will be extremely inefficient. Therefore, it seems reasonable to try to combine hill climbing with a random walk in a way that yields both efficiency and completeness.

Simulated annealing

Simulated annealing is such an algorithm. In metallurgy, **annealing** is the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to reach a low-energy crystalline state. To explain simulated annealing, we switch our point of view from hill climbing to **gradient descent** (i.e., minimizing cost) and imagine the task of getting a ping-pong ball into the deepest crevice in a very bumpy surface. If we just let the ball roll, it will come to rest at a local minimum. If we shake the surface, we can bounce the ball out of the local minimum—perhaps into a deeper local minimum, where it will spend more time. The trick is to shake just hard enough to bounce the ball out of local minima but not hard enough to dislodge it from the global minimum. The simulated-annealing solution is to start by shaking hard (i.e., at a high temperature) and then gradually reduce the intensity of the shaking (i.e., lower the temperature).

The overall structure of the simulated-annealing algorithm (Figure 4.5) is similar to hill climbing. Instead of picking the *best* move, however, it picks a *random* move. If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1. The probability decreases exponentially with the "badness" of the move—the amount ΔE by which the evaluation is worsened. The probability also decreases as the "temperature" T goes down: "bad" moves are more likely to be allowed at the start when T is high, and they become more unlikely as T decreases. If the *schedule* lowers T to 0 slowly enough, then a property of the Boltzmann distribution, $e^{\Delta E/T}$, is that

[1] Luby *et al.* (1993) suggest restarting after a fixed number of steps and show that this can be *much* more efficient than letting each search continue indefinitely.

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  current ← problem.INITIAL
  for t = 1 to ∞ do
     T ← schedule(t)
     if T = 0 then return current
     next ← a randomly selected successor of current
     ΔE ← VALUE(current) – VALUE(next)
     if ΔE > 0 then current ← next
     else current ← next only with probability $e^{-\Delta E/T}$
```

Figure 4.5 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The *schedule* input determines the value of the "temperature" T as a function of time.

all the probability is concentrated on the global maxima, which the algorithm will find with probability approaching 1.

Simulated annealing was used to solve VLSI layout problems beginning in the 1980s. It has been applied widely to factory scheduling and other large-scale optimization tasks.

4.1.3 Local beam search

Keeping just one node in memory might seem to be an extreme reaction to the problem of memory limitations. The **local beam search** algorithm keeps track of k states rather than just one. It begins with k randomly generated states. At each step, all the successors of all k states are generated. If any one is a goal, the algorithm halts. Otherwise, it selects the k best successors from the complete list and repeats.

Local beam search

At first sight, a local beam search with k states might seem to be nothing more than running k random restarts in parallel instead of in sequence. In fact, the two algorithms are quite different. In a random-restart search, each search process runs independently of the others. *In a local beam search, useful information is passed among the parallel search threads.* In effect, the states that generate the best successors say to the others, "Come over here, the grass is greener!" The algorithm quickly abandons unfruitful searches and moves its resources to where the most progress is being made.

Local beam search can suffer from a lack of diversity among the k states—they can become clustered in a small region of the state space, making the search little more than a k-times-slower version of hill climbing. A variant called **stochastic beam search**, analogous to stochastic hill climbing, helps alleviate this problem. Instead of choosing the top k successors, stochastic beam search chooses successors with probability proportional to the successor's value, thus increasing diversity.

Stochastic beam search

4.1.4 Evolutionary algorithms

Evolutionary algorithms can be seen as variants of stochastic beam search that are explicitly motivated by the metaphor of natural selection in biology: there is a population of individuals (states), in which the fittest (highest value) individuals produce offspring (successor states) that populate the next generation, a process called **recombination**. There are endless forms of evolutionary algorithms, varying in the following ways:

Evolutionary algorithms

Recombination

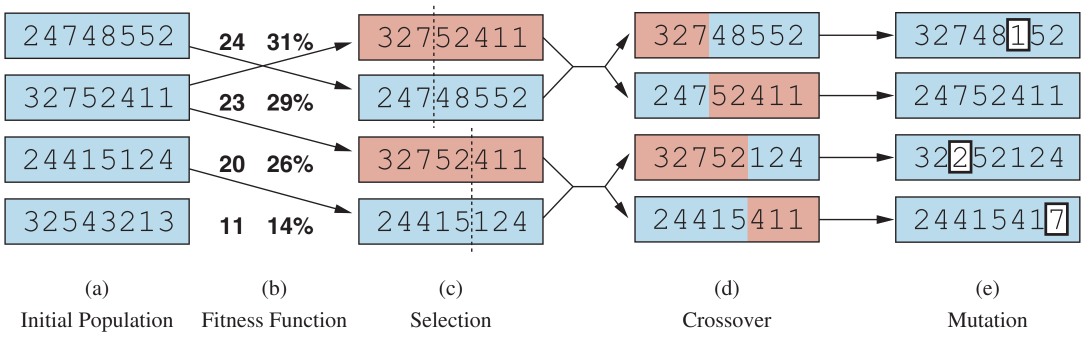


Figure 4.6 A genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by a fitness function in (b) resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

- The size of the population.
- Genetic algorithm Evolution strategies Genetic programming The representation of each individual. In **genetic algorithms**, each individual is a string over a finite alphabet (often a Boolean string), just as DNA is a string over the alphabet **ACGT**. In **evolution strategies**, an individual is a sequence of real numbers, and in **genetic programming** an individual is a computer program.
- The mixing number, ρ, which is the number of parents that come together to form offspring. The most common case is $\rho = 2$: two parents combine their "genes" (parts of their representation) to form offspring. When $\rho = 1$ we have stochastic beam search (which can be seen as asexual reproduction). It is possible to have $\rho > 2$, which occurs only rarely in nature but is easy enough to simulate on computers.
- Selection The **selection** process for selecting the individuals who will become the parents of the next generation: one possibility is to select from all individuals with probability proportional to their fitness score. Another possibility is to randomly select n individuals ($n > \rho$), and then select the ρ most fit ones as parents.
- Crossover point The recombination procedure. One common approach (assuming $\rho = 2$), is to randomly select a **crossover point** to split each of the parent strings, and recombine the parts to form two children, one with the first part of parent 1 and the second part of parent 2; the other with the second part of parent 1 and the first part of parent 2.
- Mutation rate The **mutation rate**, which determines how often offspring have random mutations to their representation. Once an offspring has been generated, every bit in its composition is flipped with probability equal to the mutation rate.
- Elitism The makeup of the next generation. This can be just the newly formed offspring, or it can include a few top-scoring parents from the previous generation (a practice called **elitism**, which guarantees that overall fitness will never decrease over time). The practice of **culling**, in which all individuals below a given threshold are discarded, can lead to a speedup (Baum *et al.*, 1995).

Figure 4.6(a) shows a population of four 8-digit strings, each representing a state of the 8-queens puzzle: the c-th digit represents the row number of the queen in column c. In (b), each state is rated by the fitness function. Higher fitness values are better, so for the 8-

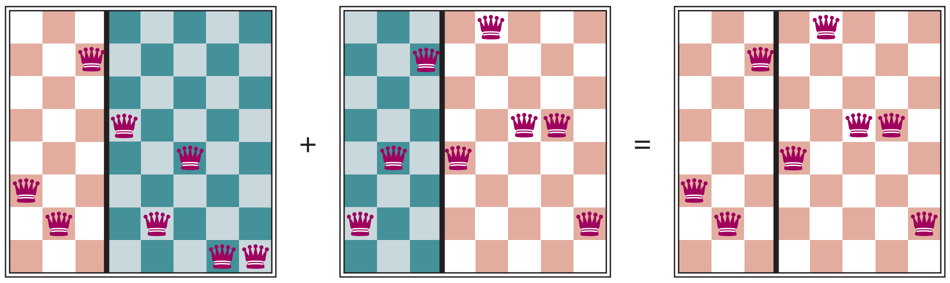


Figure 4.7 The 8-queens states corresponding to the first two parents in Figure 4.6(c) and the first offspring in Figure 4.6(d). The green columns are lost in the crossover step and the red columns are retained. (To interpret the numbers in Figure 4.6: row 1 is the bottom row, and 8 is the top row.)

queens problem we use the number of *nonattacking* pairs of queens, which has a value of $8 \times 7/2 = 28$ for a solution. The values of the four states in (b) are 24, 23, 20, and 11. The fitness scores are then normalized to probabilities, and the resulting values are shown next to the fitness values in (b).

In (c), two pairs of parents are selected, in accordance with the probabilities in (b). Notice that one individual is selected twice and one not at all. For each selected pair, a crossover point (dotted line) is chosen randomly. In (d), we cross over the parent strings at the crossover points, yielding new offspring. For example, the first child of the first pair gets the first three digits (327) from the first parent and the remaining digits (48552) from the second parent. The 8-queens states involved in this recombination step are shown in Figure 4.7.

Finally, in (e), each location in each string is subject to random mutation with a small independent probability. One digit was mutated in the first, third, and fourth offspring. In the 8-queens problem, this corresponds to choosing a queen at random and moving it to a random square in its column. It is often the case that the population is diverse early on in the process, so crossover frequently takes large steps in the state space early in the search process (as in simulated annealing). After many generations of selection towards higher fitness, the population becomes less diverse, and smaller steps are typical. Figure 4.8 describes an algorithm that implements all these steps.

Genetic algorithms are similar to stochastic beam search, but with the addition of the crossover operation. This is advantageous if there are blocks that perform useful functions. For example, it could be that putting the first three queens in positions 2, 4, and 6 (where they do not attack each other) constitutes a useful block that can be combined with other useful blocks that appear in other individuals to construct a solution. It can be shown mathematically that, if the blocks do not serve a purpose—for example if the positions of the genetic code are randomly permuted—then crossover conveys no advantage.

The theory of genetic algorithms explains how this works using the idea of a **schema**, which is a substring in which some of the positions can be left unspecified. For example, the schema 246***** describes all 8-queens states in which the first three queens are in positions 2, 4, and 6, respectively. Strings that match the schema (such as 24613578) are called **instances** of the schema. It can be shown that if the average fitness of the instances of a schema is above the mean, then the number of instances of the schema will grow over time.

Schema

Instance

Evolution and Search

The theory of **evolution** was developed by Charles Darwin in *On the Origin of Species by Means of Natural Selection* (1859) and independently by Alfred Russel Wallace (1858). The central idea is simple: variations occur in reproduction and will be preserved in successive generations approximately in proportion to their effect on reproductive fitness.

Darwin's theory was developed with no knowledge of how the traits of organisms can be inherited and modified. The probabilistic laws governing these processes were first identified by Gregor Mendel (1866), a monk who experimented with sweet peas. Much later, Watson and Crick (1953) identified the structure of the DNA molecule and its alphabet, AGTC (adenine, guanine, thymine, cytosine). In the standard model, variation occurs both by point mutations in the letter sequence and by "crossover" (in which the DNA of an offspring is generated by combining long sections of DNA from each parent).

The analogy to local search algorithms has already been described; the principal difference between stochastic beam search and evolution is the use of *sexual* reproduction, wherein successors are generated from *multiple* individuals rather than just one. The actual mechanisms of evolution are, however, far richer than most genetic algorithms allow. For example, mutations can involve reversals, duplications, and movement of large chunks of DNA; some viruses borrow DNA from one organism and insert it into another; and there are transposable genes that do nothing but copy themselves many thousands of times within the genome.

There are even genes that poison cells from potential mates that do not carry the gene, thereby increasing their own chances of replication. Most important is the fact that the *genes themselves encode the mechanisms* whereby the genome is reproduced and translated into an organism. In genetic algorithms, those mechanisms are a separate program that is not represented within the strings being manipulated.

Darwinian evolution may appear inefficient, having generated blindly some 10^{43} or so organisms without improving its search heuristics one iota. But learning does play a role in evolution. Although the otherwise great French naturalist Jean Lamarck (1809) was wrong to propose that traits acquired by adaptation during an organism's lifetime would be passed on to its offspring, James Baldwin's (1896) superficially similar theory is correct: learning can effectively relax the fitness landscape, leading to an acceleration in the rate of evolution. An organism that has a trait that is not quite adaptive for its environment will pass on the trait if it also has enough plasticity to learn to adapt to the environment in a way that is beneficial. Computer simulations (Hinton and Nowlan, 1987) confirm that this **Baldwin effect** is real, and that a consequence is that things that are hard to learn end up in the genome, but things that are easy to learn need not reside there (Morgan and Griffiths, 2015).

```
function GENETIC-ALGORITHM(population, fitness) returns an individual
   repeat
      weights ← WEIGHTED-BY(population, fitness)
      population2 ← empty list
      for i = 1 to SIZE(population) do
         parent1, parent2 ← WEIGHTED-RANDOM-CHOICES(population, weights, 2)
         child ← REPRODUCE(parent1, parent2)
         if (small random probability) then child ← MUTATE(child)
         add child to population2
      population ← population2
   until some individual is fit enough, or enough time has elapsed
   return the best individual in population, according to fitness

function REPRODUCE(parent1, parent2) returns an individual
   n ← LENGTH(parent1)
   c ← random number from 1 to n
   return APPEND(SUBSTRING(parent1, 1, c), SUBSTRING(parent2, c + 1, n))
```

Figure 4.8 A genetic algorithm. Within the function, *population* is an ordered list of individuals, *weights* is a list of corresponding fitness values for each individual, and *fitness* is a function to compute these values.

Clearly, this effect is unlikely to be significant if adjacent bits are totally unrelated to each other, because then there will be few contiguous blocks that provide a consistent benefit. Genetic algorithms work best when schemas correspond to meaningful components of a solution. For example, if the string is a representation of an antenna, then the schemas may represent components of the antenna, such as reflectors and deflectors. A good component is likely to be good in a variety of different designs. This suggests that successful use of genetic algorithms requires careful engineering of the representation.

In practice, genetic algorithms have their place within the broad landscape of optimization methods (Marler and Arora, 2004), particularly for complex structured problems such as circuit layout or job-shop scheduling, and more recently for evolving the architecture of deep neural networks (Miikkulainen *et al.*, 2019). It is not clear how much of the appeal of genetic algorithms arises from their superiority on specific tasks, and how much from the appealing metaphor of evolution.

4.2 Local Search in Continuous Spaces

In Chapter 2, we explained the distinction between discrete and continuous environments, pointing out that most real-world environments are continuous. A continuous action space has an infinite branching factor, and thus can't be handled by most of the algorithms we have covered so far (with the exception of first-choice hill climbing and simulated annealing).

This section provides a *very brief* introduction to some local search techniques for continuous spaces. The literature on this topic is vast; many of the basic techniques originated in the 17th century, after the development of calculus by Newton and Leibniz.[2] We find uses for these techniques in several places in this book, including the chapters on learning, vision, and robotics.

We begin with an example. Suppose we want to place three new airports anywhere in Romania, such that the sum of squared straight-line distances from each city on the map to its nearest airport is minimized. (See Figure 3.1 for the map of Romania.) The state space is then defined by the coordinates of the three airports: (x_1,y_1), (x_2,y_2), and (x_3,y_3). This is a *six-dimensional* space; we also say that states are defined by six **variables**. In general, states are defined by an n-dimensional vector of variables, $\mathbf{x}$. Moving around in this space corresponds to moving one or more of the airports on the map. The objective function $f(\mathbf{x}) = f(x_1,y_1,x_2,y_2,x_3,y_3)$ is relatively easy to compute for any particular state once we compute the closest cities. Let C_i be the set of cities whose closest airport (in the state $\mathbf{x}$) is airport i. Then, we have

Variable

$$f(\mathbf{x}) = f(x_1,y_1,x_2,y_2,x_3,y_3) = \sum_{i=1}^{3}\sum_{c\in C_i}(x_i-x_c)^2+(y_i-y_c)^2\,. \qquad (4.1)$$

This equation is correct not only for the state $\mathbf{x}$ but also for states in the local neighborhood of $\mathbf{x}$. However, it is not correct globally; if we stray too far from x (by altering the location of one or more of the airports by a large amount) then the set of closest cities for that airport changes, and we need to recompute C_i.

Discretization

One way to deal with a continuous state space is to **discretize** it. For example, instead of allowing the (x_i,y_i) locations to be any point in continuous two-dimensional space, we could limit them to fixed points on a rectangular grid with spacing of size δ (delta). Then instead of having an infinite number of successors, each state in the space would have only 12 successors, corresponding to incrementing one of the 6 variables by $\pm\delta$. We can then apply any of our local search algorithms to this discrete space. Alternatively, we could make the branching factor finite by sampling successor states randomly, moving in a random direction by a small amount, δ. Methods that measure progress by the change in the value of the objective function between two nearby points are called **empirical gradient** methods. Empirical gradient search is the same as steepest-ascent hill climbing in a discretized version of the state space. Reducing the value of δ over time can give a more accurate solution, but does not necessarily converge to a global optimum in the limit.

Empirical gradient

Often we have an objective function expressed in a mathematical form such that we can use calculus to solve the problem analytically rather than empirically. Many methods attempt to use the **gradient** of the landscape to find a maximum. The gradient of the objective function is a vector ∇f that gives the magnitude and direction of the steepest slope. For our problem, we have

Gradient

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3}\right).$$

In some cases, we can find a maximum by solving the equation $\nabla f = 0$. (This could be done, for example, if we were placing just one airport; the solution is the arithmetic mean of all the cities' coordinates.) In many cases, however, this equation cannot be solved in closed form. For example, with three airports, the expression for the gradient depends on what cities are

[2] Knowledge of vectors, matrices, and derivatives is useful for this section (see Appendix A).

closest to each airport in the current state. This means we can compute the gradient *locally* (but not *globally*); for example,

$$\frac{\partial f}{\partial x_1} = 2 \sum_{c \in C_1} (x_1 - x_c) . \tag{4.2}$$

Given a locally correct expression for the gradient, we can perform steepest-ascent hill climbing by updating the current state according to the formula

$$\mathbf{x} \leftarrow \mathbf{x} + \alpha \nabla f(\mathbf{x}) ,$$

where α (alpha) is a small constant often called the **step size**. There exist a huge variety of methods for adjusting α. The basic problem is that if α is too small, too many steps are needed; if α is too large, the search could overshoot the maximum. The technique of **line search** tries to overcome this dilemma by extending the current gradient direction—usually by repeatedly doubling α—until f starts to decrease again. The point at which this occurs becomes the new current state. There are several schools of thought about how the new direction should be chosen at this point.

Step size

Line search

For many problems, the most effective algorithm is the venerable **Newton–Raphson** method. This is a general technique for finding roots of functions—that is, solving equations of the form $g(x) = 0$. It works by computing a new estimate for the root x according to Newton's formula

Newton–Raphson

$$x \leftarrow x - g(x)/g'(x) .$$

To find a maximum or minimum of f, we need to find $\mathbf{x}$ such that the *gradient* is a zero vector (i.e., $\nabla f(\mathbf{x}) = \mathbf{0}$). Thus, $g(x)$ in Newton's formula becomes $\nabla f(\mathbf{x})$, and the update equation can be written in matrix–vector form as

$$\mathbf{x} \leftarrow \mathbf{x} - \mathbf{H}_f^{-1}(\mathbf{x}) \nabla f(\mathbf{x}) ,$$

where $\mathbf{H}_f(\mathbf{x})$ is the **Hessian** matrix of second derivatives, whose elements H_{ij} are given by $\partial^2 f / \partial x_i \partial x_j$. For our airport example, we can see from Equation (4.2) that $\mathbf{H}_f(\mathbf{x})$ is particularly simple: the off-diagonal elements are zero and the diagonal elements for airport i are just twice the number of cities in C_i. A moment's calculation shows that one step of the update moves airport i directly to the centroid of C_i, which is the minimum of the local expression for f from Equation (4.1).[3] For high-dimensional problems, however, computing the n^2 entries of the Hessian and inverting it may be expensive, so many approximate versions of the Newton–Raphson method have been developed.

Hessian

Local search methods suffer from local maxima, ridges, and plateaus in continuous state spaces just as much as in discrete spaces. Random restarts and simulated annealing are often helpful. High-dimensional continuous spaces are, however, big places in which it is very easy to get lost.

A final topic is **constrained optimization**. An optimization problem is constrained if solutions must satisfy some hard constraints on the values of the variables. For example, in our airport-siting problem, we might constrain sites to be inside Romania and on dry land (rather than in the middle of lakes). The difficulty of constrained optimization problems depends on the nature of the constraints and the objective function. The best-known category is that of **linear programming** problems, in which constraints must be linear inequalities

Constrained optimization

Linear programming

[3] In general, the Newton–Raphson update can be seen as fitting a quadratic surface to f at $\mathbf{x}$ and then moving directly to the minimum of that surface—which is also the minimum of f if f is quadratic.

Convex set

forming a **convex set**[4] and the objective function is also linear. The time complexity of linear programming is polynomial in the number of variables.

Convex optimization

Linear programming is probably the most widely studied and broadly useful method for optimization. It is a special case of the more general problem of **convex optimization**, which allows the constraint region to be any convex region and the objective to be any function that is convex within the constraint region. Under certain conditions, convex optimization problems are also polynomially solvable and may be feasible in practice with thousands of variables. Several important problems in machine learning and control theory can be formulated as convex optimization problems (see Chapter 20).

4.3 Search with Nondeterministic Actions

In Chapter 3, we assumed a fully observable, deterministic, known environment. Therefore, an agent can observe the initial state, calculate a sequence of actions that reach the goal, and execute the actions with its "eyes closed," never having to use its percepts.

When the environment is partially observable, however, the agent doesn't know for sure what state it is in; and when the environment is nondeterministic, the agent doesn't know what state it transitions to after taking an action. That means that rather than thinking "I'm in state s_1 and if I do action a I'll end up in state s_2," an agent will now be thinking "I'm either in state s_1 or s_3, and if I do action a I'll end up in state s_2, s_4 or s_5." We call a set of physical states that the agent believes are possible a **belief state**.

Belief state

Conditional plan

In partially observable and nondeterministic environments, the solution to a problem is no longer a sequence, but rather a **conditional plan** (sometimes called a contingency plan or a strategy) that specifies what to do depending on what percepts agent receives while executing the plan. We examine nondeterminism in this section and partial observability in the next.

4.3.1 The erratic vacuum world

The vacuum world from Chapter 2 has eight states, as shown in Figure 4.9. There are three actions—*Right*, *Left*, and *Suck*—and the goal is to clean up all the dirt (states 7 and 8). If the environment is fully observable, deterministic, and completely known, then the problem is easy to solve with any of the algorithms in Chapter 3, and the solution is an action sequence. For example, if the initial state is 1, then the action sequence [*Suck*, *Right*, *Suck*] will reach a goal state, 8.

Now suppose that we introduce nondeterminism in the form of a powerful but erratic vacuum cleaner. In the **erratic vacuum world**, the *Suck* action works as follows:

- When applied to a dirty square the action cleans the square and sometimes cleans up dirt in an adjacent square, too.
- When applied to a clean square the action sometimes deposits dirt on the carpet.[5]

To provide a precise formulation of this problem, we need to generalize the notion of a **transition model** from Chapter 3. Instead of defining the transition model by a RESULT function

[4] A set of points $\mathcal{S}$ is convex if the line joining any two points in $\mathcal{S}$ is also contained in $\mathcal{S}$. A **convex function** is one for which the space "above" it forms a convex set; by definition, convex functions have no local (as opposed to global) minima.

[5] We assume that most readers face similar problems and can sympathize with our agent. We apologize to owners of modern, efficient cleaning appliances who cannot take advantage of this pedagogical device.

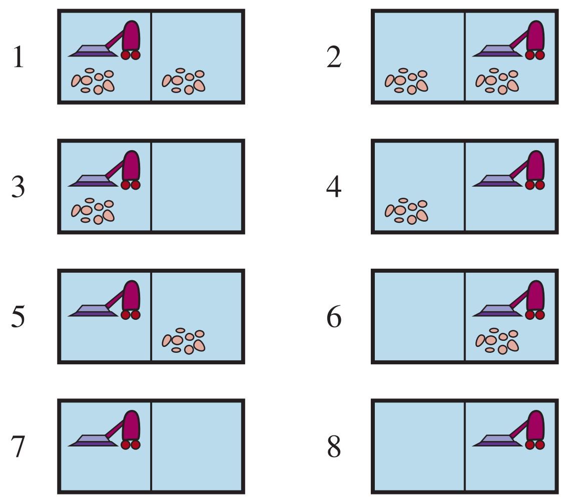


Figure 4.9 The eight possible states of the vacuum world; states 7 and 8 are goal states.

that returns a single outcome state, we use a RESULTS function that returns a set of possible outcome states. For example, in the erratic vacuum world, the *Suck* action in state 1 cleans up either just the current location, or both locations:

$$\text{RESULTS}(1, Suck) = \{5, 7\}$$

If we start in state 1, no single *sequence* of actions solves the problem, but the following **conditional plan** does:

$$[Suck, \textbf{if } State = 5 \textbf{ then } [Right, Suck] \textbf{ else } []] . \qquad (4.3)$$

Here we see that a conditional plan can contain **if–then–else** steps; this means that solutions are *trees* rather than sequences. Here the conditional in the **if** statement tests to see what the current state is; this is something the agent will be able to observe at runtime, but doesn't know at planning time. Alternatively, we could have had a formulation that tests the percept rather than the state. Many problems in the real, physical world are contingency problems, because exact prediction of the future is impossible. For this reason, many people keep their eyes open while walking around.

4.3.2 AND–OR search trees

How do we find these contingent solutions to nondeterministic problems? As in Chapter 3, we begin by constructing search trees, but here the trees have a different character. In a deterministic environment, the only branching is introduced by the agent's own choices in each state: I can do this action or that action. We call these nodes **OR nodes**. In the vacuum world, Or node
for example, at an OR node the agent chooses *Left or Right or Suck*. In a nondeterministic environment, branching is also introduced by the *environment's* choice of outcome for each action. We call these nodes **AND nodes**. For example, the *Suck* action in state 1 results in the And node
belief state $\{5, 7\}$, so the agent would need to find a plan for state 5 *and* for state 7. These two kinds of nodes alternate, leading to an **AND–OR tree** as illustrated in Figure 4.10. And–or tree

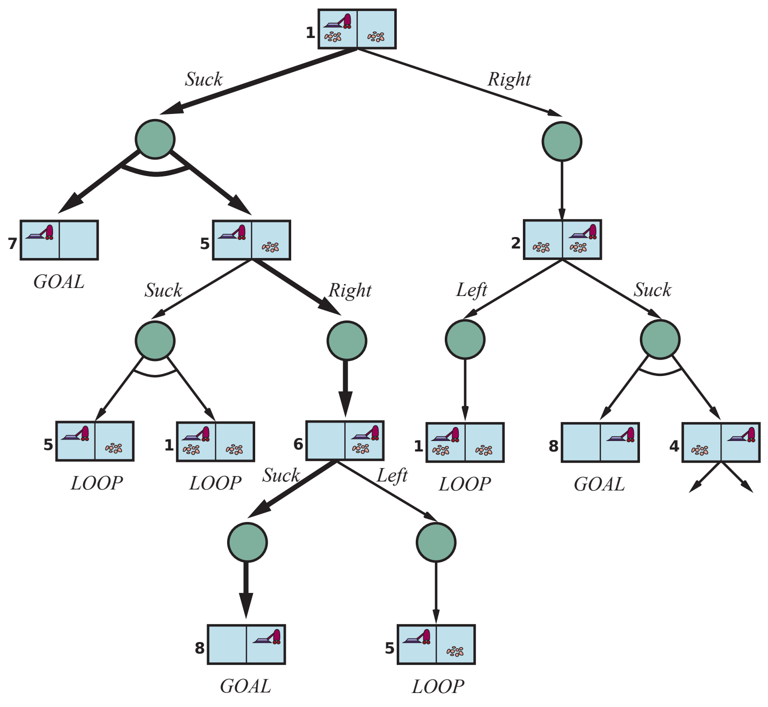


Figure 4.10 The first two levels of the search tree for the erratic vacuum world. State nodes are OR nodes where some action must be chosen. At the AND nodes, shown as circles, every outcome must be handled, as indicated by the arc linking the outgoing branches. The solution found is shown in bold lines.

A solution for an AND–OR search problem is a subtree of the complete search tree that (1) has a goal node at every leaf, (2) specifies one action at each of its OR nodes, and (3) includes every outcome branch at each of its AND nodes. The solution is shown in bold lines in the figure; it corresponds to the plan given in Equation (4.3).

Figure 4.11 gives a recursive, depth-first algorithm for AND–OR graph search. One key aspect of the algorithm is the way in which it deals with cycles, which often arise in nondeterministic problems (e.g., if an action sometimes has no effect or if an unintended effect can be corrected). If the current state is identical to a state on the path from the root, then it returns with failure. This doesn't mean that there is *no* solution from the current state; it simply means that if there *is* a noncyclic solution, it must be reachable from the earlier incarnation of the current state, so the new incarnation can be discarded. With this check, we ensure that the algorithm terminates in every finite state space, because every path must reach a goal, a dead end, or a repeated state. Notice that the algorithm does not check whether the current state is a repetition of a state on some *other* path from the root, which is important for efficiency.

AND–OR graphs can be explored either breadth-first or best-first. The concept of a heuristic function must be modified to estimate the cost of a contingent solution rather than a sequence, but the notion of admissibility carries over and there is an analog of the A* algorithm for finding optimal solutions. (See the bibliographical notes at the end of the chapter.)

```
function AND-OR-SEARCH(problem) returns a conditional plan, or failure
  return OR-SEARCH(problem, problem.INITIAL, [])

function OR-SEARCH(problem, state, path) returns a conditional plan, or failure
  if problem.IS-GOAL(state) then return the empty plan
  if IS-CYCLE(path) then return failure
  for each action in problem.ACTIONS(state) do
     plan ← AND-SEARCH(problem, RESULTS(state, action), [state] + path])
     if plan $\neq$ failure then return [action] + plan]
  return failure

function AND-SEARCH(problem, states, path) returns a conditional plan, or failure
  for each $s_i$ in states do
     $plan_i$ ← OR-SEARCH(problem, $s_i$, path)
     if $plan_i$ = failure then return failure
  return [if $s_1$ then $plan_1$ else if $s_2$ then $plan_2$ else ...if $s_{n-1}$ then $plan_{n-1}$ else $plan_n$]
```

Figure 4.11 An algorithm for searching AND–OR graphs generated by nondeterministic environments. A solution is a conditional plan that considers every nondeterministic outcome and makes a plan for each one.

4.3.3 Try, try again

Consider a *slippery* vacuum world, which is identical to the ordinary (non-erratic) vacuum world except that movement actions sometimes fail, leaving the agent in the same location. For example, moving *Right* in state 1 leads to the belief state $\{1,2\}$. Figure 4.12 shows part of the search graph; clearly, there are no longer any acyclic solutions from state 1, and AND-OR-SEARCH would return with failure. There is, however, a **cyclic solution**, which is to keep trying *Right* until it works. We can express this with a new **while** construct:

Cyclic solution

$$[Suck, \textbf{while } State = 5 \textbf{ do } Right, Suck]$$

or by adding a **label** to denote some portion of the plan and referring to that label later:

$$[Suck, L_1 : Right, \textbf{if } State = 5 \textbf{ then } L_1 \textbf{ else } Suck] .$$

When is a cyclic plan a solution? A minimum condition is that every leaf is a goal state and that a leaf is reachable from every point in the plan. In addition to that, we need to consider the cause of the nondeterminism. If it is really the case that the vacuum robot's drive mechanism works some of the time, but randomly and independently slips on other occasions, then the agent can be confident that if the action is repeated enough times, eventually it will work and the plan will succeed. But if the nondeterminism is due to some unobserved fact about the robot or environment—perhaps a drive belt has snapped and the robot will never move—then repeating the action will not help.

One way to understand this decision is to say that the initial problem formulation (fully observable, nondeterministic) is abandoned in favor of a different formulation (partially observable, deterministic) where the failure of the cyclic plan is attributed to an unobserved property of the drive belt. In Chapter 12 we discuss how to decide which of several uncertain possibilities is more likely.

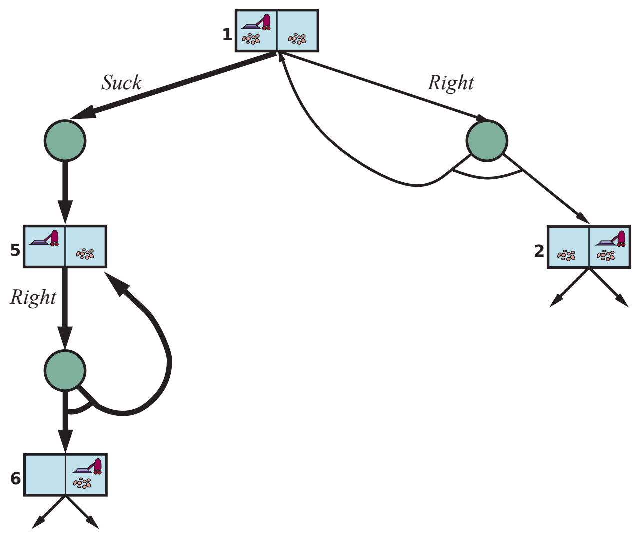


Figure 4.12 Part of the search graph for a slippery vacuum world, where we have shown (some) cycles explicitly. All solutions for this problem are cyclic plans because there is no way to move reliably.

4.4 Search in Partially Observable Environments

We now turn to the problem of partial observability, where the agent's percepts are not enough to pin down the exact state. That means that some of the agent's actions will be aimed at reducing uncertainty about the current state.

4.4.1 Searching with no observation

Sensorless
Conformant

When the agent's percepts provide *no information at all*, we have what is called a **sensorless** problem (or a **conformant** problem). At first, you might think the sensorless agent has no hope of solving a problem if it has no idea what state it starts in, but sensorless solutions are surprisingly common and useful, primarily because they *don't* rely on sensors working properly. In manufacturing systems, for example, many ingenious methods have been developed for orienting parts correctly from an unknown initial position by using a sequence of actions with no sensing at all. Sometimes a sensorless plan is better even when a conditional plan with sensing is available. For example, doctors often prescribe a broad-spectrum antibiotic rather than using the conditional plan of doing a blood test, then waiting for the results to come back, and then prescribing a more specific antibiotic. The sensorless plan saves time and money, and avoids the risk of the infection worsening before the test results are available.

Consider a sensorless version of the (deterministic) vacuum world. Assume that the agent knows the geography of its world, but not its own location or the distribution of dirt. In that case, its initial belief state is $\{1,2,3,4,5,6,7,8\}$ (see Figure 4.9). Now, if the agent moves *Right* it will be in one of the states $\{2,4,6,8\}$—the agent has gained information without perceiving anything! After [*Right*,*Suck*] the agent will always end up in one of the states $\{4,8\}$. Finally, after [*Right*,*Suck*,*Left*,*Suck*] the agent is guaranteed to reach the goal state 7, no matter what the start state. We say that the agent can **coerce** the world into state 7.

Coercion

The solution to a sensorless problem is a sequence of actions, not a conditional plan (because there is no perceiving). But we search in the space of belief states rather than physical states.[6] In belief-state space, the problem is *fully observable* because the agent always knows its own belief state. Furthermore, the solution (if any) for a sensorless problem is always a sequence of actions. This is because, as in the ordinary problems of Chapter 3, the percepts received after each action are completely predictable—they're always empty! So there are no contingencies to plan for. This is true *even if the environment is nondeterministic*.

We could introduce new algorithms for sensorless search problems. But instead, we can use the existing algorithms from Chapter 3 if we transform the underlying physical problem into a belief-state problem, in which we search over belief states rather than physical states. The original problem, P, has components $Actions_P, Result_P$ etc., and the belief-state problem has the following components:

- **States**: The belief-state space contains every possible subset of the physical states. If P has N states, then the belief-state problem has 2^N belief states, although many of those may be unreachable from the initial state.
- **Initial state**: Typically the belief state consisting of all states in P, although in some cases the agent will have more knowledge than this.
- **Actions**: This is slightly tricky. Suppose the agent is in belief state $b = \{s_1, s_2\}$, but $\textsc{Actions}_P(s_1) \neq \textsc{Actions}_P(s_2)$; then the agent is unsure of which actions are legal. If we assume that illegal actions have no effect on the environment, then it is safe to take the *union* of all the actions in any of the physical states in the current belief state b:
$$\textsc{Actions}(b) = \bigcup_{s \in b} \textsc{Actions}_P(s)\,.$$
On the other hand, if an illegal action might lead to catastrophe, it is safer to allow only the *intersection*, that is, the set of actions legal in *all* the states. For the vacuum world, every state has the same legal actions, so both methods give the same result.
- **Transition model**: For deterministic actions, the new belief state has one result state for each of the current possible states (although some result states may be the same):
$$b' = \textsc{Result}(b,a) = \{s' : s' = \textsc{Result}_P(s,a) \text{ and } s \in b\}\,. \tag{4.4}$$
With nondeterminism, the new belief state consists of all the possible results of applying the action to any of the states in the current belief state:
$$\begin{aligned} b' = \textsc{Result}(b,a) &= \{s' : s' \in \textsc{Results}_P(s,a) \text{ and } s \in b\} \\ &= \bigcup_{s \in b} \textsc{Results}_P(s,a)\,, \end{aligned}$$
The size of b' will be the same or smaller than b for deterministic actions, but may be larger than b with nondeterministic actions (see Figure 4.13).
- **Goal test**: The agent *possibly* achieves the goal if *any* state s in the belief state satisfies the goal test of the underlying problem, $\textsc{Is-Goal}_P(s)$. The agent *necessarily* achieves the goal if *every* state satisfies $\textsc{Is-Goal}_P(s)$. We aim to necessarily achieve the goal.
- **Action cost**: This is also tricky. If the same action can have different costs in different states, then the cost of taking an action in a given belief state could be one of

[6] In a fully observable environment, each belief state contains one physical state. Thus, we can view the algorithms in Chapter 3 as searching in a belief-state space of singleton belief states.

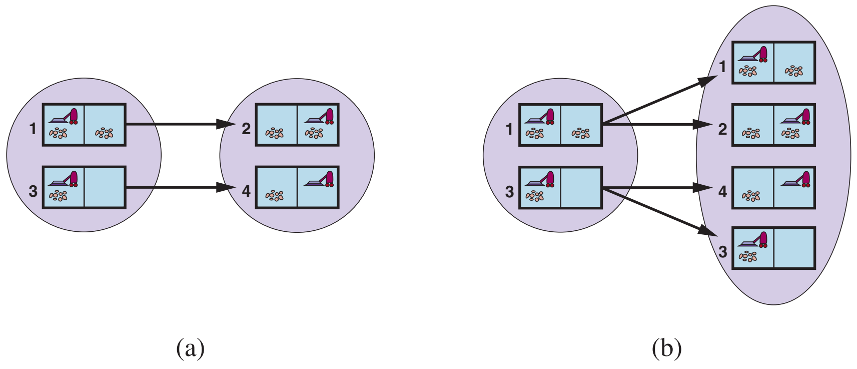


Figure 4.13 (a) Predicting the next belief state for the sensorless vacuum world with the deterministic action, *Right*. (b) Prediction for the same belief state and action in the slippery version of the sensorless vacuum world.

several values. (This gives rise to a new class of problems, which we explore in Exercise 4.MVAL.) For now we assume that the cost of an action is the same in all states and so can be transferred directly from the underlying physical problem.

Figure 4.14 shows the reachable belief-state space for the deterministic, sensorless vacuum world. There are only 12 reachable belief states out of $2^8 = 256$ possible belief states.

The preceding definitions enable the automatic construction of the belief-state problem formulation from the definition of the underlying physical problem. Once this is done, we can solve sensorless problems with any of the ordinary search algorithms of Chapter 3.

In ordinary graph search, newly reached states are tested to see if they were previously reached. This works for belief states, too; for example, in Figure 4.14, the action sequence [*Suck*,*Left*,*Suck*] starting at the initial state reaches the same belief state as [*Right*,*Left*,*Suck*], namely, $\{5,7\}$. Now, consider the belief state reached by [*Left*], namely, $\{1,3,5,7\}$. Obviously, this is not identical to $\{5,7\}$, but it is a *superset*. We can discard (prune) any such superset belief state. Why? Because a solution from $\{1,3,5,7\}$ must be a solution for each of the individual states 1, 3, 5, and 7, and thus it is a solution for any combination of these individual states, such as $\{5,7\}$; therefore we don't need to try to solve $\{1,3,5,7\}$, we can concentrate on trying to solve the strictly easier belief state $\{5,7\}$.

Conversely, if $\{1,3,5,7\}$ has already been generated and found to be solvable, then any *subset*, such as $\{5,7\}$, is guaranteed to be solvable. (If I have a solution that works when I'm very confused about what state I'm in, it will still work when I'm less confused.) This extra level of pruning may dramatically improve the efficiency of sensorless problem solving.

Even with this improvement, however, sensorless problem-solving as we have described it is seldom feasible in practice. One issue is the vastness of the belief-state space—we saw in the previous chapter that often a search space of size N is too large, and now we have search spaces of size 2^N. Furthermore, each element of the search space is a set of up to N elements. For large N, we won't be able to represent even a single belief state without running out of memory space.

One solution is to represent the belief state by some more compact description. In English, we could say the agent knows "Nothing" in the initial state; after moving *Left*, we could

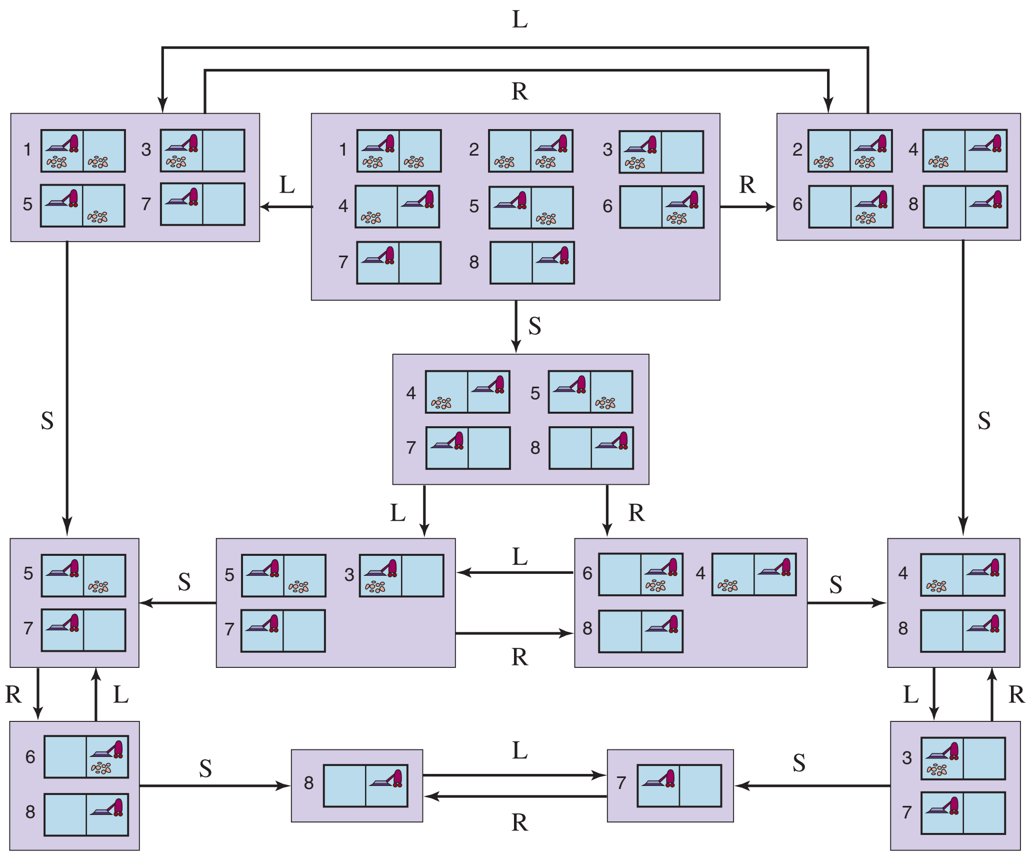


Figure 4.14 The reachable portion of the belief-state space for the deterministic, sensorless vacuum world. Each rectangular box corresponds to a single belief state. At any given point, the agent has a belief state but does not know which physical state it is in. The initial belief state (complete ignorance) is the top center box.

say, "Not in the rightmost column," and so on. Chapter 7 explains how to do this in a formal representation scheme.

Another approach is to avoid the standard search algorithms, which treat belief states as black boxes just like any other problem state. Instead, we can look *inside* the belief states and develop **incremental belief-state search** algorithms that build up the solution one physical state at a time. For example, in the sensorless vacuum world, the initial belief state is $\{1,2,3,4,5,6,7,8\}$, and we have to find an action sequence that works in all 8 states. We can do this by first finding a solution that works for state 1; then we check if it works for state 2; if not, go back and find a different solution for state 1, and so on.

Incremental belief-state search

Just as an AND–OR search has to find a solution for every branch at an AND node, this algorithm has to find a solution for every state in the belief state; the difference is that AND–OR search can find a different solution for each branch, whereas an incremental belief-state search has to find *one* solution that works for *all* the states.

The main advantage of the incremental approach is that it is typically able to detect failure quickly—when a belief state is unsolvable, it is usually the case that a small subset of the belief state, consisting of the first few states examined, is also unsolvable. In some cases, this leads to a speedup proportional to the size of the belief states, which may themselves be as large as the physical state space itself.

4.4.2 Searching in partially observable environments

Many problems cannot be solved without sensing. For example, the sensorless 8-puzzle is impossible. On the other hand, a little bit of sensing can go a long way: we can solve 8-puzzles if we can see just the upper-left corner square. The solution involves moving each tile in turn into the observable square and keeping track of its location from then on .

For a partially observable problem, the problem specification will specify a $\text{PERCEPT}(s)$ function that returns the percept received by the agent in a given state. If sensing is nondeterministic, then we can use a PERCEPTS function that returns a set of possible percepts. For fully observable problems, $\text{PERCEPT}(s) = s$ for every state s, and for sensorless problems $\text{PERCEPT}(s) = null$.

Consider a local-sensing vacuum world, in which the agent has a position sensor that yields the percept L in the left square, and R in the right square, and a dirt sensor that yields *Dirty* when the current square is dirty and *Clean* when it is clean. Thus, the PERCEPT in state 1 is $[L, Dirty]$. With partial observability, it will usually be the case that several states produce the same percept; state 3 will also produce $[L, Dirty]$. Hence, given this initial percept, the initial belief state will be $\{1,3\}$. We can think of the transition model between belief states for partially observable problems as occurring in three stages, as shown in Figure 4.15:

- The **prediction** stage computes the belief state resulting from the action, $\text{RESULT}(b,a)$, exactly as we did with sensorless problems. To emphasize that this is a prediction, we use the notation $\hat{b} = \text{RESULT}(b,a)$, where the "hat" over the b means "estimated," and we also use $\text{PREDICT}(b,a)$ as a synonym for $\text{RESULT}(b,a)$.
- The **possible percepts** stage computes the set of percepts that could be observed in the predicted belief state (using the letter o for observation):

$$\text{POSSIBLE-PERCEPTS}(\hat{b}) = \{o : o = \text{PERCEPT}(s) \text{ and } s \in \hat{b}\}\,.$$

- The **update** stage computes, for each possible percept, the belief state that would result from the percept. The updated belief state b_o is the set of states in $\hat{b}$ that could have produced the percept:

$$b_o = \text{UPDATE}(\hat{b}, o) = \{s : o = \text{PERCEPT}(s) \text{ and } s \in \hat{b}\}\,.$$

  The agent needs to deal with *possible* percepts at planning time, because it won't know the *actual* percepts until it executes the plan. Notice that nondeterminism in the physical environment can enlarge the belief state in the prediction stage, but each updated belief state b_o can be no larger than the predicted belief state $\hat{b}$; observations can only help reduce uncertainty. Moreover, for deterministic sensing, the belief states for the different possible percepts will be disjoint, forming a *partition* of the original predicted belief state.

Putting these three stages together, we obtain the possible belief states resulting from a given action and the subsequent possible percepts:

$$\begin{aligned}\text{RESULTS}(b,a) = \{b_o : b_o &= \text{UPDATE}(\text{PREDICT}(b,a), o) \text{ and} \\ o &\in \text{POSSIBLE-PERCEPTS}(\text{PREDICT}(b,a))\}\,.\end{aligned} \tag{4.5}$$

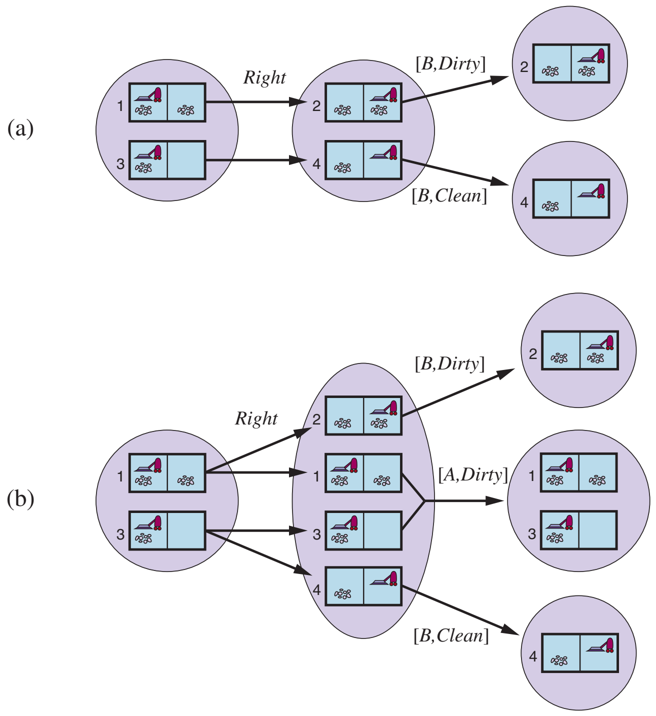


Figure 4.15 Two examples of transitions in local-sensing vacuum worlds. (a) In the deterministic world, *Right* is applied in the initial belief state, resulting in a new predicted belief state with two possible physical states; for those states, the possible percepts are $[R, Dirty]$ and $[R, Clean]$, leading to two belief states, each of which is a singleton. (b) In the slippery world, *Right* is applied in the initial belief state, giving a new belief state with four physical states; for those states, the possible percepts are $[L, Dirty]$, $[R, Dirty]$, and $[R, Clean]$, leading to three belief states as shown.

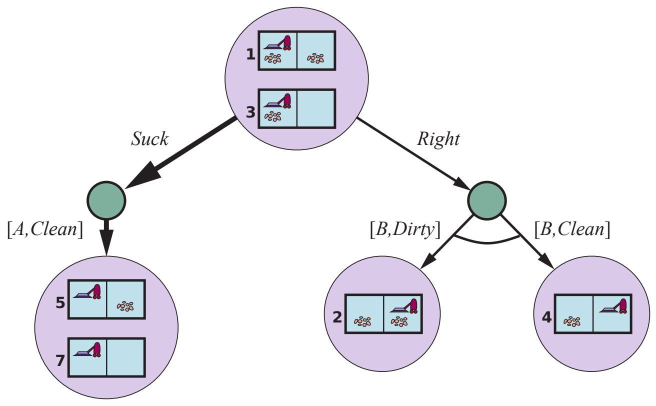


Figure 4.16 The first level of the AND–OR search tree for a problem in the local-sensing vacuum world; *Suck* is the first action in the solution.

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 $[A, Dirty]$. The solution is the conditional plan

$$[Suck, Right, \textbf{if } Bstate = \{6\} \textbf{ then } Suck \textbf{ else } []] .$$

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:

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

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.

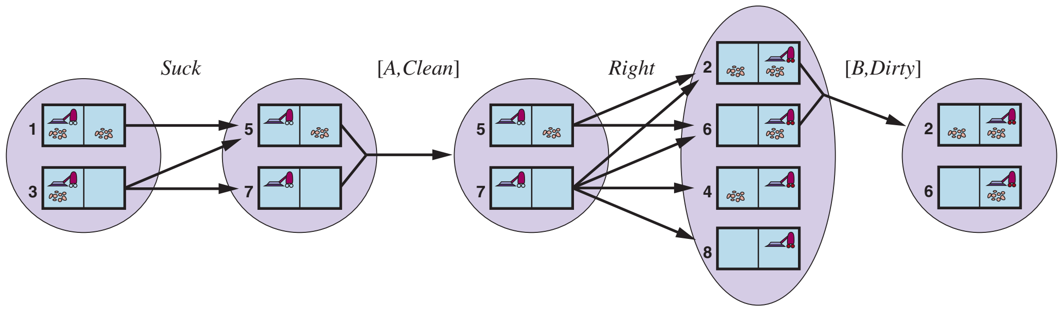


Figure 4.17 Two prediction–update cycles of belief-state maintenance in the kindergarten vacuum world with local sensing.

stochastic, continuous-state environments with the tools of probability theory, as explained in Chapter 14.

In this section we will show an example in a discrete environment with deterministic sensors and nondeterministic actions. The example concerns a robot with a particular state estimation task called **localization**: working out where it is, given a map of the world and a sequence of percepts and actions. Our robot is placed in the maze-like environment of Figure 4.18. The robot is equipped with four sonar sensors that tell whether there is an obstacle—the outer wall or a dark shaded square in the figure—in each of the four compass directions. The percept is in the form of a bit vector, one bit for each of the directions north, east, south, and west in that order, so 1011 means there are obstacles to the north, south, and west, but not east.

Localization

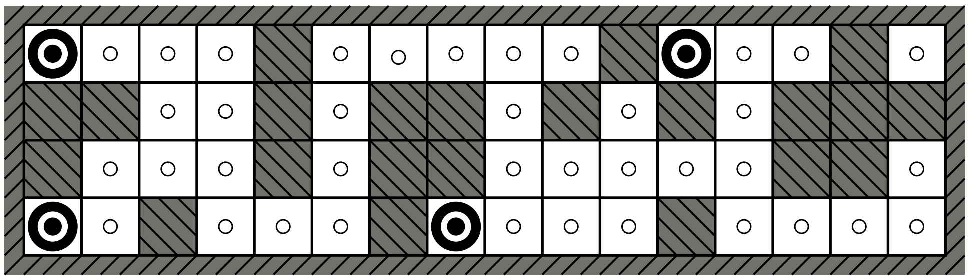
We assume that the sensors give perfectly correct data, and that the robot has a correct map of the environment. But unfortunately, the robot's navigational system is broken, so when it executes a *Right* action, it moves randomly to one of the adjacent squares. The robot's task is to determine its current location.

Suppose the robot has just been switched on, and it does not know where it is—its initial belief state b consists of the set of all locations. The robot then receives the percept 1011 and does an update using the equation $b_o = \text{UPDATE}(1011)$, yielding the 4 locations shown in Figure 4.18(a). You can inspect the maze to see that those are the only four locations that yield the percept 1011.

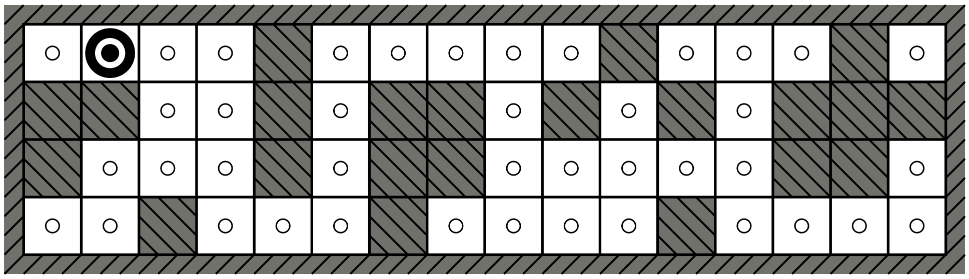
Next the robot executes a *Right* action, but the result is nondeterministic. The new belief state, $b_a = \text{PREDICT}(b_o, Right)$, contains all the locations that are one step away from the locations in b_o. When the second percept, 1010, arrives, the robot does $\text{UPDATE}(b_a, 1010)$ and finds that the belief state has collapsed down to the single location shown in Figure 4.18(b). That's the only location that could be the result of

$$\text{UPDATE}(\text{PREDICT}(\text{UPDATE}(b, 1011), Right), 1010).$$

With nondeterministic actions the PREDICT step grows the belief state, but the UPDATE step shrinks it back down—as long as the percepts provide some useful identifying information. Sometimes the percepts don't help much for localization: If there were one or more long east-west corridors, then a robot could receive a long sequence of 1010 percepts, but never know



(a) Possible locations of robot after $E_1 = 1011$



(b) Possible locations of robot after $E_1 = 1011, E_2 = 1010$

Figure 4.18 Possible positions of the robot, $\odot$, (a) after one observation, $E_1 = 1011$, and (b) after moving one square and making a second observation, $E_2 = 1010$. When sensors are noiseless and the transition model is accurate, there is only one possible location for the robot consistent with this sequence of two observations.

where in the corridor(s) it was. But for environments with reasonable variation in geography, localization often converges quickly to a single point, even when actions are nondeterministic.

What happens if the sensors are faulty? If we can reason only with Boolean logic, then we have to treat every sensor bit as being either correct or incorrect, which is the same as having no perceptual information at all. But we will see that probabilistic reasoning (Chapter 12), allows us to extract useful information from a faulty sensor as long as it is wrong less than half the time.

4.5 Online Search Agents and Unknown Environments

Offline search

Online search

So far we have concentrated on agents that use **offline search** algorithms. They compute a complete solution before taking their first action. In contrast, an **online search**[8] agent interleaves computation and action: first it takes an action, then it observes the environment and computes the next action. Online search is a good idea in dynamic or semi-dynamic environments, where there is a penalty for sitting around and computing too long. Online

[8] The term “online” here refers to algorithms that must process input as it is received rather than waiting for the entire input data set to become available. This usage of “online” is unrelated to the concept of “having an Internet connection.”

search is also helpful in nondeterministic domains because it allows the agent to focus its computational efforts on the contingencies that actually arise rather than those that *might* happen but probably won't.

Of course, there is a tradeoff: the more an agent plans ahead, the less often it will find itself up the creek without a paddle. In unknown environments, where the agent does not know what states exist or what its actions do, the agent must use its actions as experiments in order to learn about the environment.

A canonical example of online search is the **mapping problem**: a robot is placed in an unknown building and must explore to build a map that can later be used for getting from A to B. Methods for escaping from labyrinths—required knowledge for aspiring heroes of antiquity—are also examples of online search algorithms. Spatial exploration is not the only form of online exploration, however. Consider a newborn baby: it has many possible actions but knows the outcomes of none of them, and it has experienced only a few of the possible states that it can reach.

Mapping problem

4.5.1 Online search problems

An online search problem is solved by interleaving computation, sensing, and acting. We'll start by assuming a deterministic and fully observable environment (Chapter 17 relaxes these assumptions) and stipulate that the agent knows only the following:

- $\text{ACTIONS}(s)$, the legal actions in state s;
- $c(s,a,s')$, the cost of applying action a in state s to arrive at state s'. Note that this cannot be used until the agent knows that s' is the outcome.
- $\text{IS-GOAL}(s)$, the goal test.

Note in particular that the agent *cannot* determine $\text{RESULT}(s,a)$ except by actually being in s and doing a. For example, in the maze problem shown in Figure 4.19, the agent does not know that going *Up* from (1,1) leads to (1,2); nor, having done that, does it know that going *Down* will take it back to (1,1). This degree of ignorance can be reduced in some applications—for example, a robot explorer might know how its movement actions work and be ignorant only of the locations of obstacles.

Finally, the agent might have access to an admissible heuristic function $h(s)$ that estimates the distance from the current state to a goal state. For example, in Figure 4.19, the agent might know the location of the goal and be able to use the Manhattan-distance heuristic (page 97).

Typically, the agent's objective is to reach a goal state while minimizing cost. (Another possible objective is simply to explore the entire environment.) The cost is the total path cost that the agent incurs as it travels. It is common to compare this cost with the path cost the agent would incur *if it knew the search space in advance*—that is, the optimal path in the known environment. In the language of online algorithms, this comparison is called the **competitive ratio**; we would like it to be as small as possible.

Competitive ratio

Online explorers are vulnerable to **dead ends**: states from which no goal state is reachable. If the agent doesn't know what each action does, it might execute the "jump into bottomless pit" action, and thus never reach the goal. In general, *no algorithm can avoid dead ends in all state spaces.* Consider the two dead-end state spaces in Figure 4.20(a). An online search algorithm that has visited states S and A cannot tell if it is in the top state or the bottom one; the two look identical based on what the agent has seen. Therefore, there is no

Dead end

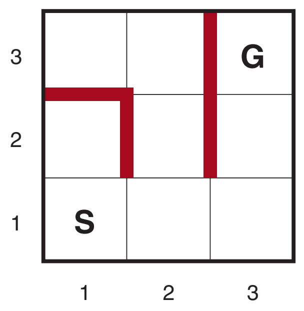


Figure 4.19 A simple maze problem. The agent starts at *S* and must reach *G* but knows nothing of the environment.

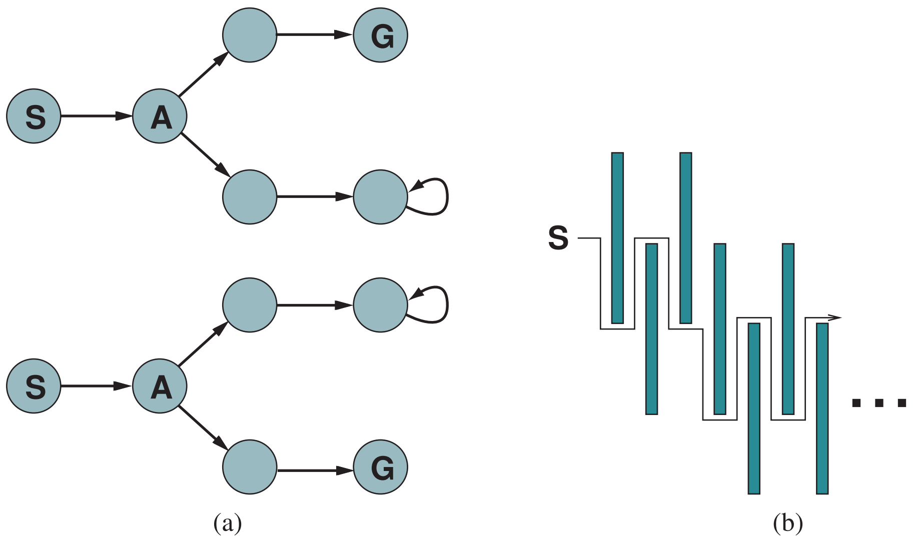


Figure 4.20 (a) Two state spaces that might lead an online search agent into a dead end. Any given agent will fail in at least one of these spaces. (b) A two-dimensional environment that can cause an online search agent to follow an arbitrarily inefficient route to the goal. Whichever choice the agent makes, the adversary blocks that route with another long, thin wall, so that the path followed is much longer than the best possible path.

way it could know how to choose the correct action in both state spaces. This is an example of an **adversary argument**—we can imagine an adversary constructing the state space while the agent explores it and putting the goals and dead ends wherever it chooses, as in Figure 4.20(b).

Adversary argument

Dead ends are a real difficulty for robot exploration—staircases, ramps, cliffs, one-way streets, and even natural terrain all present states from which some actions are **irreversible**—there is no way to return to the previous state. The exploration algorithm we will present is only guaranteed to work in state spaces that are **safely explorable**—that is, some goal state is reachable from every reachable state. State spaces with only reversible actions, such as

Irreversible action

Safely explorable

```
function ONLINE-DFS-AGENT(problem, s′) returns an action
               s, a, the previous state and action, initially null
  persistent: result, a table mapping (s, a) to s′, initially empty
               untried, a table mapping s to a list of untried actions
               unbacktracked, a table mapping s to a list of states never backtracked to

  if problem.IS-GOAL(s′) then return stop
  if s′ is a new state (not in untried) then untried[s′] ← problem.ACTIONS(s′)
  if s is not null then
      result[s, a] ← s′
      add s to the front of unbacktracked[s′]
  if untried[s′] is empty then
      if unbacktracked[s′] is empty then return stop
      else a ← an action b such that result[s′, b] = POP(unbacktracked[s′])
  else a ← POP(untried[s′])
  s ← s′
  return a
```

Figure 4.21 An online search agent that uses depth-first exploration. The agent can safely explore only in state spaces in which every action can be "undone" by some other action.

mazes and 8-puzzles, are clearly safely explorable (if they have any solution at all). We will cover the subject of safe exploration in more depth in Section 22.3.2.

Even in safely explorable environments, no bounded competitive ratio can be guaranteed if there are paths of unbounded cost. This is easy to show in environments with irreversible actions, but in fact it remains true for the reversible case as well, as Figure 4.20(b) shows. For this reason, it is common to characterize the performance of online search algorithms in terms of the size of the entire state space rather than just the depth of the shallowest goal.

4.5.2 Online search agents

After each action, an online agent in an observable environment receives a percept telling it what state it has reached; from this information, it can augment its map of the environment. The updated map is then used to plan where to go next. This interleaving of planning and action means that online search algorithms are quite different from the offline search algorithms we have seen previously: offline algorithms explore their *model* of the state space, while online algorithms explore the real world. For example, A^* can expand a node in one part of the space and then immediately expand a node in a distant part of the space, because node expansion involves simulated rather than real actions.

An online algorithm, on the other hand, can discover successors only for a state that it physically occupies. To avoid traveling all the way to a distant state to expand the next node, it seems better to expand nodes in a *local* order. Depth-first search has exactly this property because (except when the algorithm is backtracking) the next node expanded is a child of the previous node expanded.

An online depth-first exploration agent (for deterministic but unknown actions) is shown in Figure 4.21. This agent stores its map in a table, $result[s,a]$, that records the state resulting from executing action a in state s. (For nondeterministic actions, the agent could record a set

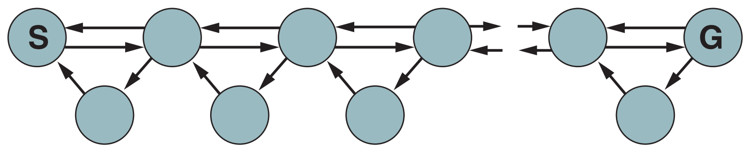


Figure 4.22 An environment in which a random walk will take exponentially many steps to find the goal.

of states under *results*$[s,a]$.) Whenever the current state has unexplored actions, the agent tries one of those actions. The difficulty comes when the agent has tried all the actions in a state. In offline depth-first search, the state is simply dropped from the queue; in an online search, the agent has to backtrack in the physical world. In depth-first search, this means going back to the state from which the agent most recently entered the current state. To achieve that, the algorithm keeps another table that lists, for each state, the predecessor states to which the agent has not yet backtracked. If the agent has run out of states to which it can backtrack, then its search is complete.

We recommend that the reader trace through the progress of ONLINE-DFS-AGENT when applied to the maze given in Figure 4.19. It is fairly easy to see that the agent will, in the worst case, end up traversing every link in the state space exactly twice. For exploration, this is optimal; for finding a goal, on the other hand, the agent's competitive ratio could be arbitrarily bad if it goes off on a long excursion when there is a goal right next to the initial state. An online variant of iterative deepening solves this problem; for an environment that is a uniform tree, the competitive ratio of such an agent is a small constant.

Because of its method of backtracking, ONLINE-DFS-AGENT works only in state spaces where the actions are reversible. There are slightly more complex algorithms that work in general state spaces, but no such algorithm has a bounded competitive ratio.

4.5.3 Online local search

Like depth-first search, **hill-climbing search** has the property of locality in its node expansions. In fact, because it keeps just one current state in memory, hill-climbing search is *already* an online search algorithm! Unfortunately, the basic algorithm is not very good for exploration because it leaves the agent sitting at local maxima with nowhere to go. Moreover, random restarts cannot be used, because the agent cannot teleport itself to a new start state.

Random walk

Instead of random restarts, one might consider using a **random walk** to explore the environment. A random walk simply selects at random one of the available actions from the current state; preference can be given to actions that have not yet been tried. It is easy to prove that a random walk will *eventually* find a goal or complete its exploration, provided that the space is finite and safely explorable.[9] On the other hand, the process can be very slow. Figure 4.22 shows an environment in which a random walk will take exponentially many steps to find the goal, because, for each state in the top row except S, backward progress is twice as likely as forward progress. The example is contrived, of course, but there are many real-world state spaces whose topology causes these kinds of "traps" for random walks.

[9] Random walks are complete on infinite one-dimensional and two-dimensional grids. On a three-dimensional grid, the probability that the walk ever returns to the starting point is only about 0.3405 (Hughes, 1995).

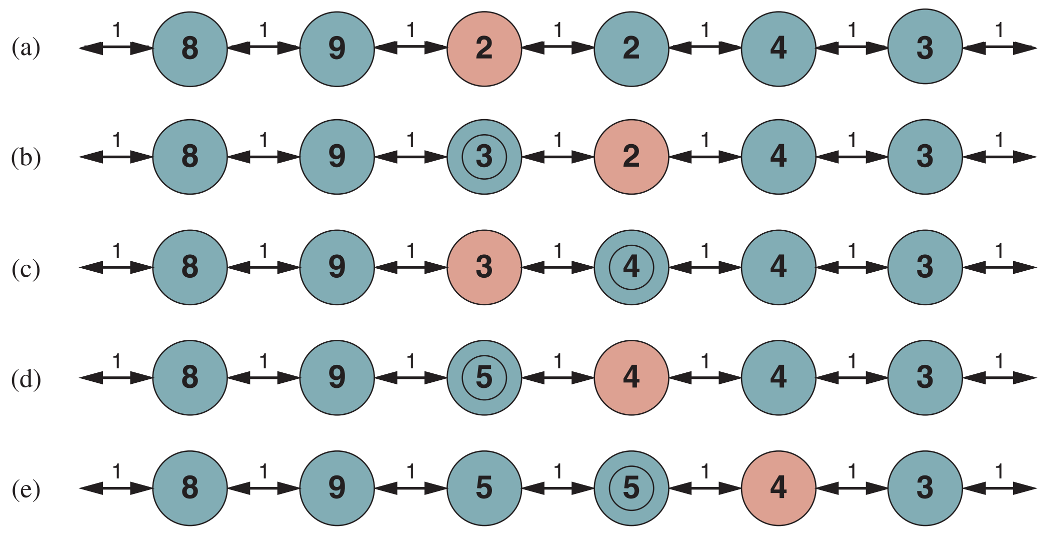


Figure 4.23 Five iterations of LRTA* on a one-dimensional state space. Each state is labeled with $H(s)$, the current cost estimate to reach a goal, and every link has an action cost of 1. The red state marks the location of the agent, and the updated cost estimates at each iteration have a double circle.

Augmenting hill climbing with *memory* rather than randomness turns out to be a more effective approach. The basic idea is to store a "current best estimate" $H(s)$ of the cost to reach the goal from each state that has been visited. $H(s)$ starts out being just the heuristic estimate $h(s)$ and is updated as the agent gains experience in the state space.

Figure 4.23 shows a simple example in a one-dimensional state space. In (a), the agent seems to be stuck in a flat local minimum at the red state. Rather than staying where it is, the agent should follow what seems to be the best path to the goal given the current cost estimates for its neighbors. The estimated cost to reach the goal through a neighbor s' is the cost to get to s' plus the estimated cost to get to a goal from there—that is, $c(s,a,s')+H(s')$. In the example, there are two actions, with estimated costs $1+9$ to the left and $1+2$ to the right, so it seems best to move right.

In (b) it is clear that the cost estimate of 2 for the red state in (a) was overly optimistic. Since the best move cost 1 and led to a state that is at least 2 steps from a goal, the red state must be at least 3 steps from a goal, so its H should be updated accordingly, as shown in Figure 4.23(b). Continuing this process, the agent will move back and forth twice more, updating H each time and "flattening out" the local minimum until it escapes to the right.

An agent implementing this scheme, which is called learning real-time A* (**LRTA***), is shown in Figure 4.24. Like ONLINE-DFS-AGENT, it builds a map of the environment in the *result* table. It updates the cost estimate for the state it has just left and then chooses the "apparently best" move according to its current cost estimates. One important detail is that actions that have not yet been tried in a state s are always assumed to lead immediately to the goal with the least possible cost, namely $h(s)$. This **optimism under uncertainty** encourages the agent to explore new, possibly promising paths.

LRTA*

Optimism under uncertainty

```
function LRTA*-AGENT(problem, s′, h) returns an action
              s, a, the previous state and action, initially null
  persistent: result, a table mapping (s, a) to s′, initially empty
              H, a table mapping s to a cost estimate, initially empty

  if IS-GOAL(s′) then return stop
  if s′ is a new state (not in H) then H[s′] ← h(s′)
  if s is not null then
      result[s, a] ← s′
      H[s] ← min_{b ∈ ACTIONS(s)} LRTA*-COST(s, b, result[s, b], H)
  a ← argmin_{b ∈ ACTIONS(s)} LRTA*-COST(problem, s′, b, result[s′, b], H)
  s ← s′
  return a

function LRTA*-COST(problem, s, a, s′, H) returns a cost estimate
  if s′ is undefined then return h(s)
  else return problem.ACTION-COST(s, a, s′) + H[s′]
```

Figure 4.24 LRTA*-AGENT selects an action according to the values of neighboring states, which are updated as the agent moves about the state space.

An LRTA* agent is guaranteed to find a goal in any finite, safely explorable environment. Unlike A*, however, it is not complete for infinite state spaces—there are cases where it can be led infinitely astray. It can explore an environment of n states in $O(n^2)$ steps in the worst case, but often does much better. The LRTA* agent is just one of a large family of online agents that one can define by specifying the action selection rule and the update rule in different ways. We discuss this family, developed originally for stochastic environments, in Chapter 22.

4.5.4 Learning in online search

The initial ignorance of online search agents provides several opportunities for learning. First, the agents learn a "map" of the environment—more precisely, the outcome of each action in each state—simply by recording each of their experiences. Second, the local search agents acquire more accurate estimates of the cost of each state by using local updating rules, as in LRTA*. In Chapter 22, we show that these updates eventually converge to *exact* values for every state, provided that the agent explores the state space in the right way. Once exact values are known, optimal decisions can be taken simply by moving to the lowest-cost successor—that is, pure hill climbing is then an optimal strategy.

If you followed our suggestion to trace the behavior of ONLINE-DFS-AGENT in the environment of Figure 4.19, you will have noticed that the agent is not very bright. For example, after it has seen that the *Up* action goes from (1,1) to (1,2), the agent still has no idea that the *Down* action goes back to (1,1) or that the *Up* action also goes from (2,1) to (2,2), from (2,2) to (2,3), and so on. In general, we would like the agent to learn that *Up* increases the y-coordinate unless there is a wall in the way, that *Down* reduces it, and so on.

For this to happen, we need two things. First, we need a formal and explicitly manipulable representation for these kinds of general rules; so far, we have hidden the information inside the black box called the RESULT function. Chapters 8 to 11 are devoted to this issue. Second, we need algorithms that can construct suitable general rules from the specific observations made by the agent. These are covered in Chapter 19.

If we anticipate that we will be called upon to solve multiple similar problems in the future then it makes sense to invest time (and memory) to make those future searches easier. There are several ways to do this, all falling under the heading of **incremental search**. We could keep the search tree in memory and reuse the parts of it that are unchanged in the new problem. We could keep the heuristic h values and update them as we gain new information—either because the world has changed or because we have computed a better estimate. Or we could keep the best-path g values, using them to piece together a new solution, and updating them when the world changes.

Incremental search

Summary

This chapter has examined search algorithms for problems in partially observable, nondeterministic, unknown, and continuous environments.

- *Local search* methods such as **hill climbing** keep only a small number of states in memory. They have been applied to optimization problems, where the idea is to find a high-scoring state, without worrying about the path to the state. Several stochastic local search algorithms have been developed, including **simulated annealing**, which returns optimal solutions when given an appropriate cooling schedule.
- Many local search methods apply also to problems in continuous spaces. **Linear programming** and **convex optimization** problems obey certain restrictions on the shape of the state space and the nature of the objective function, and admit polynomial-time algorithms that are often extremely efficient in practice. For some mathematically well-formed problems, we can find the maximum using calculus to find where the gradient is zero; for other problems we have to make do with the empirical gradient, which measures the difference in fitness between two nearby points.
- An **evolutionary algorithm** is a stochastic hill-climbing search in which a population of states is maintained. New states are generated by **mutation** and by **crossover**, which combines pairs of states.
- In **nondeterministic** environments, agents can apply AND–OR search to generate **contingent** plans that reach the goal regardless of which outcomes occur during execution.
- When the environment is partially observable, the **belief state** represents the set of possible states that the agent might be in.
- Standard search algorithms can be applied directly to belief-state space to solve **sensorless problems**, and belief-state AND–OR search can solve general partially observable problems. Incremental algorithms that construct solutions state by state within a belief state are often more efficient.
- **Exploration problems** arise when the agent has no idea about the states and actions of its environment. For safely explorable environments, **online search** agents can build a map and find a goal if one exists. Updating heuristic estimates from experience provides an effective method to escape from local minima.

Bibliographical and Historical Notes

Local search techniques have a long history in mathematics and computer science. Indeed, the Newton–Raphson method (Newton, 1671; Raphson, 1690) can be seen as a very efficient local search method for continuous spaces in which gradient information is available. Brent (1973) is a classic reference for optimization algorithms that do not require such information. Beam search, which we have presented as a local search algorithm, originated as a bounded-width variant of dynamic programming for speech recognition in the HARPY system (Lowerre, 1976). A related algorithm is analyzed in depth by Pearl (1984, Ch. 5).

The topic of local search was reinvigorated in the early 1990s by surprisingly good results for large constraint-satisfaction problems such as n-queens (Minton *et al.*, 1992) and Boolean satisfiability (Selman *et al.*, 1992) and by the incorporation of randomness, multiple simultaneous searches, and other improvements. This renaissance of what Christos Papadimitriou has called "New Age" algorithms also sparked increased interest among theoretical computer scientists (Koutsoupias and Papadimitriou, 1992; Aldous and Vazirani, 1994).

Tabu search

In the field of operations research, a variant of hill climbing called **tabu search** has gained popularity (Glover and Laguna, 1997). This algorithm maintains a tabu list of k previously visited states that cannot be revisited; as well as improving efficiency when searching graphs, this list can allow the algorithm to escape from some local minima.

Another useful improvement on hill climbing is the STAGE algorithm (Boyan and Moore, 1998). The idea is to use the local maxima found by random-restart hill climbing to get an idea of the overall shape of the landscape. The algorithm fits a smooth quadratic surface to the set of local maxima and then calculates the global maximum of that surface analytically. This becomes the new restart point. Gomes *et al.* (1998) showed that the run times of systematic backtracking algorithms often have a **heavy-tailed distribution**, which means that the probability of a very long run time is more than would be predicted if the run times were exponentially distributed. When the run time distribution is heavy-tailed, random restarts find a solution faster, on average, than a single run to completion. Hoos and Stützle (2004) provide a book-length coverage of the topic.

Heavy-tailed distribution

Simulated annealing was first described by Kirkpatrick *et al.* (1983), who borrowed directly from the **Metropolis algorithm** (which is used to simulate complex systems in physics (Metropolis *et al.*, 1953) and was supposedly invented at a Los Alamos dinner party). Simulated annealing is now a field in itself, with hundreds of papers published every year.

Finding optimal solutions in continuous spaces is the subject matter of several fields, including **optimization theory**, **optimal control theory**, and the **calculus of variations**. The basic techniques are explained well by Bishop (1995); Press *et al.* (2007) cover a wide range of algorithms and provide working software.

Researchers have taken inspiration for search and optimization algorithms from a wide variety of fields of study: metallurgy (simulated annealing); biology (genetic algorithms); neuroscience (neural networks); mountaineering (hill climbing); economics (market-based algorithms (Dias *et al.*, 2006)); physics (particle swarms (Li and Yao, 2012) and spin glasses (Mézard *et al.*, 1987)); animal behavior (reinforcement learning, grey wolf optimizers (Mirjalili and Lewis, 2014)); ornithology (Cuckoo search (Yang and Deb, 2014)); entomology (ant colony (Dorigo *et al.*, 2008), bee colony (Karaboga and Basturk, 2007), firefly (Yang, 2009) and glowworm (Krishnanand and Ghose, 2009) optimization); and others.

Linear programming (LP) was first studied systematically by the mathematician Leonid Kantorovich (1939). It was one of the first applications of computers; the **simplex algorithm** (Dantzig, 1949) is still used despite worst-case exponential complexity. Karmarkar (1984) developed the far more efficient family of **interior-point** methods, which was shown to have polynomial complexity for the more general class of convex optimization problems by Nesterov and Nemirovski (1994). Excellent introductions to convex optimization are provided by Ben-Tal and Nemirovski (2001) and Boyd and Vandenberghe (2004).

Work by Sewall Wright (1931) on the concept of a **fitness landscape** was an important precursor to the development of genetic algorithms. In the 1950s, several statisticians, including Box (1957) and Friedman (1959), used evolutionary techniques for optimization problems, but it wasn't until Rechenberg (1965) introduced **evolution strategies** to solve optimization problems for airfoils that the approach gained popularity. In the 1960s and 1970s, John Holland (1975) championed genetic algorithms, both as a useful optimization tool and as a method to expand our understanding of adaptation (Holland, 1995).

The **artificial life** movement (Langton, 1995) took this idea one step further, viewing the products of genetic algorithms as *organisms* rather than solutions to problems. The the Baldwin effect discussed in the chapter was proposed roughly simultaneously by Conwy Lloyd Morgan (1896) and James (Baldwin, 1896). Computer simulations have helped to clarify its implications (Hinton and Nowlan, 1987; Ackley and Littman, 1991; Morgan and Griffiths, 2015). Smith and Szathmáry (1999), Ridley (2004), and Carroll (2007) provide general background on evolution.

Most comparisons of genetic algorithms to other approaches (especially stochastic hill climbing) have found that the genetic algorithms are slower to converge (O'Reilly and Oppacher, 1994; Mitchell *et al.*, 1996; Juels and Wattenberg, 1996; Baluja, 1997). Such findings are not universally popular within the GA community, but recent attempts within that community to understand population-based search as an approximate form of Bayesian learning (see Chapter 20) might help close the gap between the field and its critics (Pelikan *et al.*, 1999). The theory of **quadratic dynamical systems** may also explain the performance of GAs (Rabani *et al.*, 1998). There are some impressive practical applications of GAs, in areas as diverse as antenna design (Lohn *et al.*, 2001), computer-aided design (Renner and Ekart, 2003), climate models (Stanislawska *et al.*, 2015), medicine (Ghaheri *et al.*, 2015), and designing deep neural networks (Miikkulainen *et al.*, 2019).

The field of **genetic programming** is a subfield of genetic algorithms in which the representations are programs rather than bit strings. The programs are represented in the form of syntax trees, either in a standard programming language or in specially designed formats to represent electronic circuits, robot controllers, and so on. Crossover involves splicing together subtrees in such a way that the offspring are guaranteed to be well-formed expressions.

Interest in genetic programming was spurred by the work of John Koza (1992, 1994), but it goes back at least to early experiments with machine code by Friedberg (1958) and with finite-state automata by Fogel *et al.* (1966). As with genetic algorithms, there is debate about the effectiveness of the technique. Koza *et al.* (1999) describe experiments in the use of genetic programming to design circuit devices.

The journals *Evolutionary Computation* and *IEEE Transactions on Evolutionary Computation* cover evolutionary algorithms; articles are also found in *Complex Systems*, *Adaptive Behavior*, and *Artificial Life*. The main conference is the *Genetic and Evolutionary Com-*

putation Conference (GECCO). Good overview texts on genetic algorithms include those by Mitchell (1996), Fogel (2000), Langdon and Poli (2002), and Poli *et al.* (2008).

The unpredictability and partial observability of real environments were recognized early on in robotics projects that used planning techniques, including Shakey (Fikes *et al.*, 1972) and FREDDY (Michie, 1972). The problems received more attention after the publication of McDermott's (1978a) influential article *Planning and Acting*.

The first work to make explicit use of AND–OR trees seems to have been Slagle's SAINT program for symbolic integration, mentioned in Chapter 1. Amarel (1967) applied the idea to propositional theorem proving, a topic discussed in Chapter 7, and introduced a search algorithm similar to AND-OR-GRAPH-SEARCH. The algorithm was further developed by Nilsson (1971), who also described AO*—which, as its name suggests, finds optimal solutions. AO* was further improved by Martelli and Montanari (1973).

AO* is a top-down algorithm; a bottom-up generalization of A* is A*LD, for A* Lightest Derivation (Felzenszwalb and McAllester, 2007). Interest in AND–OR search underwent a revival in the early 2000s, with new algorithms for finding cyclic solutions (Jimenez and Torras, 2000; Hansen and Zilberstein, 2001) and new techniques inspired by dynamic programming (Bonet and Geffner, 2005).

The idea of transforming partially observable problems into belief-state problems originated with Astrom (1965) for the much more complex case of probabilistic uncertainty (see Chapter 17). Erdmann and Mason (1988) studied the problem of robotic manipulation without sensors, using a continuous form of belief-state search. They showed that it was possible to orient a part on a table from an arbitrary initial position by a well-designed sequence of tilting actions. More practical methods, based on a series of precisely oriented diagonal barriers across a conveyor belt, use the same algorithmic insights (Wiegley *et al.*, 1996).

The belief-state approach was reinvented in the context of sensorless and partially observable search problems by Genesereth and Nourbakhsh (1993). Additional work was done on sensorless problems in the logic-based planning community (Goldman and Boddy, 1996; Smith and Weld, 1998). This work has emphasized concise representations for belief states, as explained in Chapter 11. Bonet and Geffner (2000) introduced the first effective heuristics for belief-state search; these were refined by Bryce *et al.* (2006). The incremental approach to belief-state search, in which solutions are constructed incrementally for subsets of states within each belief state, was studied in the planning literature by Kurien *et al.* (2002); several new incremental algorithms were introduced for nondeterministic, partially observable problems by Russell and Wolfe (2005). Additional references for planning in stochastic, partially observable environments appear in Chapter 17.

Algorithms for exploring unknown state spaces have been of interest for many centuries. Depth-first search in a reversible maze can be implemented by keeping one's left hand on the wall; loops can be avoided by marking each junction. The more general problem of exploring **Eulerian graphs** (i.e., graphs in which each node has equal numbers of incoming and outgoing edges) was solved by an algorithm due to Hierholzer (1873).

Eulerian graph

The first thorough algorithmic study of the exploration problem for arbitrary graphs was carried out by Deng and Papadimitriou (1990), who developed a completely general algorithm but showed that no bounded competitive ratio is possible for exploring a general graph. Papadimitriou and Yannakakis (1991) examined the question of finding paths to a goal in geometric path-planning environments (where all actions are reversible). They showed that a small competitive ratio is achievable with square obstacles, but with general rectangular obstacles no bounded ratio can be achieved. (See Figure 4.20.)

In a dynamic environment, the state of the world can spontaneously change without any action by the agent. For example, the agent can plan an optimal driving route from A to B, but an accident or unusually bad rush hour traffic can spoil the plan. Incremental search algorithms such as Lifelong Planning A* (Koenig *et al.*, 2004) and D* Lite (Koenig and Likhachev, 2002) deal with this situation.

The LRTA* algorithm was developed by Korf (1990) as part of an investigation into **real-time search** for environments in which the agent must act after searching for only a fixed amount of time (a common situation in two-player games). LRTA* is in fact a special case of reinforcement learning algorithms for stochastic environments (Barto *et al.*, 1995). Its policy of optimism under uncertainty—always head for the closest unvisited state—can result in an exploration pattern that is less efficient in the uninformed case than simple depth-first search (Koenig, 2000). Dasgupta *et al.* (1994) show that online iterative deepening search is optimally efficient for finding a goal in a uniform tree with no heuristic information.

Several informed variants on the LRTA* theme have been developed with different methods for searching and updating within the known portion of the graph (Pemberton and Korf, 1992). As yet, there is no good theoretical understanding of how to find goals with optimal efficiency when using heuristic information. Sturtevant and Bulitko (2016) provide an analysis of some pitfalls that occur in practice.

CHAPTER 5

ADVERSARIAL SEARCH AND GAMES

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

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.

Adversarial search

5.1 Game Theory

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.

Economy

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.

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.

Pruning

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 5.3), or we can average the outcomes of many fast simulations of the game from that state all the way to the end (Section 5.4).

Section 5.5 discusses games that include an element of chance (through rolling dice or shuffling cards) and Section 5.6 covers games of **imperfect information** (such as poker and bridge, where not all cards are visible to all players).

Imperfect information

5.1.1 Two-player zero-sum games

The games most commonly studied within AI (such as chess and Go) are what game theorists call deterministic, two-player, turn-taking, **perfect information**, **zero-sum games**. "Perfect information" is a synonym for "fully observable,"[1] and "zero-sum" means that what is good for one player is just as bad for the other: there is no "win-win" outcome. For games we often use the term **move** as a synonym for "action" and **position** as a synonym for "state."

Perfect information

Zero-sum games

Move

Position

We will call our two players MAX and MIN, for reasons that will soon become obvious. MAX moves first, and then the players take turns moving until the game is over. At the end of the game, points are awarded to the winning player and penalties are given to the loser. A game can be formally defined with the following elements:

- S_0: The **initial state**, which specifies how the game is set up at the start.
- TO-MOVE(s): The player whose turn it is to move in state s.
- ACTIONS(s): The set of legal moves in state s.
- RESULT(s, a): The **transition model**, which defines the state resulting from taking action a in state s.
- IS-TERMINAL(s): A **terminal test**, which is true when the game is over and false otherwise. States where the game has ended are called **terminal states**.
- UTILITY(s, p): A **utility function** (also called an objective function or payoff function), which defines the final numeric value to player p when the game ends in terminal state s. In chess, the outcome is a win, loss, or draw, with values 1, 0, or $1/2$.[2] Some games have a wider range of possible outcomes—for example, the payoffs in backgammon range from 0 to 192.

Transition model

Terminal test

Terminal state

Much as in Chapter 3, the initial state, ACTIONS function, and RESULT function define the **state space graph**—a graph where the vertices are states, the edges are moves and a state might be reached by multiple paths. As in Chapter 3, we can superimpose a **search tree** over part of that graph to determine what move to make. We define the complete **game tree** as a search tree that follows every sequence of moves all the way to a terminal state. The game tree may be infinite if the state space itself is unbounded or if the rules of the game allow for infinitely repeating positions.

State space graph

Search tree

Game tree

Figure 5.1 shows part of the game tree for tic-tac-toe (noughts and crosses). From the initial state, MAX has nine possible moves. Play alternates between MAX's placing an X and MIN's placing an O until we reach leaf nodes corresponding to terminal states such that one player has three squares in a row or all the squares are filled. The number on each leaf node indicates the utility value of the terminal state from the point of view of MAX; high values are good for MAX and bad for MIN (which is how the players get their names).

For tic-tac-toe the game tree is relatively small—fewer than $9! = 362,880$ terminal nodes (with only 5,478 distinct states). But for chess there are over 10^{40} nodes, so the game tree is best thought of as a theoretical construct that we cannot realize in the physical world.

[1] Some authors make a distinction, using "imperfect information game" for one like poker where the players get private information about their own hands that the other players do not have, and "partially observable game" to mean one like StarCraft II where each player can see the nearby environment, but not the environment far away.

[2] Chess is considered a "zero-sum" game, even though the sum of the outcomes for the two players is +1 for each game, not zero. "Constant-sum" would have been a more accurate term, but zero-sum is traditional and makes sense if you imagine each player is charged an entry fee of $1/2$.

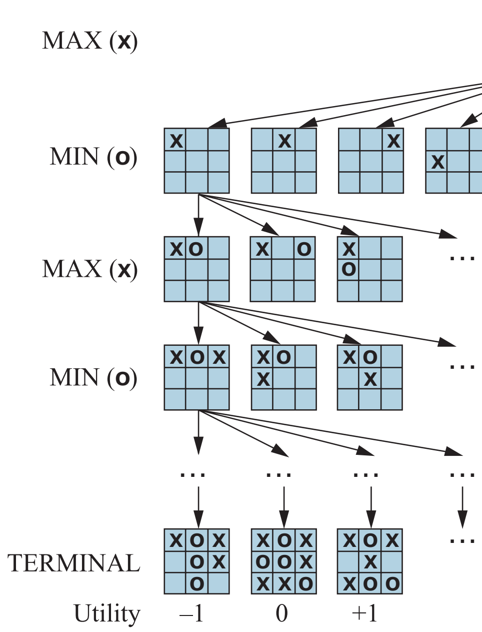


Figure 5.1 A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

5.2 Optimal Decisions in Games

MAX wants to find a sequence of actions leading to a win, but MIN has something to say about it. This means that MAX's strategy must be a conditional plan—a contingent strategy specifying a response to each of MIN's possible moves. In games that have a binary outcome (win or lose), we could use AND–OR search (page 125) to generate the conditional plan. In fact, for such games, the definition of a winning strategy for the game is identical to the definition of a solution for a nondeterministic planning problem: in both cases the desirable outcome must be guaranteed no matter what the "other side" does. For games with multiple outcome scores, we need a slightly more general algorithm called **minimax search**.

Minimax search

Consider the trivial game in Figure 5.2. The possible moves for MAX at the root node are labeled a_1, a_2, and a_3. The possible replies to a_1 for MIN are b_1, b_2, b_3, and so on. This particular game ends after one move each by MAX and MIN. (Note: In some games, the word "move" means that both players have taken an action; therefore the word **ply** is used to unambiguously mean one move by one player, bringing us one level deeper in the game tree.) The utilities of the terminal states in this game range from 2 to 14.

Ply

Given a game tree, the optimal strategy can be determined by working out the **minimax value** of each state in the tree, which we write as MINIMAX(s). The minimax value is the utility (for MAX) of being in that state, *assuming that both players play optimally* from there to the end of the game. The minimax value of a terminal state is just its utility. In a non-terminal state, MAX prefers to move to a state of maximum value when it is MAX's turn to

Minimax value

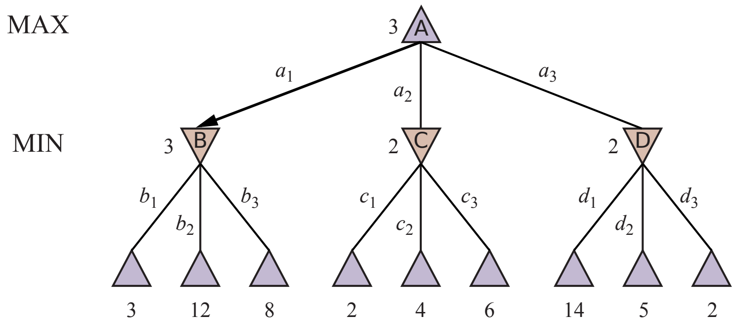


Figure 5.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

move, and MIN prefers a state of minimum value (that is, minimum value for MAX and thus maximum value for MIN). So we have:

$$\textsc{Minimax}(s) = \begin{cases} \textsc{Utility}(s, \textsc{max}) & \text{if } \textsc{Is-Terminal}(s) \\ \max_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{To-Move}(s) = \textsc{max} \\ \min_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{To-Move}(s) = \textsc{min} \end{cases}$$

Let us apply these definitions to the game tree in Figure 5.2. The terminal nodes on the bottom level get their utility values from the game's UTILITY function. The first MIN node, labeled *B*, has three successor states with values 3, 12, and 8, so its minimax value is 3. Similarly, the other two MIN nodes have minimax value 2. The root node is a MAX node; its successor states have minimax values 3, 2, and 2; so it has a minimax value of 3. We can also identify the **minimax decision** at the root: action a_1 is the optimal choice for MAX because it leads to the state with the highest minimax value.

Minimax decision

This definition of optimal play for MAX assumes that MIN also plays optimally. What if MIN does not play optimally? Then MAX will do at least as well as against an optimal player, possibly better. However, that does not mean that it is always best to play the minimax optimal move when facing a suboptimal opponent. Consider a situation where optimal play by both sides will lead to a draw, but there is one risky move for MAX that leads to a state in which there are 10 possible response moves by MIN that all seem reasonable, but 9 of them are a loss for MIN and one is a loss for MAX. If MAX believes that MIN does not have sufficient computational power to discover the optimal move, MAX might want to try the risky move, on the grounds that a 9/10 chance of a win is better than a certain draw.

5.2.1 The minimax search algorithm

Now that we can compute MINIMAX(s), we can turn that into a search algorithm that finds the best move for MAX by trying all actions and choosing the one whose resulting state has the highest MINIMAX value. Figure 5.3 shows the algorithm. It is a recursive algorithm that proceeds all the way down to the leaves of the tree and then **backs up** the minimax values through the tree as the recursion unwinds. For example, in Figure 5.2, the algorithm

```
function MINIMAX-SEARCH(game, state) returns an action
   player ← game.TO-MOVE(state)
   value, move ← MAX-VALUE(game, state)
   return move

function MAX-VALUE(game, state) returns a (utility, move) pair
   if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
   v ← −∞
   for each a in game.ACTIONS(state) do
      v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
      if v2 > v then
         v, move ← v2, a
   return v, move

function MIN-VALUE(game, state) returns a (utility, move) pair
   if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
   v ← +∞
   for each a in game.ACTIONS(state) do
      v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
      if v2 < v then
         v, move ← v2, a
   return v, move
```

Figure 5.3 An algorithm for calculating the optimal move using minimax—the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there.

first recurses down to the three bottom-left nodes and uses the UTILITY function on them to discover that their values are 3, 12, and 8, respectively. Then it takes the minimum of these values, 3, and returns it as the backed-up value of node B. A similar process gives the backed-up values of 2 for C and 2 for D. Finally, we take the maximum of 3, 2, and 2 to get the backed-up value of 3 for the root node.

The minimax algorithm performs a complete depth-first exploration of the game tree. If the maximum depth of the tree is m and there are b legal moves at each point, then the time complexity of the minimax algorithm is $O(b^m)$. The space complexity is $O(bm)$ for an algorithm that generates all actions at once, or $O(m)$ for an algorithm that generates actions one at a time (see page 80). The exponential complexity makes MINIMAX impractical for complex games; for example, chess has a branching factor of about 35 and the average game has depth of about 80 ply, and it is not feasible to search $35^{80} \approx 10^{123}$ states. MINIMAX does, however, serve as a basis for the mathematical analysis of games. By approximating the minimax analysis in various ways, we can derive more practical algorithms.

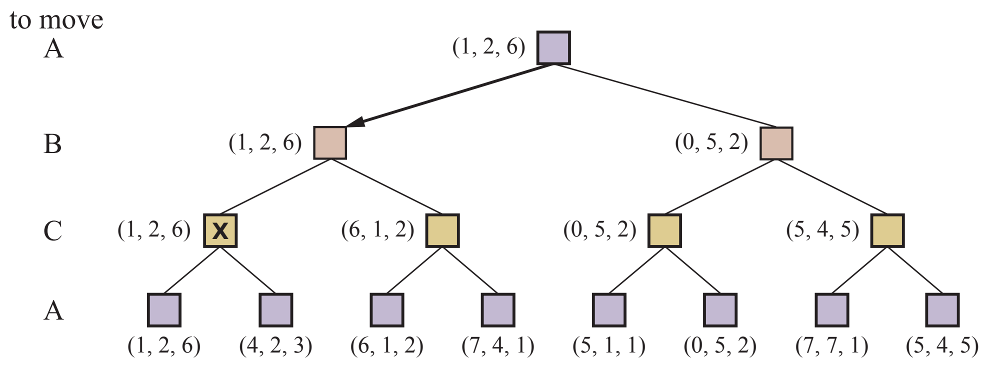


Figure 5.4 The first three ply of a game tree with three players (*A*, *B*, *C*). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

5.2.2 Optimal decisions in multiplayer games

Many popular games allow more than two players. Let us examine how to extend the minimax idea to multiplayer games. This is straightforward from the technical viewpoint, but raises some interesting new conceptual issues.

First, we need to replace the single value for each node with a *vector* of values. For example, in a three-player game with players *A*, *B*, and *C*, a vector $\langle v_A, v_B, v_C \rangle$ is associated with each node. For terminal states, this vector gives the utility of the state from each player's viewpoint. (In two-player, zero-sum games, the two-element vector can be reduced to a single value because the values are always opposite.) The simplest way to implement this is to have the UTILITY function return a vector of utilities.

Now we have to consider nonterminal states. Consider the node marked *X* in the game tree shown in Figure 5.4. In that state, player *C* chooses what to do. The two choices lead to terminal states with utility vectors $\langle v_A = 1, v_B = 2, v_C = 6 \rangle$ and $\langle v_A = 4, v_B = 2, v_C = 3 \rangle$. Since 6 is bigger than 3, *C* should choose the first move. This means that if state *X* is reached, subsequent play will lead to a terminal state with utilities $\langle v_A = 1, v_B = 2, v_C = 6 \rangle$. Hence, the backed-up value of *X* is this vector. In general, the backed-up value of a node *n* is the utility vector of the successor state with the highest value for the player choosing at *n*.

Anyone who plays multiplayer games, such as Diplomacy or Settlers of Catan, quickly becomes aware that much more is going on than in two-player games. Multiplayer games usually involve **alliances**, whether formal or informal, among the players. Alliances are made and broken as the game proceeds. How are we to understand such behavior? Are alliances a natural consequence of optimal strategies for each player in a multiplayer game? It turns out that they can be. Alliance

For example, suppose *A* and *B* are in weak positions and *C* is in a stronger position. Then it is often optimal for both *A* and *B* to attack *C* rather than each other, lest *C* destroy each of them individually. In this way, collaboration emerges from purely selfish behavior. Of course, as soon as *C* weakens under the joint onslaught, the alliance loses its value, and either *A* or *B* could violate the agreement.

In some cases, explicit alliances merely make concrete what would have happened anyway. In other cases, a social stigma attaches to breaking an alliance, so players must balance the immediate advantage of breaking an alliance against the long-term disadvantage of being perceived as untrustworthy. See Section 18.2 for more on these complications.

If the game is not zero-sum, then collaboration can also occur with just two players. Suppose, for example, that there is a terminal state with utilities $\langle v_A = 1000, v_B = 1000 \rangle$ and that 1000 is the highest possible utility for each player. Then the optimal strategy is for both players to do everything possible to reach this state—that is, the players will automatically cooperate to achieve a mutually desirable goal.

5.2.3 Alpha–Beta Pruning

The number of game states is exponential in the depth of the tree. No algorithm can completely eliminate the exponent, but we can sometimes cut it in half, computing the correct minimax decision without examining every state by **pruning** (see page 90) large parts of the tree that make no difference to the outcome. The particular technique we examine is called **alpha–beta pruning**.

Alpha–beta pruning

Consider again the two-ply game tree from Figure 5.2. Let's go through the calculation of the optimal decision once more, this time paying careful attention to what we know at each point in the process. The steps are explained in Figure 5.5. The outcome is that we can identify the minimax decision without ever evaluating two of the leaf nodes.

Another way to look at this is as a simplification of the formula for MINIMAX. Let the two unevaluated successors of node C in Figure 5.5 have values x and y. Then the value of the root node is given by

$$\begin{aligned}
\textsc{Minimax}(root) &= \max(\min(3,12,8), \min(2,x,y), \min(14,5,2)) \\
&= \max(3, \min(2,x,y), 2) \\
&= \max(3, z, 2) \qquad \text{where } z = \min(2,x,y) \leq 2 \\
&= 3.
\end{aligned}$$

In other words, the value of the root and hence the minimax decision are *independent* of the values of the leaves x and y, and therefore they can be pruned.

Alpha–beta pruning can be applied to trees of any depth, and it is often possible to prune entire subtrees rather than just leaves. The general principle is this: consider a node n somewhere in the tree (see Figure 5.6), such that Player has a choice of moving to n. If Player has a better choice either at the same level (e.g. m' in Figure 5.6) or at any point higher up in the tree (e.g. m in Figure 5.6), then Player will never move to n. So once we have found out enough about n (by examining some of its descendants) to reach this conclusion, we can prune it.

Remember that minimax search is depth-first, so at any one time we just have to consider the nodes along a single path in the tree. Alpha–beta pruning gets its name from the two extra parameters in MAX-VALUE$(state, \alpha, \beta)$ (see Figure 5.7) that describe bounds on the backed-up values that appear anywhere along the path:

$\alpha =$ the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX. Think: $\alpha =$ "at least."

$\beta =$ the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN. Think: $\beta =$ "at most."

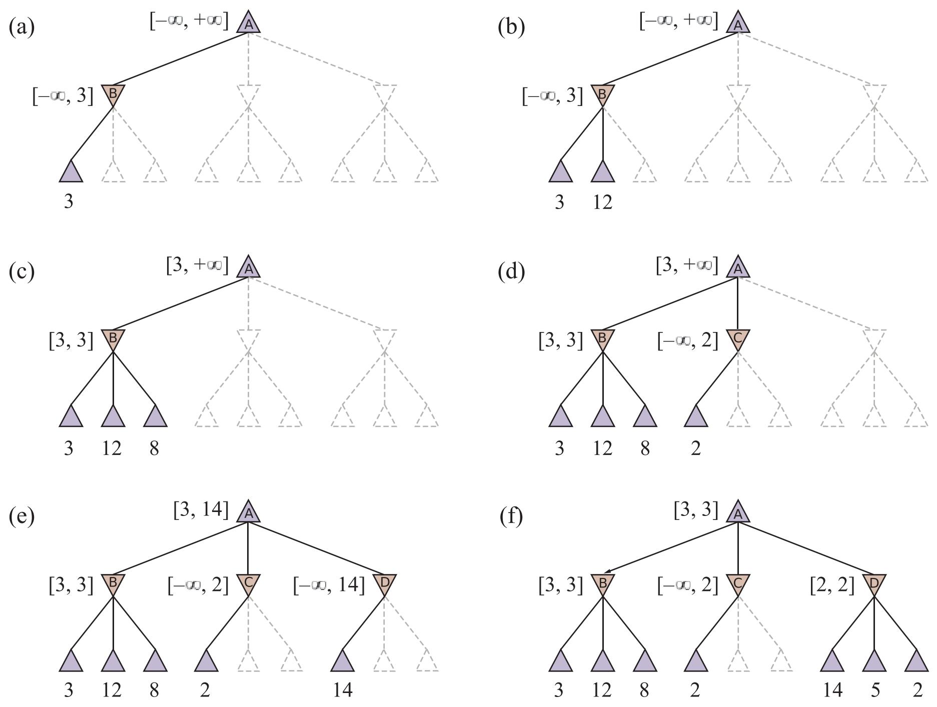


Figure 5.5 Stages in the calculation of the optimal decision for the game tree in Figure 5.2. At each point, we show the range of possible values for each node. (a) The first leaf below *B* has the value 3. Hence, *B*, which is a MIN node, has a value of *at most* 3. (b) The second leaf below *B* has a value of 12; MIN would avoid this move, so the value of *B* is still at most 3. (c) The third leaf below *B* has a value of 8; we have seen all *B*'s successor states, so the value of *B* is exactly 3. Now we can infer that the value of the root is *at least* 3, because MAX has a choice worth 3 at the root. (d) The first leaf below *C* has the value 2. Hence, *C*, which is a MIN node, has a value of *at most* 2. But we know that *B* is worth 3, so MAX would never choose *C*. Therefore, there is no point in looking at the other successor states of *C*. This is an example of alpha–beta pruning. (e) The first leaf below *D* has the value 14, so *D* is worth *at most* 14. This is still higher than MAX's best alternative (i.e., 3), so we need to keep exploring *D*'s successor states. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14. (f) The second successor of *D* is worth 5, so again we need to keep exploring. The third successor is worth 2, so now *D* is worth exactly 2. MAX's decision at the root is to move to *B*, giving a value of 3.

Alpha–beta search updates the values of α and β as it goes along and prunes the remaining branches at a node (i.e., terminates the recursive call) as soon as the value of the current node is known to be worse than the current α or β value for MAX or MIN, respectively. The complete algorithm is given in Figure 5.7. Figure 5.5 traces the progress of the algorithm on a game tree.

5.2.4 Move ordering

The effectiveness of alpha–beta pruning is highly dependent on the order in which the states are examined. For example, in Figure 5.5(e) and (f), we could not prune any successors of *D* at all because the worst successors (from the point of view of MIN) were generated first. If the

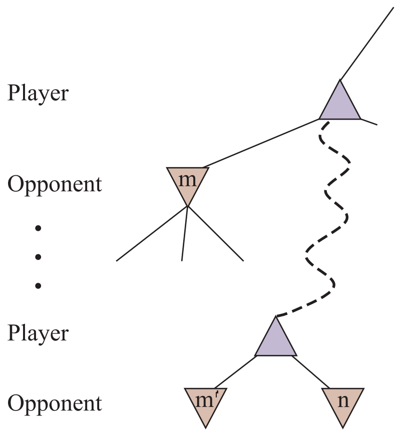


Figure 5.6 The general case for alpha–beta pruning. If m or m' is better than n for Player, we will never get to n in play.

```
function ALPHA-BETA-SEARCH(game, state) returns an action
   player ← game.TO-MOVE(state)
   value, move ← MAX-VALUE(game, state, $-\infty$, $+\infty$)
   return move

function MAX-VALUE(game, state, $\alpha$, $\beta$) returns a (utility, move) pair
   if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
   $v \leftarrow -\infty$
   for each a in game.ACTIONS(state) do
      v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), $\alpha$, $\beta$)
      if $v2 > v$ then
         v, move ← v2, a
         $\alpha \leftarrow$ MAX($\alpha$, v)
      if $v \geq \beta$ then return v, move
   return v, move

function MIN-VALUE(game, state, $\alpha$, $\beta$) returns a (utility, move) pair
   if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
   $v \leftarrow +\infty$
   for each a in game.ACTIONS(state) do
      v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), $\alpha$, $\beta$)
      if $v2 < v$ then
         v, move ← v2, a
         $\beta \leftarrow$ MIN($\beta$, v)
      if $v \leq \alpha$ then return v, move
   return v, move
```

Figure 5.7 The alpha–beta search algorithm. Notice that these functions are the same as the MINIMAX-SEARCH functions in Figure 5.3, except that we maintain bounds in the variables α and β, and use them to cut off search when a value is outside the bounds.

third successor of D had been generated first, with value 2, we would have been able to prune the other two successors. This suggests that it might be worthwhile to try to first examine the successors that are likely to be best.

If this could be done perfectly, alpha–beta would need to examine only $O(b^{m/2})$ nodes to pick the best move, instead of $O(b^m)$ for minimax. This means that the effective branching factor becomes $\sqrt{b}$ instead of b—for chess, about 6 instead of 35. Put another way, alpha–beta with perfect move ordering can solve a tree roughly twice as deep as minimax in the same amount of time. With random move ordering, the total number of nodes examined will be roughly $O(b^{3m/4})$ for moderate b. Now, obviously we cannot achieve *perfect* move ordering—in that case the ordering function could be used to play a perfect game! But we can often get fairly close. For chess, a fairly simple ordering function (such as trying captures first, then threats, then forward moves, and then backward moves) gets you to within about a factor of 2 of the best-case $O(b^{m/2})$ result.

Adding dynamic move-ordering schemes, such as trying first the moves that were found to be best in the past, brings us quite close to the theoretical limit. The past could be the previous move—often the same threats remain—or it could come from previous exploration of the current move through a process of **iterative deepening** (see page 80). First, search one ply deep and record the ranking of moves based on their evaluations. Then search one ply deeper, using the previous ranking to inform move ordering; and so on. The increased search time from iterative deepening can be more than made up from better move ordering. The best moves are known as **killer moves**, and to try them first is called the killer move heuristic.

Killer moves

In Section 3.3.3, we noted that redundant paths to repeated states can cause an exponential increase in search cost, and that keeping a table of previously reached states can address this problem. In game tree search, repeated states can occur because of **transpositions**—different permutations of the move sequence that end up in the same position, and the problem can be addressed with a **transposition table** that caches the heuristic value of states.

Transposition

Transposition table

For example, suppose White has a move w_1 that can be answered by Black with b_1 and an unrelated move w_2 on the other side of the board that can be answered by b_2, and that we search the sequence of moves $[w_1, b_1, w_2, b_2]$; let's call the resulting state s. After exploring a large subtree below s, we find its backed-up value, which we store in the transposition table. When we later search the sequence of moves $[w_2, b_2, w_1, b_1]$, we end up in s again, and we can look up the value instead of repeating the search. In chess, use of transposition tables is very effective, allowing us to double the reachable search depth in the same amount of time.

Even with alpha–beta pruning and clever move ordering, minimax won't work for games like chess and Go, because there are still too many states to explore in the time available. In the very first paper on computer game-playing, *Programming a Computer for Playing Chess* (Shannon, 1950), Claude Shannon recognized this problem and proposed two strategies: a **Type A strategy** considers all possible moves to a certain depth in the search tree, and then uses a heuristic evaluation function to estimate the utility of states at that depth. It explores a *wide but shallow* portion of the tree. A **Type B strategy** ignores moves that look bad, and follows promising lines "as far as possible." It explores a *deep but narrow* portion of the tree.

Type A strategy

Type B strategy

Historically, most chess programs have been Type A (which we cover in the next section), whereas Go programs are more often Type B (covered in Section 5.4), because the branching factor is much higher in Go. More recently, Type B programs have shown world-champion-level play across a variety of games, including chess (Silver *et al.*, 2018).

5.3 Heuristic Alpha–Beta Tree Search

To make use of our limited computation time, we can cut off the search early and apply a heuristic **evaluation function** to states, effectively treating nonterminal nodes as if they were terminal. In other words, we replace the UTILITY function with EVAL, which estimates a state's utility. We also replace the terminal test by a **cutoff test**, which must return true for terminal states, but is otherwise free to decide when to cut off the search, based on the search depth and any property of the state that it chooses to consider. That gives us the formula H-MINIMAX(s, d) for the heuristic minimax value of state s at search depth d:

Cutoff test

$$\text{H-MINIMAX}(s,d) = \begin{cases} \text{EVAL}(s,\text{MAX}) & \text{if IS-CUTOFF}(s,d) \\ \max_{a\in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a),d+1) & \text{if TO-MOVE}(s) = \text{MAX} \\ \min_{a\in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a),d+1) & \text{if TO-MOVE}(s) = \text{MIN}. \end{cases}$$

5.3.1 Evaluation functions

A heuristic evaluation function $\text{EVAL}(s,p)$ returns an *estimate* of the expected utility of state s to player p, just as the heuristic functions of Chapter 3 return an estimate of the distance to the goal. For terminal states, it must be that $\text{EVAL}(s,p) = \text{UTILITY}(s,p)$ and for nonterminal states, the evaluation must be somewhere between a loss and a win: $\text{UTILITY}(loss,p) \leq \text{EVAL}(s,p) \leq \text{UTILITY}(win,p)$.

Beyond those requirements, what makes for a good evaluation function? First, the computation must not take too long! (The whole point is to search faster.) Second, the evaluation function should be strongly correlated with the actual chances of winning. One might well wonder about the phrase "chances of winning." After all, chess is not a game of chance: we know the current state with certainty, and no dice are involved; if neither player makes a mistake, the outcome is predetermined. But if the search must be cut off at nonterminal states, then the algorithm will necessarily be *uncertain* about the final outcomes of those states (even though that uncertainty could be resolved with infinite computing resources).

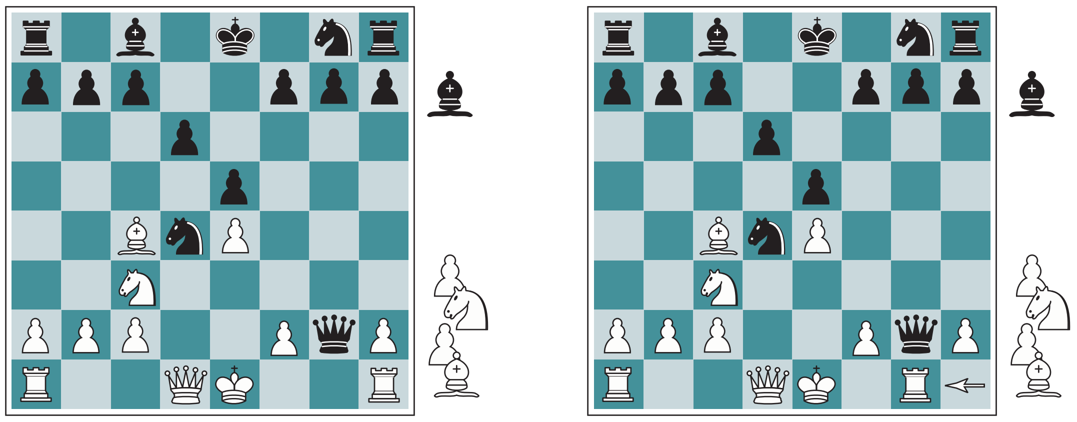
Features

Let us make this idea more concrete. Most evaluation functions work by calculating various **features** of the state—for example, in chess, we would have features for the number of white pawns, black pawns, white queens, black queens, and so on. The features, taken together, define various *categories* or *equivalence classes* of states: the states in each category have the same values for all the features. For example, one category might contain all two-pawn versus one-pawn endgames. Any given category will contain some states that lead (with perfect play) to wins, some that lead to draws, and some that lead to losses.

Expected value

The evaluation function does not know which states are which, but it can return a single value that estimates the *proportion* of states with each outcome. For example, suppose our experience suggests that 82% of the states encountered in the two-pawns versus one-pawn category lead to a win (utility +1); 2% to a loss (0), and 16% to a draw (1/2). Then a reasonable evaluation for states in the category is the **expected value**: $(0.82 \times +1) + (0.02 \times 0) + (0.16 \times 1/2) = 0.90$. In principle, the expected value can be determined for each category of states, resulting in an evaluation function that works for any state.

In practice, this kind of analysis requires too many categories and hence too much experience to estimate all the probabilities. Instead, most evaluation functions compute separate numerical contributions from each feature and then *combine* them to find the total value. For



(a) White to move

(b) White to move

Figure 5.8 Two chess positions that differ only in the position of the rook at lower right. In (a), Black has an advantage of a knight and two pawns, which should be enough to win the game. In (b), White will capture the queen, giving it an advantage that should be strong enough to win.

centuries, chess players have developed ways of judging the value of a position using just this idea. For example, introductory chess books give an approximate **material value** for each piece: each pawn is worth 1, a knight or bishop is worth 3, a rook 5, and the queen 9. Other features such as "good pawn structure" and "king safety" might be worth half a pawn, say. These feature values are then simply added up to obtain the evaluation of the position.

Material value

Mathematically, this kind of evaluation function is called a **weighted linear function** because it can be expressed as

Weighted linear function

$$\text{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)\,,$$

where each f_i is a feature of the position (such as "number of white bishops") and each w_i is a weight (saying how important that feature is). The weights should be normalized so that the sum is always within the range of a loss (0) to a win (+1). A secure advantage equivalent to a pawn gives a substantial likelihood of winning, and a secure advantage equivalent to three pawns should give almost certain victory, as illustrated in Figure 5.8(a). We said that the evaluation function should be strongly correlated with the actual chances of winning, but it need not be linearly correlated: if state s is twice as likely to win as state s' we don't require that EVAL(S) be twice EVAL(S'); all we require is that EVAL(S) > EVAL(S').

Adding up the values of features seems like a reasonable thing to do, but in fact it involves a strong assumption: that the contribution of each feature is *independent* of the values of the other features. For this reason, current programs for chess and other games also use *nonlinear* combinations of features. For example, a pair of bishops might be worth more than twice the value of a single bishop, and a bishop is worth more in the endgame than earlier—when the *move number* feature is high or the *number of remaining pieces* feature is low.

Where do the features and weights come from? They're not part of the rules of chess, but they are part of the culture of human chess-playing experience. In games where this kind of experience is not available, the weights of the evaluation function can be estimated by the machine learning techniques of Chapter 22. Applying these techniques to chess has confirmed that a bishop is indeed worth about three pawns, and it appears that centuries of human experience can be replicated in just a few hours of machine learning.

5.3.2 Cutting off search

The next step is to modify ALPHA-BETA-SEARCH so that it will call the heuristic EVAL function when it is appropriate to cut off the search. We replace the two lines in Figure 5.7 that mention IS-TERMINAL with the following line:

if *game*.IS-CUTOFF(*state*, *depth*) **then return** *game*.EVAL(*state*, *player*), *null*

We also must arrange for some bookkeeping so that the current *depth* is incremented on each recursive call. The most straightforward approach to controlling the amount of search is to set a fixed depth limit so that IS-CUTOFF(*state*, *depth*) returns *true* for all *depth* greater than some fixed depth *d* (as well as for all terminal states). The depth *d* is chosen so that a move is selected within the allocated time. A more robust approach is to apply iterative deepening. (See Chapter 3.) When time runs out, the program returns the move selected by the deepest completed search. As a bonus, if in each round of iterative deepening we keep entries in the transposition table, subsequent rounds will be faster, and we can use the evaluations to improve move ordering.

These simple approaches can lead to errors due to the approximate nature of the evaluation function. Consider again the simple evaluation function for chess based on material advantage. Suppose the program searches to the depth limit, reaching the position in Figure 5.8(b), where Black is ahead by a knight and two pawns. It would report this as the heuristic value of the state, thereby declaring that the state is a probable win by Black. But White's next move captures Black's queen with no compensation. Hence, the position is actually favorable for White, but this can be seen only by looking ahead.

Quiescence

The evaluation function should be applied only to positions that are **quiescent**—that is, positions in which there is no pending move (such as a capturing the queen) that would wildly swing the evaluation. For nonquiescent positions the IS-CUTOFF returns false, and the search continues until quiescent positions are reached. This extra **quiescence search** is sometimes restricted to consider only certain types of moves, such as capture moves, that will quickly resolve the uncertainties in the position.

Quiescence search

Horizon effect

The **horizon effect** is more difficult to eliminate. It arises when the program is facing an opponent's move that causes serious damage and is ultimately unavoidable, but can be temporarily avoided by the use of delaying tactics. Consider the chess position in Figure 5.9. It is clear that there is no way for the black bishop to escape. For example, the white rook can capture it by moving to h1, then a1, then a2; a capture at depth 6 ply.

But Black does have a sequence of moves that pushes the capture of the bishop "over the horizon." Suppose Black searches to depth 8 ply. Most moves by Black will lead to the eventual capture of the bishop, and thus will be marked as "bad" moves. But Black will also consider the sequence of moves that starts by checking the king with a pawn, and enticing the king to capture the pawn. Black can then do the same thing with a second pawn. That takes up enough moves that the capture of the bishop would not be discovered during the remainder of Black's search. Black thinks that the line of play has saved the bishop at the price of two

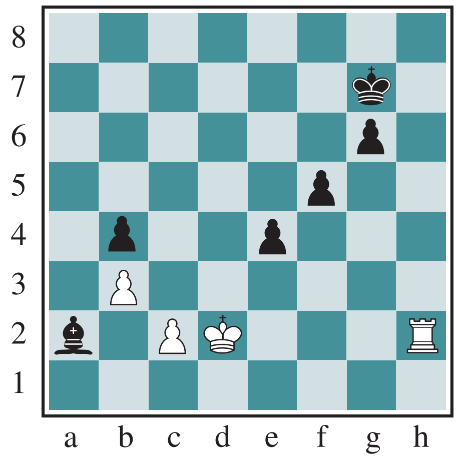


Figure 5.9 The horizon effect. With Black to move, the black bishop is surely doomed. But Black can forestall that event by checking the white king with its pawns, encouraging the king to capture the pawns. This pushes the inevitable loss of the bishop over the horizon, and thus the pawn sacrifices are seen by the search algorithm as good moves rather than bad ones.

pawns, when actually all it has done is waste pawns and push the inevitable capture of the bishop beyond the horizon that Black can see.

Singular extension

One strategy to mitigate the horizon effect is to allow **singular extensions**, moves that are "clearly better" than all other moves in a given position, even when the search would normally be cut off at that point. In our example, a search will have revealed that three moves of the white rook—h2 to h1, then h1 to a1, and then a1 capturing the bishop on a2—are each in turn clearly better moves, so even if a sequence of pawn moves pushes us to the horizon, these clearly better moves will be given a chance to extend the search. This makes the tree deeper, but because there are usually few singular extensions, the strategy does not add many total nodes to the tree, and has proven to be effective in practice.

5.3.3 Forward pruning

Forward pruning

Alpha–beta pruning prunes branches of the tree that can have no effect on the final evaluation, but **forward pruning** prunes moves that appear to be poor moves, but might possibly be good ones. Thus, the strategy saves computation time at the risk of making an error. In Shannon's terms, this is a Type B strategy. Clearly, most human chess players do this, considering only a few moves from each position (at least consciously).

One approach to forward pruning is **beam search** (see page 115): on each ply, consider only a "beam" of the n best moves (according to the evaluation function) rather than considering all possible moves. Unfortunately, this approach is rather dangerous because there is no guarantee that the best move will not be pruned away.

The PROBCUT, or probabilistic cut, algorithm (Buro, 1995) is a forward-pruning version of alpha–beta search that uses statistics gained from prior experience to lessen the chance that the best move will be pruned. Alpha–beta search prunes any node that is *provably* outside the current (α, β) window. PROBCUT also prunes nodes that are *probably* outside the window. It computes this probability by doing a shallow search to compute the backed-up value

v of a node and then using past experience to estimate how likely it is that a score of v at depth d in the tree would be outside (α, β). Buro applied this technique to his Othello program, LOGISTELLO, and found that a version of his program with PROBCUT beat the regular version 64% of the time, even when the regular version was given twice as much time.

Late move reduction

Another technique, **late move reduction**, works under the assumption that move ordering has been done well, and therefore moves that appear later in the list of possible moves are less likely to be good moves. But rather than pruning them away completely, we just reduce the depth to which we search these moves, thereby saving time. If the reduced search comes back with a value above the current α value, we can re-run the search with the full depth.

Combining all the techniques described here results in a program that can play creditable chess (or other games). Let us assume we have implemented an evaluation function for chess, a reasonable cutoff test with a quiescence search. Let us also assume that, after months of tedious bit-bashing, we can generate and evaluate around a million nodes per second on the latest PC. The branching factor for chess is about 35, on average, and 35^5 is about 50 million, so if we used minimax search, we could look ahead only five ply in about a minute of computation; the rules of competition would not give us enough time to search six ply. Though not incompetent, such a program can be defeated by an average human chess player, who can occasionally plan six or eight ply ahead.

With alpha–beta search and a large transposition table we get to about 14 ply, which results in an expert level of play. We could trade in our PC for a workstation with 8 GPUs, getting us over a billion nodes per second, but to obtain grandmaster status we would still need an extensively tuned evaluation function and a large database of endgame moves. Top chess programs like STOCKFISH have all of these, often reaching depth 30 or more in the search tree and far exceeding the ability of any human player.

5.3.4 Search versus lookup

Somehow it seems like overkill for a chess program to start a game by considering a tree of a billion game states, only to conclude that it will play pawn to e4 (the most popular first move). Books describing good play in the opening and endgame in chess have been available for more than a century (Tattersall, 1911). It is not surprising, therefore, that many game-playing programs use *table lookup* rather than search for the opening and ending of games.

For the openings, the computer is mostly relying on the expertise of humans. The best advice of human experts on how to play each opening can be copied from books and entered into tables for the computer's use. In addition, computers can gather statistics from a database of previously played games to see which opening sequences most often lead to a win. For the first few moves there are few possibilities, and most positions will be in the table. Usually after about 10 or 15 moves we end up in a rarely seen position, and the program must switch from table lookup to search.

Near the end of the game there are again fewer possible positions, and thus it is easier to do lookup. But here it is the computer that has the expertise: computer analysis of endgames goes far beyond human abilities. Novice humans can win a king-and-rook-versus-king (KRK) endgame by following a few simple rules. Other endings, such as king, bishop, and knight versus king (KBNK), are difficult to master and have no succinct strategy description.

A computer, on the other hand, can completely *solve* the endgame by producing a **policy**, which is a mapping from every possible state to the best move in that state. Then the computer can play perfectly by looking up the right move in this table. The table is constructed by **retrograde** minimax search: start by considering all ways to place the KBNK pieces on the board. Some of the positions are wins for white; mark them as such. Then reverse the rules of chess to do reverse moves rather than moves. Any move by White that, no matter what move Black responds with, ends up in a position marked as a win, must also be a win. Continue this search until all possible positions are resolved as win, loss, or draw, and you have an infallible lookup table for all endgames with those pieces. This has been done not only for KBNK endings, but for all endings with seven or fewer pieces. The tables contain 400 trillion positions. An eight-piece table would require 40 quadrillion positions.

Retrograde

5.4 Monte Carlo Tree Search

The game of Go illustrates two major weaknesses of heuristic alpha–beta tree search: First, Go has a branching factor that starts at 361, which means alpha–beta search would be limited to only 4 or 5 ply. Second, it is difficult to define a good evaluation function for Go because material value is not a strong indicator and most positions are in flux until the endgame. In response to these two challenges, modern Go programs have abandoned alpha–beta search and instead use a strategy called **Monte Carlo tree search (MCTS)**.[3]

Monte Carlo tree search (MCTS)

The basic MCTS strategy does not use a heuristic evaluation function. Instead, the value of a state is estimated as the average utility over a number of **simulations** of complete games starting from the state. A simulation (also called a **playout** or **rollout**) chooses moves first for one player, than for the other, repeating until a terminal position is reached. At that point the rules of the game (not fallible heuristics) determine who has won or lost, and by what score. For games in which the only outcomes are a win or a loss, "average utility" is the same as "win percentage."

Simulation

Playout

Rollout

How do we choose what moves to make during the playout? If we just choose randomly, then after multiple simulations we get an answer to the question "what is the best move if both players play randomly?" For some simple games, that happens to be the same answer as "what is the best move if both players play well?," but for most games it is not. To get useful information from the playout we need a **playout policy** that biases the moves towards good ones. For Go and other games, playout policies have been successfully learned from self-play by using neural networks. Sometimes game-specific heuristics are used, such as "consider capture moves" in chess or "take the corner square" in Othello.

Playout policy

Given a playout policy, we next need to decide two things: from what positions do we start the playouts, and how many playouts do we allocate to each position? The simplest answer, called **pure Monte Carlo search**, is to do N simulations starting from the current state of the game, and track which of the possible moves from the current position has the highest win percentage.

Pure Monte Carlo search

For some stochastic games this converges to optimal play as N increases, but for most games it is not sufficient—we need a **selection policy** that selectively focuses the computational resources on the important parts of the game tree. It balances two factors: **exploration** of states that have had few playouts, and **exploitation** of states that have done well in past playouts, to get a more accurate estimate of their value. (See Section 17.3 for more on the

Selection policy

Exploration

Exploitation

[3] "Monte Carlo" algorithms are randomized algorithms named after the Casino de Monte-Carlo in Monaco.

exploration/exploitation tradeoff.) Monte Carlo tree search does that by maintaining a search tree and growing it on each iteration of the following four steps, as shown in Figure 5.10:

- **Selection**: Starting at the root of the search tree, we choose a move (guided by the selection policy), leading to a successor node, and repeat that process, moving down the tree to a leaf. Figure 5.10(a) shows a search tree with the root representing a state where white has just moved, and white has won 37 out of the 100 playouts done so far. The thick arrow shows the selection of a move by black that leads to a node where black has won 60/79 playouts. This is the best win percentage among the three moves, so selecting it is an example of exploitation. But it would also have been reasonable to select the 2/11 node for the sake of exploration—with only 11 playouts, the node still has high uncertainty in its valuation, and might end up being best if we gain more information about it. Selection continues on to the leaf node marked 27/35.
- **Expansion**: We grow the search tree by generating a new child of the selected node; Figure 5.10(b) shows the new node marked with 0/0. (Some versions generate more than one child in this step.)
- **Simulation**: We perform a playout from the newly generated child node, choosing moves for both players according to the playout policy. These moves are *not* recorded in the search tree. In the figure, the simulation results in a win for black.
- **Back-propagation**: We now use the result of the simulation to update all the search tree nodes going up to the root. Since black won the playout, black nodes are incremented in both the number of wins and the number of playouts, so 27/35 becomes 28/26 and 60/79 becomes 61/80. Since white lost, the white nodes are incremented in the number of playouts only, so 16/53 becomes 16/54 and the root 37/100 becomes 37/101.

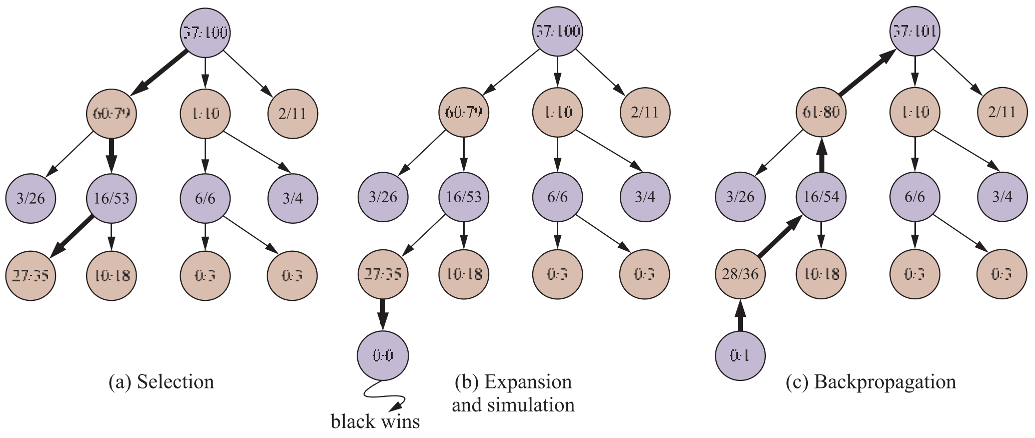


Figure 5.10 One iteration of the process of choosing a move with Monte Carlo tree search (MCTS) using the upper confidence bounds applied to trees (UCT) selection metric, shown after 100 iterations have already been done. In (a) we select moves, all the way down the tree, ending at the leaf node marked 27/35 (for 27 wins for black out of 35 playouts). In (b) we expand the selected node and do a simulation (playout), which ends in a win for black. In (c), the results of the simulation are back-propagated up the tree.

```
function MONTE-CARLO-TREE-SEARCH(state) returns an action
   tree ← NODE(state)
   while IS-TIME-REMAINING() do
      leaf ← SELECT(tree)
      child ← EXPAND(leaf)
      result ← SIMULATE(child)
      BACK-PROPAGATE(result, child)
   return the move in ACTIONS(state) whose node has highest number of playouts
```

Figure 5.11 The Monte Carlo tree search algorithm. A game tree, *tree*, is initialized, and then we repeat a cycle of SELECT / EXPAND / SIMULATE / BACK-PROPAGATE until we run out of time, and return the move that led to the node with the highest number of playouts.

We repeat these four steps either for a set number of iterations, or until the allotted time has expired, and then return the move with the highest number of playouts.

One very effective selection policy is called "upper confidence bounds applied to trees" or **UCT**. The policy ranks each possible move based on an upper confidence bound formula called **UCB1**. (See Section 17.3.3 for more details.) For a node n, the formula is:

UCT

UCB1

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(\text{PARENT}(n))}{N(n)}}$$

where $U(n)$ is the total utility of all playouts that went through node n, $N(n)$ is the number of playouts through node n, and $\text{PARENT}(n)$ is the parent node of n in the tree. Thus $\frac{U(n)}{N(n)}$ is the exploitation term: the average utility of n. The term with the square root is the exploration term: it has the count $N(n)$ in the denominator, which means the term will be high for nodes that have only been explored a few times. In the numerator it has the log of the number of times we have explored the parent of n. This means that if we are selecting n some non-zero percentage of the time, the exploration term goes to zero as the counts increase, and eventually the playouts are given to the node with highest average utility.

C is a constant that balances exploitation and exploration. There is a theoretical argument that C should be $\sqrt{2}$, but in practice, game programmers try multiple values for C and choose the one that performs best. (Some programs use slightly different formulas; for example, ALPHAZERO adds in a term for move probability, which is calculated by a neural network trained from past self-play.) With $C = 1.4$, the 60/79 node in Figure 5.10 has the highest UCB1 score, but with $C = 1.5$, it would be the 2/11 node.

Figure 5.11 shows the complete UCT MCTS algorithm. When the iterations terminate, the move with the highest number of playouts is returned. You might think that it would be better to return the node with the highest average utility, but the idea is that a node with 65/100 wins is better than one with 2/3 wins, because the latter has a lot of uncertainty. In any event, the UCB1 formula ensures that the node with the most playouts is almost always the node with the highest win percentage, because the selection process favors win percentage more and more as the number of playouts goes up.

The time to compute a playout is linear, not exponential, in the depth of the game tree, because only one move is taken at each choice point. That gives us plenty of time for multiple playouts. For example: consider a game with a branching factor of 32, where the average game lasts 100 ply. If we have enough computing power to consider a billion game states before we have to make a move, then minimax can search 6 ply deep, alpha–beta with perfect move ordering can search 12 ply, and Monte Carlo search can do 10 million playouts. Which approach will be better? That depends on the accuracy of the heuristic function versus the selection and playout policies.

The conventional wisdom has been that Monte Carlo search has an advantage over alpha–beta for games like Go where the branching factor is very high (and thus alpha–beta can't search deep enough), or when it is difficult to define a good evaluation function. What alpha–beta does is choose the path to a node that has the highest achievable evaluation function score, given that the opponent will be trying to minimize the score. Thus, if the evaluation function is inaccurate, alpha–beta will be inaccurate. A miscalculation on a single node can lead alpha–beta to erroneously choose (or avoid) a path to that node. But Monte Carlo search relies on the aggregate of many playouts, and thus is not as vulnerable to a single error. It is possible to combine MCTS and evaluation functions by doing a playout for a certain number of moves, but then truncating the playout and applying an evaluation function.

Early playout termination

It is also possible to combine aspects of alpha–beta and Monte Carlo search. For example, in games that can last many moves, we may want to use **early playout termination**, in which we stop a playout that is taking too many moves, and either evaluate it with a heuristic evaluation function or just declare it a draw.

Monte Carlo search can be applied to brand-new games, in which there is no body of experience to draw upon to define an evaluation function. As long as we know the rules of the game, Monte Carlo search does not need any additional information. The selection and playout policies can make good use of hand-crafted expert knowledge when it is available, but good policies can be learned using neural networks trained by self-play alone.

Monte Carlo search has a disadvantage when it is likely that a single move can change the course of the game, because the stochastic nature of Monte Carlo search means it might fail to consider that move. In other words, Type B pruning in Monte Carlo search means that a vital line of play might not be explored at all. Monte Carlo search also has a disadvantage when there are game states that are "obviously" a win for one side or the other (according to human knowledge and to an evaluation function), but where it will still take many moves in a playout to verify the winner. It was long held that alpha–beta search was better suited for games like chess with low branching factor and good evaluation functions, but recently Monte Carlo approaches have demonstrated success in chess and other games.

The general idea of simulating moves into the future, observing the outcome, and using the outcome to determine which moves are good ones is one kind of **reinforcement learning**, which is covered in Chapter 22.

5.5 Stochastic Games

Stochastic game

Stochastic games bring us a little closer to the unpredictability of real life by including a random element, such as the throwing of dice. Backgammon is a typical stochastic game that combines luck and skill. In the backgammon position of Figure 5.12, White has rolled a 6–5 and has four possible moves (each of which moves one piece forward (clockwise) 5 positions, and one piece forward 6 positions).

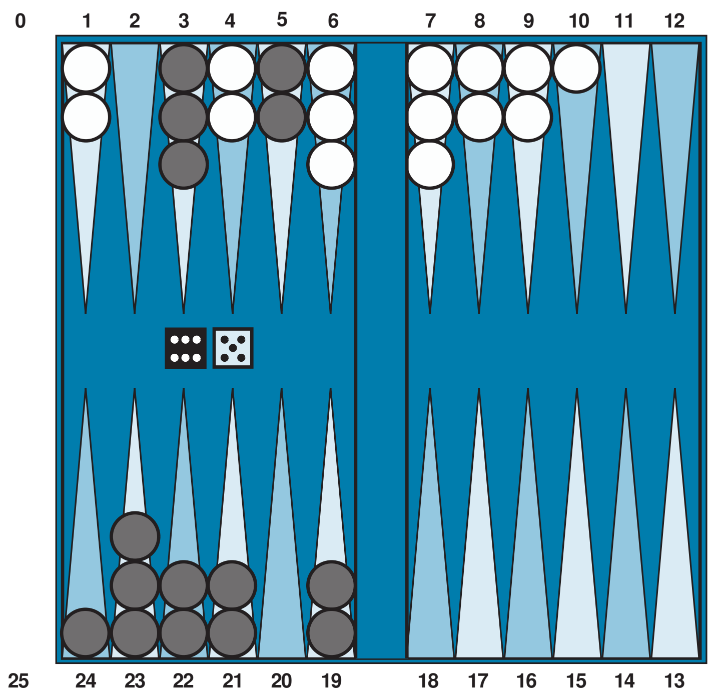


Figure 5.12 A typical backgammon position. The goal of the game is to move all one's pieces off the board. Black moves clockwise toward 25, and White moves counterclockwise toward 0. A piece can move to any position unless multiple opponent pieces are there; if there is one opponent, it is captured and must start over. In the position shown, Black has rolled 6–5 and must choose among four legal moves: (5–11,5–10), (5–11,19–24), (5–10,10–16), and (5–11,11–16), where the notation (5–11,11–16) means move one piece from position 5 to 11, and then move a piece from 11 to 16.

At this point Black knows what moves can be made, but does not know what White is going to roll and thus does not know what White's legal moves will be. That means Black cannot construct a standard game tree of the sort we saw in chess and tic-tac-toe. A game tree in backgammon must include **chance nodes** in addition to MAX and MIN nodes. Chance nodes are shown as circles in Figure 5.13. The branches leading from each chance node denote the possible dice rolls; each branch is labeled with the roll and its probability. There are 36 ways to roll two dice, each equally likely; but because a 6–5 is the same as a 5–6, there are only 21 distinct rolls. The six doubles (1–1 through 6–6) each have a probability of 1/36, so we say $P(1\text{–}1) = 1/36$. The other 15 distinct rolls each have a 1/18 probability.

Chance nodes

The next step is to understand how to make correct decisions. Obviously, we still want to pick the move that leads to the best position. However, positions do not have definite minimax values. Instead, we can only calculate the **expected value** of a position: the average over all possible outcomes of the chance nodes.

Expected value

This leads us to the **expectiminimax value** for games with chance nodes, a generalization of the minimax value for deterministic games. Terminal nodes and MAX and MIN nodes work exactly the same way as before (with the caveat that the legal moves for MAX and MIN will depend on the outcome of the dice roll in the previous chance node). For chance nodes we

Expectiminimax value

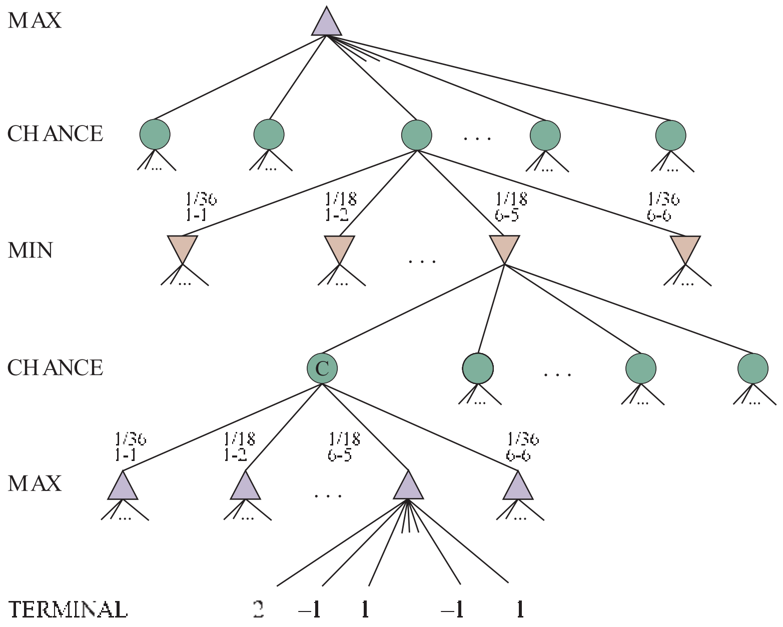


Figure 5.13 Schematic game tree for a backgammon position.

compute the expected value, which is the sum of the value over all outcomes, weighted by the probability of each chance action:

$$\textsc{Expectiminimax}(s) = \begin{cases} \textsc{Utility}(s, \textsc{max}) & \text{if } \textsc{Is-Terminal}(s) \\ \max_a \textsc{Expectiminimax}(\textsc{Result}(s,a)) & \text{if } \textsc{To-Move}(s) = \textsc{max} \\ \min_a \textsc{Expectiminimax}(\textsc{Result}(s,a)) & \text{if } \textsc{To-Move}(s) = \textsc{min} \\ \sum_r P(r) \textsc{Expectiminimax}(\textsc{Result}(s,r)) & \text{if } \textsc{To-Move}(s) = \textsc{chance} \end{cases}$$

where r represents a possible dice roll (or other chance event) and $\textsc{Result}(s,r)$ is the same state as s, with the additional fact that the result of the dice roll is r.

5.5.1 Evaluation functions for games of chance

As with minimax, the obvious approximation to make with expectiminimax is to cut the search off at some point and apply an evaluation function to each leaf. One might think that evaluation functions for games such as backgammon should be just like evaluation functions for chess—they just need to give higher values to better positions. But in fact, the presence of chance nodes means that one has to be more careful about what the values mean.

Figure 5.14 shows what happens: with an evaluation function that assigns the values [1, 2, 3, 4] to the leaves, move a_1 is best; with values [1, 20, 30, 400], move a_2 is best. Hence, the program behaves totally differently if we make a change to some of the evaluation values, even if the preference order remains the same.

It turns out that to avoid this problem, the evaluation function must return values that are a positive linear transformation of the **probability** of winning (or of the expected utility, for games that have outcomes other than win/lose). This relation to probability is an important

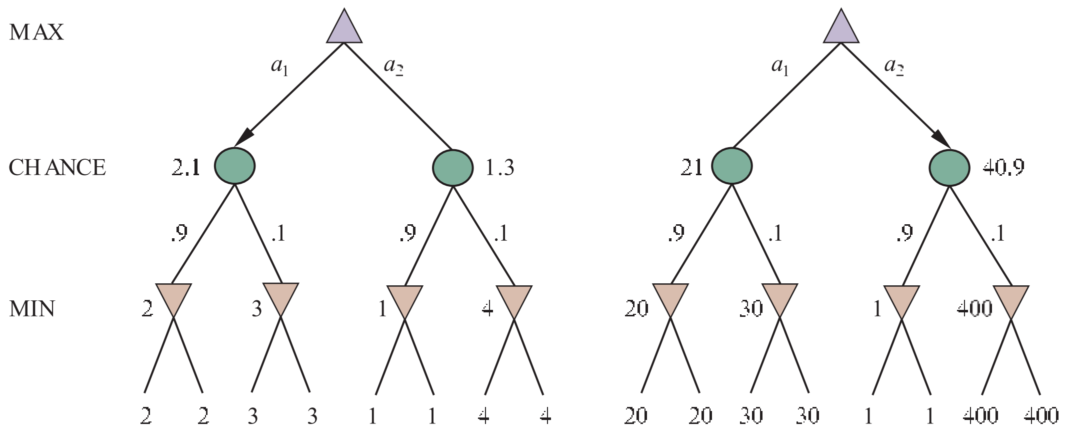


Figure 5.14 An order-preserving transformation on leaf values changes the best move.

and general property of situations in which uncertainty is involved, and we discuss it further in Chapter 16.

If the program knew in advance all the dice rolls that would occur for the rest of the game, solving a game with dice would be just like solving a game without dice, which minimax does in $O(b^m)$ time, where b is the branching factor and m is the maximum depth of the game tree. Because expectiminimax is also considering all the possible dice-roll sequences, it will take $O(b^m n^m)$, where n is the number of distinct rolls.

Even if the search is limited to some small depth d, the extra cost compared with that of minimax makes it unrealistic to consider looking ahead very far in most games of chance. In backgammon n is 21 and b is usually around 20, but in some situations can be as high as 4000 for dice rolls that are doubles. We could probably only manage three ply of search.

Another way to think about the problem is this: the advantage of alpha–beta is that it ignores future developments that just are not going to happen, given best play. Thus, it concentrates on likely occurrences. But in a game where a throw of two dice precedes each move, there are *no* likely sequences of moves; even the most likely move occurs only 2/36 of the time, because for the move to take place, the dice would first have to come out the right way to make it legal. This is a general problem whenever uncertainty enters the picture: the possibilities are multiplied enormously, and forming detailed plans of action becomes pointless because the world probably will not play along.

It may have occurred to you that something like alpha–beta pruning could be applied to game trees with chance nodes. It turns out that it can. The analysis for MIN and MAX nodes is unchanged, but we can also prune chance nodes, using a bit of ingenuity. Consider the chance node C in Figure 5.13 and what happens to its value as we evaluate its children. Is it possible to find an upper bound on the value of C before we have looked at all its children? (Recall that this is what alpha–beta needs in order to prune a node and its subtree.)

At first sight, it might seem impossible because the value of C is the *average* of its children's values, and in order to compute the average of a set of numbers, we must look at all the numbers. But if we put bounds on the possible values of the utility function, then we can 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.

5.6 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

5.6.1 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 opening move. Keeping track of the belief state as the game progresses is exactly the problem of **state estimation**, for which the update step is given in Equation (4.6) on page 132. We can map Kriegspiel state estimation directly onto the partially observable, nondeterministic framework of Section 4.4 if we consider the opponent as the source of nondeterminism; that is, the RESULTS of White's move are composed from the (predictable) outcome of White's own move and the unpredictable outcome given by Black's reply.[4]

Given a current belief state, White may ask, "Can I win the game?" For a partially observable game, the notion of a **strategy** is altered; instead of specifying a move to make for each possible *move* the opponent might make, we need a move for every possible *percept sequence* that might be received.

Guaranteed checkmate

For Kriegspiel, a winning strategy, or **guaranteed checkmate**, is one that, for each possible percept sequence, leads to an actual checkmate for every possible board state in the current belief state, regardless of how the opponent moves. With this definition, the opponent's belief state is irrelevant—the strategy has to work even if the opponent can see all the pieces. This greatly simplifies the computation. Figure 5.15 shows part of a guaranteed checkmate for the KRK (king and rook versus king) endgame. In this case, Black has just one piece (the king), so a belief state for White can be shown in a single board by marking each possible position of the Black king.

The general AND-OR search algorithm can be applied to the belief-state space to find guaranteed checkmates, just as in Section 4.4. The incremental belief-state algorithm mentioned in Section 4.4.2 often finds midgame checkmates up to depth 9—well beyond the abilities of most human players.

Probabilistic checkmate

In addition to guaranteed checkmates, Kriegspiel admits an entirely new concept that makes no sense in fully observable games: **probabilistic checkmate**. Such checkmates are still required to work in every board state in the belief state; they are probabilistic with respect to randomization of the winning player's moves. To get the basic idea, consider the problem of finding a lone black king using just the white king. Simply by moving randomly, the white king will *eventually* bump into the black king even if the latter tries to avoid this fate, since Black cannot keep guessing the right evasive moves indefinitely. In the terminology of probability theory, detection occurs *with probability* 1.

The KBNK endgame—king, bishop and knight versus king—is won in this sense; White presents Black with an infinite random sequence of choices, for one of which Black will guess incorrectly and reveal his position, leading to checkmate. On the other hand, the KBBK endgame is won with probability $1 - \epsilon$. White can force a win only by leaving one of his bishops unprotected for one move. If Black happens to be in the right place and captures the bishop (a move that would be illegal if the bishops are protected), the game is drawn. White can choose to make the risky move at some randomly chosen point in the middle of a very long sequence, thus reducing ϵ to an arbitrarily small constant, but cannot reduce ϵ to zero.

Accidental checkmate

Sometimes a checkmate strategy works for *some* of the board states in the current belief state but not others. Trying such a strategy may succeed, leading to an **accidental checkmate**—accidental in the sense that White could not *know* that it would be checkmate—if Black's pieces happen to be in the right places. (Most checkmates in games between humans

[4] Sometimes, the belief state will become too large to represent just as a list of board states, but we will ignore this issue for now; Chapters 7 and 8 suggest methods for compactly representing very large belief states.

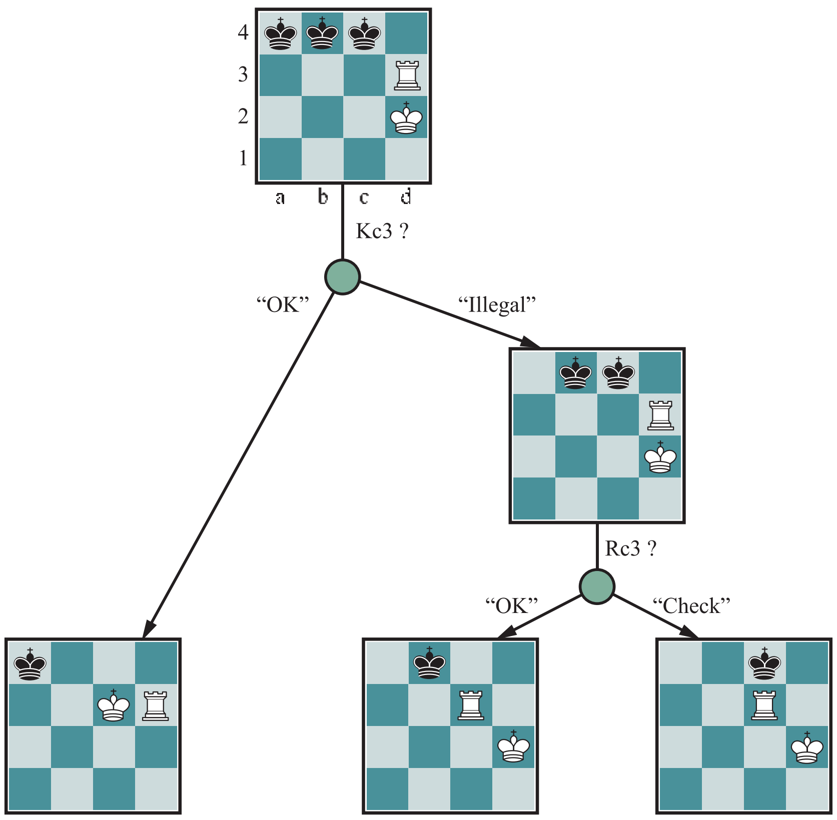


Figure 5.15 Part of a guaranteed checkmate in the KRK endgame, shown on a reduced board. In the initial belief state, Black's king is in one of three possible locations. By a combination of probing moves, the strategy narrows this down to one. Completion of the checkmate is left as an exercise.

are of this accidental nature.) This idea leads naturally to the question of *how likely* it is that a given strategy will win, which leads in turn to the question of *how likely* it is that each board state in the current belief state is the true board state.

One's first inclination might be to propose that all board states in the current belief state are equally likely—but this can't be right. Consider, for example, White's belief state after Black's first move of the game. By definition (assuming that Black plays optimally), Black must have played an optimal move, so all board states resulting from suboptimal moves ought to be assigned zero probability.

This argument is not quite right either, because *each player's goal is not just to move pieces to the right squares but also to minimize the information that the opponent has about their location.* Playing any *predictable* "optimal" strategy provides the opponent with information. Hence, optimal play in partially observable games requires a willingness to play somewhat *randomly*. (This is why restaurant hygiene inspectors do *random* inspection visits.) This means occasionally selecting moves that may seem "intrinsically" weak—but they gain strength from their very unpredictability, because the opponent is unlikely to have prepared any defense against them.

From these considerations, it seems that the probabilities associated with the board states in the current belief state can only be calculated given an optimal randomized strategy; in turn, computing that strategy seems to require knowing the probabilities of the various states the board might be in. This conundrum can be resolved by adopting the game-theoretic notion of an **equilibrium** solution, which we pursue further in Chapter 17. An equilibrium specifies an optimal randomized strategy for each player. Computing equilibria is too expensive for Kriegspiel. At present, the design of effective algorithms for general Kriegspiel play is an open research topic. Most systems perform bounded-depth look-ahead in their own belief-state space, ignoring the opponent's belief state. Evaluation functions resemble those for the observable game but include a component for the size of the belief state—smaller is better! We will return to partially observable games under the topic of Game Theory in Section 18.2.

5.6.2 Card games

Card games such as bridge, whist, hearts, and poker feature *stochastic* partial observability, where the missing information is generated by the random dealing of cards.

At first sight, it might seem that these card games are just like dice games: the cards are dealt randomly and determine the moves available to each player, but all the "dice" are rolled at the beginning! Even though this analogy turns out to be incorrect, it suggests an algorithm: treat the start of the game as a chance node with every possible deal as an outcome, and then use the EXPECTIMINIMAX formula to pick the best move. Note that in this approach the only chance node is the root node; after that the game becomes fully observable. This approach is sometimes called *averaging over clairvoyance* because it assumes that once the actual deal has occurred, the game becomes fully observable to both players. Despite its intuitive appeal, the strategy can lead one astray. Consider the following story:

> Day 1: Road A leads to a pot of gold; Road B leads to a fork. You can see that the left fork leads to two pots of gold, and the right fork leads to you being run over by a bus.
>
> Day 2: Road A leads to a pot of gold; Road B leads to a fork. You can see that the right fork leads to two pots of gold, and the left fork leads to you being run over by a bus.
>
> Day 3: Road A leads to a pot of gold; Road B leads to a fork. You are told that one fork leads to two pots of gold, and one fork leads to you being run over by a bus. Unfortunately you don't know which fork is which.

Averaging over clairvoyance leads to the following reasoning: on Day 1, B is the right choice; on Day 2, B is the right choice; on Day 3, the situation is the same as either Day 1 or Day 2, so B must still be the right choice.

Now we can see how averaging over clairvoyance fails: it does not consider the *belief state* that the agent will be in after acting. A belief state of total ignorance is not desirable, especially when one possibility is certain death. Because it assumes that every future state will automatically be one of perfect knowledge, the clairvoyance approach never selects actions that *gather information* (like the first move in Figure 5.15); nor will it choose actions that hide information from the opponent or provide information to a partner, because it assumes that they already know the information; and it will never **bluff** in poker,[5] because it assumes the opponent can see its cards. In Chapter 17, we show how to construct algorithms that do

Bluff

[5] Bluffing—betting as if one's hand is good, even when it's not—is a core part of poker strategy.

all these things by virtue of solving the true partially observable decision problem, resulting in an optimal equilibrium strategy (see Section 18.2).

Despite the drawbacks, averaging over clairvoyance can be an effective strategy, with some tricks to make it work better. In most card games, the number of possible deals is rather large. For example, in bridge play, each player sees just two of the four hands; there are two unseen hands of 13 cards each, so the number of deals is $\binom{26}{13} =$ 10,400,600. Solving even one deal is quite difficult, so solving ten million is out of the question. One way to deal with this huge number is with **abstraction**: i.e. by treating similar hands as identical. For example, it is very important which aces and kings are in a hand, but whether the hand has a 4 or 5 is not as important, and can be abstracted away.

Another way to deal with the large number is forward pruning: consider only a small random sample of N deals, and again calculate the EXPECTIMINIMAX score. Even for fairly small N—say, 100 to 1,000—this method gives a good approximation. It can also be applied to deterministic games such as Kriegspiel, where we sample over possible states of the game rather than over possible deals, as long as we have some way to estimate how likely each state is. It can also be helpful to do heuristic search with a depth cutoff rather than to search the entire game tree.

So far we have assumed that each deal is equally likely. That makes sense for games like whist and hearts. But for bridge, play is preceded by a bidding phase in which each team indicates how many tricks it expects to win. Since players bid based on the cards they hold, the other players learn something about the probability $P(s)$ of each deal. Taking this into account in deciding how to play the hand is tricky, for the reasons mentioned in our description of Kriegspiel: players may bid in such a way as to minimize the information conveyed to their opponents.

Computers have reached a superhuman level of performance in poker. The poker program Libratus took on four of the top poker players in the world in a 20-day match of no-limit Texas hold 'em and decisively beat them all. Since there are so many possible states in poker, Libratus uses abstraction to reduce the state space: it might consider the two hands AAA72 and AAA64 to be equivalent (they're both "three aces and some low cards"), and it might consider a bet of 200 dollars to be the same as 201 dollars. But Libratus also monitors the other players, and if it detects they are exploiting an abstraction, it will do some additional computation overnight to plug that hole. Overall it used 25 million CPU hours on a supercomputer to pull off the win.

The computational costs incurred by Libratus (and similar costs by ALPHAZERO and other systems) suggests that world champion game play may not be achievable for researchers with limited budgets. To some extent that is true: just as you should not expect to be able to assemble a champion Formula One race car out of spare parts in your garage, there is an advantage to having access to supercomputers or specialty hardware such as Tensor Processing Units. That is particularly true when training a system, but training could also be done via crowdsourcing. For example the open-source LEELAZERO system is a reimplementation of ALPHAZERO that trains through self-play on the computers of volunteer participants. Once trained, the computational requirements for actual tournament play are modest. ALPHASTAR won StarCraft II games running on a commodity desktop with a single GPU, and ALPHAZERO could have been run in that mode.

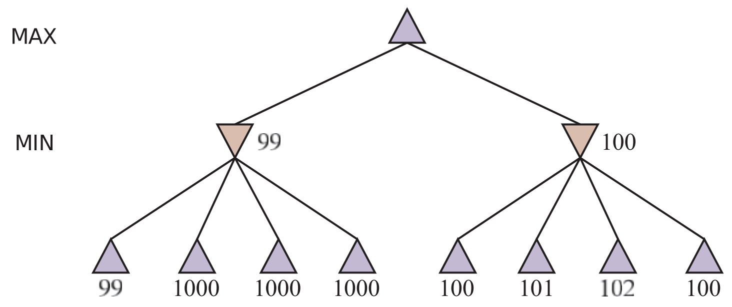


Figure 5.16 A two-ply game tree for which heuristic minimax may make an error.

5.7 Limitations of Game Search Algorithms

Because calculating optimal decisions in complex games is intractable, all algorithms must make some assumptions and approximations. Alpha–beta search uses the heuristic evaluation function as an approximation, and Monte Carlo search computes an approximate average over a random selection of playouts. The choice of which algorithm to use depends in part on the features of each game: when the branching factor is high or it is difficult to define an evaluation function, Monte Carlo search is preferred. But both algorithms suffer from fundamental limitations.

One limitation of alpha–beta search is its vulnerability to errors in the heuristic function. Figure 5.16 shows a two-ply game tree for which minimax suggests taking the right-hand branch because $100 > 99$. That is the correct move if the evaluations are all exactly accurate. But suppose that the evaluation of each node has an error that is independent of other nodes and is randomly distributed with a standard deviation of σ. Then the left-hand branch is actually better 71% of the time when $\sigma = 5$, and 58% of the time when $\sigma = 2$ (because one of the four right-hand leaves is likely to slip below 99 in these cases). If errors in the evaluation function are *not* independent, then the chance of a mistake rises. It is difficult to compensate for this because we don't have a good model of the dependencies between the values of sibling nodes.

A second limitation of both alpha–beta and Monte Carlo is that they are designed to calculate (bounds on) the values of legal moves. But sometimes there is one move that is obviously best (for example when there is only one legal move), and in that case, there is no point wasting computation time to figure out the value of the move—it is better to just make the move. A better search algorithm would use the idea of the *utility of a node expansion*, selecting node expansions of high utility—that is, ones that are likely to lead to the discovery of a significantly better move. If there are no node expansions whose utility is higher than their cost (in terms of time), then the algorithm should stop searching and make a move. This works not only for clear-favorite situations but also for the case of *symmetrical* moves, for which no amount of search will show that one move is better than another.

This kind of reasoning about what computations to do is called **metareasoning** (reasoning about reasoning). It applies not just to game playing but to any kind of reasoning at all. All computations are done in the service of trying to reach better decisions, all have costs, and all have some likelihood of resulting in a certain improvement in decision quality. Monte

Metareasoning

Carlo search does attempt to do metareasoning to allocate resources to the most important parts of the tree, but does not do so in an optimal way.

A third limitation is that both alpha-beta and Monte Carlo do all their reasoning at the level of individual moves. Clearly, humans play games differently: they can reason at a more abstract level, considering a higher-level goal—for example, trapping the opponent's queen—and using the goal to *selectively* generate plausible plans. In Chapter 11 we will study this type of **planning**, and in Section 11.4 we will show how to plan with a hierarchy of abstract to concrete representations.

A fourth issue is the ability to incorporate **machine learning** into the game search process. Early game programs relied on human expertise to hand-craft evaluation functions, opening books, search strategies, and efficiency tricks. We are just beginning to see programs like ALPHAZERO (Silver *et al.*, 2018), which relied on machine learning from self-play rather than game-specific human-generated expertise. We cover machine learning in depth starting with Chapter 19.

Summary

We have looked at a variety of games to understand what optimal play means, to understand how to play well in practice, and to get a feel for how an agent should act in any type of adversarial environment. The most important ideas are as follows:

- A game can be defined by the **initial state** (how the board is set up), the legal **actions** in each state, the **result** of each action, a **terminal test** (which says when the game is over), and a **utility function** that applies to terminal states to say who won and what the final score is.
- In two-player, discrete, deterministic, turn-taking zero-sum games with **perfect information**, the **minimax** algorithm can select optimal moves by a depth-first enumeration of the game tree.
- The **alpha–beta** search algorithm computes the same optimal move as minimax, but achieves much greater efficiency by eliminating subtrees that are provably irrelevant.
- Usually, it is not feasible to consider the whole game tree (even with alpha–beta), so we need to cut the search off at some point and apply a heuristic **evaluation function** that estimates the utility of a state.
- An alternative called **Monte Carlo tree search** (MCTS) evaluates states not by applying a heuristic function, but by playing out the game all the way to the end and using the rules of the game to see who won. Since the moves chosen during the **playout** may not have been optimal moves, the process is repeated multiple times and the evaluation is an average of the results.
- Many game programs precompute tables of best moves in the opening and endgame so that they can look up a move rather than search.
- Games of chance can be handled by **expectiminimax**, an extension to the minimax algorithm that evaluates a **chance node** by taking the average utility of all its children, weighted by the probability of each child.
- In games of **imperfect information**, such as Kriegspiel and poker, optimal play requires reasoning about the current and future **belief states** of each player. A simple

approximation can be obtained by averaging the value of an action over each possible configuration of missing information.

- Programs have soundly defeated champion human players at chess, checkers, Othello, Go, poker, and many other games. Humans retain the edge in a few games of imperfect information, such as bridge and Kriegspiel. In video games such as StarCraft and Dota 2, programs are competitive with human experts, but part of their success may be due to their ability to perform many actions very quickly.

Bibliographical and Historical Notes

In 1846, Charles Babbage discussed the feasibility of computer chess and checkers (Morrison and Morrison, 1961). He did not understand the exponential complexity of search trees, claiming "the combinations involved in the Analytical Engine enormously surpassed any required, even by the game of chess." Babbage also designed, but did not build, a special-purpose machine for playing tic-tac-toe. The first game-playing machine was built around 1890 by the Spanish engineer Leonardo Torres y Quevedo. It specialized in the "KRK" (king and rook versus king) chess endgame, guaranteeing a win when the side with the rook has the move. The **minimax** algorithm is traced to a 1912 paper by Ernst Zermelo, the developer of modern set theory.

Game playing was one of the first tasks undertaken in AI, with early efforts by such pioneers as Konrad Zuse (1945), Norbert Wiener in his book *Cybernetics* (1948), and Alan Turing (1953). But it was Claude Shannon's article *Programming a Computer for Playing Chess* (1950) that laid out all the major ideas: a representation for board positions, an evaluation function, quiescence search, and some ideas for selective game-tree search. Slater (1950) had the idea of an evaluation function as a linear combination of features, and stressed the mobility feature in chess.

John McCarthy conceived the idea of **alpha–beta** search in 1956, although the idea did not appear in print until later (Hart and Edwards, 1961). Knuth and Moore (1975) proved the correctness of alpha–beta and analysed its time complexity, while Pearl (1982b) showed alpha–beta to be asymptotically optimal among all fixed-depth game-tree search algorithms.

Berliner (1979) introduced B*, a heuristic search algorithm that maintains interval bounds on the possible value of a node in the game tree rather than giving it a single point-valued estimate. David McAllester's (1988) conspiracy number search expands leaf nodes that, by changing their values, could cause the program to prefer a new move at the root of the tree. MGSS* (Russell and Wefald, 1989) uses the decision-theoretic techniques of Chapter 16 to estimate the value of expanding each leaf in terms of the expected improvement in decision quality at the root.

The SSS* algorithm (Stockman, 1979) can be viewed as a two-player A* that never expands more nodes than alpha–beta. The memory requirements make it impractical, but a linear-space version has been developed from the RBFS algorithm (Korf and Chickering, 1996). Baum and Smith (1997) propose a probability-based replacement for minimax, showing that it results in better choices in certain games. The **expectiminimax** algorithm was proposed by Donald Michie (1966). Bruce Ballard (1983) extended alpha–beta pruning to cover trees with chance nodes.

Pearl's book *Heuristics* (1984) thoroughly analyzes many game-playing algorithms.

Monte Carlo simulation was pioneered by Metropolis and Ulam (1949) for calculations related to the development of the atomic bomb. Monte Carlo tree search (MCTS) was introduced by Abramson (1987). Tesauro and Galperin (1997) showed how a Monte Carlo search could be combined with an evaluation function for the game of backgammon. Early playout termination is studied by Lorentz (2015). ALPHAGO terminated playouts and applied an evaluation function (Silver *et al.*, 2016). Kocsis and Szepesvari (2006) refined the approach with the "Upper Confidence Bounds applied to Trees" selection mechanism. Chaslot *et al.* (2008) show how MCTS can be applied to a variety of games and Browne *et al.* (2012) give a survey.

Koller and Pfeffer (1997) describe a system for completely solving **partially observable** games. It handles larger games than previous systems, but not the full version of complex games like poker and bridge. Frank *et al.* (1998) describe several variants of Monte Carlo search for partially observable games, including one where MIN has complete information but MAX does not. Schofield and Thielscher (2015) adapt a general game-playing system for partially observable games.

Ferguson hand-derived randomized strategies for winning Kriegspiel with a bishop and knight (1992) or two bishops (1995) against a king. The first Kriegspiel programs concentrated on finding endgame checkmates and performed AND–OR search in belief-state space (Sakuta and Iida, 2002; Bolognesi and Ciancarini, 2003). Incremental belief-state algorithms enabled much more complex midgame checkmates to be found (Russell and Wolfe, 2005; Wolfe and Russell, 2007), but efficient state estimation remains the primary obstacle to effective general play (Parker *et al.*, 2005). Ciancarini and Favini (2010) apply MCTS to Kriegspiel, and Wang *et al.* (2018b) describe a belief-state version of MCTS for Phantom Go.

Chess milestones have been marked by successive winners of the Fredkin Prize: BELLE (Condon and Thompson, 1982), the first program to achieve master status; DEEP THOUGHT (Hsu *et al.*, 1990), the first to reach international master status; and Deep Blue (Campbell *et al.*, 2002; Hsu, 2004), which defeated world champion Garry Kasparov in a 1997 exhibition match. Deep Blue ran alpha–beta search at over 100 million positions per second, and could generate singular extensions to occasionally reach a depth of 40 ply.

The top chess programs today (e.g., STOCKFISH, KOMODO, HOUDINI) far exceed any human player. These programs have reduced the effective branching factor to less than 3 (compared with the actual branching factor of about 35), searching to about 20 ply at a speed of about a million nodes per second on a standard 1-core computer. They use pruning techniques such as the **null move** heuristic, which generates a good lower bound on the value of a position, using a shallow search in which the opponent gets to move twice at the beginning. Also important is **futility pruning**, which helps decide in advance which moves will cause a beta cutoff in the successor nodes. SUNFISH is a simplified chess program for teaching purposes; the core is less than 200 lines of Python.

Null move

Futility pruning

The idea of retrograde analysis for computing endgame tables is due to Bellman (1965). Using this idea, Ken Thompson (1986, 1996) and Lewis Stiller (1992, 1996) solved all chess endgames with up to five pieces. Stiller discovered one case where a forced mate existed but required 262 moves; this caused some consternation because the rules of chess require a capture or pawn move to occur within 50 moves, or else a draw is declared. In 2012 Vladimir Makhnychev and Victor Zakharov compiled the Lomonosov Endgame Tablebase, which solved all endgame positions with up to seven pieces—some require over 500 moves without a capture. The 7-piece table consumes 140 terabytes; an 8-piece table would be 100 times larger.

In 2017, ALPHAZERO (Silver *et al.*, 2018) defeated STOCKFISH (the 2017 TCEC computer chess champion) in a 1000-game trial, with 155 wins and 6 losses. Additional matches also resulted in decisive wins for ALPHAZERO, even when it was given only 1/10th the time allotted to STOCKFISH.

Grandmaster Larry Kaufman was surprised at the sucess of this Monte Carlo program and noted, “It may well be that the current dominance of minimax chess engines may be at an end, but it’s too soon to say so.” Garry Kasparov commented “It’s a remarkable achievement, even if we should have expected it after ALPHAGO. It approaches the Type B human-like approach to machine chess dreamt of by Claude Shannon and Alan Turing instead of brute force.” He went on to predict “Chess has been shaken to its roots by ALPHAZERO, but this is only a tiny example of what is to come. Hidebound disciplines like education and medicine will also be shaken” (Sadler and Regan, 2019).

Checkers was the first of the classic games played by a computer (Strachey, 1952). Arthur Samuel (1959, 1967) developed a checkers program that learned its own evaluation function through self-play using a form of reinforcement learning. It is quite an achievement that Samuel was able to create a program that played better than he did, on an IBM 704 computer with only 10,000 words of memory and a 0.000001 GHz processor. MENACE—the Machine Educable Noughts And Crosses Engine (Michie, 1963)—also used reinforcement learning to become competent at tic-tac-toe. Its processor was even slower: a collection of 304 matchboxes holding colored beads to represent the best learned move in each position.

In 1992, Jonathan Schaeffer’s CHINOOK checkers program challenged the legendary Marion Tinsley, who had been world champion for over 20 years. Tinsley won the match, but lost two games—the fourth and fifth losses in his entire career. After Tinsley retired for health reasons, CHINOOK took the crown. The saga was chronicled by Schaeffer (2008).

In 2007 Schaeffer and his team “solved” checkers (Schaeffer *et al.*, 2007): the game is a draw with perfect play. Richard Bellman (1965) had predicted this: “In checkers, the number of possible moves in any given situation is so small that we can confidently expect a complete digital computer solution to the problem of optimal play in this game.” Bellman did not anticipate the scale of the effort: the endgame table for 10 pieces has 39 trillion entries. Given this table, it took 18 CPU-years of alpha–beta search to solve the game.

I. J. Good, who was taught the Game of **Go** by Alan Turing, wrote (1965a) “ I think it will be even more difficult to programme a computer to play a reasonable game of Go than of chess.” He was right: through 2015, Go programs played only at an amateur level. The early literature is summarized by Bouzy and Cazenave (2001) and Müller (2002).

Visual pattern recognition was proposed as a promising technique for Go by Zobrist (1970), while Schraudolph *et al.* (1994) analyzed the use of reinforcement learning, Lubberts and Miikkulainen (2001) recommended neural networks, and Brügmann (1993) introduced Monte Carlo tree search to Go. ALPHAGO (Silver *et al.*, 2016) put those four ideas together to defeat top-ranked professionals Lee Sedol (by a score of 4–1 in 2015) and Ke Jie (by 3–0 in 2016).

Ke Jie remarked “After humanity spent thousands of years improving our tactics, computers tell us that humans are completely wrong. I would go as far as to say not a single human has touched the edge of the truth of Go." Lee Sedol retired from Go, lamenting, "Even if I became the number one, there is an entity that cannot be defeated."

In 2018, ALPHAZERO surpassed ALPHAGO at Go, and also defeated top programs in chess and shogi, learning through self-play without any expert human knowledge and without access to any past games. (It does, of course, rely on humans to define the basic architecture as Monte Carlo tree search with deep neural networks and reinforcement learning, and to encode the rules of the game.) The success of ALPHAZERO has led to increased interest in reinforcement learning as a key component of general AI (see Chapter 22). Going one step further, the MUZERO system operates without even being told the rules of the game it is playing—it has to figure out the rules by making plays. MUZERO achieved state-of-the-art results in Pacman, chess, Go, and 75 Atari games (Schrittwieser *et al.*, 2019). It learns to generalize; for example, it learns that in Pacman the "up" action moves the player up a square (unless there is a wall there), even though it has only observed the result of the "up" action in a small percentage of the locations on the board.

Othello, also called Reversi, has a smaller search space than chess, but defining an evaluation function is difficult, because material advantage is not as important as mobility. Programs have been at superhuman level since 1997 (Buro, 2002).

Backgammon, a game of chance, was analyzed mathematically by Gerolamo Cardano (1663), and taken up for computer play with the BKG program (Berliner, 1980b), which used a manually constructed evaluation function and searched only to depth 1. It was the first program to defeat a human world champion at a major game (Berliner, 1980a), although Berliner readily acknowledged that BKG was very lucky with the dice. Gerry Tesauro's (1995) TD-GAMMON learned its evaluation function using neural networks trained by self-play. It consistently played at world champion level and caused human analysts to change their opinion on the best opening move for several dice rolls.

Poker, like Go, has seen surprising advances in recent years. Bowling *et al.* (2015) used game theory (see Section 18.2) to determine the exact optimal strategy for a version of poker with just two players and a fixed number of raises with fixed bet sizes. In 2017, for the first time, champion poker players were beaten at heads-up (two player) no-limit Texas hold 'em in two separate matches against the programs Libratus (Brown and Sandholm, 2017) and DeepStack (Moravčík *et al.*, 2017). In 2019, Pluribus (Brown and Sandholm, 2019) defeated top-ranked professional human players in Texas hold 'em games with six players. Multiplayer games introduce some strategic concerns that we will cover in Chapter 18. Petosa and Balch (2019) implement a multiplayer version of ALPHAZERO.

Bridge: Smith *et al.* (1998) report on how BRIDGE BARON won the 1998 computer bridge championship, using hierarchical plans (see Chapter 11) and high-level actions, such as finessing and squeezing, that are familiar to bridge players. Ginsberg (2001) describes how his GIB program, based on Monte Carlo simulation (first proposed for bridge by Levy (1989)), won the following computer championship and did surprisingly well against expert human players. In the 21st century, the computer bridge championship has been dominated by two commercial programs, JACK and WBRIDGE5. Neither has been described in published articles, but both are believed to use Monte Carlo techniques. In general, bridge programs are at human champion level when actually playing the hands, but lag behind in the bidding phase, because they do not completely understand the conventions used by humans to communicate with their partners. Bridge programmers have concentrated more on producing useful and educational programs that encourage people to take up the game, rather than on defeating human champions.

Scrabble is a game where amateur human players have difficulty coming up with high-scoring words, but for a computer, it is easy to find the highest possible score for a given hand (Gordon, 1994); the hard part is planning ahead in a partially observable, stochastic game. Nevertheless, in 2006, the QUACKLE program defeated the former world champion, David Boys, 3–2. Boys took it well, stating, "It's still better to be a human than to be a computer." A good description of a top program, MAVEN, is given by Sheppard (2002).

Video games such as **StarCraft II** involve hundreds of partially observable units moving in real time with high-dimensional near-continuous[6] observation and action spaces with complex rules. Oriol Vinyals, who was Spain's StarCraft champion at age 15, described how the game can serve as a testbed and grand challenge for reinforcement learning (Vinyals *et al.*, 2017a). In 2019, Vinyals and the team at DeepMind unveiled the ALPHASTAR program, based on deep learning and reinforcement learning, which defeated expert human players 10 games to 1, and ranks in the top 0.02% of officially ranked human players (Vinyals *et al.*, 2019). ALPHASTAR took steps to limit the number of actions per minute it could perform in critical bursts, in response to critics who felt it had an unfair advantage.

Computers have defeated top humans in other popular video games such as Super Smash Bros. (Firoiu *et al.*, 2017), Quake III (Jaderberg *et al.*, 2019), and Dota 2 (Fernandez and Mahlmann, 2018), all using deep learning techniques.

Physical games such as **robotic soccer** (Visser *et al.*, 2008; Barrett and Stone, 2015), **billiards** (Lam and Greenspan, 2008; Archibald *et al.*, 2009), and **ping-pong** (Silva *et al.*, 2015) have attracted some attention in AI. They combine all the complications of video games with the messiness of the real world.

Computer game competitions occur annually, including the Computer Olympiads since 1989. The General Game Competition (Love *et al.*, 2006) tests programs that must learn to play an unknown game given only a logical description of the rules of the game. The International Computer Games Association (ICGA) publishes the *ICGA Journal* and runs two alternating biennial conferences, The International Conference on Computers and Games (ICCG or CG) and the International Conference on Advances in Computer Games (ACG). The IEEE publishes *IEEE Transactions on Games* and runs an annual Conference on Computational Intelligence and Games.

6 To a human player, it appears that objects move continuously, but they are actually discrete at the level of a pixel on the screen.

CHAPTER 6

CONSTRAINT SATISFACTION PROBLEMS

In which we see how treating states as more than just little black boxes leads to new search methods and a deeper understanding of problem structure.

Chapters 3 and 4 explored the idea that problems can be solved by searching the state space: a graph where the nodes are states and the edges between them are actions. We saw that domain-specific heuristics could estimate the cost of reaching the goal from a given state, but that from the point of view of the search algorithm, each state is atomic, or indivisible—a black box with no internal structure. For each problem we need domain-specific code to describe the transitions between states.

In this chapter we break open the black box by using a **factored representation** for each state: a set of **variables**, each of which has a **value**. A problem is solved when each variable has a value that satisfies all the constraints on the variable. A problem described this way is called a **constraint satisfaction problem**, or **CSP**.

Constraint satisfaction problem

CSP search algorithms take advantage of the structure of states and use *general* rather than domain-specific heuristics to enable the solution of complex problems. The main idea is to eliminate large portions of the search space all at once by identifying variable/value combinations that violate the constraints. CSPs have the additional advantage that the actions and transition model can be deduced from the problem description.

6.1 Defining Constraint Satisfaction Problems

A constraint satisfaction problem consists of three components, $\mathcal{X}, \mathcal{D}$, and $\mathcal{C}$:

$\mathcal{X}$ is a set of variables, $\{X_1, \ldots, X_n\}$.

$\mathcal{D}$ is a set of domains, $\{D_1, \ldots, D_n\}$, one for each variable.

$\mathcal{C}$ is a set of constraints that specify allowable combinations of values.

A domain, D_i, consists of a set of allowable values, $\{v_1, \ldots, v_k\}$, for variable X_i. For example, a Boolean variable would have the domain $\{true, false\}$. Different variables can have different domains of different sizes. Each constraint $\mathcal{C}_j$ consists of a pair $\langle scope, rel \rangle$, where *scope* is a tuple of variables that participate in the constraint and *rel* is a **relation** that defines the values that those variables can take on. A relation can be represented as an explicit set of all tuples of values that satisfy the constraint, or as a function that can compute whether a tuple is a member of the relation. For example, if X_1 and X_2 both have the domain $\{1,2,3\}$, then the constraint saying that X_1 must be greater than X_2 can be written as $\langle (X_1, X_2), \{(3,1),(3,2),(2,1)\} \rangle$ or as $\langle (X_1, X_2), X_1 > X_2 \rangle$.

Relation

CSPs deal with **assignments** of values to variables, $\{X_i = v_i, X_j = v_j, \ldots\}$. An assignment that does not violate any constraints is called a **consistent** or legal assignment. A **complete assignment** is one in which every variable is assigned a value, and a **solution** to a CSP is a consistent, complete assignment. A **partial assignment** is one that leaves some variables unassigned, and a **partial solution** is a partial assignment that is consistent. Solving a CSP is an NP-complete problem in general, although there are important subclasses of CSPs that can be solved very efficiently.

Assignments

Consistent

Complete assignment

Solution

Partial assignment

Partial solution

6.1.1 Example problem: Map coloring

Suppose that, having tired of Romania, we are looking at a map of Australia showing each of its states and territories (Figure 6.1(a)). We are given the task of coloring each region either red, green, or blue in such a way that no two neighboring regions have the same color. To formulate this as a CSP, we define the variables to be the regions:

$$\mathcal{X} = \{WA, NT, Q, NSW, V, SA, T\}.$$

The domain of every variable is the set $D_i = \{red, green, blue\}$. The constraints require neighboring regions to have distinct colors. Since there are nine places where regions border, there are nine constraints:

$$\begin{aligned}\mathcal{C} \;=\; &\{SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V,\\ &WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}.\end{aligned}$$

Here we are using abbreviations; $SA \neq WA$ is a shortcut for $\langle (SA, WA), SA \neq WA \rangle$, where $SA \neq WA$ can be fully enumerated in turn as

$$\{(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)\}.$$

There are many possible solutions to this problem, such as

$$\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = red\}.$$

It can be helpful to visualize a CSP as a **constraint graph**, as shown in Figure 6.1(b). The nodes of the graph correspond to variables of the problem, and an edge connects any two variables that participate in a constraint.

Constraint graph

Why formulate a problem as a CSP? One reason is that the CSPs yield a natural representation for a wide variety of problems; it is often easy to formulate a problem as a CSP. Another is that years of development work have gone into making CSP solvers fast and efficient. A third is that a CSP solver can quickly prune large swathes of the search space that an atomic state-space searcher cannot. For example, once we have chosen $\{SA = blue\}$ in the Australia problem, we can conclude that none of the five neighboring variables can take on the value *blue*. A search procedure that does not use constraints would have to consider $3^5 = 243$ assignments for the five neighboring variables; with constraints we have only $2^5 = 32$ assignments to consider, a reduction of 87%.

In atomic state-space search we can only ask: is this specific state a goal? No? What about this one? With CSPs, once we find out that a partial assignment violates a constraint, we can immediately discard further refinements of the partial assignment. Furthermore, we can see *why* the assignment is not a solution—we see which variables violate a constraint—so we can focus attention on the variables that matter. As a result, many problems that are intractable for atomic state-space search can be solved quickly when formulated as a CSP.

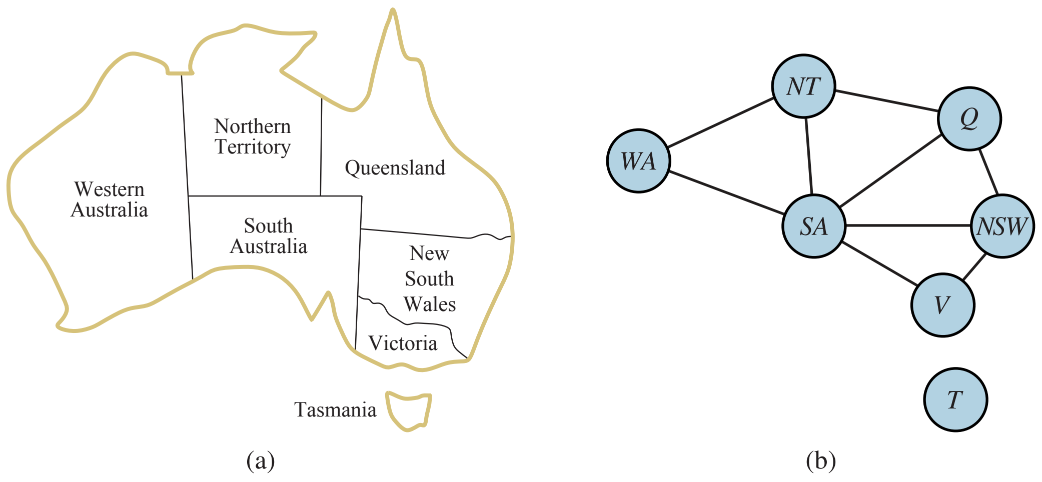


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

6.1.2 Example problem: Job-shop scheduling

Factories have the problem of scheduling a day's worth of jobs, subject to various constraints. In practice, many of these problems are solved with CSP techniques. Consider the problem of scheduling the assembly of a car. The whole job is composed of tasks, and we can model each task as a variable, where the value of each variable is the time that the task starts, expressed as an integer number of minutes. Constraints can assert that one task must occur before another—for example, a wheel must be installed before the hubcap is put on—and that only so many tasks can go on at once. Constraints can also specify that a task takes a certain amount of time to complete.

We consider a small part of the car assembly, consisting of 15 tasks: install axles (front and back), affix all four wheels (right and left, front and back), tighten nuts for each wheel, affix hubcaps, and inspect the final assembly. We can represent the tasks with 15 variables:

$$\begin{aligned}\mathcal{X} \;=\; &\{Axle_F, Axle_B, Wheel_{RF}, Wheel_{LF}, Wheel_{RB}, Wheel_{LB}, Nuts_{RF},\\ &Nuts_{LF}, Nuts_{RB}, Nuts_{LB}, Cap_{RF}, Cap_{LF}, Cap_{RB}, Cap_{LB}, Inspect\}\,.\end{aligned}$$

Precedence constraint

Next, we represent **precedence constraints** between individual tasks. Whenever a task T_1 must occur before task T_2, and task T_1 takes duration d_1 to complete, we add an arithmetic constraint of the form

$$T_1 + d_1 \leq T_2\,.$$

In our example, the axles have to be in place before the wheels are put on, and it takes 10 minutes to install an axle, so we write

$$\begin{aligned}&Axle_F + 10 \leq Wheel_{RF}; \quad Axle_F + 10 \leq Wheel_{LF};\\ &Axle_B + 10 \leq Wheel_{RB}; \quad Axle_B + 10 \leq Wheel_{LB}\,.\end{aligned}$$

Next we say that for each wheel, we must affix the wheel (which takes 1 minute), then tighten the nuts (2 minutes), and finally attach the hubcap (1 minute, but not represented yet):

$$\begin{aligned}
&Wheel_{RF}+1 \leq Nuts_{RF};\; Nuts_{RF}+2 \leq Cap_{RF};\\
&Wheel_{LF}+1 \leq Nuts_{LF};\; Nuts_{LF}+2 \leq Cap_{LF};\\
&Wheel_{RB}+1 \leq Nuts_{RB};\; Nuts_{RB}+2 \leq Cap_{RB};\\
&Wheel_{LB}+1 \leq Nuts_{LB};\; Nuts_{LB}+2 \leq Cap_{LB}.
\end{aligned}$$

Suppose we have four workers to install wheels, but they have to share one tool that helps put the axle in place. We need a **disjunctive constraint** to say that $Axle_F$ and $Axle_B$ must not overlap in time; either one comes first or the other does:

Disjunctive constraint

$$(Axle_F + 10 \leq Axle_B) \quad \textbf{or} \quad (Axle_B + 10 \leq Axle_F).$$

This looks like a more complicated constraint, combining arithmetic and logic. But it still reduces to a set of pairs of values that $Axle_F$ and $Axle_B$ can take on.

We also need to assert that the inspection comes last and takes 3 minutes. For every variable except *Inspect* we add a constraint of the form $X + d_X \leq Inspect$. Finally, suppose there is a requirement to get the whole assembly done in 30 minutes. We can achieve that by limiting the domain of all variables:

$$D_i = \{0,1,2,3,\ldots,30\}.$$

This particular problem is trivial to solve, but CSPs have been successfully applied to job-shop scheduling problems like this with thousands of variables.

6.1.3 Variations on the CSP formalism

The simplest kind of CSP involves variables that have **discrete**, **finite domains**. Map-coloring problems and scheduling with time limits are both of this kind. The 8-queens problem (Figure 4.3) can also be viewed as a finite-domain CSP, where the variables $Q_1,\ldots,Q_8$ correspond to the queens in columns 1 to 8, and the domain of each variable specifies the possible row numbers for the queen in that column, $D_i = \{1,2,3,4,5,6,7,8\}$. The constraints say that no two queens can be in the same row or diagonal.

Discrete domain

Finite domain

A discrete domain can be **infinite**, such as the set of integers or strings. (If we didn't put a deadline on the job-scheduling problem, there would be an infinite number of start times for each variable.) With infinite domains, we must use implicit constraints like $T_1 + d_1 \leq T_2$ rather than explicit tuples of values. Special solution algorithms (which we do not discuss here) exist for **linear constraints** on integer variables—that is, constraints, such as the one just given, in which each variable appears only in linear form. It can be shown that no algorithm exists for solving general **nonlinear constraints** on integer variables—the problem is undecidable.

Infinite

Linear constraints

Nonlinear constraints

Constraint satisfaction problems with **continuous domains** are common in the real world and are widely studied in the field of operations research. For example, the scheduling of experiments on the Hubble Space Telescope requires very precise timing of observations; the start and finish of each observation and maneuver are continuous-valued variables that must obey a variety of astronomical, precedence, and power constraints. The best-known category of continuous-domain CSPs is that of **linear programming** problems, where constraints must be linear equalities or inequalities. Linear programming problems can be solved in time polynomial in the number of variables. Problems with different types of constraints and objective functions have also been studied—quadratic programming, second-order conic programming, and so on. These problems constitute an important area of applied mathematics.

Continuous domains

Unary constraint

In addition to examining the types of variables that can appear in CSPs, it is useful to look at the types of constraints. The simplest type is the **unary constraint**, which restricts the value of a single variable. For example, in the map-coloring problem it could be the case that South Australians won't tolerate the color green; we can express that with the unary constraint $\langle (SA), SA \neq green \rangle$. (The initial specification of the domain of a variable can also be seen as a unary constraint.)

Binary constraint

Binary CSP

A **binary constraint** relates two variables. For example, $SA \neq NSW$ is a binary constraint. A **binary CSP** is one with only unary and binary constraints; it can be represented as a constraint graph, as in Figure 6.1(b).

We can also define higher-order constraints. The ternary constraint $Between(X,Y,Z)$, for example, can be defined as $\langle (X,Y,Z), X < Y < Z \textbf{ or } X > Y > Z \rangle$.

Global constraint

A constraint involving an arbitrary number of variables is called a **global constraint**. (The name is traditional but confusing because a global constraint need not involve *all* the variables in a problem). One of the most common global constraints is *Alldiff*, which says that all of the variables involved in the constraint must have different values. In Sudoku problems (see Section 6.2.6), all variables in a row, column, or 3×3 box must satisfy an *Alldiff* constraint.

Cryptarithmetic

Another example is provided by **cryptarithmetic** puzzles (Figure 6.2(a)). Each letter in a cryptarithmetic puzzle represents a different digit. For the case in Figure 6.2(a), this would be represented as the global constraint $Alldiff(F,T,U,W,R,O)$. The addition constraints on the four columns of the puzzle can be written as the following n-ary constraints:

$$\begin{aligned}
&O + O = R + 10 \cdot C_1 \\
&C_1 + W + W = U + 10 \cdot C_2 \\
&C_2 + T + T = O + 10 \cdot C_3 \\
&C_3 = F\,,
\end{aligned}$$

Constraint hypergraph

where C_1, C_2, and C_3 are auxiliary variables representing the digit carried over into the tens, hundreds, or thousands column. These constraints can be represented in a **constraint hypergraph**, such as the one shown in Figure 6.2(b). A hypergraph consists of ordinary nodes (the circles in the figure) and hypernodes (the squares), which represent n-ary constraints—constraints involving n variables.

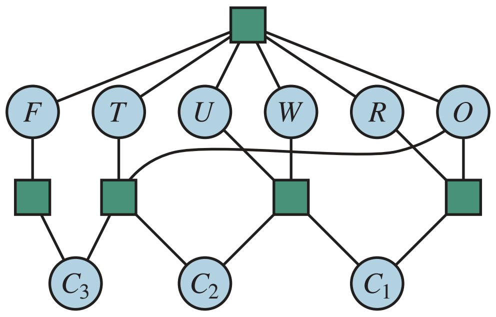
Dual graph

Alternatively, as Exercise 6.NARY asks you to prove, every finite-domain constraint can be reduced to a set of binary constraints if enough auxiliary variables are introduced. This means that we could transform any CSP into one with only binary constraints—which certainly makes the life of the algorithm designer simpler. Another way to convert an n-ary CSP to a binary one is the **dual graph** transformation: create a new graph in which there will be one variable for each constraint in the original graph, and one binary constraint for each pair of constraints in the original graph that share variables.

For example, consider a CSP with the variables $\mathcal{X} = \{X,Y,Z\}$, each with the domain $\{1,2,3,4,5\}$, and with the two constraints $C_1 : \langle (X,Y,Z), X+Y=Z \rangle$ and $C_2 : \langle (X,Y), X+1=Y \rangle$. Then the dual graph would have the variables $\mathcal{X} = \{C_1, C_2\}$, where the domain of the C_1 variable in the dual graph is the set of $\{(x_i, y_j, z_k)\}$ tuples from the C_1 constraint in the original problem, and similarly the domain of C_2 is the set of $\{(x_i, y_j)\}$ tuples. The dual graph has the binary constraint $\langle (C_1, C_2), R_1 \rangle$, where R_1 is a new relation that defines the constraint between C_1 and C_2; in this case it would be $R_1 = \{((1,2,3),(1,2)),((2,3,5),(2,3))\}$.

$$\begin{array}{cccc} & T & W & O \\ + & T & W & O \\ \hline F & O & U & R \end{array}$$

(a)



(b)

Figure 6.2 (a) A cryptarithmetic problem. Each letter stands for a distinct digit; the aim is to find a substitution of digits for letters such that the resulting sum is arithmetically correct, with the added restriction that no leading zeroes are allowed. (b) The constraint hypergraph for the cryptarithmetic problem, showing the *Alldiff* constraint (square box at the top) as well as the column addition constraints (four square boxes in the middle). The variables C_1, C_2, and C_3 represent the carry digits for the three columns from right to left.

There are however two reasons why we might prefer a global constraint such as *Alldiff* rather than a set of binary constraints. First, it is easier and less error-prone to write the problem description using *Alldiff*. Second, it is possible to design special-purpose inference algorithms for global constraints that are more efficient than operating with primitive constraints. We describe these inference algorithms in Section 6.2.5.

The constraints we have described so far have all been absolute constraints, violation of which rules out a potential solution. Many real-world CSPs include **preference constraints** indicating which solutions are preferred. For example, in a university class-scheduling problem there are absolute constraints that no professor can teach two classes at the same time. But we also may allow preference constraints: Prof. R might prefer teaching in the morning, whereas Prof. N prefers teaching in the afternoon. A schedule that has Prof. R teaching at 2 p.m. would still be an allowable solution (unless Prof. R happens to be the department chair) but would not be an optimal one.

Preference constraints

Preference constraints can often be encoded as costs on individual variable assignments—for example, assigning an afternoon slot for Prof. R costs 2 points against the overall objective function, whereas a morning slot costs 1. With this formulation, CSPs with preferences can be solved with optimization search methods, either path-based or local. We call such a problem a **constrained optimization problem**, or COP. Linear programs are one class of COPs.

Constrained optimization problem

6.2 Constraint Propagation: Inference in CSPs

An atomic state-space search algorithm makes progress in only one way: by expanding a node to visit the successors. A CSP algorithm has choices. It can generate successors by choosing a new variable assignment, or it can do a specific type of inference called **constraint propagation**: using the constraints to reduce the number of legal values for a variable, which

Constraint propagation

in turn can reduce the legal values for another variable, and so on. The idea is that this will leave fewer choices to consider when we make the next choice of a variable assignment. Constraint propagation may be intertwined with search, or it may be done as a preprocessing step, before search starts. Sometimes this preprocessing can solve the whole problem, so no search is required at all.

Local consistency

The key idea is **local consistency**. If we treat each variable as a node in a graph (see Figure 6.1(b)) and each binary constraint as an edge, then the process of enforcing local consistency in each part of the graph causes inconsistent values to be eliminated throughout the graph. There are different types of local consistency, which we now cover in turn.

6.2.1 Node consistency

Node consistency

A single variable (corresponding to a node in the CSP graph) is **node-consistent** if all the values in the variable's domain satisfy the variable's unary constraints. For example, in the variant of the Australia map-coloring problem (Figure 6.1) where South Australians dislike green, the variable SA starts with domain $\{red, green, blue\}$, and we can make it node consistent by eliminating $green$, leaving SA with the reduced domain $\{red, blue\}$. We say that a graph is node-consistent if every variable in the graph is node-consistent.

It is easy to eliminate all the unary constraints in a CSP by reducing the domain of variables with unary constraints at the start of the solving process. As mentioned earlier, it is also possible to transform all n-ary constraints into binary ones. Because of this, some CSP solvers work with only binary constraints, expecting the user to eliminate the other constraints ahead of time. We make that assumption for the rest of this chapter, except where noted.

6.2.2 Arc consistency

Arc consistency

A variable in a CSP is **arc-consistent**[1] if every value in its domain satisfies the variable's binary constraints. More formally, X_i is arc-consistent with respect to another variable X_j if for every value in the current domain D_i there is some value in the domain D_j that satisfies the binary constraint on the arc (X_i, X_j). A graph is arc-consistent if every variable is arc-consistent with every other variable. For example, consider the constraint $Y = X^2$ where the domain of both X and Y is the set of decimal digits. We can write this constraint explicitly as

$$\langle (X,Y), \{(0,0),(1,1),(2,4),(3,9)\} \rangle .$$

To make X arc-consistent with respect to Y, we reduce X's domain to $\{0,1,2,3\}$. If we also make Y arc-consistent with respect to X, then Y's domain becomes $\{0,1,4,9\}$, and the whole CSP is arc-consistent. On the other hand, arc consistency can do nothing for the Australia map-coloring problem. Consider the following inequality constraint on (SA, WA):

$$\{(red, green),(red, blue),(green, red),(green, blue),(blue, red),(blue, green)\} .$$

No matter what value you choose for SA (or for WA), there is a valid value for the other variable. So applying arc consistency has no effect on the domains of either variable.

The most popular algorithm for enforcing arc consistency is called AC-3 (see Figure 6.3). To make every variable arc-consistent, the AC-3 algorithm maintains a queue of arcs to consider. Initially, the queue contains all the arcs in the CSP. (Each binary constraint becomes two arcs, one in each direction.) AC-3 then pops off an arbitrary arc (X_i, X_j) from the queue

[1] We have been using the term "edge" rather than "arc," so it would make more sense to call this "edge-consistent," but the name "arc-consistent" is historical.

```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
  queue ← a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    $(X_i, X_j)$ ← POP(queue)
    if REVISE(csp, $X_i$, $X_j$) then
      if size of $D_i = 0$ then return false
      for each $X_k$ in $X_i$.NEIGHBORS - $\{X_j\}$ do
        add $(X_k, X_i)$ to queue
  return true

function REVISE(csp, $X_i$, $X_j$) returns true iff we revise the domain of $X_i$
  revised ← false
  for each x in $D_i$ do
    if no value y in $D_j$ allows (x,y) to satisfy the constraint between $X_i$ and $X_j$ then
      delete x from $D_i$
      revised ← true
  return revised
```

Figure 6.3 The arc-consistency algorithm AC-3. After applying AC-3, either every arc is arc-consistent, or some variable has an empty domain, indicating that the CSP cannot be solved. The name "AC-3" was used by the algorithm's inventor (Mackworth, 1977) because it was the third version developed in the paper.

and makes X_i arc-consistent with respect to X_j. If this leaves D_i unchanged, the algorithm just moves on to the next arc. But if this revises D_i (makes the domain smaller), then we add to the queue all arcs (X_k, X_i) where X_k is a neighbor of X_i. We need to do that because the change in D_i might enable further reductions in D_k, even if we have previously considered X_k. If D_i is revised down to nothing, then we know the whole CSP has no consistent solution, and AC-3 can immediately return failure. Otherwise, we keep checking, trying to remove values from the domains of variables until no more arcs are in the queue. At that point, we are left with a CSP that is equivalent to the original CSP—they both have the same solutions—but the arc-consistent CSP will be faster to search because its variables have smaller domains. In some cases, it solves the problem completely (by reducing every domain to size 1) and in others it proves that no solution exists (by reducing some domain to size 0).

The complexity of AC-3 can be analyzed as follows. Assume a CSP with n variables, each with domain size at most d, and with c binary constraints (arcs). Each arc (X_k, X_i) can be inserted in the queue only d times because X_i has at most d values to delete. Checking consistency of an arc can be done in $O(d^2)$ time, so we get $O(cd^3)$ total worst-case time.

6.2.3 Path consistency

Suppose we are to color the map of Australia with just two colors, red and blue. Arc consistency does nothing because every constraint can be satisfied individually with red at one end and blue at the other. But clearly there is no solution to the problem: because Western Australia, Northern Territory, and South Australia all touch each other, we need at least three colors for them alone.

Arc consistency tightens down the domains (unary constraints) using the arcs (binary constraints). To make progress on problems like map coloring, we need a stronger notion of consistency. **Path consistency** tightens the binary constraints by using implicit constraints that are inferred by looking at triples of variables.

Path consistency

A two-variable set $\{X_i, X_j\}$ is path-consistent with respect to a third variable X_m if, for every assignment $\{X_i = a, X_j = b\}$ consistent with the constraints (if any) on $\{X_i, X_j\}$, there is an assignment to X_m that satisfies the constraints on $\{X_i, X_m\}$ and $\{X_m, X_j\}$. The name refers to the overall consistency of the path from X_i to X_j with X_m in the middle.

Let's see how path consistency fares in coloring the Australia map with two colors. We will make the set $\{WA, SA\}$ path-consistent with respect to NT. We start by enumerating the consistent assignments to the set. In this case, there are only two: $\{WA = red, SA = blue\}$ and $\{WA = blue, SA = red\}$. We can see that with both of these assignments NT can be neither *red* nor *blue* (because it would conflict with either WA or SA). Because there is no valid choice for NT, we eliminate both assignments, and we end up with no valid assignments for $\{WA, SA\}$. Therefore, we know that there can be no solution to this problem.

6.2.4 *K*-consistency

K-consistency

Stronger forms of propagation can be defined with the notion of ***k*-consistency**. A CSP is k-consistent if, for any set of $k-1$ variables and for any consistent assignment to those variables, a consistent value can always be assigned to any kth variable. 1-consistency says that, given the empty set, we can make any set of one variable consistent: this is what we called node consistency. 2-consistency is the same as arc consistency. For binary constraint graphs, 3-consistency is the same as path consistency.

Strongly k-consistent

A CSP is **strongly *k*-consistent** if it is k-consistent and is also $(k-1)$-consistent, $(k-2)$-consistent, ... all the way down to 1-consistent. Now suppose we have a CSP with n nodes and make it strongly n-consistent (i.e., strongly k-consistent for $k=n$). We can then solve the problem as follows: First, we choose a consistent value for X_1. We are then guaranteed to be able to choose a value for X_2 because the graph is 2-consistent, for X_3 because it is 3-consistent, and so on. For each variable X_i, we need only search through the d values in the domain to find a value consistent with $X_1, \ldots, X_{i-1}$. The total run time is only $O(n^2 d)$.

Of course, there is no free lunch: constraint satisfaction is NP-complete in general, and any algorithm for establishing n-consistency must take time exponential in n in the worst case. Worse, n-consistency also requires space that is exponential in n. In practice, determining the appropriate level of consistency checking is mostly an empirical science. Computing 2-consistency is common, and 3-consistency less common.

6.2.5 Global constraints

Remember that a **global constraint** is one involving an arbitrary number of variables (but not necessarily all variables). Global constraints occur frequently in real problems and can be handled by special-purpose algorithms that are more efficient than the general-purpose methods described so far. For example, the *Alldiff* constraint says that all the variables involved must have distinct values (as in the cryptarithmetic problem above and Sudoku puzzles below). One simple form of inconsistency detection for *Alldiff* constraints works as follows: if m variables are involved in the constraint, and if they have n possible distinct values altogether, and $m > n$, then the constraint cannot be satisfied.

This leads to the following simple algorithm: First, remove any variable in the constraint that has a singleton domain, and delete that variable's value from the domains of the remaining variables. Repeat as long as there are singleton variables. If at any point an empty domain is produced or there are more variables than domain values left, then an inconsistency has been detected.

This method can detect the inconsistency in the assignment $\{WA = red, NSW = red\}$ for Figure 6.1. Notice that the variables *SA*, *NT*, and *Q* are effectively connected by an *Alldiff* constraint because each pair must have two different colors. After applying AC-3 with the partial assignment, the domains of *SA*, *NT*, and *Q* are all reduced to $\{green, blue\}$. That is, we have three variables and only two colors, so the *Alldiff* constraint is violated. Thus, a simple consistency procedure for a higher-order constraint is sometimes more effective than applying arc consistency to an equivalent set of binary constraints.

Resource constraint

Another important higher-order constraint is the **resource constraint**, sometimes called the *Atmost* constraint. For example, in a scheduling problem, let $P_1, \ldots, P_4$ denote the numbers of personnel assigned to each of four tasks. The constraint that no more than 10 personnel are assigned in total is written as $Atmost(10, P_1, P_2, P_3, P_4)$. We can detect an inconsistency simply by checking the sum of the minimum values of the current domains; for example, if each variable has the domain $\{3, 4, 5, 6\}$, the *Atmost* constraint cannot be satisfied. We can also enforce consistency by deleting the maximum value of any domain if it is not consistent with the minimum values of the other domains. Thus, if each variable in our example has the domain $\{2, 3, 4, 5, 6\}$, the values 5 and 6 can be deleted from each domain.

Bounds propagation

For large resource-limited problems with integer values—such as logistical problems involving moving thousands of people in hundreds of vehicles—it is usually not possible to represent the domain of each variable as a large set of integers and gradually reduce that set by consistency-checking methods. Instead, domains are represented by upper and lower bounds and are managed by **bounds propagation**. For example, in an airline-scheduling problem, let's suppose there are two flights, F_1 and F_2, for which the planes have capacities 165 and 385, respectively. The initial domains for the numbers of passengers on flights F_1 and F_2 are then

$$D_1 = [0, 165] \quad \text{and} \quad D_2 = [0, 385].$$

Now suppose we have the additional constraint that the two flights together must carry 420 people: $F_1 + F_2 = 420$. Propagating bounds constraints, we reduce the domains to

$$D_1 = [35, 165] \quad \text{and} \quad D_2 = [255, 385].$$

Bounds-consistent

We say that a CSP is **bounds-consistent** if for every variable X, and for both the lower-bound and upper-bound values of X, there exists some value of Y that satisfies the constraint between X and Y for every variable Y. This kind of bounds propagation is widely used in practical constraint problems.

6.2.6 Sudoku

Sudoku

The popular **Sudoku** puzzle has introduced millions of people to constraint satisfaction problems, although they may not realize it. A Sudoku board consists of 81 squares, some of which are initially filled with digits from 1 to 9. The puzzle is to fill in all the remaining squares such that no digit appears twice in any row, column, or 3×3 box (see Figure 6.4). A row, column, or box is called a **unit**.

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

(a)

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

(b)

Figure 6.4 (a) A Sudoku puzzle and (b) its solution.

The Sudoku puzzles that appear in newspapers and puzzle books have the property that there is exactly one solution. Although some can be tricky to solve by hand, taking tens of minutes, a CSP solver can handle thousands of puzzles per second.

A Sudoku puzzle can be considered a CSP with 81 variables, one for each square. We use the variable names *A1* through *A9* for the top row (left to right), down to *I1* through *I9* for the bottom row. The empty squares have the domain $\{1,2,3,4,5,6,7,8,9\}$ and the pre-filled squares have a domain consisting of a single value. In addition, there are 27 different *Alldiff* constraints, one for each unit (row, column, and box of 9 squares):

$$\begin{array}{l}
Alldiff(A1,A2,A3,A4,A5,A6,A7,A8,A9)\\
Alldiff(B1,B2,B3,B4,B5,B6,B7,B8,B9)\\
\ldots\\
Alldiff(A1,B1,C1,D1,E1,F1,G1,H1,I1)\\
Alldiff(A2,B2,C2,D2,E2,F2,G2,H2,I2)\\
\ldots\\
Alldiff(A1,A2,A3,B1,B2,B3,C1,C2,C3)\\
Alldiff(A4,A5,A6,B4,B5,B6,C4,C5,C6)\\
\ldots
\end{array}$$

Let us see how far arc consistency can take us. Assume that the *Alldiff* constraints have been expanded into binary constraints (such as $A1 \neq A2$) so that we can apply the AC-3 algorithm directly. Consider variable *E6* from Figure 6.4(a)—the empty square between the 2 and the 8 in the middle box. From the constraints in the box, we can remove 1, 2, 7, and 8 from *E6*'s domain. From the constraints in its column, we can eliminate 5, 6, 2, 8, 9, and 3 (although 2 and 8 were already removed). That leaves *E6* with a domain of $\{4\}$; in other words, we know the answer for *E6*. Now consider variable *I6*—the square in the bottom middle box surrounded by 1, 3, and 3. Applying arc consistency in its column, we eliminate 5, 6, 2, 4 (since we now know *E6* must be 4), 8, 9, and 3. We eliminate 1 by arc consistency with *I5*, and we are left with only the value 7 in the domain of *I6*. Now there are 8 known values in column 6, so arc consistency can infer that *A6* must be 1. Inference continues along these lines, and eventually, AC-3 can solve the entire puzzle—all the variables have their domains reduced to a single value, as shown in Figure 6.4(b).

Of course, Sudoku would soon lose its appeal if every puzzle could be solved by a mechanical application of AC-3, and indeed AC-3 works only for the easiest Sudoku puzzles. Slightly harder ones can be solved by PC-2, but at a greater computational cost: there are 255,960 different path constraints to consider in a Sudoku puzzle. To solve the hardest puzzles and to make efficient progress, we will have to be more clever.

Indeed, the appeal of Sudoku puzzles for the human solver is the need to be resourceful in applying more complex inference strategies. Aficionados give them colorful names, such as "naked triples." That strategy works as follows: in any unit (row, column or box), find three squares that each have a domain that contains the same three numbers or a subset of those numbers. For example, the three domains might be $\{1,8\}$, $\{3,8\}$, and $\{1,3,8\}$. From that we don't know which square contains 1, 3, or 8, but we do know that the three numbers must be distributed among the three squares. Therefore we can remove 1, 3, and 8 from the domains of every *other* square in the unit.

It is interesting to note how far we can go without saying much that is specific to Sudoku. We do of course have to say that there are 81 variables, that their domains are the digits 1 to 9, and that there are 27 *Alldiff* constraints. But beyond that, all the strategies—arc consistency, path consistency, and so on—apply generally to all CSPs, not just to Sudoku problems. Even naked triples is really a strategy for enforcing consistency of *Alldiff* constraints and is not specific to Sudoku *per se*. This is the power of the CSP formalism: for each new problem area, we only need to define the problem in terms of constraints; then the general constraint-solving mechanisms can take over.

6.3 Backtracking Search for CSPs

Sometimes we can finish the constraint propagation process and still have variables with multiple possible values. In that case we have to **search** for a solution. In this section we cover backtracking search algorithms that work on partial assignments; in the next section we look at local search algorithms over complete assignments.

Consider how a standard depth-limited search (from Chapter 3) could solve CSPs. A state would be a partial assignment, and an action would extend the assignment, adding, say, $NSW = red$ or $SA = blue$ for the Australia map-coloring problem. For a CSP with n variables of domain size d we would end up with a search tree where all the complete assignments (and thus all the solutions) are leaf nodes at depth n. But notice that the branching factor at the top level would be nd because any of d values can be assigned to any of n variables. At the next level, the branching factor is $(n-1)d$, and so on for n levels. So the tree has $n! \cdot d^n$ leaves, even though there are only d^n possible complete assignments!

We can get back that factor of $n!$ by recognizing a crucial property of CSPs: **commutativity**. A problem is commutative if the order of application of any given set of actions does not matter. In CSPs, it makes no difference if we first assign $NSW = red$ and then $SA = blue$, or the other way around. Therefore, we need only consider a *single* variable at each node in the search tree. At the root we might make a choice between $SA = red$, $SA = green$, and

Commutativity

```
function BACKTRACKING-SEARCH(csp) returns a solution or failure
  return BACKTRACK(csp, { })

function BACKTRACK(csp, assignment) returns a solution or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
  for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do
     if value is consistent with assignment then
        add {var = value} to assignment
        inferences ← INFERENCE(csp, var, assignment)
        if inferences ≠ failure then
           add inferences to csp
           result ← BACKTRACK(csp, assignment)
           if result ≠ failure then return result
           remove inferences from csp
        remove {var = value} from assignment
  return failure
```

Figure 6.5 A simple backtracking algorithm for constraint satisfaction problems. The algorithm is modeled on the recursive depth-first search of Chapter 3. The functions SELECT-UNASSIGNED-VARIABLE and ORDER-DOMAIN-VALUES, implement the general-purpose heuristics discussed in Section 6.3.1. The INFERENCE function can optionally impose arc-, path-, or k-consistency, as desired. If a value choice leads to failure (noticed either by INFERENCE or by BACKTRACK), then value assignments (including those made by INFERENCE) are retracted and a new value is tried.

$SA = blue$, but we would never choose between $NSW = red$ and $SA = blue$. With this restriction, the number of leaves is d^n, as we would hope. At each level of the tree we do have to choose which variable we will deal with, but we never have to backtrack over that choice.

Figure 6.5 shows a backtracking search procedure for CSPs. It repeatedly chooses an unassigned variable, and then tries all values in the domain of that variable in turn, trying to extend each one into a solution via a recursive call. If the call succeeds, the solution is returned, and if it fails, the assignment is restored to the previous state, and we try the next value. If no value works then we return failure. Part of the search tree for the Australia problem is shown in Figure 6.6, where we have assigned variables in the order $WA, NT, Q, \ldots$.

Notice that BACKTRACKING-SEARCH keeps only a single representation of a state (assignment) and alters that representation rather than creating new ones (see page 80).

Whereas the uninformed search algorithms of Chapter 3 could be improved only by supplying them with *domain-specific* heuristics, it turns out that backtracking search can be improved using *domain-independent* heuristics that take advantage of the factored representation of CSPs. In the following four sections we show how this is done:

- (6.3.1) Which variable should be assigned next (SELECT-UNASSIGNED-VARIABLE), and in what order should its values be tried (ORDER-DOMAIN-VALUES)?
- (6.3.2) What inferences should be performed at each step in the search (INFERENCE)?
- (6.3.3) Can we BACKTRACK more than one step when appropriate?
- (6.3.4) Can we save and reuse partial results from the search?

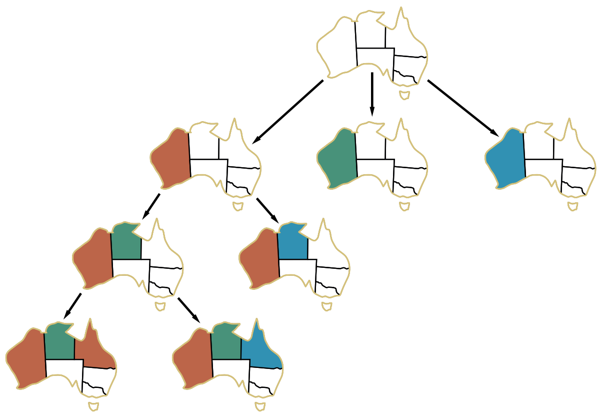


Figure 6.6 Part of the search tree for the map-coloring problem in Figure 6.1.

6.3.1 Variable and value ordering

The backtracking algorithm contains the line

$var \leftarrow$ SELECT-UNASSIGNED-VARIABLE(csp, $assignment$) .

The simplest strategy for SELECT-UNASSIGNED-VARIABLE is static ordering: choose the variables in order, $\{X_1, X_2, \ldots\}$. The next simplest is to choose randomly. Neither strategy is optimal. For example, after the assignments for $WA = red$ and $NT = green$ in Figure 6.6, there is only one possible value for SA, so it makes sense to assign $SA = blue$ 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.

Minimum-remaining-values

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 6.1, SA is the variable with highest degree, 5; the other variables have degree 2 or 3, except for T, which has degree 0. In fact, once SA is chosen, applying the degree heuristic solves the problem without any false steps—you can choose *any* consistent color at each choice point and still arrive at a solution with no backtracking. The minimum-remaining-values heuristic is usually a more powerful guide, but the degree heuristic can be useful as a tie-breaker.

Degree heuristic

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

Least-constraining-value

graph. For example, suppose that in Figure 6.1 we have generated the partial assignment with $WA = red$ and $NT = green$ and that our next choice is for Q. Blue would be a bad choice because it eliminates the last legal value left for Q's neighbor, SA. The least-constraining-value heuristic therefore prefers red to blue. In general, the heuristic is trying to leave the maximum flexibility for subsequent variable assignments.

Why should variable selection be fail-first, but value selection be fail-last? Every variable has to be assigned eventually, so by choosing the ones that are likely to fail first, we will on average have fewer successful assignments to backtrack over. For value ordering, the trick is that we only need one solution; therefore it makes sense to look for the most likely values first. If we wanted to enumerate all solutions rather than just find one, then value ordering would be irrelevant.

6.3.2 Interleaving search and inference

We saw how AC-3 can reduce the domains of variables *before* we begin the search. But inference can be even more powerful *during* the course of a search: every time we make a choice of a value for a variable, we have a brand-new opportunity to infer new domain reductions on the neighboring variables.

Forward checking

One of the simplest forms of inference is called **forward checking**. Whenever a variable X is assigned, the forward-checking process establishes arc consistency for it: for each unassigned variable Y that is connected to X by a constraint, delete from Y's domain any value that is inconsistent with the value chosen for X.

Figure 6.7 shows the progress of backtracking search on the Australia CSP with forward checking. There are two important points to notice about this example. First, notice that after $WA = red$ and $Q = green$ are assigned, the domains of NT and SA are reduced to a single value; we have eliminated branching on these variables altogether by propagating information from WA and Q. A second point to notice is that after $V = blue$, the domain of SA is empty. Hence, forward checking has detected that the partial assignment $\{WA = red, Q = green, V = blue\}$ is inconsistent with the constraints of the problem, and the algorithm backtracks immediately.

For many problems the search will be more effective if we combine the MRV heuristic with forward checking. Consider Figure 6.7 after assigning $\{WA = red\}$. Intuitively, it seems that that assignment constrains its neighbors, NT and SA, so we should handle those variables next, and then all the other variables will fall into place. That's exactly what happens with MRV: NT and SA each have two values, so one of them is chosen first, then the other, then Q, NSW, and V in order. Finally T still has three values, and any one of them works. We can view forward checking as an efficient way to incrementally compute the information that the MRV heuristic needs to do its job.

Although forward checking detects many inconsistencies, it does not detect all of them. The problem is that it doesn't look ahead far enough. For example, consider the $Q = green$ row of Figure 6.7. We've made WA and Q arc-consistent, but we've left both NT and SA with blue as their only possible value, which is an inconsistency, since they are neighbors.

Maintaining Arc Consistency

The algorithm called MAC (for **Maintaining Arc Consistency**) detects inconsistencies like this. After a variable X_i is assigned a value, the INFERENCE procedure calls AC-3, but instead of a queue of all arcs in the CSP, we start with only the arcs (X_j, X_i) for all X_j that are unassigned variables that are neighbors of X_i. From there, AC-3 does constraint propagation in the usual way, and if any variable has its domain reduced to the empty set, the call to AC-3

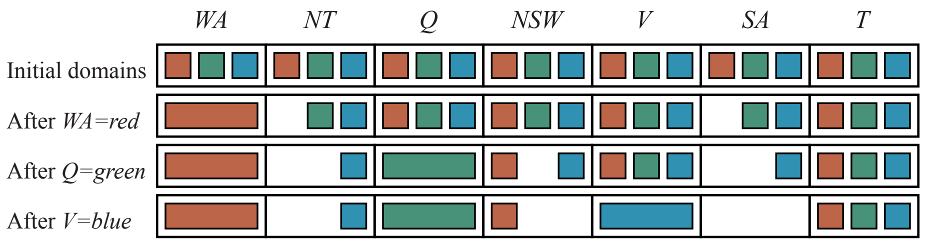


Figure 6.7 The progress of a map-coloring search with forward checking. $WA = red$ is assigned first; then forward checking deletes *red* from the domains of the neighboring variables *NT* and *SA*. After $Q = green$ is assigned, *green* is deleted from the domains of *NT*, *SA*, and *NSW*. After $V = blue$ is assigned, *blue* is deleted from the domains of *NSW* and *SA*, leaving *SA* with no legal values.

fails and we know to backtrack immediately. We can see that MAC is strictly more powerful than forward checking because forward checking does the same thing as MAC on the initial arcs in MAC's queue; but unlike MAC, forward checking does not recursively propagate constraints when changes are made to the domains of variables.

6.3.3 Intelligent backtracking: Looking backward

The BACKTRACKING-SEARCH algorithm in Figure 6.5 has a very simple policy for what to do when a branch of the search fails: back up to the preceding variable and try a different value for it. This is called **chronological backtracking** because the *most recent* decision point is revisited. In this subsection, we consider better possibilities.

Chronological backtracking

Consider what happens when we apply simple backtracking in Figure 6.1 with a fixed variable ordering Q, NSW, V, T, SA, WA, NT. Suppose we have generated the partial assignment $\{Q = red, NSW = green, V = blue, T = red\}$. When we try the next variable, SA, we see that every value violates a constraint. We back up to T and try a new color for Tasmania! Obviously this is silly—recoloring Tasmania cannot possibly help in resolving the problem with South Australia.

A more intelligent approach is to backtrack to a variable that might fix the problem—a variable that was responsible for making one of the possible values of SA impossible. To do this, we will keep track of a set of assignments that are in conflict with some value for SA. The set (in this case $\{Q = red, NSW = green, V = blue\}$), is called the **conflict set** for SA. The **backjumping** method backtracks to the *most recent* assignment in the conflict set; in this case, backjumping would jump over Tasmania and try a new value for V. This method is easily implemented by a modification to BACKTRACK such that it accumulates the conflict set while checking for a legal value to assign. If no legal value is found, the algorithm should return the most recent element of the conflict set along with the failure indicator.

Conflict set

Backjumping

The sharp-eyed reader may have noticed that forward checking can supply the conflict set with no extra work: whenever forward checking based on an assignment $X = x$ deletes a value from Y's domain, it should add $X = x$ to Y's conflict set. If the last value is deleted from Y's domain, then the assignments in the conflict set of Y are added to the conflict set of X. That is, we now know that $X = x$ leads to a contradiction (in Y), and thus a different assignment should be tried for X.

The eagle-eyed reader may have noticed something odd: backjumping occurs when every value in a domain is in conflict with the current assignment; but forward checking detects this event and prevents the search from ever reaching such a node! In fact, it can be shown that *every* branch pruned by backjumping is also pruned by forward checking. Hence, simple backjumping is redundant in a forward-checking search or, indeed, in a search that uses stronger consistency checking, such as MAC—you need only do one or the other.

Despite the observations of the preceding paragraph, the idea behind backjumping remains a good one: to backtrack based on the reasons for failure. Backjumping notices failure when a variable's domain becomes empty, but in many cases a branch is doomed long before this occurs. Consider again the partial assignment $\{WA = red, NSW = red\}$ (which, from our earlier discussion, is inconsistent). Suppose we try $T = red$ next and then assign NT, Q, V, SA. We know that no assignment can work for these last four variables, so eventually we run out of values to try at NT. Now, the question is, where to backtrack? Backjumping cannot work, because NT *does* have values consistent with the preceding assigned variables—NT doesn't have a complete conflict set of preceding variables that caused it to fail. We know, however, that the four variables NT, Q, V, and SA, *taken together*, failed because of a set of preceding variables, which must be those variables that directly conflict with the four.

This leads to a different–and deeper–notion of the conflict set for a variable such as NT: it is that set of preceding variables that caused NT, *together with any subsequent variables*, to have no consistent solution. In this case, the set is WA and NSW, so the algorithm should backtrack to NSW and skip over Tasmania. A backjumping algorithm that uses conflict sets defined in this way is called **conflict-directed backjumping**.

Conflict-directed backjumping

We must now explain how these new conflict sets are computed. The method is in fact quite simple. The "terminal" failure of a branch of the search always occurs because a variable's domain becomes empty; that variable has a standard conflict set. In our example, SA fails, and its conflict set is (say) $\{WA, NT, Q\}$. We backjump to Q, and Q *absorbs* the conflict set from SA (minus Q itself, of course) into its own direct conflict set, which is $\{NT, NSW\}$; the new conflict set is $\{WA, NT, NSW\}$. That is, there is no solution from Q onward, given the preceding assignment to $\{WA, NT, NSW\}$. Therefore, we backtrack to NT, the most recent of these. NT absorbs $\{WA, NT, NSW\} - \{NT\}$ into its own direct conflict set $\{WA\}$, giving $\{WA, NSW\}$ (as stated in the previous paragraph). Now the algorithm backjumps to NSW, as we would hope. To summarize: let X_j be the current variable, and let $conf(X_j)$ be its conflict set. If every possible value for X_j fails, backjump to the most recent variable X_i in $conf(X_j)$ and recompute the conflict set for X_i as follows:

$$conf(X_i) \leftarrow conf(X_i) \cup conf(X_j) - \{X_i\}\,.$$

6.3.4 Constraint learning

When we reach a contradiction, backjumping can tell us how far to back up, so we don't waste time changing variables that won't fix the problem. But we would also like to avoid running into the same problem again. When the search arrives at a contradiction, we know that some subset of the conflict set is responsible for the problem. **Constraint learning** is the idea of finding a minimum set of variables from the conflict set that causes the problem. This set of variables, along with their corresponding values, is called a **no-good**. We then record the no-good, either by adding a new constraint to the CSP to forbid this combination of assignments or by keeping a separate cache of no-goods.

Constraint learning

No-good

For example, consider the state $\{WA = red, NT = green, Q = blue\}$ in the bottom row of Figure 6.6. Forward checking can tell us this state is a no-good because there is no valid assignment to SA. In this particular case, recording the no-good would not help, because once we prune this branch from the search tree, we will never encounter this combination again. But suppose that the search tree in Figure 6.6 were actually part of a larger search tree that started by first assigning values for V and T. Then it would be worthwhile to record $\{WA = red, NT = green, Q = blue\}$ as a no-good because we are going to run into the same problem again for each possible set of assignments to V and T.

No-goods can be effectively used by forward checking or by backjumping. Constraint learning is one of the most important techniques used by modern CSP solvers to achieve efficiency on complex problems.

6.4 Local Search for CSPs

Local search algorithms (see Section 4.1) turn out to be very effective in solving many CSPs. They use a complete-state formulation (as introduced in Section 4.1.1) where each state assigns a value to every variable, and the search changes the value of one variable at a time. As an example, we'll use the 8-queens problem, as defined as a CSP on page 183. In Figure 6.8 we start on the left with a complete assignment to the 8 variables; typically this will violate several constraints. We then randomly choose a conflicted variable, which turns out to be Q_8, the rightmost column. We'd like to change the value to something that brings us closer to a solution; the most obvious approach is to select the value that results in the minimum number of conflicts with other variables—the **min-conflicts** heuristic.

Min-conflicts

In the figure we see there are two rows that only violate one constraint; we pick $Q_8 = 3$ (that is, we move the queen to the 8th column, 3rd row). On the next iteration, in the middle board of the figure, we select Q_6 as the variable to change, and note that moving the queen to the 8th row results in no conflicts. At this point there are no more conflicted variables, so we have a solution. The algorithm is shown in Figure 6.9.[2]

Min-conflicts is surprisingly effective for many CSPs. Amazingly, on the n-queens problem, if you don't count the initial placement of queens, the run time of min-conflicts is roughly *independent of problem size*. It solves even the *million*-queens problem in an average of 50 steps (after the initial assignment). This remarkable observation was the stimulus leading to a great deal of research in the 1990s on local search and the distinction between easy and hard problems, which we take up in Section 7.6.3. Roughly speaking, n-queens is easy for local search because solutions are densely distributed throughout the state space. Min-conflicts also works well for hard problems. For example, it has been used to schedule observations for the Hubble Space Telescope, reducing the time taken to schedule a week of observations from three weeks (!) to around 10 minutes.

All the local search techniques from Section 4.1 are candidates for application to CSPs, and some of those have proved especially effective. The landscape of a CSP under the min-conflicts heuristic usually has a series of plateaus. There may be millions of variable assignments that are only one conflict away from a solution. Plateau search—allowing sideways moves to another state with the same score—can help local search find its way off this

[2] Local search can easily be extended to constrained optimization problems (COPs). In that case, all the techniques for hill climbing and simulated annealing can be applied to optimize the objective function.

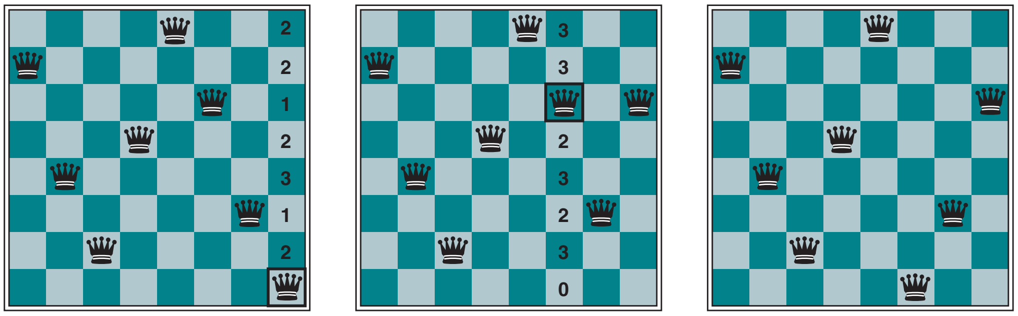


Figure 6.8 A two-step solution using min-conflicts for an 8-queens problem. At each stage, a queen is chosen for reassignment in its column. The number of conflicts (in this case, the number of attacking queens) is shown in each square. The algorithm moves the queen to the min-conflicts square, breaking ties randomly.

```
function MIN-CONFLICTS(csp, max_steps) returns a solution or failure
  inputs: csp, a constraint satisfaction problem
          max_steps, the number of steps allowed before giving up

  current ← an initial complete assignment for csp
  for i = 1 to max_steps do
     if current is a solution for csp then return current
     var ← a randomly chosen conflicted variable from csp.VARIABLES
     value ← the value v for var that minimizes CONFLICTS(csp, var, v, current)
     set var = value in current
  return failure
```

Figure 6.9 The MIN-CONFLICTS local search algorithm for CSPs. The initial state may be chosen randomly or by a greedy assignment process that chooses a minimal-conflict value for each variable in turn. The CONFLICTS function counts the number of constraints violated by a particular value, given the rest of the current assignment.

plateau. This wandering on the plateau can be directed with a technique called **tabu search**: keeping a small list of recently visited states and forbidding the algorithm to return to those states. Simulated annealing can also be used to escape from plateaus.

Constraint weighting

Another technique called **constraint weighting** aims to concentrate the search on the important constraints. Each constraint is given a numeric weight, initially all 1. At each step of the search, the algorithm chooses a variable/value pair to change that will result in the lowest total weight of all violated constraints. The weights are then adjusted by incrementing the weight of each constraint that is violated by the current assignment. This has two benefits: it adds topography to plateaus, making sure that it is possible to improve from the current state, and it also adds learning: over time the difficult constraints are assigned higher weights.

Another advantage of local search is that it can be used in an online setting (see Section 4.5) when the problem changes. Consider a scheduling problem for an airline's weekly flights. The schedule may involve thousands of flights and tens of thousands of personnel assignments, but bad weather at one airport can render the schedule infeasible. We would like to repair the schedule with a minimum number of changes. This can be easily done with a local search algorithm starting from the current schedule. A backtracking search with the new set of constraints usually requires much more time and might find a solution with many changes from the current schedule.

6.5 The Structure of Problems

In this section, we examine ways in which the *structure* of the problem, as represented by the constraint graph, can be used to find solutions quickly. Most of the approaches here also apply to other problems besides CSPs, such as probabilistic reasoning.

The only way we can possibly hope to deal with the vast real world is to decompose it into subproblems. Looking again at the constraint graph for Australia (Figure 6.1(b), repeated as Figure 6.12(a)), one fact stands out: Tasmania is not connected to the mainland.[3] Intuitively, it is obvious that coloring Tasmania and coloring the mainland are **independent subproblems**—any solution for the mainland combined with any solution for Tasmania yields a solution for the whole map.

Independent subproblems

Connected component

Independence can be ascertained simply by finding **connected components** of the constraint graph. Each component corresponds to a subproblem CSP_i. If assignment S_i is a solution of CSP_i, then $\bigcup_i S_i$ is a solution of $\bigcup_i CSP_i$. Why is this important? Suppose each CSP_i has c variables from the total of n variables, where c is a constant. Then there are n/c subproblems, each of which takes at most d^c work to solve, where d is the size of the domain. Hence, the total work is $O(d^c n/c)$, which is *linear* in n; without the decomposition, the total work is $O(d^n)$, which is exponential in n. Let's make this more concrete: dividing a Boolean CSP with 100 variables into four subproblems reduces the worst-case solution time from the lifetime of the universe down to less than a second.

Completely independent subproblems are delicious, then, but rare. Fortunately, some other graph structures are also easy to solve. For example, a constraint graph is a **tree** when any two variables are connected by only one path. We will show that *any tree-structured CSP can be solved in time linear in the number of variables.*[4] The key is a new notion of consistency, called **directional arc consistency** or DAC. A CSP is defined to be directional arc-consistent under an ordering of variables $X_1, X_2, \ldots, X_n$ if and only if every X_i is arc-consistent with each X_j for $j > i$.

Directional arc consistency

To solve a tree-structured CSP, first pick any variable to be the root of the tree, and choose an ordering of the variables such that each variable appears after its parent in the tree. Such an ordering is called a **topological sort**. Figure 6.10(a) shows a sample tree and (b) shows one possible ordering. Any tree with n nodes has $n-1$ edges, so we can make this graph directed arc-consistent in $O(n)$ steps, each of which must compare up to d possible domain values for two variables, for a total time of $O(nd^2)$. Once we have a directed arc-consistent graph, we can just march down the list of variables and choose any remaining value. Since each edge from a parent to its child is arc-consistent, we know that for any value we choose for the parent, there will be a valid value left to choose for the child. That means we won't

Topological sort

[3] A careful cartographer or patriotic Tasmanian might object that Tasmania should not be colored the same as its nearest mainland neighbor, to avoid the impression that it *might* be part of that state.

[4] Sadly, very few regions of the world have tree-structured maps, although Sulawesi comes close.

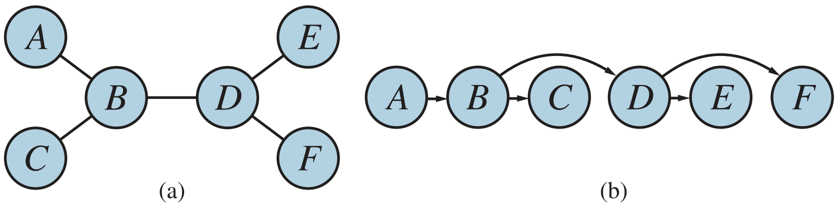


Figure 6.10 (a) The constraint graph of a tree-structured CSP. (b) A linear ordering of the variables consistent with the tree with A as the root. This is known as a **topological sort** of the variables.

```
function TREE-CSP-SOLVER(csp) returns a solution, or failure
  inputs: csp, a CSP with components $X$, $D$, $C$

  $n \leftarrow$ number of variables in $X$
  assignment ← an empty assignment
  root ← any variable in $X$
  $X \leftarrow$ TOPOLOGICALSORT($X$, root)
  for $j = n$ down to 2 do
    MAKE-ARC-CONSISTENT(PARENT($X_j$), $X_j$)
    if it cannot be made consistent then return failure
  for $i = 1$ to $n$ do
    assignment[$X_i$] ← any consistent value from $D_i$
    if there is no consistent value then return failure
  return assignment
```

Figure 6.11 The TREE-CSP-SOLVER algorithm for solving tree-structured CSPs. If the CSP has a solution, we will find it in linear time; if not, we will detect a contradiction.

have to backtrack; we can move linearly through the variables. The complete algorithm is shown in Figure 6.11.

Now that we have an efficient algorithm for trees, we can consider whether more general constraint graphs can be *reduced* to trees somehow. There are two ways to do this: by removing nodes (Section 6.5.1) or by collapsing nodes together (Section 6.5.2).

6.5.1 Cutset conditioning

The first way to reduce a constraint graph to a tree involves assigning values to some variables so that the remaining variables form a tree. Consider the constraint graph for Australia, shown again in Figure 6.12(a). Without South Australia, the graph would become a tree, as in (b). Fortunately, we can delete South Australia (in the graph, not the country) by fixing a value for *SA* and deleting from the domains of the other variables any values that are inconsistent with the value chosen for *SA*.

Now, any solution for the CSP after *SA* and its constraints are removed will be consistent with the value chosen for *SA*. (This works for binary CSPs; the situation is more complicated with higher-order constraints.) Therefore, we can solve the remaining tree with the algorithm

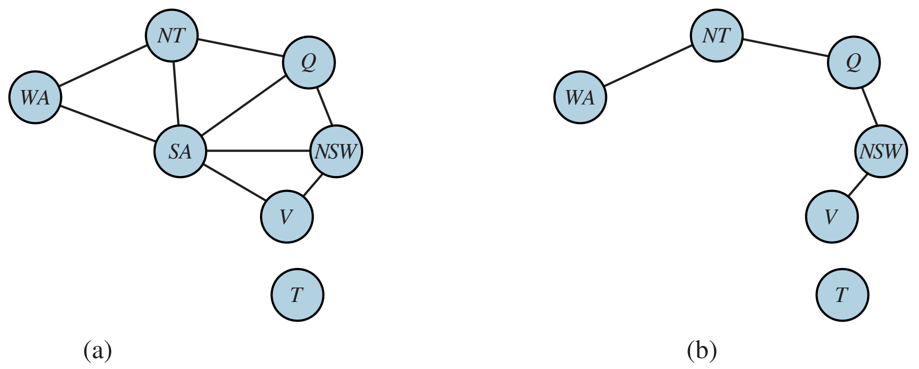


Figure 6.12 (a) The original constraint graph from Figure 6.1. (b) After the removal of *SA*, the constraint graph becomes a forest of two trees.

given above and thus solve the whole problem. Of course, in the general case (as opposed to map coloring), the value chosen for *SA* could be the wrong one, so we would need to try each possible value. The general algorithm is as follows:

1. Choose a subset S of the CSP's variables such that the constraint graph becomes a tree after removal of S. S is called a **cycle cutset**.
2. For each possible assignment to the variables in S that satisfies all constraints on S,
   (a) remove from the domains of the remaining variables any values that are inconsistent with the assignment for S, and
   (b) if the remaining CSP has a solution, return it together with the assignment for S.

Cycle cutset

If the cycle cutset has size c, then the total run time is $O(d^c \cdot (n-c)d^2)$: we have to try each of the d^c combinations of values for the variables in S, and for each combination we must solve a tree problem of size $n-c$. If the graph is "nearly a tree," then c will be small and the savings over straight backtracking will be huge—for our 100-Boolean-variable example, if we could find a cutset of size $c=20$, this would get us down from the lifetime of the Universe to a few minutes. In the worst case, however, c can be as large as $(n-2)$. Finding the *smallest* cycle cutset is NP-hard, but several efficient approximation algorithms are known. The overall algorithmic approach is called **cutset conditioning**; it comes up again in Chapter 13, where it is used for reasoning about probabilities.

Cutset conditioning

6.5.2 Tree decomposition

The second way to reduce a constraint graph to a tree is based on constructing a **tree decomposition** of the constraint graph: a transformation of the original graph into a tree where each node in the tree consists of a set of variables, as in Figure 6.13. A tree decomposition must satisfy these three requirements:

Tree decomposition

- Every variable in the original problem appears in at least one of the tree nodes.
- If two variables are connected by a constraint in the original problem, they must appear together (along with the constraint) in at least one of the tree nodes.
- If a variable appears in two nodes in the tree, it must appear in every node along the path connecting those nodes.

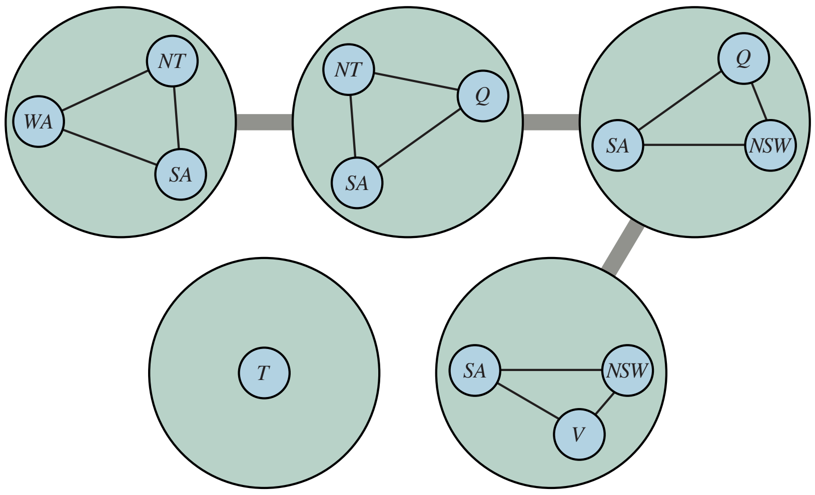


Figure 6.13 A tree decomposition of the constraint graph in Figure 6.12(a).

The first two conditions ensure that all the variables and constraints are represented in the tree decomposition. The third condition seems rather technical, but allows us to say that any variable from the original problem must have the same value wherever it appears: the constraints in the tree say that a variable in one node of the tree must have the same value as the corresponding variable in the adjacent node in the tree. For example, *SA* appears in all four of the connected nodes in Figure 6.13, so each edge in the tree decomposition therefore includes the constraint that the value of *SA* in one node must be the same as the value of *SA* in the next. You can verify from Figure 6.12 that this decomposition makes sense.

Once we have a tree-structured graph, we can apply TREE-CSP-SOLVER to get a solution in $O(nd^2)$ time, where n is the number of tree nodes and d is the size of the largest domain. But note that in the tree, a domain is a set of *tuples* of values, not just individual values.

For example, the top left node in Figure 6.13 represents, at the level of the original problem, a subproblem with variables $\{WA, NT, SA\}$, domain $\{red, green, blue\}$, and constraints $WA \neq NT, SA \neq NT, WA \neq SA$. At the level of the tree, the node represents a single variable, which we can call *SANTWA*, whose value must be a three-tuple of colors, such as $(red, green, blue)$, but not $(red, red, blue)$, because that would violate the constraint $SA \neq NT$ from the original problem. We can then move from that node to the adjacent one, with the variable we can call *SANTQ*, and find that there is only one tuple, $(red, green, blue)$, that is consistent with the choice for *SANTWA*. The exact same process is repeated for the next two nodes, and independently we can make any choice for T.

We can solve any tree decomposition problem in $O(nd^2)$ time with TREE-CSP-SOLVER, which will be efficient as long as d remains small. Going back to our example with 100 Boolean variables, if each node has 10 variables, then $d = 2^{10}$ and we should be able to solve the problem in seconds. But if there is a node with 30 variables, it would take centuries.

A given graph admits many tree decompositions; in choosing a decomposition, the aim is to make the subproblems as small as possible. (Putting all the variables into one node is technically a tree, but is not helpful.) The **tree width** of a tree decomposition of a graph is

Tree width

one less than the size of the largest node; the tree width of the graph itself is defined to be the minimum width among all its tree decompositions. If a graph has tree width w then the problem can be solved in $O(nd^{w+1})$ time given the corresponding tree decomposition. Hence, *CSPs with constraint graphs of bounded tree width are solvable in polynomial time.*

Unfortunately, finding the decomposition with minimal tree width is NP-hard, but there are heuristic methods that work well in practice. Which is better: the cutset decomposition with time $O(d^c \cdot (n-c)d^2)$, or the tree decomposition with time $O(nd^{w+1})$? Whenever you have a cycle-cutset of size c, there is also a tree width of size $w < c+1$, and it may be far smaller in some cases. So time consideration favors tree decomposition, but the advantage of the cycle-cutset approach is that it can be executed in linear memory, while tree decomposition requires memory exponential in w.

6.5.3 Value symmetry

So far, we have looked at the structure of the constraint graph. There can also be important structure in the *values* of variables, or in the structure of the constraint relations themselves. Consider the map-coloring problem with d colors. For every consistent solution, there is actually a set of $d!$ solutions formed by permuting the color names. For example, on the Australia map we know that WA, NT, and SA must all have different colors, but there are $3! = 6$ ways to assign three colors to three regions. This is called **value symmetry**. We would like to reduce the search space by a factor of $d!$ by breaking the symmetry in assignments. We do this by introducing a **symmetry-breaking constraint**. For our example, we might impose an arbitrary ordering constraint, $NT < SA < WA$, that requires the three values to be in alphabetical order. This constraint ensures that only one of the $d!$ solutions is possible: $\{NT = blue, SA = green, WA = red\}$.

Value symmetry

Symmetry-breaking constraint

For map coloring, it was easy to find a constraint that eliminates the symmetry. In general it is NP-hard to eliminate all symmetry, but breaking value symmetry has proved to be important and effective on a wide range of problems.

Summary

- **Constraint satisfaction problems** (CSPs) represent a state with a set of variable/value pairs and represent the conditions for a solution by a set of constraints on the variables. Many important real-world problems can be described as CSPs.
- A number of **inference** techniques use the constraints to rule out certain variable assignments. These include node, arc, path, and k-consistency.
- **Backtracking search**, a form of depth-first search, is commonly used for solving CSPs. Inference can be interwoven with search.
- The **minimum-remaining-values** and **degree** heuristics are domain-independent methods for deciding which variable to choose next in a backtracking search. The **least-constraining-value** heuristic helps in deciding which value to try first for a given variable. Backtracking occurs when no legal assignment can be found for a variable. **Conflict-directed backjumping** backtracks directly to the source of the problem. **Constraint learning** records the conflicts as they are encountered during search in order to avoid the same conflict later in the search.

- Local search using the **min-conflicts** heuristic has also been applied to constraint satisfaction problems with great success.
- The complexity of solving a CSP is strongly related to the structure of its constraint graph. Tree-structured problems can be solved in linear time. **Cutset conditioning** can reduce a general CSP to a tree-structured one and is quite efficient (requiring only linear memory) if a small cutset can be found. **Tree decomposition** techniques transform the CSP into a tree of subproblems and are efficient if the **tree width** of the constraint graph is small; however they need memory exponential in the tree width of the constraint graph. Combining cutset conditioning with tree decomposition can allow a better tradeoff of memory versus time.

Bibliographical and Historical Notes

Diophantine equations

The Greek mathematician Diophantus (c. 200–284) presented and solved problems involving algebraic constraints on equations, although he didn't develop a generalized methodology. We now call equations over integer domains **Diophantine equations**. The Indian mathematician Brahmagupta (c. 650) was the first to show a general solution over the domain of integers for the equation $ax + by = c$. Systematic methods for solving linear equations by variable elimination were studied by Gauss (1829); the solution of linear inequality constraints goes back to Fourier (1827).

Finite-domain constraint satisfaction problems also have a long history. For example, **graph coloring** (of which map coloring is a special case) is an old problem in mathematics. The four-color conjecture (that every planar graph can be colored with four or fewer colors) was first made by Francis Guthrie, a student of De Morgan, in 1852. It resisted solution—despite several published claims to the contrary—until a proof was devised by Appel and Haken (1977) (see the book *Four Colors Suffice* (Wilson, 2004)). Purists were disappointed that part of the proof relied on a computer, so Georges Gonthier (2008), using the COQ theorem prover, derived a formal proof that Appel and Haken's proof program was correct.

Specific classes of constraint satisfaction problems occur throughout the history of computer science. One of the most influential early examples was SKETCHPAD (Sutherland, 1963), which solved geometric constraints in diagrams and was the forerunner of modern drawing programs and CAD tools. The identification of CSPs as a *general* class is due to Ugo Montanari (1974). The reduction of higher-order CSPs to purely binary CSPs with auxiliary variables (see Exercise 6.NARY) is due originally to the 19th-century logician Charles Sanders Peirce. It was introduced into the CSP literature by Dechter (1990b) and was elaborated by Bacchus and van Beek (1998). CSPs with preferences among solutions are studied widely in the optimization literature; see Bistarelli *et al.* (1997) for a generalization of the CSP framework to allow for preferences.

Constraint propagation methods were popularized by Waltz's (1975) success on polyhedral line-labeling problems for computer vision. Waltz showed that in many problems, propagation completely eliminates the need for backtracking. Montanari (1974) introduced the notion of constraint graphs and propagation by path consistency. Alan Mackworth (1977) proposed the AC-3 algorithm for enforcing arc consistency as well as the general idea of combining backtracking with some degree of consistency enforcement. AC-4, a more efficient arc-consistency algorithm developed by Mohr and Henderson (1986), runs in $O(cd^2)$ worst-case time but can be slower than AC-3 on average cases. The PC-2 algorithm (Mackworth, 1977) achieves path consistency in much the same way that AC-3 achieves arc consistency.

Soon after Mackworth's paper appeared, researchers began experimenting with the trade-off between the cost of consistency enforcement and the benefits in terms of search reduction. Haralick and Elliott (1980) favored the minimal forward-checking algorithm described by McGregor (1979), whereas Gaschnig (1979) suggested full arc-consistency checking after each variable assignment—an algorithm later called MAC by Sabin and Freuder (1994). The latter paper provides somewhat convincing evidence that on harder CSPs, full arc-consistency checking pays off. Freuder (1978, 1982) investigated the notion of k-consistency and its relationship to the complexity of solving CSPs. Dechter and Dechter (1987) introduced directional arc consistency. Apt (1999) describes a generic algorithmic framework within which consistency propagation algorithms can be analyzed, and surveys are given by Bessière (2006) and Barták *et al.* (2010).

Constraint logic programming

Special methods for handling higher-order or global constraints were developed first within the context of **constraint logic programming**. Marriott and Stuckey (1998) provide excellent coverage of research in this area. The *Alldiff* constraint was studied by Regin (1994), Stergiou and Walsh (1999), and van Hoeve (2001). There are more complex inference algorithms for *Alldiff* (see van Hoeve and Katriel, 2006) that propagate more constraints but are more computationally expensive to run. Bounds constraints were incorporated into constraint logic programming by Van Hentenryck *et al.* (1998). A survey of global constraints is provided by van Hoeve and Katriel (2006).

Sudoku has become the most widely known CSP and was described as such by Simonis (2005). Agerbeck and Hansen (2008) describe some of the strategies and show that Sudoku on an $n^2 \times n^2$ board is in the class of *NP*-hard problems.

In 1850, C. F. Gauss described a recursive backtracking algorithm for solving the 8-queens problem, which had been published in the German chess magazine *Schachzeitung* in 1848. Gauss called his method *Tatonniren*, derived from the French word *tâtonner*—to grope around, as if in the dark.

According to Donald Knuth (personal communication), R. J. Walker introduced the term *backtrack* in the 1950s. Walker (1960) described the basic backtracking algorithm and used it to find all solutions to the 13-queens problem. Golomb and Baumert (1965) formulated, with examples, the general class of combinatorial problems to which backtracking can be applied, and introduced what we call the MRV heuristic. Bitner and Reingold (1975) provided an influential survey of backtracking techniques. Brelaz (1979) used the degree heuristic as a tiebreaker after applying the MRV heuristic. The resulting algorithm, despite its simplicity, is still the best method for k-coloring arbitrary graphs. Haralick and Elliott (1980) proposed the least-constraining-value heuristic.

The basic backjumping method is due to John Gaschnig (1977, 1979). Kondrak and van Beek (1997) showed that this algorithm is essentially subsumed by forward checking. Conflict-directed backjumping was devised by Prosser (1993). Dechter (1990a) introduced graph-based backjumping, which bounds the complexity of backjumping-based algorithms as a function of the constraint graph (Dechter and Frost, 2002).

Dependency-directed backtracking

A very general form of intelligent backtracking was developed early on by Stallman and Sussman (1977). Their technique of **dependency-directed backtracking** combines backjumping with no-good learning (McAllester, 1990) and led to the development of **truth maintenance systems** (Doyle, 1979), which we discuss in Section 10.6.2. The connection between the two areas is analyzed by de Kleer (1989).

Constraint learning

The work of Stallman and Sussman also introduced the idea of **constraint learning**, in which partial results obtained by search can be saved and reused later in the search. The idea was formalized by Dechter (1990a). **Backmarking** (Gaschnig, 1979) is a particularly simple method in which consistent and inconsistent pairwise assignments are saved and used to avoid rechecking constraints. Backmarking can be combined with conflict-directed backjumping; Kondrak and van Beek (1997) present a hybrid algorithm that provably subsumes either method taken separately.

The method of **dynamic backtracking** (Ginsberg, 1993) retains successful partial assignments from later subsets of variables when backtracking over an earlier choice that does not invalidate the later success. Moskewicz *et al.* (2001) show how these techniques and others are used to create an efficient SAT solver. Empirical studies of several randomized backtracking methods were done by Gomes *et al.* (2000) and Gomes and Selman (2001). Van Beek (2006) surveys backtracking.

Local search in constraint satisfaction problems was popularized by the work of Kirkpatrick *et al.* (1983) on simulated annealing (see Chapter 4), which is widely used for VLSI layout and scheduling problems. Beck *et al.* (2011) give an overview of recent work on job-shop scheduling. The min-conflicts heuristic was first proposed by Gu (1989) and was developed independently by Minton *et al.* (1992). Sosic and Gu (1994) showed how it could be applied to solve the 3,000,000 queens problem in less than a minute. The astounding success of local search using min-conflicts on the n-queens problem led to a reappraisal of the nature and prevalence of “easy” and “hard” problems. Peter Cheeseman *et al.* (1991) explored the difficulty of randomly generated CSPs and discovered that almost all such problems either are trivially easy or have no solutions. Only if the parameters of the problem generator are set in a certain narrow range, within which roughly half of the problems are solvable, do we find “hard” problem instances. We discuss this phenomenon further in Chapter 7.

Konolige (1994) showed that local search is inferior to backtracking search on problems with a certain degree of local structure; this led to work that combined local search and inference, such as that by Pinkas and Dechter (1995). Hoos and Tsang (2006) provide a survey of local search techniques, and textbooks are offered by Hoos and Stützle (2004) and Aarts and Lenstra (2003).

Work relating the structure and complexity of CSPs originates with Freuder (1985) and Mackworth and Freuder (1985), who showed that search on arc-consistent trees works without any backtracking. A similar result, with extensions to acyclic hypergraphs, was developed in the database community (Beeri *et al.*, 1983). Bayardo and Miranker (1994) present an algorithm for tree-structured CSPs that runs in linear time without any preprocessing. Dechter (1990a) describes the cycle-cutset approach.

Since those papers were published, there has been a great deal of progress in developing more general results relating the complexity of solving a CSP to the structure of its constraint graph. The notion of tree width was introduced by the graph theorists Robertson and Seymour (1986). Dechter and Pearl (1987, 1989), building on the work of Freuder, applied a related notion (which they called **induced width** but is identical to tree width) to constraint satisfaction problems and developed the tree decomposition approach sketched in Section 6.5.

Drawing on this work and on results from database theory, Gottlob *et al.* (1999a, 1999b) developed a notion, **hypertree width**, that is based on the characterization of the CSP as a hypergraph. In addition to showing that any CSP with hypertree width w can be solved in time $O(n^{w+1} \log n)$, they also showed that hypertree width subsumes all previously defined measures of "width" in the sense that there are cases where the hypertree width is bounded and the other measures are unbounded.

The RELSAT algorithm of Bayardo and Schrag (1997) combined constraint learning and backjumping and was shown to outperform many other algorithms of the time. This led to AND-OR search algorithms applicable to both CSPs and probabilistic reasoning (Dechter and Mateescu, 2007). Brown *et al.* (1988) introduce the idea of symmetry breaking in CSPs, and Gent *et al.* (2006) give a survey.

The field of **distributed constraint satisfaction** looks at solving CSPs when there is a collection of agents, each of which controls a subset of the constraint variables. There have been annual workshops on this problem since 2000, and good coverage elsewhere (Collin *et al.*, 1999; Pearce *et al.*, 2008).

Comparing CSP algorithms is mostly an empirical science: few theoretical results show that one algorithm dominates another on all problems; instead, we need to run experiments to see which algorithms perform better on typical instances of problems. As Hooker (1995) points out, we need to be careful to distinguish between competitive testing—as occurs in competitions among algorithms based on run time—and scientific testing, whose goal is to identify the properties of an algorithm that determine its efficacy on a class of problems.

The textbooks by Apt (2003), Dechter (2003), Tsang (1993), and Lecoutre (2009), and the collection by Rossi *et al.* (2006), are excellent resources on constraint processing. There are several good survey articles, including those by Dechter and Frost (2002), and Barták *et al.* (2010). Carbonnel and Cooper (2016) survey tractable classes of CSPs. Kondrak and van Beek (1997) give an analytical survey of backtracking search algorithms, and Bacchus and van Run (1995) give a more empirical survey. Constraint programming is covered in the books by Apt (2003) and Fruhwirth and Abdennadher (2003). Papers on constraint satisfaction appear regularly in *Artificial Intelligence* and in the specialist journal *Constraints*; the latest SAT solvers are described in the annual International SAT Competition. The primary conference venue is the International Conference on Principles and Practice of Constraint Programming, often called *CP*.

CHAPTER 7

LOGICAL AGENTS

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

Knowledge-based agents

Reasoning

Representation

Humans, it seems, know things; and what they know helps them do things. In AI, **knowledge-based agents** use a process of **reasoning** over an internal **representation** of knowledge to decide what actions to take.

The problem-solving agents of Chapters 3 and 4 know things, but only in a very limited, inflexible sense. They know what actions are available and what the result of performing a specific action from a specific state will be, but they don't know general facts. A route-finding agent doesn't know that it is impossible for a road to be a negative number of kilometers long. An 8-puzzle agent doesn't know that two tiles cannot occupy the same space. The knowledge they have is very useful for finding a path from the start to a goal, but not for anything else.

The atomic representations used by problem-solving agents are also very limiting. In a partially observable environment, for example, a problem-solving agent's only choice for representing what it knows about the current state is to list all possible concrete states. I could give a human the goal of driving to a U.S. town with population less than 10,000, but to say that to a problem-solving agent, I could formally describe the goal only as an explicit set of the 16,000 or so towns that satisfy the description.

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

We begin in Section 7.1 with the overall agent design. Section 7.2 introduces a simple new environment, the wumpus world, and illustrates the operation of a knowledge-based agent without going into any technical detail. Then we explain the general principles of **logic** in Section 7.3 and the specifics of **propositional logic** in Section 7.4. Propositional logic is a factored representation; while less expressive than **first-order logic** (Chapter 8), which is the canonical structured representation, propositional logic illustrates all the basic concepts

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

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

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

of logic. It also comes with well-developed inference technologies, which we describe in sections 7.5 and 7.6. Finally, Section 7.7 combines the concept of knowledge-based agents with the technology of propositional logic to build some simple agents for the wumpus world.

7.1 Knowledge-Based Agents

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

Knowledge base

Sentence

Knowledge representation language

Axiom

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

Inference

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**.

Background knowledge

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

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

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

Knowledge level

knowledge level, where we need specify only what the agent knows and what its goals are, in order to determine its behavior.

For example, an automated taxi might have the goal of taking a passenger from San Francisco to Marin County and might know that the Golden Gate Bridge is the only link between the two locations. Then we can expect it to cross the Golden Gate Bridge *because it knows that that will achieve its goal*. Notice that this analysis is independent of how the taxi

Implementation level

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.

A knowledge-based agent can be built simply by TELLing it what it needs to know. Starting with an empty knowledge base, the agent designer can TELL sentences one by one until

Declarative

Procedural

the agent knows how to operate in its environment. This is called the **declarative** approach to system building. In contrast, the **procedural** approach encodes desired behaviors directly as program code. In the 1970s and 1980s, advocates of the two approaches engaged in heated debates. We now understand that a successful agent often combines both declarative and procedural elements in its design, and that declarative knowledge can often be compiled into more efficient procedural code.

We can also provide a knowledge-based agent with mechanisms that allow it to learn for itself. These mechanisms, which are discussed in Chapter 19, create general knowledge about the environment from a series of percepts. A learning agent can be fully autonomous.

7.2 The Wumpus World

Wumpus world

In this section we describe an environment in which knowledge-based agents can show their worth. The **wumpus world** is a cave consisting of rooms connected by passageways. Lurking somewhere in the cave is the terrible wumpus, a beast that eats anyone who enters its room. The wumpus can be shot by an agent, but the agent has only one arrow. Some rooms contain bottomless pits that will trap anyone who wanders into these rooms (except for the wumpus, which is too big to fall in). The only redeeming feature of this bleak environment is the possibility of finding a heap of gold. Although the wumpus world is rather tame by modern computer game standards, it illustrates some important points about intelligence.

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

- **Performance measure**: +1000 for climbing out of the cave with the gold, –1000 for falling into a pit or being eaten by the wumpus, –1 for each action taken, and –10 for using up the arrow. The game ends either when the agent dies or when the agent climbs out of the cave.

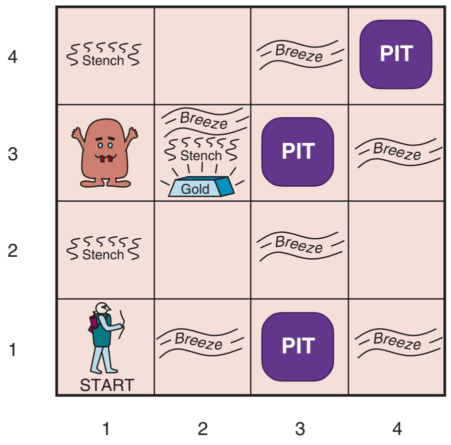


Figure 7.2 A typical wumpus world. The agent is in the bottom left corner, facing east (rightward).

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

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

[1] Presumably the square containing the wumpus also has a stench, but any agent entering that square is eaten before being able to perceive anything.

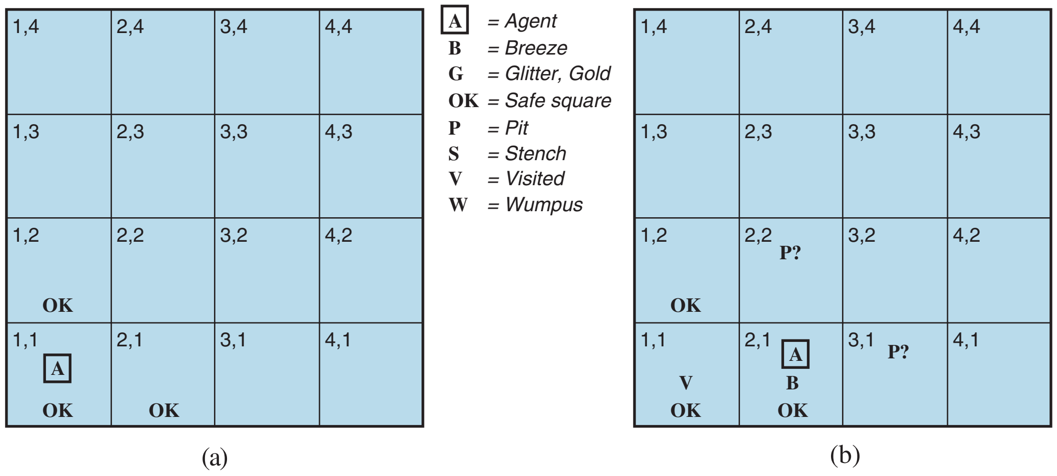


Figure 7.3 The first step taken by the agent in the wumpus world. (a) The initial situation, after percept $[None, None, None, None, None]$. (b) After moving to [2,1] and perceiving $[None, Breeze, None, None, None]$.

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

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

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

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

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

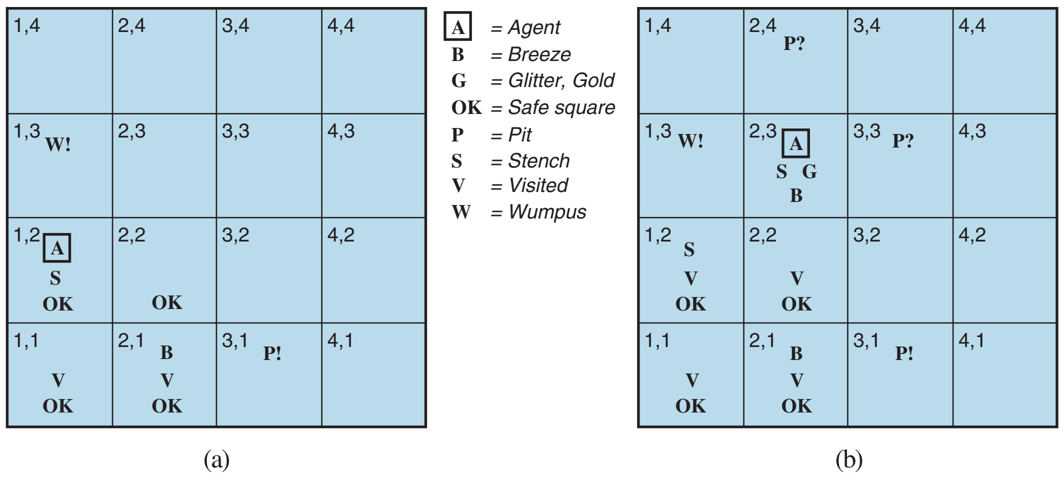


Figure 7.4 Two later stages in the progress of the agent. (a) After moving to [1,1] and then [1,2], and perceiving [*Stench*, *None*, *None*, *None*, *None*]. (b) After moving to [2,2] and then [2,3], and perceiving [*Stench*, *Breeze*, *Glitter*, *None*, *None*].

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

The agent perceives a stench in [1,2], resulting in the state of knowledge shown in Figure 7.4(a). The stench in [1,2] means that there must be a wumpus nearby. But the wumpus cannot be in [1,1], by the rules of the game, and it cannot be in [2,2] (or the agent would have detected a stench when it was in [2,1]). Therefore, the agent can infer that the wumpus is in [1,3]. The notation W! indicates this inference. Moreover, the lack of a breeze in [1,2] implies that there is no pit in [2,2]. Yet the agent has already inferred that there must be a pit in either [2,2] or [3,1], so this means it must be in [3,1]. This is a fairly difficult inference, because it combines knowledge gained at different times in different places and relies on the lack of a percept to make one crucial step.

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

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

7.3 Logic

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

Syntax

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

Semantics

Truth

Possible world

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

Model

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

Satisfaction

Entailment

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

$$\alpha \models \beta$$

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

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

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

We can apply the same kind of analysis to the wumpus-world reasoning example given in the preceding section. Consider the situation in Figure 7.3(b): the agent has detected nothing in [1,1] and a breeze in [2,1]. These percepts, combined with the agent's knowledge of the rules of the wumpus world, constitute the KB. The agent is interested in whether the adjacent squares [1,2], [2,2], and [3,1] contain pits. Each of the three squares might or might

[2] **Fuzzy logic**, discussed in Chapter 13, allows for degrees of truth.

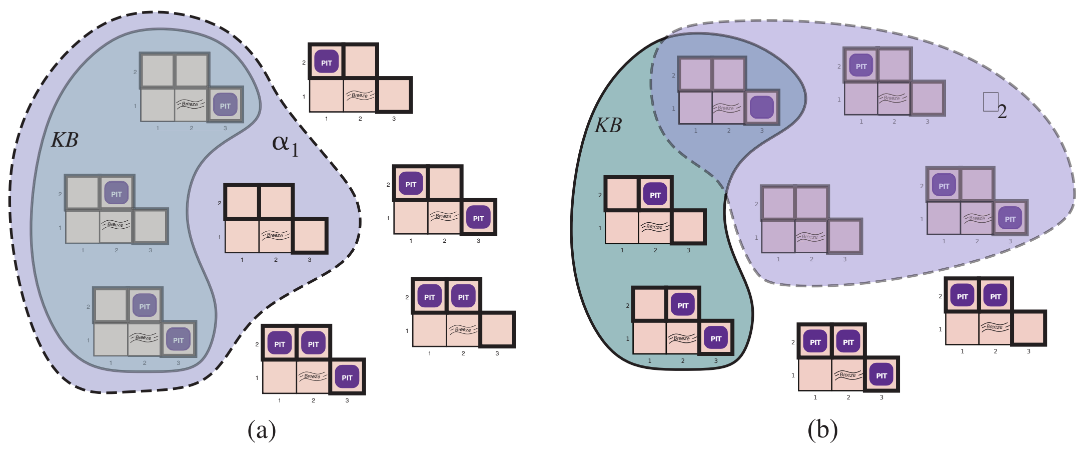


Figure 7.5 Possible models for the presence of pits in squares [1,2], [2,2], and [3,1]. The KB corresponding to the observations of nothing in [1,1] and a breeze in [2,1] is shown by the solid line. (a) Dotted line shows models of α_1 (no pit in [1,2]). (b) Dotted line shows models of α_2 (no pit in [2,2]).

not contain a pit, so (ignoring other aspects of the world for now) there are $2^3 = 8$ possible models. These eight models are shown in Figure 7.5.[3]

The KB can be thought of as a set of sentences or as a single sentence that asserts all the individual sentences. The KB is false in models that contradict what the agent knows—for example, the KB is false in any model in which [1,2] contains a pit, because there is no breeze in [1,1]. There are in fact just three models in which the KB is true, and these are shown surrounded by a solid line in Figure 7.5. Now let us consider two possible conclusions:

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

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

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

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

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

Hence, *KB* does not entail α_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, that is, that $M(KB) \subseteq M(\alpha)$.

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 12 shows how.

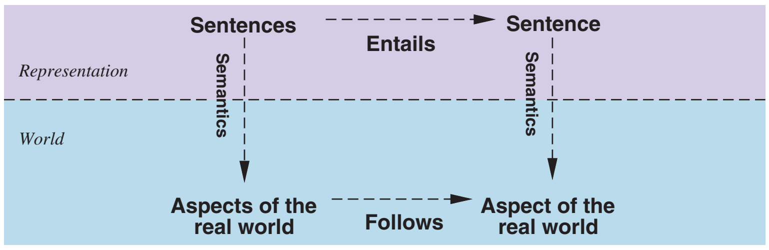


Figure 7.6 Sentences are physical configurations of the agent, and reasoning is a process of constructing new physical configurations from old ones. Logical reasoning should ensure that the new configurations represent aspects of the world that actually follow from the aspects that the old configurations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

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 by virtue of other aspects of the real world being the case.[7] This correspondence between world and representation is illustrated in Figure 7.6.

Grounding

The final issue to consider is **grounding**—the connection between logical reasoning processes and the real environment in which the agent exists. In particular, *how do we know that*

[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."

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 27.) A simple answer is that the agent’s sensors create the connection. For example, our wumpus-world agent has a smell sensor. The agent program creates a suitable sentence whenever there is a smell. Then, whenever that sentence is in the knowledge base, it is true in the real world. Thus, the meaning and truth of percept sentences are defined by the processes of sensing and sentence construction that produce them. What about the rest of the agent’s knowledge, such as its belief that wumpuses cause smells in adjacent squares? This is not a direct representation of a single percept, but a general rule—derived, perhaps, from perceptual experience but not identical to a statement of that experience. General rules like this are produced by a sentence construction process called **learning**, which is the subject of Part V. Learning is fallible. It could be the case that wumpuses cause smells *except on February 29 in leap years*, which is when they take their baths. Thus, *KB* may not be true in the real world, but with good learning procedures, there is reason for optimism.

7.4 Propositional Logic: A Very Simple Logic

Propositional logic

We now present **propositional logic**. We describe its syntax (the structure of sentences) and its semantics (the way in which the truth of sentences is determined). From these, we derive a simple, syntactic algorithm for logical inference that implements the semantic notion of entailment. Everything takes place, of course, in the wumpus world.

7.4.1 Syntax

Atomic sentences

Proposition symbol

The **syntax** of propositional logic defines the allowable sentences. The **atomic sentences** consist of a single **proposition symbol**. Each such symbol stands for a proposition that can be true or false. We use symbols that start with an uppercase letter and may contain other letters or subscripts, for example: P, Q, R, $W_{1,3}$ and *FacingEast*. The names are arbitrary but are often chosen to have some mnemonic value—we 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 parentheses and operators called **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**).

Conjunction

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

Disjunction

$\vee$ (or). A sentence whose main connective is $\vee$, such as $(W_{1,3} \wedge P_{3,1}) \vee W_{2,2}$, is a **disjunction**; its parts are **disjuncts**—in this example, $(W_{1,3} \wedge P_{3,1})$ and $W_{2,2}$.

Implication

Premise

Conclusion

Rules

$\Rightarrow$ (implies). A sentence such as $(W_{1,3} \wedge P_{3,1}) \Rightarrow \neg W_{2,2}$ is called an **implication** (or conditional). Its **premise** or **antecedent** is $(W_{1,3} \wedge P_{3,1})$, 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**.

$$
\begin{array}{rcl}
\mathit{Sentence} & \rightarrow & \mathit{AtomicSentence} \mid \mathit{ComplexSentence} \\
\mathit{AtomicSentence} & \rightarrow & \mathit{True} \mid \mathit{False} \mid P \mid Q \mid R \mid \ldots \\
\mathit{ComplexSentence} & \rightarrow & (\, \mathit{Sentence} \,) \\
 & \mid & \neg\, \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \land \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \lor \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \;\Rightarrow\; \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \;\Leftrightarrow\; \mathit{Sentence} \\
 & & \\
\textsc{Operator Precedence} & : & \neg, \land, \lor, \Rightarrow, \Leftrightarrow
\end{array}
$$

Figure 7.7 A BNF (Backus–Naur Form) grammar of sentences in propositional logic, along with operator precedences, from highest to lowest.

Figure 7.7 gives a formal grammar of propositional logic. (BNF notation is explained on page 1030.) The BNF grammar is augmented with an operator precedence list to remove ambiguity when multiple operators are used. The “not” operator ($\neg$) has the highest precedence, which means that in the sentence $\neg A \land B$ the $\neg$ binds most tightly, giving us the equivalent of $(\neg A) \land B$ rather than $\neg(A \land B)$. (The notation for ordinary arithmetic is the same: $-2+4$ is 2, not –6.) When appropriate, we also use parentheses and square brackets to clarify the intended sentence structure and improve readability.

7.4.2 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 sets 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

Truth value

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

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

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

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

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*.

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

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

Truth table

The rules can also be expressed with **truth tables** that specify the truth value of a complex sentence for each possible assignment of truth values to its components. Truth tables for the five connectives are given in Figure 7.8. From these tables, the truth value of any sentence s can be computed with respect to any model m by a simple recursive evaluation. For example, the sentence $\neg P_{1,2} \wedge (P_{2,2} \vee P_{3,1})$, evaluated in m_1, gives $true \wedge (false \vee true) = true \wedge true = true$. Exercise 7.TRUV asks you to write the algorithm PL-TRUE?(s, m), which computes the truth value of a propositional logic sentence s in a model m.

The truth tables for "and," "or," and "not" are in close accord with our intuitions about the English words. The main point of possible confusion is that $P \vee Q$ is true when P is true or Q is true *or both*. A different connective, called "exclusive or" ("xor" for short), yields false when both disjuncts are true.[8] There is no consensus on the symbol for exclusive or; some choices are $\dot{\vee}$ or $\neq$ or $\oplus$.

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

The biconditional, $P \Leftrightarrow Q$, is true whenever both $P \Rightarrow Q$ and $Q \Rightarrow P$ are true. In English, this is often written as "P if and only if Q." Many of the rules of the wumpus world are best

[8] Latin uses two separate words: "vel" is inclusive or and "aut" is exclusive or.

written using $\Leftrightarrow$. For example, a square is breezy *if* a neighboring square has a pit, and a square is breezy *only if* a neighboring square has a pit. So we need a biconditional,

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

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

7.4.3 A simple knowledge base

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

$P_{x,y}$ is true if there is a pit in $[x,y]$.
$W_{x,y}$ is true if there is a wumpus in $[x,y]$, dead or alive.
$B_{x,y}$ is true if there is a breeze in $[x,y]$.
$S_{x,y}$ is true if there is a stench in $[x,y]$.
$L_{x,y}$ is true if the agent is in location $[x,y]$.

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

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

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

- 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:

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

- 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).

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

7.4.4 A simple inference procedure

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

Figure 7.9 reproduces in a more precise form the reasoning illustrated in Figure 7.5. A general algorithm for deciding entailment in propositional logic is shown in Figure 7.10. Like the BACKTRACKING-SEARCH algorithm on page 192, TT-ENTAILS? performs a recursive

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	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*	$\underline{\textit{true}}$
false	*true*	*false*	*false*	*false*	*true*	*false*	*true*	*true*	*true*	*true*	*true*	$\underline{\textit{true}}$
false	*true*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	$\underline{\textit{true}}$
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 (the ones underlined in the right-hand column). In all 3 rows, $P_{1,2}$ is false, so there is no pit in [1,2]. On the other hand, there might (or might not) be a pit in [2,2].

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

  symbols ← a list of the proposition symbols in KB and α
  return TT-CHECK-ALL(KB, α, symbols, { })

function TT-CHECK-ALL(KB, α, symbols, model) returns true or false
  if EMPTY?(symbols) then
     if PL-TRUE?(KB, model) then return PL-TRUE?(α, model)
     else return true      // when KB is false, always return true
  else
     P ← FIRST(symbols)
     rest ← REST(symbols)
     return (TT-CHECK-ALL(KB, α, rest, model ∪ {P = true})
             and
             TT-CHECK-ALL(KB, α, rest, model ∪ {P = false }))
```

Figure 7.10 A truth-table enumeration algorithm for deciding propositional entailment. (TT stands for truth table.) PL-TRUE? returns *true* if a sentence holds within a model. The variable *model* represents a partial model—an assignment to some of the symbols. The keyword **and** here is an infix function symbol in the pseudocode programming language, not an operator in proposition logic; it takes two arguments and returns *true* or *false*.

enumeration of a finite space of assignments to symbols. The algorithm is **sound** because it implements directly the definition of entailment, and **complete** because it works for any *KB* and α and always terminates—there are only finitely many models to examine.

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

7.5 Propositional Theorem Proving

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

Theorem proving

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

Logical equivalence

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

Validity

Tautology

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

Deduction theorem

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

(Exercise 7.DEDU asks for a proof.) Hence, we can decide if $\alpha \models \beta$ by checking that $(\alpha \Rightarrow \beta)$ is true in every model—which is essentially what the inference algorithm in Figure 7.10 does—or by proving that $(\alpha \Rightarrow \beta)$ is equivalent to *True*. Conversely, the deduction theorem states that every valid implication sentence describes a legitimate inference.

Satisfiability

The final concept we will need is **satisfiability**. A sentence is satisfiable if it is true in, or satisfied by, *some* model. For example, the knowledge base given earlier, $(R_1 \land R_2 \land R_3 \land R_4 \land R_5)$, is satisfiable because there are three models in which it is true, as shown in Figure 7.9. Satisfiability can be checked by enumerating the possible models until one is found that satisfies the sentence. The problem of determining the satisfiability of sentences

$$
\begin{array}{rcll}
(\alpha \land \beta) & \equiv & (\beta \land \alpha) & \text{commutativity of } \land \\
(\alpha \lor \beta) & \equiv & (\beta \lor \alpha) & \text{commutativity of } \lor \\
((\alpha \land \beta) \land \gamma) & \equiv & (\alpha \land (\beta \land \gamma)) & \text{associativity of } \land \\
((\alpha \lor \beta) \lor \gamma) & \equiv & (\alpha \lor (\beta \lor \gamma)) & \text{associativity of } \lor \\
\neg(\neg\alpha) & \equiv & \alpha & \text{double-negation elimination} \\
(\alpha \Rightarrow \beta) & \equiv & (\neg\beta \Rightarrow \neg\alpha) & \text{contraposition} \\
(\alpha \Rightarrow \beta) & \equiv & (\neg\alpha \lor \beta) & \text{implication elimination} \\
(\alpha \Leftrightarrow \beta) & \equiv & ((\alpha \Rightarrow \beta) \land (\beta \Rightarrow \alpha)) & \text{biconditional elimination} \\
\neg(\alpha \land \beta) & \equiv & (\neg\alpha \lor \neg\beta) & \text{De Morgan} \\
\neg(\alpha \lor \beta) & \equiv & (\neg\alpha \land \neg\beta) & \text{De Morgan} \\
(\alpha \land (\beta \lor \gamma)) & \equiv & ((\alpha \land \beta) \lor (\alpha \land \gamma)) & \text{distributivity of } \land \text{ over } \lor \\
(\alpha \lor (\beta \land \gamma)) & \equiv & ((\alpha \lor \beta) \land (\alpha \lor \gamma)) & \text{distributivity of } \lor \text{ over } \land
\end{array}
$$

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 6 ask whether the constraints are satisfiable by some assignment.

SAT

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

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

Proving β from α by checking the unsatisfiability of $(\alpha \land \neg\beta)$ 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 $(\alpha \land \neg\beta)$ is unsatisfiable.

Reductio ad absurdum
Refutation
Contradiction

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

Inference rules
Proof
Modus Ponens

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

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 $(WumpusAhead \land WumpusAlive) \Rightarrow Shoot$ and $(WumpusAhead \land WumpusAlive)$ 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:

And-Elimination

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

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

By considering the possible truth values of α and β, one can easily show once and for all that Modus Ponens and And-Elimination are sound. 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

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

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]:

1. Apply biconditional elimination to R_2 to obtain
$$R_6: \quad (B_{1,1} \Rightarrow (P_{1,2} \lor P_{2,1})) \land ((P_{1,2} \lor P_{2,1}) \Rightarrow B_{1,1})\,.$$
2. Apply And-Elimination to R_6 to obtain
$$R_7: \quad ((P_{1,2} \lor P_{2,1}) \Rightarrow B_{1,1})\,.$$
3. Logical equivalence for contrapositives gives
$$R_8: \quad (\neg B_{1,1} \Rightarrow \neg(P_{1,2} \lor P_{2,1}))\,.$$
4. Apply Modus Ponens with R_8 and the percept R_4 (i.e., $\neg B_{1,1}$), to obtain
$$R_9: \quad \neg(P_{1,2} \lor P_{2,1})\,.$$
5. 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.

Any of the search algorithms in Chapter 3 can be used to find a sequence of steps that constitutes a proof like this. We just need to define a proof problem as follows:

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

Thus, searching for proofs is an alternative to enumerating models. In many practical cases *finding a proof can be more efficient because the proof can ignore irrelevant propositions, no matter how many of them there are.* For example, the proof just given 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_2; the other propositions in R_2 appear only in R_4 and R_2; so R_1, R_3, and R_5 have no bearing on the proof. The same would hold even if we added a million more sentences to the knowledge base; the simple truth-table algorithm, on the other hand, would be overwhelmed by the exponential explosion of models.

One final property of logical systems is **monotonicity**, which says that the set of entailed sentences can only *increase* as information is added to the knowledge base.[9] For any sentences α and β,

Monotonicity

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

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*.

7.5.2 Proof by resolution

We have argued that the inference rules covered so far are *sound*, but we have not discussed the question of *completeness* for the inference algorithms that use them. Search algorithms such as iterative deepening search (page 81) 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:

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

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]:

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

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

Resolvent

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

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

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

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

Unit resolution

[9] **Nonmonotonic** logics, which violate the monotonicity property, capture a common property of human reasoning: changing one's mind. They are discussed in Section 10.6.

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

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

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

You can resolve only one pair of complementary literals at a time. For example, we can resolve P and $\neg P$ to deduce

$$\frac{P \vee \neg Q \vee R, \qquad \neg P \vee Q}{\neg Q \vee Q \vee R}\,,$$

but you can't resolve on both P and Q at once to infer R. There is one more technical aspect of the resolution rule: the resulting clause should contain only one copy of each literal.[10] 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 by factoring.

Factoring

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

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

Conjunctive normal form

The resolution rule applies only to clauses (that is, disjunctions of literals), so it would seem to be relevant only to knowledge bases and queries consisting of clauses. How, then, can it lead to a complete inference procedure for all of propositional logic? The answer is that *every sentence of propositional logic is logically equivalent to a conjunction of clauses.*

Conjunctive normal form
CNF

A sentence expressed as a conjunction of clauses is said to be in **conjunctive normal form** or **CNF** (see Figure 7.12). We now describe a procedure for converting to CNF. We illustrate the procedure by converting the sentence $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$ into CNF. The steps are as follows:

[10] 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.

$$
\begin{array}{rcl}
\textit{CNFSentence} & \rightarrow & \textit{Clause}_1 \land \cdots \land \textit{Clause}_n \\
\textit{Clause} & \rightarrow & \textit{Literal}_1 \lor \cdots \lor \textit{Literal}_m \\
\textit{Fact} & \rightarrow & \textit{Symbol} \\
\textit{Literal} & \rightarrow & \textit{Symbol} \mid \neg\textit{Symbol} \\
\textit{Symbol} & \rightarrow & P \mid Q \mid R \mid \ldots \\
\textit{HornClauseForm} & \rightarrow & \textit{DefiniteClauseForm} \mid \textit{GoalClauseForm} \\
\textit{DefiniteClauseForm} & \rightarrow & \textit{Fact} \mid (\textit{Symbol}_1 \land \cdots \land \textit{Symbol}_l) \Rightarrow \textit{Symbol} \\
\textit{GoalClauseForm} & \rightarrow & (\textit{Symbol}_1 \land \cdots \land \textit{Symbol}_l) \Rightarrow \textit{False}
\end{array}
$$

Figure 7.12 A grammar for conjunctive normal form, Horn clauses, and definite clauses. A CNF clause such as $\neg A \lor \neg B \lor C$ can be written in definite clause form as $A \land B \Rightarrow C$.

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

$$(B_{1,1} \Rightarrow (P_{1,2} \lor P_{2,1})) \land ((P_{1,2} \lor P_{2,1}) \Rightarrow B_{1,1}).$$

2. Eliminate $\Rightarrow$, replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \lor \beta$:

$$(\neg B_{1,1} \lor P_{1,2} \lor P_{2,1}) \land (\neg(P_{1,2} \lor P_{2,1}) \lor B_{1,1}).$$

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:

   $\neg(\neg\alpha) \equiv \alpha$ (double-negation elimination)
   $\neg(\alpha \land \beta) \equiv (\neg\alpha \lor \neg\beta)$ (De Morgan)
   $\neg(\alpha \lor \beta) \equiv (\neg\alpha \land \neg\beta)$ (De Morgan)

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

$$(\neg B_{1,1} \lor P_{1,2} \lor P_{2,1}) \land ((\neg P_{1,2} \land \neg P_{2,1}) \lor B_{1,1}).$$

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

$$(\neg B_{1,1} \lor P_{1,2} \lor P_{2,1}) \land (\neg P_{1,2} \lor B_{1,1}) \land (\neg P_{2,1} \lor B_{1,1}).$$

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

A resolution algorithm

Inference procedures based on resolution work by using the principle of proof by contradiction introduced on page 223. That is, to show that $KB \models \alpha$, we show that $(KB \land \neg\alpha)$ is unsatisfiable. We do this by proving a contradiction.

A resolution algorithm is shown in Figure 7.13. First, $(KB \land \neg\alpha)$ is converted into CNF. Then, the resolution rule is applied to the resulting clauses. Each pair that contains complementary literals is resolved to produce a new clause, which is added to the set if it is not already present. The process continues until one of two things happens:

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

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

Figure 7.13 A simple resolution algorithm for propositional logic. PL-RESOLVE returns the set of all possible clauses obtained by resolving its two inputs.

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. Moreover, the empty clause arises only from resolving two contradictory 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 7.14. The second row of the figure shows clauses obtained by resolving pairs in the first row. Then, when $P_{1,2}$ is resolved with $\neg P_{1,2}$, we obtain the empty clause, shown as a small square. Inspection of Figure 7.14 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

To conclude our discussion of resolution, we now show why PL-RESOLUTION is complete. To do this, we introduce the **resolution closure** $RC(S)$ of a set of clauses S, which is the set of all clauses derivable by repeated application of the resolution rule to clauses in S or their derivatives. The resolution closure is what PL-RESOLUTION computes as the final value of the variable *clauses*. It is easy to see that $RC(S)$ must be finite: thanks to the factoring step, there are only finitely many distinct clauses that can be constructed out of the symbols $P_1, \ldots, P_k$ that appear in S. Hence, PL-RESOLUTION always terminates.

Resolution closure

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

Ground resolution theorem

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

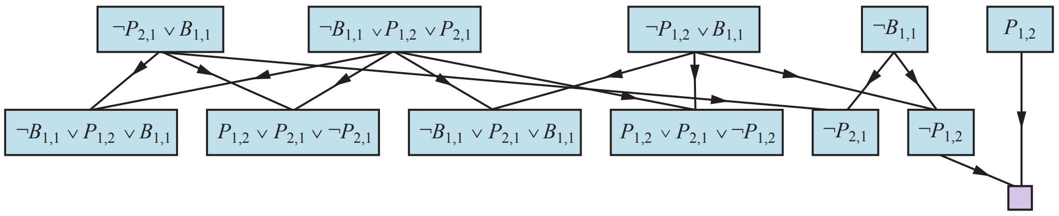

$\neg P_{2,1} \vee B_{1,1}$
$\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}$
$\neg P_{1,2} \vee B_{1,1}$
$\neg B_{1,1}$
$P_{1,2}$
$\neg B_{1,1} \vee P_{1,2} \vee B_{1,1}$
$P_{1,2} \vee P_{2,1} \vee \neg P_{2,1}$
$\neg B_{1,1} \vee P_{2,1} \vee B_{1,1}$
$P_{1,2} \vee P_{2,1} \vee \neg P_{1,2}$
$\neg P_{2,1}$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

Figure 7.14 Partial application of PL-RESOLUTION to a simple inference in the wumpus world to prove the query $\neg P_{1,2}$. Each of the leftmost four clauses in the top row is paired with each of the other three, and the resolution rule is applied to yield the clauses on the bottom row. We see that the third and fourth clauses on the top row combine to yield the clause $\neg P_{1,2}$, which is then resolved with $P_{1,2}$ to yield the empty clause, meaning that the query is proven.

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

This assignment to $P_1, \ldots, P_k$ is a model of S. To see this, assume the opposite—that, at some stage i in the sequence, assigning symbol P_i causes some clause C to become false. For this to happen, it must be the case that all the *other* literals in C must already have been falsified by assignments to $P_1, \ldots, P_{i-1}$. Thus, C must now look like either $(false \vee false \vee \cdots false \vee P_i)$ or like $(false \vee false \vee \cdots false \vee \neg P_i)$. If just one of these two is in $RC(S)$, then the algorithm will assign the appropriate truth value to P_i to make C true, so C can only be falsified if *both* of these clauses are in $RC(S)$.

Now, since $RC(S)$ is closed under resolution, it will contain the resolvent of these two clauses, and that resolvent will have all of its literals already falsified by the assignments to $P_1, \ldots, P_{i-1}$. This contradicts our assumption that the first falsified clause appears at stage i. Hence, we have proved that the construction never falsifies a clause in $RC(S)$; that is, it produces a model of $RC(S)$. Finally, because S is contained in $RC(S)$, any model of $RC(S)$ is a model of S itself.

7.5.3 Horn clauses and definite clauses

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

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

Definite clause

Slightly more general is the **Horn clause**, which is a disjunction of literals of which *at*

Horn clause

most one is positive. So all definite clauses are Horn clauses, as are clauses with no positive literals; these are called **goal clauses**. Horn clauses are closed under resolution: if you resolve two Horn clauses, you get back a Horn clause. One more class is the k-CNF sentence, which is a CNF sentence where each clause has at most k literals.

Goal clauses

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

1. Every definite clause can be written as an implication whose premise is a conjunction of positive literals and whose conclusion is a single positive literal. (See Exercise 7.DISJ.) For example, the definite clause $(\neg L_{1,1} \vee \neg Breeze \vee B_{1,1})$ can be written as the implication $(L_{1,1} \wedge Breeze) \Rightarrow B_{1,1}$. In the implication form, the sentence is easier to understand: it says that if the agent is in [1,1] and there is a breeze percept, then [1,1] is breezy. In Horn form, the premise is called the **body** and the conclusion is called the **head**. A sentence consisting of a single positive literal, such as $L_{1,1}$, is called a **fact**. It too can be written in implication form as $True \Rightarrow L_{1,1}$, but it is simpler to write just $L_{1,1}$.

Body

Head

Fact

2. Inference with Horn clauses can be done through the **forward-chaining** and **backward-chaining** algorithms, which we explain next. Both of these algorithms are natural, in that the inference steps are obvious and easy for humans to follow. This type of inference is the basis for **logic programming**, which is discussed in Chapter 9.

Forward-chaining

Backward-chaining

3. Deciding entailment with Horn clauses can be done in time that is *linear* in the size of the knowledge base—a pleasant surprise.

7.5.4 Forward and backward chaining

The forward-chaining algorithm PL-FC-ENTAILS?(KB, q) determines if a single proposition symbol q—the query—is entailed by a knowledge base of definite clauses. It begins from known facts (positive literals) in the knowledge base. If all the premises of an implication are known, then its conclusion is added to the set of known facts. For example, if $L_{1,1}$ and *Breeze* are known and $(L_{1,1} \wedge Breeze) \Rightarrow B_{1,1}$ 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 algorithm is shown in Figure 7.15; the main point to remember is that it runs in linear time.

The best way to understand the algorithm is through an example and a picture. Figure 7.16(a) shows a simple knowledge base of Horn clauses with A and B as known facts. Figure 7.16(b) shows the same knowledge base drawn as an **AND–OR graph** (see Chapter 4). In AND–OR graphs, multiple edges joined by an arc indicate a conjunction—every edge must be proved—while multiple edges without an arc indicate a disjunction—any edge 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.

It is easy to see that forward chaining is **sound**: every inference is essentially an application of Modus Ponens. Forward chaining is also **complete**: every entailed atomic sentence will be derived. The easiest way to see this is to consider the final state of the *inferred* table (after the algorithm reaches a fixed point where no new inferences are possible). The table contains *true* for each symbol inferred during the process, and *false* for all other symbols. We can view the table as a logical model; moreover, *every definite clause in the original KB is true in this model.*

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

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

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

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, because we would now be licensed to add b to the KB. We can conclude, therefore, that the set of atomic sentences inferred at the fixed point defines a model of the original KB. Furthermore, any atomic sentence q that is entailed by the KB must be true in all its models and in this model in particular. Hence, every entailed atomic sentence q must be inferred by the algorithm.

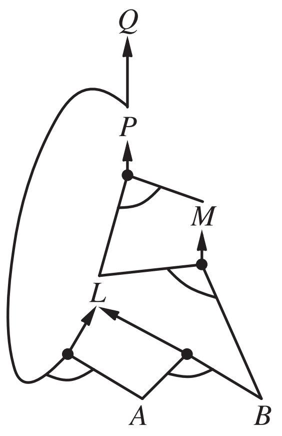
Data-driven

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

The backward-chaining algorithm, as its name suggests, works backward from the query. If the query q is known to be true, then no work is needed. Otherwise, the algorithm finds

$$\begin{aligned}
&P \Rightarrow Q\\
&L \wedge M \Rightarrow P\\
&B \wedge L \Rightarrow M\\
&A \wedge P \Rightarrow L\\
&A \wedge B \Rightarrow L\\
&A\\
&B
\end{aligned}$$

(a)



(b)

Figure 7.16 (a) A set of Horn clauses. (b) The corresponding AND–OR graph.

those implications in the knowledge base whose conclusion is q. If all the premises of one of those implications can be proved true (by backward chaining), then q is true. When applied to the query Q in Figure 7.16, it works back down the graph until it reaches a set of known facts, A and B, that forms the basis for a proof. The algorithm is essentially identical to the AND-OR-GRAPH-SEARCH algorithm in Figure 4.11. As with forward chaining, an efficient implementation runs in linear time.

Goal-directed reasoning

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

7.6 Effective Propositional Model Checking

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

The algorithms we describe are for checking satisfiability: the SAT problem. (As noted in Section 7.5, testing entailment, $\alpha \models \beta$, can be done by testing *un*satisfiability of $\alpha \wedge \neg\beta$.) We mentioned on page 223 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 propositional satisfiability algorithms closely resemble the backtracking algorithms of Section 6.3 and the local search algorithms of Section 6.4. They are, however, extremely important in their own right because so many combinatorial problems in computer science can be reduced to checking the satisfiability of a propositional sentence. Any improvement in satisfiability algorithms has huge consequences for our ability to handle complexity in general.

7.6.1 A complete backtracking algorithm

Davis–Putnam algorithm

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

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

Pure symbol

- *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 in the remaining clauses C appears only as a positive literal; therefore C becomes pure.
- *Unit clause heuristic*: A **unit clause** was defined earlier as a clause with just one literal. In the context of DPLL, it also means clauses in which all literals but one are already assigned *false* by the model. For example, if the model contains $B = true$, then $(\neg B \vee \neg C)$ simplifies to $\neg C$, which is a unit clause. Obviously, for this clause to be true, C must be set to *false*. The unit clause heuristic assigns all such symbols before branching on the remainder. One important consequence of the heuristic is that any attempt to prove (by refutation) a literal that is already in the knowledge base will succeed immediately (Exercise 7.KNOW). Notice also that assigning one unit clause can create another unit clause—for example, when C is set to *false*, $(C \vee A)$ becomes a unit clause, causing *true* to be assigned to A. This "cascade" of forced assignments is called **unit propagation**. It resembles the process of forward chaining with definite clauses, and indeed, if the CNF expression contains only definite clauses then DPLL essentially replicates forward chaining. (See Exercise 7.DPLL.)

Unit propagation

The DPLL algorithm is shown in Figure 7.17, which gives the essential skeleton of the search process without the implementation details.

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

```
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, model ∪ {P=value})
  P, value ← FIND-UNIT-CLAUSE(clauses, model)
  if P is non-null then return DPLL(clauses, symbols – P, model ∪ {P=value})
  P ← FIRST(symbols); rest ← REST(symbols)
  return DPLL(clauses, rest, model ∪ {P=true}) or
            DPLL(clauses, rest, model ∪ {P=false}))
```

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

1. **Component analysis** (as seen with Tasmania in CSPs): As DPLL assigns truth values to variables, the set of clauses may become separated into disjoint subsets, called **components**, that share no unassigned variables. Given an efficient way to detect when this occurs, a solver can gain considerable speed by working on each component separately.
2. **Variable and value ordering** (as seen in Section 6.3.1 for CSPs): Our simple implementation of DPLL uses an arbitrary variable ordering and always tries the value *true* before *false*. The **degree heuristic** (see page 193) suggests choosing the variable that appears most frequently over all remaining clauses.
3. **Intelligent backtracking** (as seen in Section 6.3.3 for CSPs): Many problems that cannot be solved in hours of run time with chronological backtracking can be solved in seconds with intelligent backtracking that backs up all the way to the relevant point of conflict. All SAT solvers that do intelligent backtracking use some form of **conflict clause learning** to record conflicts so that they won't be repeated later in the search. Usually a limited-size set of conflicts is kept, and rarely used ones are dropped.
4. **Random restarts** (as seen on page 113 for hill climbing): Sometimes a run appears not to be making progress. In this case, we can start over from the top of the search tree, rather than trying to continue. After restarting, different random choices (in variable and value selection) are made. Clauses that are learned in the first run are retained after the restart and can help prune the search space. Restarting does not guarantee that a solution will be found faster, but it does reduce the variance on the time to solution.
5. **Clever indexing** (as seen in many algorithms): The speedup methods used in DPLL itself, as well as the tricks used in modern solvers, require fast indexing of such things

```
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 value flips allowed before giving up

  model ← a random assignment of true/false to the symbols in clauses
  for each i = 1 to max_flips do
     if model satisfies clauses then return model
     clause ← a randomly selected clause from clauses that is false in model
     if RANDOM(0, 1) ≤ p then
        flip the value in model of a randomly selected symbol from clause
     else flip whichever symbol in clause maximizes the number of satisfied clauses
  return failure
```

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

as "the set of clauses in which variable X_i appears as a positive literal." This task is complicated by the fact that the algorithms are interested only in the clauses that have not yet been satisfied by previous assignments to variables, so the indexing structures must be updated dynamically as the computation proceeds.

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

7.6.2 Local search algorithms

We have seen several local search algorithms so far in this book, including HILL-CLIMBING (page 111) and SIMULATED-ANNEALING (page 115). These algorithms can be applied directly to satisfiability problems, provided that we choose the right evaluation function. Because the goal is to find an assignment that satisfies every clause, an evaluation function that counts the number of unsatisfied clauses will do the job. In fact, this is exactly the measure used by the MIN-CONFLICTS algorithm for CSPs (page 198). All these algorithms take steps in the space of complete assignments, flipping the truth value of one symbol at a time. The space usually contains many local minima, to escape from which various forms of randomness are required. In recent years, there has been a great deal of experimentation to find a good balance between greediness and randomness.

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

When WALKSAT returns a model, the input sentence is indeed satisfiable, but when it returns *failure*, there are two possible causes: either the sentence is unsatisfiable or we need to give the algorithm more time. If we set $max\_flips = \infty$ and $p > 0$, WALKSAT will eventually return a model (if one exists), because the random-walk steps will eventually hit upon the solution. Alas, if *max_flips* is infinity and the sentence is unsatisfiable, then the algorithm never terminates!

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

7.6.3 The landscape of random SAT problems

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

Underconstrained

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

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

Sixteen of the 32 possible assignments are models of this sentence, so, on average, it would take just two random guesses to find a model. This is an easy satisfiability problem, as are most such underconstrained problems. On the other hand, an *overconstrained* problem has many clauses relative to the number of variables and is likely to have no solutions. Overconstrained problems are often easy to solve, because the constraints quickly lead either to a solution or to a dead end from which there is no escape.

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

Given a source of random sentences, we can measure the probability of satisfiability. Figure 7.19(a) plots the probability for $CNF_3(m,50)$, that is, sentences with 50 variables and 3 literals per clause, as a function of the clause/symbol ratio, m/n. As we expect, for small m/n the probability of satisfiability is close to 1, and at large m/n the probability is close to 0. The probability drops fairly sharply around $m/n = 4.3$. Empirically, we find that the "cliff" stays in roughly the same place (for $k = 3$) and gets sharper and sharper as n increases.

Satisfiability threshold conjecture

Theoretically, the **satisfiability threshold conjecture** says that for every $k \geq 3$, there is a threshold ratio r_k such that, as n goes to infinity, the probability that $CNF_k(rn,n)$ is satisfiable becomes 1 for all values of r below the threshold, and 0 for all values above. The conjecture remains unproven, even for special cases like $k = 3$. Whether it is a theorem or not, this kind

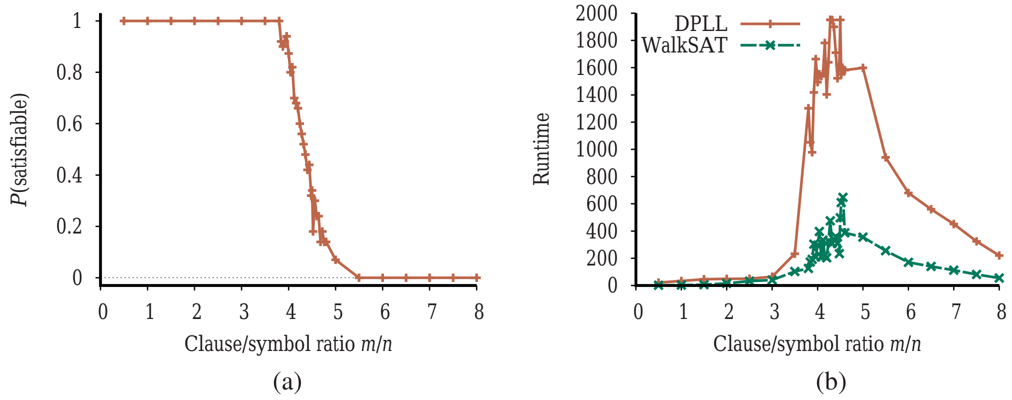


Figure 7.19 (a) Graph showing the probability that a random 3-CNF sentence with $n = 50$ symbols is satisfiable, as a function of the clause/symbol ratio m/n. (b) Graph of the median run time (measured in number of iterations) for both DPLL and WALKSAT on random 3-CNF sentences. The most difficult problems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

of thresholding effect is certainly common, for satisfiability problems as well as other types of NP-hard problems.

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

7.7 Agents Based on Propositional Logic

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

7.7.1 The current state of the world

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

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

$$\begin{aligned} &B_{1,1} \;\Leftrightarrow\; (P_{1,2} \lor P_{2,1}) \\ &S_{1,1} \;\Leftrightarrow\; (W_{1,2} \lor W_{2,1}) \\ &\cdots \end{aligned}$$

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

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

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

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

So far, so good. Now let's consider the agent's percepts. We are using $S_{1,1}$ to mean there is a stench in [1,1]; can we use a single proposition, *Stench* to mean that the agent perceives a stench? Unfortunately we can't: if there was no stench at the previous time step, then $\neg$*Stench* would already be asserted, and the new assertion would simply result in a contradiction. The problem is solved when we realize that a percept asserts something *only about the current time*. Thus, if the time step (as supplied to MAKE-PERCEPT-SENTENCE in Figure 7.1) is 4, then we add $Stench^4$ to the knowledge base, rather than *Stench*—neatly avoiding any contradiction with $\neg Stench^3$. The same goes for the breeze, bump, glitter, and scream percepts.

The idea of associating propositions with time steps extends to any aspect of the world that changes over time. For example, the initial knowledge base includes $L^0_{1,1}$—the agent is in square $[1,1]$ at time 0—as well as $FacingEast^0$, $HaveArrow^0$, and $WumpusAlive^0$. We use the noun **fluent** (from the Latin *fluens*, flowing) to refer to an aspect of the world that changes. "Fluent" is a synonym for "state variable," in the sense described in the discussion of factored representations in Section 2.4.7 on page 58. Symbols associated with permanent aspects of the world do not need a time superscript and are sometimes called **atemporal variables**.

Fluent

Atemporal variable

We can connect stench and breeze percepts directly to the properties of the squares where they are experienced as follows.[11] For any time step t and any square $[x,y]$, we assert

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

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

[11] Section 7.4.3 conveniently glossed over this requirement.

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

Effect axiom

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

$$L^0_{1,1} \land FacingEast^0 \land Forward^0 \Rightarrow (L^1_{2,1} \land \neg L^1_{1,1})\,. \tag{7.1}$$

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

Let us suppose that the agent does decide to move *Forward* at time 0 and asserts this fact into its knowledge base. Given the effect axiom in Equation (7.1), combined with the initial assertions about the state at time 0, the agent can now deduce that it is in $[2,1]$. That is, $\text{ASK}(KB, L^1_{2,1}) = true$. So far, so good. Unfortunately, if we $\text{ASK}(KB, HaveArrow^1)$, the answer is *false*, that is, the agent cannot prove it still has the arrow; nor can it prove it *doesn't* have it! The information has been lost because the effect axiom fails to state what remains *unchanged* as the result of an action. The need to do this gives rise to the **frame problem**.[12] One possible solution to the frame problem would be to add **frame axioms** explicitly asserting all the propositions that remain the same. For example, for each time t we would have

Frame problem

Frame axiom

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

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

Representational frame problem

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

Locality

Inferential frame problem

The solution to the problem involves changing one's focus from writing axioms about *actions* to writing axioms about *fluents*. Thus for each fluent F, we will have an axiom that defines the truth value of F^{t+1} in terms of fluents (including F itself) at time t and the actions that may have occurred at time t. Now, the truth value of F^{t+1} can be set in one of two ways:

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

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:

Successor-state axiom

$$F^{t+1} \Leftrightarrow ActionCausesF^t \lor (F^t \land \neg ActionCausesNotF^t)\,.$$

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

$$HaveArrow^{t+1} \Leftrightarrow (HaveArrow^t \land \neg Shoot^t)\,. \tag{7.2}$$

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

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

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

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

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

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

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

Solving the representational and inferential frame problems is a big step forward, but a pernicious problem remains: we need to confirm that *all* the necessary preconditions of an action hold for it to have its intended effect. We said that the *Forward* action moves the agent ahead unless there is a wall in the way, but there are many other unusual exceptions that could cause the action to fail: the agent might trip and fall, be stricken with a heart attack, be carried away by giant bats, etc. Specifying all these exceptions is called the **qualification problem**. There is no complete solution within logic; system designers have to use good judgment in deciding how detailed they want to be in specifying their model, and what details they want to leave out. We will see in Chapter 12 that probability theory allows us to summarize all the exceptions without explicitly naming them.

Qualification problem

7.7.2 A hybrid agent

The ability to deduce various aspects of the state of the world can be combined fairly straightforwardly with condition–action rules (see Section 2.4.2) and with problem-solving algorithms from Chapters 3 and 4 to produce a **hybrid agent** for the wumpus world. Figure 7.20 shows one possible way to do this. The agent program maintains and updates a knowledge base as well as a current plan. The initial knowledge base contains the *atemporal* axioms—those that don't depend on t, such as the axiom relating the breeziness of squares to the presence of pits. At each time step, the new percept sentence is added along with all the axioms that depend on t, such as the successor-state axioms. (The next section explains why the agent doesn't need axioms for *future* time steps.) Then, the agent uses logical inference, by ASKing questions of the knowledge base, to work out which squares are safe and which have yet to be visited.

Hybrid agent

The main body of the agent program constructs a plan based on a decreasing priority of goals. First, if there is a glitter, the program constructs a plan to grab the gold, follow a route back to the initial location, and climb out of the cave. Otherwise, if there is no current plan, the program plans a route to the closest safe square that it has not visited yet, making sure the route goes through only safe squares.

Route planning is done with A* search, not with ASK. If there are no safe squares to explore, the next step—if the agent still has an arrow—is to try to make a safe square by shooting at one of the possible wumpus locations. These are determined by asking where $\text{ASK}(KB, \neg W_{x,y})$ is false—that is, where it is *not* known that there is *not* a wumpus. The function PLAN-SHOT (not shown) uses PLAN-ROUTE to plan a sequence of actions that will line up this shot. If this fails, the program looks for a square to explore that is not provably unsafe—that is, a square for which $\text{ASK}(KB, \neg OK^t_{x,y})$ returns false. If there is no such square, then the mission is impossible and the agent retreats to $[1,1]$ and climbs out of the cave.

7.7.3 Logical state estimation

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

Caching

As we saw in Section 4.4, the history of percepts and all their ramifications can be replaced by the **belief state**—that is, some representation of the set of all possible current states

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

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  TELL the KB the temporal "physics" sentences for time t
  safe ← {[x, y] : ASK(KB, $OK^t_{x,y}$) = true}
  if ASK(KB, $Glitter^t$) = true then
    plan ← [Grab] + PLAN-ROUTE(current, {[1,1]}, safe) + [Climb]
  if plan is empty then
    unvisited ← {[x, y] : ASK(KB, $L^{t'}_{x,y}$) = false for all $t' \leq t$}
    plan ← PLAN-ROUTE(current, unvisited ∩ safe, safe)
  if plan is empty and ASK(KB, $HaveArrow^t$) = true then
    possible_wumpus ← {[x, y] : ASK(KB, ¬ $W_{x,y}$) = false}
    plan ← PLAN-SHOT(current, possible_wumpus, safe)
  if plan is empty then          // no choice but to take a risk
    not_unsafe ← {[x, y] : ASK(KB, ¬ $OK^t_{x,y}$) = false}
    plan ← PLAN-ROUTE(current, unvisited ∩ not_unsafe, safe)
  if plan is empty then
    plan ← PLAN-ROUTE(current, {[1, 1]}, safe) + [Climb]
  action ← POP(plan)
  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t ← t + 1
  return action

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

  problem ← ROUTE-PROBLEM(current, goals,allowed)
  return SEARCH(problem)        // Any search algorithm from Chapter 3
```

Figure 7.20 A hybrid agent program for the wumpus world. It uses a propositional knowledge base to infer the state of the world, and a combination of problem-solving search and domain-specific code to choose actions. Each time HYBRID-WUMPUS-AGENT is called, it adds the percept to the knowledge base, and then either relies on a previously-defined plan or creates a new plan, and pops off the first step of the plan as the action to do next.

of the world.[13] The process of updating the belief state as new percepts arrive is called **state estimation** (see page 132). Whereas in Section 4.4 the belief state was an explicit list of states, here we can use a logical sentence involving the proposition symbols associated with the current time step, as well as the atemporal symbols. For example, the logical sentence

$$WumpusAlive^1 \land L^1_{2,1} \land B_{2,1} \land (P_{3,1} \lor P_{2,2}) \tag{7.4}$$

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

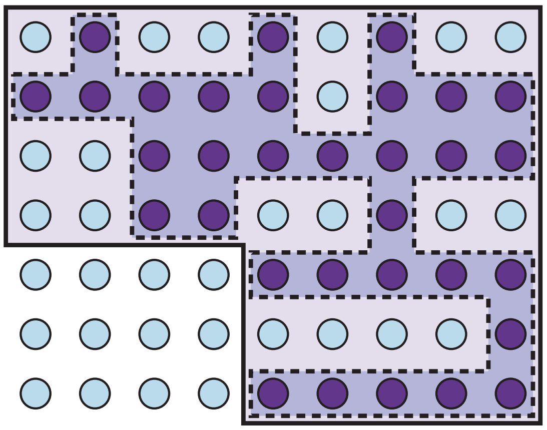


Figure 7.21 Depiction of a 1-CNF belief state (bold outline) as a simply representable, conservative approximation to the exact (wiggly) belief state (shaded region with dashed outline). Each possible world is shown as a circle; the shaded ones are consistent with all the percepts.

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

Maintaining an exact belief state as a logical formula turns out not to be easy. If there are n fluent symbols for time t, then there are 2^n possible states—that is, assignments of truth values to those symbols. Now, the set of belief states is the powerset (set of all subsets) of the set of physical states. There are 2^n physical states, hence 2^{2^n} belief states. Even if we used the most compact possible encoding of logical formulas, with each belief state represented by a unique binary number, we would need numbers with $\log_2(2^{2^n}) = 2^n$ bits to label the current belief state. That is, exact state estimation may require logical formulas whose size is exponential in the number of symbols.

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

It is important to understand that this scheme may lose some information as time goes along. For example, if the sentence in Equation (7.4) were the true belief state, then neither $P_{3,1}$ nor $P_{2,2}$ would be provable individually and neither would appear in the 1-CNF belief state. (Exercise 7.HYBR explores one possible solution to this problem.) On the other hand, because every literal in the 1-CNF belief state is proved from the previous belief state, and the initial belief state is a true assertion, we know that the entire 1-CNF belief state must be true. Thus *the set of possible states represented by the 1-CNF belief state includes all states that are in fact possible given the full percept history.* As illustrated in Figure 7.21, the 1-CNF belief state acts as a simple outer envelope, or **conservative approximation**, around the exact belief state. We see this idea of conservative approximations to complicated sets as a recurring theme in many areas of AI.

Conservative approximation

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

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

Figure 7.22 The SATPLAN algorithm. The planning problem is translated into a CNF sentence in which the goal is asserted to hold at a fixed time step t and axioms are included for each time step up to t. If the satisfiability algorithm finds a model, then a plan is extracted by looking at those proposition symbols that refer to actions and are assigned *true* in the model. If no model exists, then the process is repeated with the goal moved one step later.

7.7.4 Making plans by propositional inference

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

1. Construct a sentence that includes
   (a) $Init^0$, a collection of assertions about the initial state;
   (b) $Transition^1, \ldots, Transition^t$, the successor-state axioms for all possible actions at each time up to some maximum time t;
   (c) the assertion that the goal is achieved at time t: $HaveGold^t \wedge ClimbedOut^t$.
2. Present the whole sentence to a SAT solver. If the solver finds a satisfying model, then the goal is achievable; if the sentence is unsatisfiable, then the problem is unsolvable.
3. Assuming a model is found, extract from the model those variables that represent actions and are assigned *true*. Together they represent a plan to achieve the goals.

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

The key step in using SATPLAN is the construction of the knowledge base. It might seem, on casual inspection, that the wumpus world axioms in Section 7.7.1 suffice for steps 1(a) and 1(b) above. There is, however, a significant difference between the requirements for entailment (as tested by ASK) and those for satisfiability.

Consider, for example, the agent's location, initially $[1,1]$, and suppose the agent's unambitious goal is to be in $[2,1]$ at time 1. The initial knowledge base contains $L^0_{1,1}$ and the goal is $L^1_{2,1}$. Using ASK, we can prove $L^1_{2,1}$ if $Forward^0$ is asserted, and, reassuringly, we cannot prove $L^1_{2,1}$ if, say, $Shoot^0$ is asserted instead. Now, SATPLAN will find the plan $[Forward^0]$; so far, so good.

Unfortunately, SATPLAN also finds the plan $[Shoot^0]$. How could this be? To find out, we inspect the model that SATPLAN constructs: it includes the assignment $L^0_{2,1}$, that is, the agent can be in $[2,1]$ at time 1 by being there at time 0 and shooting. One might ask, "Didn't we say the agent is in $[1,1]$ at time 0?" Yes, we did, but we didn't tell the agent that it can't be in two places at once! For entailment, $L^0_{2,1}$ is unknown and cannot, therefore, be used in a proof; for satisfiability, on the other hand, $L^0_{2,1}$ is unknown and can, therefore, be set to whatever value helps to make the goal true.

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

SATPLAN has more surprises in store, however. The first is that it finds models with impossible actions, such as shooting with no arrow. To understand why, we need to look more carefully at what the successor-state axioms (such as Equation (7.3)) say about actions whose preconditions are not satisfied. The axioms *do* predict correctly that nothing will happen when such an action is executed (see Exercise 7.SATP), but they do *not* say that the action cannot be executed! To avoid generating plans with illegal actions, we must add **precondition axioms** stating that an action occurrence requires the preconditions to be satisfied.[14] For example, we need to say, for each time t, that

Precondition axioms

$$Shoot^t \Rightarrow HaveArrow^t .$$

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

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

Action exclusion axiom

$$\neg A^t_i \vee \neg A^t_j .$$

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

To summarize, SATPLAN finds models for a sentence containing the initial state, the goal, the successor-state axioms, the precondition axioms, and the action exclusion axioms. It can be shown that this collection of axioms is sufficient, in the sense that there are no longer any spurious "solutions." Any model satisfying the propositional sentence will be a

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

valid plan for the original problem. Modern SAT-solving technology makes the approach quite practical. For example, a DPLL-style solver has no difficulty in generating the solution for the wumpus world instance shown in Figure 7.2.

This section has described a declarative approach to agent construction: the agent works by a combination of asserting sentences in the knowledge base and performing logical inference. This approach has some weaknesses hidden in phrases such as "for each time t" and "for each square $[x,y]$." For any practical agent, these phrases have to be implemented by code that generates instances of the general sentence schema automatically for insertion into the knowledge base. For a wumpus world of reasonable size—one comparable to a smallish computer game—we might need a 100×100 board and 1000 time steps, leading to knowledge bases with tens or hundreds of millions of sentences.

Not only does this become rather impractical, but it also illustrates a deeper problem: we know something about the wumpus world—namely, that the "physics" works the same way across all squares and all time steps—that we cannot express directly in the language of propositional logic. To solve this problem, we need a more expressive language, one in which phrases like "for each time t" and "for each square $[x,y]$" can be written in a natural way. First-order logic, described in Chapter 8, is such a language; in first-order logic a wumpus world of any size and duration can be described in about ten logic sentences rather than ten million or ten trillion.

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 $\alpha \land \neg\beta$.
- Inference is the process of deriving new sentences from old ones. **Sound** inference algorithms derive *only* sentences that are entailed; **complete** algorithms derive *all* sentences that are entailed.
- **Propositional logic** is a simple language consisting of **proposition symbols** and **logical connectives**. It can handle propositions that are known to be true, known to be false, or completely unknown.

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

Bibliographical and Historical Notes

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

Logic itself had its origins in ancient Greek philosophy and mathematics. Plato discussed the syntactic structure of sentences, their truth and falsity, their meaning, and the validity of logical arguments. The first known systematic study of logic was Aristotle's Organon. His **syllogisms** were what we now call inference rules, although they lacked the compositionality of our current rules. Syllogism

The Megarian and Stoic schools began the systematic study of the basic logical connectives in the fifth century BCE. Truth tables are 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 222) and were clearer about proof than was Aristotle (Mates, 1953).

The idea of reducing logical inference to a purely mechanical process is due to Wilhelm Leibniz (1646–1716). 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 it didn't handle all of propositional logic, other mathematicians soon filled in the missing pieces. 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 the Stanhope Demonstrator, constructed by the third Earl of Stanhope (1753–1816). William Stanley Jevons, one of the mathematicians who extended Boole's work, constructed his "logical piano" in 1869 to do inferences in Boolean logic. An entertaining history of these early mechanical inference devices is given by Martin Gardner (1968). The first computer programs for logical inference were Martin Davis's 1954 program for proofs in Presburger arithmetic (Davis, 1957), and the Logic Theorist of Newell, Shaw, and Simon (1957).

Emil Post (1921) and Ludwig Wittgenstein (1922) independently used truth tables as a method of testing validity of propositional logic sentences. The Davis–Putnam algorithm (Davis and Putnam, 1960) was the first algorithm for propositional resolution, and the improved DPLL backtracking algorithm (Davis *et al.*, 1962) proved to be more efficient. The resolution rule and a proof of its completeness were developed in full generality for first-order logic by J. A. Robinson (1965).

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

Early investigations showed that DPLL has polynomial average-case complexity for certain natural distributions of problems. Even better, Franco and Paull (1983) showed that the same problems could be solved in *constant* time simply by guessing random assignments. Motivated by the empirical success of local search, 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. Schöning (1999) exhibited a randomized hill-climbing algorithm whose *worst-case* expected run time on 3-SAT problems is $O(1.333^n)$—still exponential, but substantially faster than previous worst-case bounds. The current record is $O(1.32216^n)$ (Rolf, 2006).

Efficiency gains in propositional solvers have been rapid. Given ten minutes of computing time, the original DPLL algorithm on 1962 hardware could solve only problems with 10 or 15 variables (on a 2019 laptop it would be about 30 variables). By 1995 the SATZ solver (Li and Anbulagan, 1997) could handle 1,000 variables, thanks to optimized data structures for indexing variables. Two crucial contributions were the **watched literal** indexing technique of Zhang and Stickel (1996), which makes unit propagation very efficient, and the introduction of clause (i.e., constraint) learning techniques from the CSP community by Bayardo and Schrag (1997). Using these ideas, and spurred by the prospect of solving industrial-scale circuit verification problems, Moskewicz *et al.* (2001) developed the CHAFF solver, which could handle problems with millions of variables. Beginning in 2002, annual SAT competitions have been held; most of the winning entries have been variants of CHAFF. The landscape of solvers is surveyed by Gomes *et al.* (2008).

Watched literal

Local search algorithms for satisfiability were tried by various authors throughout the 1980s, 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 this chapter is due to Selman *et al.* (1996).

The "phase transition" in satisfiability of random *k*-SAT problems was first observed by Simon and Dubois (1989) and has given rise to a great deal of theoretical and empirical research—due, in part, to the connection to phase transition phenomena in statistical physics. Crawford and Auton (1993) located the 3-SAT transition at a clause/variable ratio of around 4.26, noting that this coincides with a sharp peak in the run time of their SAT solver. Cook and Mitchell (1997) provide an excellent summary of the early literature on the problem. Algorithms such as **survey propagation** (Parisi and Zecchina, 2002; Maneva *et al.*, 2007) take advantage of special properties of random SAT instances near the satisfiability threshold and greatly outperform general SAT solvers on such instances. The current state of theoretical understanding is summarized by Achlioptas (2009).

Survey propagation

Good sources for information on satisfiability, both theoretical and practical, include the *Handbook of Satisfiability* (Biere *et al.*, 2009), Donald Knuth's (2015) fascicle on satisfiability, and the regular *International Conferences on Theory and Applications of Satisfiability Testing*, known as SAT.

The idea of building agents with propositional logic can be traced back to the seminal paper of McCulloch and Pitts (1943), which is well known for initiating the field of neural networks, but actually was concerned with the implementation of a Boolean circuit-based agent design in the brain. Stan Rosenschein (Rosenschein, 1985; Kaelbling and Rosenschein, 1990) developed ways to compile circuit-based agents from declarative descriptions of the task environment. Rod Brooks (1986, 1989) demonstrates the effectiveness of circuit-based designs for controlling robots (see Chapter 26). 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, both reasoning and circuits are necessary. Williams *et al.* (2003) describe a hybrid agent—not too different from our wumpus agent—that controls NASA spacecraft, planning sequences of actions and diagnosing and recovering from faults.

The general problem of keeping track of a partially observable environment was introduced for state-based representations in Chapter 4. Its instantiation for propositional representations was studied by Amir and Russell (2003), who identified several classes of environments that admit efficient state-estimation algorithms and showed that for several other classes the problem is intractable. The **temporal-projection** problem, which involves determining what propositions hold true after an action sequence is executed, can be seen as a special case of state estimation with empty percepts. Many authors have studied this problem because of its importance in planning; some important hardness results were established by Liberatore (1997). The idea of representing a belief state with propositions can be traced to Wittgenstein (1922).

Temporal-projection

The approach to logical state estimation using temporal indexes on propositional variables was proposed by Kautz and Selman (1992). Later generations of SATPLAN were able to take advantage of the advances in SAT solvers and remain among the most effective ways of solving difficult planning problems (Kautz, 2006).

The **frame problem** was first recognized by McCarthy and Hayes (1969). Many researchers considered the problem unsolvable within first-order logic, and it spurred a great deal of research into nonmonotonic logics. Philosophers from Dreyfus (1972) to Crockett

(1994) have cited the frame problem as one symptom of the inevitable failure of the entire AI enterprise. The solution of the frame problem with successor-state axioms is due to Ray Reiter (1991). Thielscher (1999) identifies the inferential frame problem as a separate idea and provides a solution. In retrospect, one can see that Rosenschein's (1985) agents were using circuits that implemented successor-state axioms, but Rosenschein did not notice that the frame problem was thereby largely solved.

Modern propositional solvers have been applied to a variety of industrial applications, such as the synthesis of computer hardware (Nowick *et al.*, 1993). The SATMC satisfiability checker was used to detect a previously unknown vulnerability in a Web browser sign-on protocol (Armando *et al.*, 2008).

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

CHAPTER 8

FIRST-ORDER LOGIC

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

Propositional logic sufficed to illustrate the basic concepts of logic, inference, and knowledge-based agents. Unfortunately, propositional logic is limited in what it can say. In this chapter, we examine **first-order logic**,[1] which can concisely represent much more. We begin in Section 8.1 with a discussion of representation languages in general; Section 8.2 covers the syntax and semantics of first-order logic; Sections 8.3 and 8.4 illustrate the use of first-order logic for simple representations.

First-order logic

8.1 Representation Revisited

In this section, we discuss the nature of representation languages. Programming languages (such as C++ or Java or Python) are the largest class of formal languages in common use. Data structures within programs can be used to represent facts; for example, a program could use a 4×4 array to represent the contents of the wumpus world. Thus, the programming language statement $World[2,2] \leftarrow Pit$ is a fairly natural way to assert that there is a pit in square [2,2]. Putting together a string of such statements is sufficient for running a simulation of the wumpus world.

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

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

Propositional logic is a declarative language because its semantics is based on a truth relation between sentences and possible worlds. It also has sufficient expressive power to deal with partial information, using disjunction and negation. Propositional logic has a third property that is desirable in representation languages, namely, **compositionality**. In a compositional language, the meaning of a sentence is a function of the meaning of its parts. For

Compositionality

[1] First-order logic is also called **first-order predicate calculus**; it may be abbreviated as **FOL** or **FOPC**.

example, the meaning of "$S_{1,4} \wedge S_{1,2}$" is related to the meanings of "$S_{1,4}$" and "$S_{1,2}$." It would be very strange if "$S_{1,4}$" meant that there is a stench in square [1,4] and "$S_{1,2}$" meant that there is a stench in square [1,2], but "$S_{1,4} \wedge S_{1,2}$" meant that France and Poland drew 1–1 in last week's ice hockey qualifying match.

However, propositional logic, as a factored representation, lacks the expressive power to *concisely* describe an environment with many objects. For example, we were forced to write a separate rule about breezes and pits for each square, such as

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

In English, on the other hand, it seems easy enough to say, once and for all, "Squares adjacent to pits are breezy." The syntax and semantics of English make it possible to describe the environment concisely: English, like first-order logic, is a structured representation.

8.1.1 The language of thought

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

The modern view of natural language is that it serves as a medium for *communication* rather than pure representation. When a speaker points and says, "Look!" the listener comes to know that, say, Superman has finally appeared over the rooftops. Yet we would not want to say that the sentence "Look!" represents that fact. Rather, the meaning of the sentence depends both on the sentence itself and on the *context* in which the sentence was spoken. Clearly, one could not store a sentence such as "Look!" in a knowledge base and expect to recover its meaning without also storing a representation of the context—which raises the question of how the context itself can be represented.

Natural languages also suffer from *ambiguity*, a problem for a representation language. As Pinker (1995) puts it: "When people think about *spring*, surely they are not confused as to whether they are thinking about a season or something that goes *boing*—and if one word can correspond to two thoughts, thoughts can't be words."

The famous **Sapir–Whorf hypothesis** Whorf (1956) claims that our understanding of the world is strongly influenced by the language we speak. It is certainly true that different speech communities divide up the world differently. The French have two words "chaise" and "fauteuil," for a concept that English speakers cover with one: "chair." But English speakers can easily recognize the category *fauteuil* and give it a name—roughly "open-arm chair"—so does language really make a difference? Whorf relied mainly on intuition and speculation, and his ideas have been largely dismissed, but in the intervening years we actually have real data from anthropological, psychological, and neurological studies.

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

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

"This section covers the topic of knowledge representation languages . . ."

Wanner (1974) did a similar experiment and found that subjects made the right choice at chance level—about 50% of the time—but remembered the content of what they read with better than 90% accuracy. This suggests that people interpret the words they read and form an internal *nonverbal* representation, and that the exact words are not consequential.

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

Language also seems to influence thought through seemingly arbitrary grammatical features such as the gender of nouns. For example, "bridge" is masculine in Spanish and feminine in German. Boroditsky (2003) asked subjects to choose English adjectives to describe a photograph of a particular bridge. Spanish speakers chose *big*, *dangerous*, *strong*, and *towering*, whereas German speakers chose *beautiful*, *elegant*, *fragile*, and *slender*.

Words can serve as anchor points that affect how we perceive the world. Loftus and Palmer (1974) showed experimental subjects a movie of an auto accident. Subjects who were asked "How fast were the cars going when they contacted each other?" reported an average of 32 mph, while subjects who were asked the question with the word "smashed" instead of "contacted" reported 41mph for the same cars in the same movie. Overall, there are measurable but small differences in cognitive processing by speakers of different languages, but no convincing evidence that this leads to a major difference in world view.

In a logical reasoning system that uses conjunctive normal form (CNF), we can see that the linguistic forms "$\neg(A \lor B)$" and "$\neg A \land \neg B$" are the same because we can look inside the system and see that the two sentences are stored as the same canonical CNF form. It is starting to become possible to do something similar with the human brain. Mitchell *et al.* (2008) put subjects in an functional magnetic resonance imaging (fMRI) machine, showed them words such as "celery," and imaged their brains. A machine learning program trained on (word, image) pairs was able to predict correctly 77% of the time on binary choice tasks (e.g., "celery" or "airplane"). The system can even predict at above-chance levels for words it has never seen an fMRI image of before (by considering the images of related words) and for people it has never seen before (proving that fMRI reveals some level of common representation across people). This type of work is still in its infancy, but fMRI (and other imaging technology such as intracranial electrophysiology (Sahin *et al.*, 2009)) promises to give us much more concrete ideas of what human knowledge representations are like.

From the viewpoint of formal logic, representing the same knowledge in two different ways makes absolutely no difference; the same facts will be derivable from either representation. In practice, however, one representation might require fewer steps to derive a conclusion, meaning that a reasoner with limited resources could get to the conclusion using one representation but not the other. For *nondeductive* tasks such as learning from experience, outcomes are *necessarily* dependent on the form of the representations used. We show in Chapter 19 that when a learning program considers two possible theories of the world, both of which are consistent with all the data, the most common way of breaking the tie is to choose the most succinct theory—and that depends on the language used to represent theories. Thus, the influence of language on thought is unavoidable for any agent that does learning.

8.1.2 Combining the best of formal and natural languages

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

Object

Relation

Function

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

Property

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

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

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

Ontological commitment

The primary difference between propositional and first-order logic lies in the **ontological commitment** made by each language—that is, what it assumes about the nature of *reality*. Mathematically, this commitment is expressed through the nature of the formal models with respect to which the truth of sentences is defined. For example, propositional logic assumes that there are facts that either hold or do not hold in the world. Each fact can be in one of two states—true or false—and each model assigns *true* or *false* to each proposition symbol (see Section 7.4.2). First-order logic assumes more; namely, that the world consists of objects with certain relations among them that do or do not hold. (See Figure 8.1.) The formal models are correspondingly more complicated than those for propositional logic.

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

Figure 8.1 Formal languages and their ontological and epistemological commitments.

This ontological commitment is a great strength of logic (both propositional and first-order), because it allows us to start with true statements and infer other true statements. It is especially powerful in domains where every proposition has clear boundaries, such as mathematics or the wumpus world, where a square either does or doesn't have a pit; there is no possibility of a square with a vaguely pit-like indentation. But in the real world, many propositions have vague boundaries: Is Vienna a large city? Does this restaurant serve delicious food? Is that person tall? It depends who you ask, and their answer might be "kind of."

One response is to refine the representation: if a crude line dividing cities into "large" and "not large" leaves out too much information for the application in question, then one can increase the number of size categories or use a *Population* function symbol. Another proposed solution comes from **Fuzzy logic**, which makes the ontological commitment that propositions have a **degree of truth** between 0 and 1. For example, the sentence "Vienna is a large city" might be true to degree 0.8 in fuzzy logic, while "Paris is a large city" might be true to degree 0.9. This corresponds better to our intuitive conception of the world, but it makes it harder to do inference: instead of one rule to determine the truth of $A \land B$, fuzzy logic needs different rules depending on the domain. Another possibility, covered in Section 24.1, is to assign each concept to a point in a multidimensional space, and then measure the distance between the concept "large city" and the concept "Vienna" or "Paris."

Fuzzy logic

Degree of truth

Various special-purpose logics make still further ontological commitments; for example, **temporal logic** assumes that facts hold at particular *times* and that those times (which may be points or intervals) are ordered. Thus, special-purpose logics give certain kinds of objects (and the axioms about them) "first class" status within the logic, rather than simply defining them within the knowledge base. **Higher-order logic** views the relations and functions referred to by first-order logic as objects in themselves. This allows one to make assertions about *all* relations—for example, one could wish to define what it means for a relation to be transitive. Unlike most special-purpose logics, higher-order logic is strictly more expressive than first-order logic, in the sense that some sentences of higher-order logic cannot be expressed by any finite number of first-order logic sentences.

Temporal logic

Higher-order logic

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

Epistemological commitment

Systems using **probability theory**, on the other hand, can have any *degree of belief*, or *subjective likelihood*, ranging from 0 (total disbelief) to 1 (total belief). It is important not to confuse the degree of belief in probability theory with the degree of truth in fuzzy logic. Indeed, some fuzzy systems allow uncertainty (degree of belief) about degrees of truth. For example, a probabilistic wumpus-world agent might believe that the wumpus is in [1,3] with probability 0.75 and in [2, 3] with probability 0.25 (although the wumpus is definitely in one particular square).

8.2 Syntax and Semantics of First-Order Logic

We begin this section by specifying more precisely the way in which the possible worlds of first-order logic reflect the ontological commitment to objects and relations. Then we introduce the various elements of the language, explaining their semantics as we go along. The main points are how the language facilitates concise representations and how its semantics leads to sound reasoning procedures.

8.2.1 Models for first-order logic

Chapter 7 said that the models of a logical language are the formal structures that constitute the possible worlds under consideration. Each model links the vocabulary of the logical sentences to elements of the possible world, so that the truth of any sentence can be determined. Thus, models for propositional logic link proposition symbols to predefined truth values.

Models for first-order logic are much more interesting. First, they have objects in them! The **domain** of a model is the set of objects or **domain elements** it contains. The domain is required to be *nonempty*—every possible world must contain at least one object. (See Exercise 8.EMPT for a discussion of empty worlds.) Mathematically speaking, it doesn't matter *what* these objects are—all that matters is *how many* there are in each particular model—but for pedagogical purposes we'll use a concrete example. Figure 8.2 shows a model with five objects: Richard the Lionheart, King of England from 1189 to 1199; his younger brother, the evil King John, who ruled from 1199 to 1215; the left legs of Richard and John; and a crown.

Domain

Domain elements

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

Tuple

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

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

Certain kinds of relationships are best considered as functions, in that a given object must be related to exactly one object in this way. For example, each person has one left leg, so the model has a unary "left leg" function—a mapping from a one-element tuple to an object—that

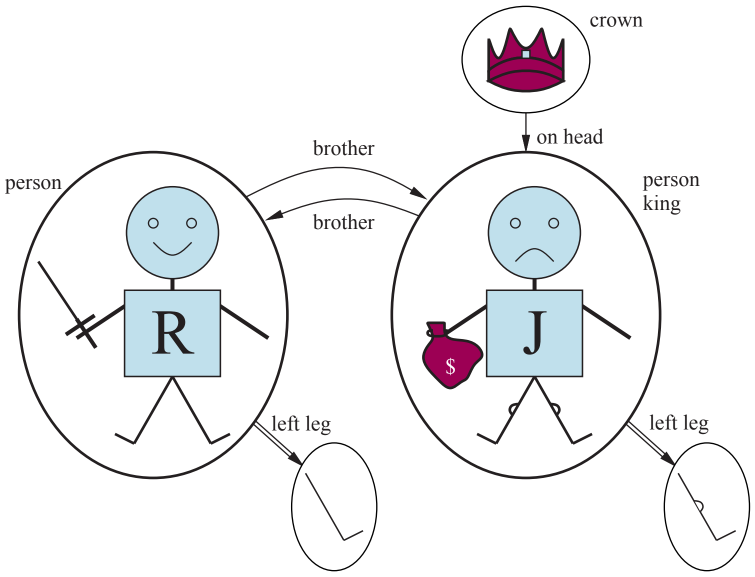


Figure 8.2 A model containing five objects, two binary relations (brother and on-head), three unary relations (person, king, and crown), and one unary function (left-leg).

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

$$
\begin{array}{rcl}
\mathit{Sentence} & \rightarrow & \mathit{AtomicSentence} \mid \mathit{ComplexSentence} \\
\mathit{AtomicSentence} & \rightarrow & \mathit{Predicate} \mid \mathit{Predicate}(\mathit{Term}, \ldots) \mid \mathit{Term} = \mathit{Term} \\
\mathit{ComplexSentence} & \rightarrow & (\, \mathit{Sentence} \,) \\
 & \mid & \neg\, \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \land \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \lor \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \;\Rightarrow\; \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \;\Leftrightarrow\; \mathit{Sentence} \\
 & \mid & \mathit{Quantifier}\ \mathit{Variable}, \ldots\ \mathit{Sentence} \\
 & & \\
\mathit{Term} & \rightarrow & \mathit{Function}(\mathit{Term}, \ldots) \\
 & \mid & \mathit{Constant} \\
 & \mid & \mathit{Variable} \\
 & & \\
\mathit{Quantifier} & \rightarrow & \forall \mid \exists \\
\mathit{Constant} & \rightarrow & A \mid X_1 \mid \mathit{John} \mid \cdots \\
\mathit{Variable} & \rightarrow & a \mid x \mid s \mid \cdots \\
\mathit{Predicate} & \rightarrow & \mathit{True} \mid \mathit{False} \mid \mathit{After} \mid \mathit{Loves} \mid \mathit{Raining} \mid \cdots \\
\mathit{Function} & \rightarrow & \mathit{Mother} \mid \mathit{LeftLeg} \mid \cdots \\
\textsc{Operator Precedence} & : & \neg, =, \land, \lor, \Rightarrow, \Leftrightarrow
\end{array}
$$

Figure 8.3 The syntax of first-order logic with equality, specified in Backus–Naur form (see page 1030 if you are not familiar with this notation). Operator precedences are specified, from highest to lowest. The precedence of quantifiers is such that a quantifier holds over everything to the right of it.

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

Interpretation

Intended interpretation

- *Richard* refers to Richard the Lionheart and *John* refers to the evil King John.
- *Brother* refers to the brotherhood relation—that is, the set of tuples of objects given in Equation (8.1); *OnHead* is a relation that holds between the crown and King John; *Person*, *King*, and *Crown* are unary relations that identify persons, kings, and crowns.
- *LeftLeg* refers to the "left leg" function as defined in Equation (8.2).

There are many other possible interpretations, of course. For example, one interpretation maps *Richard* to the crown and *John* to King John's left leg. There are five objects in the model, so there are 25 possible interpretations just for the constant symbols *Richard* and *John*.

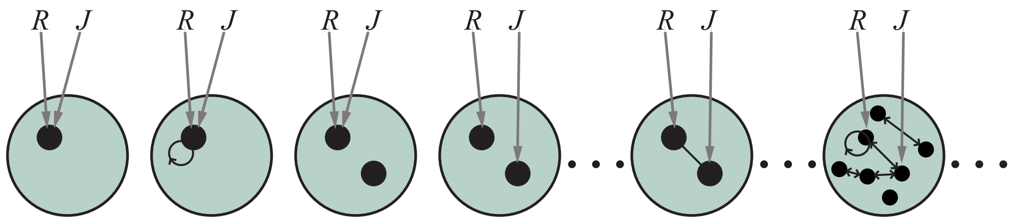


Figure 8.4 Some members of the set of all models for a language with two constant symbols, R and J, and one binary relation symbol. The interpretation of each constant symbol is shown by a gray arrow. Within each model, the related objects are connected by arrows.

Notice that not all the objects need have a name—for example, the intended interpretation does not name the crown or the legs. It is also possible for an object to have several names; there is an interpretation under which both *Richard* and *John* refer to the crown.[2] If you find this possibility confusing, remember that, in propositional logic, it is perfectly possible to have a model in which *Cloudy* and *Sunny* are both true; it is the job of the knowledge base to rule out models that are inconsistent with our knowledge.

In summary, a model in first-order logic consists of a set of objects and an interpretation that maps constant symbols to objects, function symbols to functions on those objects, and predicate symbols to relations. Just as with propositional logic, entailment, validity, and so on are defined in terms of *all possible models*. To get an idea of what the set of all possible models looks like, see Figure 8.4. It shows that models vary in how many objects they contain—from one to infinity—and in the way the constant symbols map to objects.

Because the number of first-order models is unbounded, we cannot check entailment by enumerating them all (as we did for propositional logic). Even if the number of objects is restricted, the number of combinations can be very large. (See Exercise 8.MCNT.) For the example in Figure 8.4, there are 137,506,194,466 models with six or fewer objects.

8.2.3 Terms

Term

A **term** is a logical expression that refers to an object. Constant symbols are terms, but it is not always convenient to have a distinct symbol to name every object. In English we might use the expression "King John's left leg" rather than giving a name to his leg. This is what function symbols are for: instead of using a constant symbol, we use $LeftLeg(John)$.[3]

In the general case, a complex term is formed by a function symbol followed by a parenthesized list of terms as arguments to the function symbol. It is important to remember that a complex term is just a complicated kind of name. It is not a "subroutine call" that "returns a value." There is no *LeftLeg* subroutine that takes a person as input and returns a leg. We can reason about left legs (e.g., stating the general rule that everyone has one and then deducing

[2] Later, in Section 8.2.8, we examine a semantics in which every object must have exactly one name.

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

that John must have one) without ever providing a definition of *LeftLeg*. This is something that cannot be done with subroutines in programming languages.

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

8.2.4 Atomic sentences

Atomic sentence

Atom

Now that we have terms for referring to objects and predicate symbols for referring to relations, we can combine them to make **atomic sentences** that state facts. An **atomic sentence** (or **atom** for short) is formed from a predicate symbol optionally followed by a parenthesized list of terms, such as

$$Brother(Richard, John).$$

This states, under the intended interpretation given earlier, that Richard the Lionheart is the brother of King John.[4] Atomic sentences can have complex terms as arguments. Thus,

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

states that Richard the Lionheart's father is married to King John's mother (again, under a suitable interpretation).[5]

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

8.2.5 Complex sentences

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

$$\neg Brother(LeftLeg(Richard), John)$$
$$Brother(Richard, John) \land Brother(John, Richard)$$
$$King(Richard) \lor King(John)$$
$$\neg King(Richard) \Rightarrow King(John).$$

8.2.6 Quantifiers

Quantifier

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

Universal quantification ($\forall$)

Recall the difficulty we had in Chapter 7 with the expression of general rules in propositional logic. Rules such as "Squares neighboring the wumpus are smelly" and "All kings

[4] We usually follow the argument-ordering convention that $P(x,y)$ is read as "x is a P of y."

[5] This ontology only recognizes one father and one mother for each person. A more complex ontology could recognize biological mother, birth mother, adoptive mother, etc.

are persons" are the bread and butter of first-order logic. We deal with the first of these in Section 8.3. The second rule, "All kings are persons," is written in first-order logic as

$$\forall x \;\; King(x) \;\Rightarrow\; Person(x)\,.$$

Universal quantifier

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

Variable

Ground term

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

Extended interpretation

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

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

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

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

Let us look carefully at this set of assertions. Since, in our model, King John is the only king, the second sentence asserts that he is a person, as we would hope. But what about the other four sentences, which appear to make claims about legs and crowns? Is that part of the meaning of "All kings are persons"? In fact, the other four assertions are true in the model, but make no claim whatsoever about the personhood qualifications of legs, crowns, or indeed Richard. This is because none of these objects is a king. Looking at the truth table for $\Rightarrow$ (Figure 7.8 on page 219), we see that the implication is true whenever its premise is false—*regardless* of the truth of the conclusion. Thus, by asserting the universally quantified sentence, which is equivalent to asserting a whole list of individual implications, we end up asserting the conclusion of the rule just for those objects for which the premise is true and saying nothing at all about those objects for which the premise is false. Thus, the truth-table definition of $\Rightarrow$ turns out to be perfect for writing general rules with universal quantifiers.

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

$$\forall x \;\; King(x) \land Person(x)$$

would be equivalent to asserting

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

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

Existential quantification ($\exists$)

Existential quantifier

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

$$\exists x \;\; Crown(x) \wedge OnHead(x, John) .$$

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

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

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

The fifth assertion is true in the model, so the original existentially quantified sentence is true in the model. Notice that, by our definition, the sentence would also be true in a model in which King John was wearing two crowns. This is entirely consistent with the original sentence "King John has a crown on his head." [6]

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

$$\exists x \;\; Crown(x) \;\Rightarrow\; OnHead(x, John) .$$

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

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

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

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

Nested quantifiers

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

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

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

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

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

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

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

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

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

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

$$\forall x\ (\mathit{Crown}(x) \lor (\exists x\ \mathit{Brother}(\mathit{Richard},x)))\,.$$

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

Connections between $\forall$ and $\exists$

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

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

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

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

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

$$\begin{array}{llllll}
\neg\exists x\ P & \equiv & \forall x\ \neg P & \qquad \neg(P \lor Q) & \equiv & \neg P \land \neg Q \\
\neg\forall x\ P & \equiv & \exists x\ \neg P & \qquad \neg(P \land Q) & \equiv & \neg P \lor \neg Q \\
\forall x\ P & \equiv & \neg\exists x\ \neg P & \qquad P \land Q & \equiv & \neg(\neg P \lor \neg Q) \\
\exists x\ P & \equiv & \neg\forall x\ \neg P & \qquad P \lor Q & \equiv & \neg(\neg P \land \neg Q)\,.
\end{array}$$

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

8.2.7 Equality

Equality symbol

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

$$Father(John) = Henry$$

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

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

$$\exists x, y \;\; Brother(x, Richard) \wedge Brother(y, Richard) \wedge \neg(x = y) .$$

The sentence

$$\exists x, y \;\; Brother(x, Richard) \wedge Brother(y, Richard)$$

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

8.2.8 Database semantics

Continuing the example from the previous section, suppose that we believe that Richard has two brothers, John and Geoffrey.[7] We could write

$$Brother(John, Richard) \wedge Brother(Geoffrey, Richard) , \qquad (8.3)$$

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

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

This logical sentence seems much more cumbersome than the corresponding English sentence. But if we fail to translate the English properly, our logical reasoning system will make mistakes. Can we devise a semantics that allows a more straightforward logical sentence?

Unique-names assumption

Closed-world assumption

Domain closure

One proposal that is very popular in database systems works as follows. First, we insist that every constant symbol refer to a distinct object—the **unique-names assumption**. Second, we assume that atomic sentences not known to be true are in fact false—the **closed-world assumption**. Finally, we invoke **domain closure**, meaning that each model contains no more domain elements than those named by the constant symbols.

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

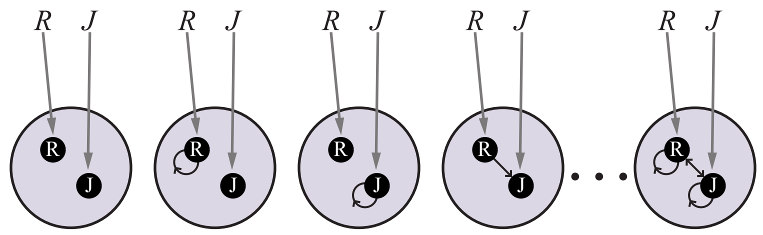


Figure 8.5 Some members of the set of all models for a language with two constant symbols, *R* and *J*, and one binary relation symbol, under database semantics. The interpretation of the constant symbols is fixed, and there is a distinct object for each constant symbol.

Under the resulting semantics, Equation (8.3) does indeed state that Richard has exactly two brothers, John and Geoffrey. We call this **database semantics** to distinguish it from the standard semantics of first-order logic. Database semantics is also used in logic programming systems, as explained in Section 9.4.4.

Database semantics

It is instructive to consider the set of all possible models under database semantics for the same case as shown in Figure 8.4 (page 259). Figure 8.5 shows some of the models, ranging from the model with no tuples satisfying the relation to the model with all tuples satisfying the relation. With two objects, there are four possible two-element tuples, so there are $2^4 = 16$ different subsets of tuples that can satisfy the relation. Thus, there are 16 possible models in all—a lot fewer than the infinitely many models for the standard first-order semantics. On the other hand, the database semantics requires definite knowledge of what the world contains.

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

8.3 Using First-Order Logic

Now that we have defined an expressive logical language, let's learn how to use it. In this section, we provide example sentences in some simple **domains**. In knowledge representation, a domain is just some part of the world about which we wish to express some knowledge.

Domain

We begin with a brief description of the TELL/ASK interface for first-order knowledge bases. Then we look at the domains of family relationships, numbers, sets, and lists, and at the wumpus world. Section 8.4.2 contains a more substantial example (electronic circuits) and Chapter 10 covers everything in the universe.

8.3.1 Assertions and queries in first-order logic

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

Assertion

person, and all kings are persons:

$$\textsc{Tell}(KB, King(John))\,.$$
$$\textsc{Tell}(KB, Person(Richard))\,.$$
$$\textsc{Tell}(KB, \forall x\ King(x) \Rightarrow Person(x))\,.$$

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

$$\textsc{Ask}(KB, King(John))$$

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

$$\textsc{Ask}(KB, Person(John))$$

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

$$\textsc{Ask}(KB, \exists x\ Person(x))\,.$$

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

$$\textsc{AskVars}(KB, Person(x))$$

Substitution
Binding list

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

8.3.2 The kinship domain

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

Clearly, the objects in our domain are people. Unary predicates include *Male* and *Female*, among others. Kinship relations—parenthood, brotherhood, marriage, and so on—are represented by binary predicates: *Parent*, *Sibling*, *Brother*, *Sister*, *Child*, *Daughter*, *Son*, *Spouse*, *Wife*, *Husband*, *Grandparent*, *Grandchild*, *Cousin*, *Aunt*, and *Uncle*. We use functions for *Mother* and *Father*, because every person has exactly one of each of these, biologically (although we could introduce additional functions for adoptive mothers, surrogate mothers, etc.).

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

$$\forall m,c\ \ Mother(c)=m \Leftrightarrow Female(m) \land Parent(m,c)\,.$$

One's husband is one's male spouse:

$$\forall w,h\ \ Husband(h,w) \Leftrightarrow Male(h) \land Spouse(h,w)\,.$$

Parent and child are inverse relations:

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

A grandparent is a parent of one's parent:

$$\forall g,c \;\; Grandparent(g,c) \;\Leftrightarrow\; \exists p \;\; Parent(g,p) \land Parent(p,c) \,.$$

A sibling is another child of one's parent:

$$\forall x,y \;\; Sibling(x,y) \;\Leftrightarrow\; x \neq y \land \exists p \;\; Parent(p,x) \land Parent(p,y) \,.$$

We could go on for several more pages like this, and Exercise 8.KINS asks you to do just that.

Each of these sentences can be viewed as an **axiom** of the kinship domain, as explained in Section 7.1. Axioms are commonly associated with purely mathematical domains—we will see some axioms for numbers shortly—but they are needed in all domains. They provide the basic factual information from which useful conclusions can be derived. Our kinship axioms are also **definitions**; they have the form $\forall x,y \;\; P(x,y) \;\Leftrightarrow\; \ldots$. The axioms define the *Mother* function and the *Husband*, *Male*, *Parent*, *Grandparent*, and *Sibling* predicates in terms of other predicates. Our definitions "bottom out" at a basic set of predicates (*Child*, *Female*, etc.) in terms of which the others are ultimately defined.

Definition

This is a natural way in which to build up the representation of a domain, and it is analogous to the way in which software packages are built up by successive definitions of subroutines from primitive library functions. Notice that there is not necessarily a unique set of primitive predicates; we could equally well have used *Parent* instead of *Child*. In some domains, as we show, there is no clearly identifiable basic set.

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

Theorem

$$\forall x,y \;\; Sibling(x,y) \;\Leftrightarrow\; Sibling(y,x) \,.$$

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

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

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

$$\forall x \;\; Person(x) \;\Leftrightarrow\; \ldots$$

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

$$\begin{aligned} &\forall x \;\; Person(x) \;\Rightarrow\; \ldots \\ &\forall x \;\; \ldots \;\Rightarrow\; Person(x) \,. \end{aligned}$$

Axioms can also be "just plain facts," such as $Male(Jim)$ and $Spouse(Jim, Laura)$. Such facts form the descriptions of specific problem instances, enabling specific questions to be answered. If all goes well, the answers to these questions will then be theorems that follow from the axioms.

Often, one finds that the expected answers are not forthcoming—for example, from $Spouse(Jim, Laura)$ one expects (under the laws of many countries) to be able to infer that $\neg Spouse(George, Laura)$; but this does not follow from the axioms given earlier—even after we add $Jim \neq George$ as suggested in Section 8.2.8. This is a sign that an axiom is missing. Exercise 8.HILL asks the reader to supply it.

8.3.3 Numbers, sets, and lists

Natural numbers

Peano axioms

Numbers are perhaps the most vivid example of how a large theory can be built up from a tiny kernel of axioms. We describe here the theory of **natural numbers** or nonnegative integers. We need a predicate *NatNum* that will be true of natural numbers; we need one constant symbol, 0; and we need one function symbol, S (successor). The **Peano axioms** define natural numbers and addition.[8] Natural numbers are defined recursively:

$$NatNum(0)\,.$$
$$\forall n \;\; NatNum(n) \Rightarrow NatNum(S(n))\,.$$

That is, 0 is a natural number, and for every object n, if n is a natural number, then $S(n)$ is a natural number. So the natural numbers are 0, $S(0)$, $S(S(0))$, and so on. We also need axioms to constrain the successor function:

$$\forall n \;\; 0 \neq S(n)\,.$$
$$\forall m, n \;\; m \neq n \Rightarrow S(m) \neq S(n)\,.$$

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

$$\forall m \;\; NatNum(m) \Rightarrow +(0, m) = m\,.$$
$$\forall m, n \;\; NatNum(m) \land NatNum(n) \Rightarrow +(S(m), n) = S(+(m, n))\,.$$

Infix

Prefix

The first of these axioms says that adding 0 to any natural number m gives m itself. Notice the use of the binary function symbol "+" in the term $+(m, 0)$; in ordinary mathematics, the term would be written $m + 0$ using **infix** notation. (The notation we have used for first-order logic is called **prefix**.) To make our sentences about numbers easier to read, we allow the use of infix notation. We can also write $S(n)$ as $n + 1$, so the second axiom becomes

$$\forall m, n \;\; NatNum(m) \land NatNum(n) \Rightarrow (m+1) + n = (m+n) + 1\,.$$

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

Syntactic sugar

The use of infix notation is an example of **syntactic sugar**, that is, an extension to or abbreviation of the standard syntax that does not change the semantics. Any sentence that uses sugar can be "desugared" to produce an equivalent sentence in ordinary first-order logic. Another example is using square brackets rather than parentheses to make it easier to see what left bracket matches with what right bracket. Yet another example is collapsing quantifiers: replacing $\forall x \; \forall y \; P(x, y)$ with $\forall x, y \; P(x, y)$.

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

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

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

Set

We will use the normal vocabulary of set theory as syntactic sugar. The empty set is a constant written as $\{\}$. There is one unary predicate, *Set*, which is true of sets. The binary predicates are $x \in s$ (x is a member of set s) and $s_1 \subseteq s_2$ (set s_1 is a subset of s_2, possibly equal to s_2). The binary functions are $s_1 \cap s_2$ (intersection), $s_1 \cup s_2$ (union), and $Add(x,s)$ (the set resulting from adding element x to set s). One possible set of axioms is as follows:

1. The only sets are the empty set and those made by adding something to a set:
$$\forall s\ Set(s) \Leftrightarrow (s=\{\}) \lor (\exists x, s_2\ Set(s_2) \land s=Add(x,s_2)) .$$
2. The empty set has no elements added into it. In other words, there is no way to decompose $\{\}$ into a smaller set and an element:
$$\neg\exists x, s\ Add(x,s)=\{\} .$$
3. Adding an element already in the set has no effect:
$$\forall x, s\ x\in s \Leftrightarrow s=Add(x,s) .$$
4. The only members of a set are the elements that were added into it. We express this recursively, saying that x is a member of s if and only if s is equal to some element y added to some set s_2, where either y is the same as x or x is a member of s_2:
$$\forall x, s\ x\in s \Leftrightarrow \exists y, s_2\ (s=Add(y,s_2) \land (x=y \lor x\in s_2)) .$$
5. A set is a subset of another set if and only if all of the first set's members are members of the second set:
$$\forall s_1, s_2\ s_1 \subseteq s_2 \Leftrightarrow (\forall x\ x\in s_1 \Rightarrow x\in s_2) .$$
6. Two sets are equal if and only if each is a subset of the other:
$$\forall s_1, s_2\ (s_1=s_2) \Leftrightarrow (s_1 \subseteq s_2 \land s_2 \subseteq s_1) .$$
7. An object is in the intersection of two sets if and only if it is a member of both sets:
$$\forall x, s_1, s_2\ x\in (s_1 \cap s_2) \Leftrightarrow (x\in s_1 \land x\in s_2) .$$
8. An object is in the union of two sets if and only if it is a member of either set:
$$\forall x, s_1, s_2\ x\in (s_1 \cup s_2) \Leftrightarrow (x\in s_1 \lor x\in s_2) .$$

Lists are similar to sets. The differences are that lists are ordered and the same element can appear more than once in a list. We can use the vocabulary of Lisp for lists: *Nil* is the constant list with no elements; *Cons*, *Append*, *First*, and *Rest* are functions; and *Find* is the predicate that does for lists what *Member* does for sets. *List* is a predicate that is true only of lists. As with sets, it is common to use syntactic sugar in logical sentences involving lists. The empty list is $[\,]$. The term $Cons(x, Nil)$ (i.e., the list containing the element x followed by nothing) is written as $[x]$. A list of several elements, such as $[A,B,C]$, corresponds to the nested term $Cons(A, Cons(B, Cons(C, Nil)))$. Exercise 8.LIST asks you to write out the axioms for lists.

List

8.3.4 The wumpus world

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

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

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

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

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

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

$$\textsc{AskVars}(KB, BestAction(a, 5))\,,$$

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

$$\begin{array}{l}
\forall t, s, g, w, c\ \ Percept([s, Breeze, g, w, c], t) \ \Rightarrow\ Breeze(t) \\
\forall t, s, g, w, c\ \ Percept([s, None, g, w, c], t) \ \Rightarrow\ \neg Breeze(t) \\
\forall t, s, b, w, c\ \ Percept([s, b, Glitter, w, c], t) \ \Rightarrow\ Glitter(t) \\
\forall t, s, b, w, c\ \ Percept([s, b, None, w, c], t) \ \Rightarrow\ \neg Glitter(t)
\end{array}$$

and so on. These rules exhibit a trivial form of the reasoning process called **perception**, which we study in depth in Chapter 25. Notice the quantification over time t. In propositional logic, we would need copies of each sentence for each time step.

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

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

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

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

$$\begin{array}{l}
\forall x, y, a, b\ \ Adjacent([x, y], [a, b]) \ \Leftrightarrow \\
\qquad (x = a \land (y = b - 1 \lor y = b + 1)) \lor (y = b \land (x = a - 1 \lor x = a + 1))\,.
\end{array}$$

We could name each pit, but this would be inappropriate for a different reason: there is no reason to distinguish among pits.[9] It is simpler to use a unary predicate *Pit* that is true of squares containing pits. Finally, since there is exactly one wumpus, a constant *Wumpus* is just as good as a unary predicate (and perhaps more dignified from the wumpus's viewpoint).

The agent's location changes over time, so we write $At(Agent, s, t)$ to mean that the agent is at square s at time t. We can fix the wumpus to a specific location forever with $\forall t\, At(Wumpus, [1,3], t)$. We can then say that objects can be at only one location at a time:

$$\forall x, s_1, s_2, t\;\; At(x, s_1, t) \land At(x, s_2, t) \;\Rightarrow\; s_1 = s_2\,.$$

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

$$\forall s, t\;\; At(Agent, s, t) \land Breeze(t) \;\Rightarrow\; Breezy(s)\,.$$

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

Having discovered which places are breezy (or smelly) and, very importantly, *not* breezy (or *not* smelly), the agent can deduce where the pits are (and where the wumpus is). Whereas propositional logic necessitates a separate axiom for each square (see R_2 and R_3 on page 220) and would need a different set of axioms for each geographical layout of the world, first-order logic just needs one axiom:

$$\forall s\;\; Breezy(s) \;\Leftrightarrow\; \exists r\;\; Adjacent(r, s) \land Pit(r)\,. \tag{8.4}$$

Similarly, in first-order logic we can quantify over time, so we need just one successor-state axiom for each predicate, rather than a different copy for each time step. For example, the axiom for the arrow (Equation (7.2) on page 240) becomes

$$\forall t\;\; HaveArrow(t+1) \;\Leftrightarrow\; (HaveArrow(t) \land \neg Action(Shoot, t))\,.$$

From these two example sentences, we can see that the first-order logic formulation is no less concise than the original English-language description given in Chapter 7. The reader is invited to construct analogous axioms for the agent's location and orientation; in these cases, the axioms quantify over both space and time. As in the case of propositional state estimation, an agent can use logical inference with axioms of this kind to keep track of aspects of the world that are not directly observed. Chapter 11 goes into more depth on the subject of first-order successor-state axioms and their uses for constructing plans.

8.4 Knowledge Engineering in First-Order Logic

The preceding section illustrated the use of first-order logic to represent knowledge in three simple domains. This section describes the general process of knowledge-base construction—a process called **knowledge engineering**. A knowledge engineer is someone who investigates a particular domain, learns what concepts are important in that domain, and creates a formal representation of the objects and relations in the domain. We illustrate the knowledge engineering process in an electronic circuit domain. The approach we take is suitable for developing *special-purpose* knowledge bases whose domain is carefully circumscribed and

Knowledge engineering

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

whose range of queries is known in advance. *General-purpose* knowledge bases, which cover a broad range of human knowledge and are intended to support tasks such as natural language understanding, are discussed in Chapter 10.

8.4.1 The knowledge engineering process

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

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

   Knowledge acquisition

   For the wumpus world, which is defined by an artificial set of rules, the relevant knowledge is easy to identify. (Notice, however, that the definition of adjacency was not supplied explicitly in the wumpus-world rules.) For real domains, the issue of relevance can be quite difficult—for example, a system for simulating VLSI designs might or might not need to take into account stray capacitances and skin effects.
3. *Decide on a vocabulary of predicates, functions, and constants.* That is, translate the important domain-level concepts into logic-level names. This involves many questions of knowledge-engineering *style*. Like programming style, this can have a significant impact on the eventual success of the project. For example, should pits be represented by objects or by a unary predicate on squares? Should the agent's orientation be a function or a predicate? Should the wumpus's location depend on time? Once the choices have been made, the result is a vocabulary that is known as the **ontology** of the domain. The word *ontology* means a particular theory of the nature of being or existence. The ontology determines what kinds of things exist, but does not determine their specific properties and interrelationships.

   Ontology
4. *Encode general knowledge about the domain.* The knowledge engineer writes down the axioms for all the vocabulary terms. This pins down (to the extent possible) the meaning of the terms, enabling the expert to check the content. Often, this step reveals misconceptions or gaps in the vocabulary that must be fixed by returning to step 3 and iterating through the process.
5. *Encode a description of the problem instance.* If the ontology is well thought out, this step is easy. It involves writing simple atomic sentences about instances of concepts that are already part of the ontology. For a logical agent, problem instances are supplied by the sensors, whereas a "disembodied" knowledge base is given sentences in the same way that traditional programs are given input data.

6. *Pose queries to the inference procedure and get answers.* This is where the reward is: we can let the inference procedure operate on the axioms and problem-specific facts to derive the facts we are interested in knowing. Thus, we avoid the need for writing an application-specific solution algorithm.
7. *Debug and evaluate the knowledge base.* Alas, the answers to queries will seldom be correct on the first try. More precisely, the answers will be correct *for the knowledge base as written*, assuming that the inference procedure is sound, but they will not be the ones that the user is expecting. For example, if an axiom is missing, some queries will not be answerable from the knowledge base. A considerable debugging process could ensue. Missing axioms or axioms that are too weak can be easily identified by noticing places where the chain of reasoning stops unexpectedly. For example, if the knowledge base includes a diagnostic rule (see Exercise 8.WUMD) for finding the wumpus,

   $$\forall s \; Smelly(s) \Rightarrow Adjacent(Home(Wumpus), s),$$

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

   $$\forall x \; NumOfLegs(x, 4) \Rightarrow Mammal(x)$$

   is false for reptiles, amphibians, and tables. *The falsehood of this sentence can be determined independently of the rest of the knowledge base.* In contrast, a typical error in a program looks like this:

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

   It is impossible to tell whether `offset` should be `position` or `position + 1` without understanding the surrounding context.

When you get to the point where there are no obvious errors in your knowledge base, it is tempting to declare success. But unless there are obviously no errors, it is better to formally evaluate your system by running it on a test suite of queries and measuring how many you get right. Without objective measurement, it is too easy to convince yourself that the job is done. To understand this seven-step process better, we now apply it to an extended example—the domain of electronic circuits.

8.4.2 The electronic circuits domain

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

Identify the questions

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

Assemble the relevant knowledge

What do we know about digital circuits? For our purposes, they are composed of wires and gates. Signals flow along wires to the input terminals of gates, and each gate produces a signal on the output terminal that flows along another wire. To determine what these signals will be, we need to know how the gates transform their input signals. There are four types of gates: AND, OR, and XOR gates have two input terminals, and NOT gates have one. All gates have one output terminal. Circuits, like gates, have input and output terminals.

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

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

Decide on a vocabulary

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

Next we consider terminals, which are identified by the predicate $Terminal(x)$. A circuit can have one or more input terminals and one or more output terminals. We use the function

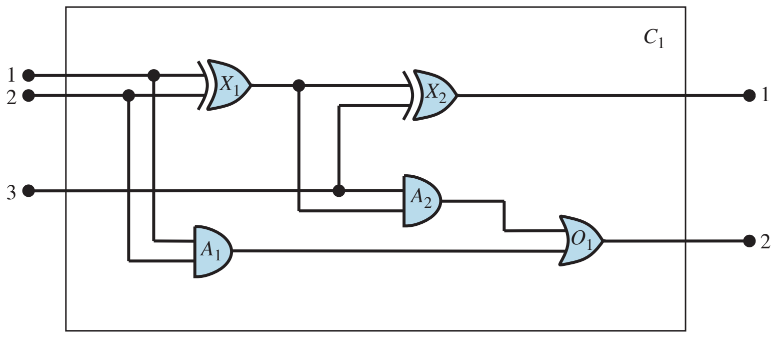


Figure 8.6 A digital circuit C_1, purporting to be a one-bit full adder. The first two inputs are the two bits to be added, and the third input is a carry bit. The first output is the sum, and the second output is a carry bit for the next adder. The circuit contains two XOR gates, two AND gates, and one OR gate.

$In(1,X_1)$ to denote the first input terminal for circuit X_1. A similar function $Out(n,c)$ is used for output terminals. The function $Arity(c,i,j)$ says that circuit c has i input and j output terminals. The connectivity between gates can be represented by a predicate, *Connected*, which takes two terminals as arguments, as in $Connected(Out(1,X_1),In(1,X_2))$.

Finally, we need to know whether a signal is on or off. One possibility is to use a unary predicate, $On(t)$, which is true when the signal at a terminal is on. This makes it a little difficult, however, to pose questions such as "What are all the possible values of the signals at the output terminals of circuit C_1 ?" We therefore introduce as objects two signal values, 1 and 0, representing "on" and "off" respectively, and a function $Signal(t)$ that denotes the signal value for the terminal t.

Encode general knowledge of the domain

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

1. If two terminals are connected, then they have the same signal:
   $\forall t_1,t_2\ Terminal(t_1) \land Terminal(t_2) \land Connected(t_1,t_2) \Rightarrow$
   $\quad Signal(t_1) = Signal(t_2)$.
2. The signal at every terminal is either 1 or 0:
   $\forall t\ Terminal(t) \Rightarrow Signal(t) = 1 \lor Signal(t) = 0$.
3. *Connected* is commutative:
   $\forall t_1,t_2\ Connected(t_1,t_2) \Leftrightarrow Connected(t_2,t_1)$.
4. There are four types of gates:
   $\forall g\ Gate(g) \land k = Type(g) \Rightarrow k = AND \lor k = OR \lor k = XOR \lor k = NOT$.
5. An AND gate's output is 0 if and only if any of its inputs is 0:
   $\forall g\ Gate(g) \land Type(g) = AND \Rightarrow$
   $\quad Signal(Out(1,g)) = 0 \Leftrightarrow \exists n\ Signal(In(n,g)) = 0$.
6. An OR gate's output is 1 if and only if any of its inputs is 1:
   $\forall g\ Gate(g) \land Type(g) = OR \Rightarrow$
   $\quad Signal(Out(1,g)) = 1 \Leftrightarrow \exists n\ Signal(In(n,g)) = 1$.
7. An XOR gate's output is 1 if and only if its inputs are different:
   $\forall g\ Gate(g) \land Type(g) = XOR \Rightarrow$
   $\quad Signal(Out(1,g)) = 1 \Leftrightarrow Signal(In(1,g)) \neq Signal(In(2,g))$.
8. A NOT gate's output is different from its input:
   $\forall g\ Gate(g) \land Type(g) = NOT \Rightarrow$
   $\quad Signal(Out(1,g)) \neq Signal(In(1,g))$.
9. The gates (except for NOT) have two inputs and one output.
   $\forall g\ Gate(g) \land Type(g) = NOT \Rightarrow Arity(g,1,1)$.
   $\forall g\ Gate(g) \land k = Type(g) \land (k = AND \lor k = OR \lor k = XOR) \Rightarrow$
   $\quad Arity(g,2,1)$
10. A circuit has terminals, up to its input and output arity, and nothing beyond its arity:
   $\forall c,i,j\ Circuit(c) \land Arity(c,i,j) \Rightarrow$
   $\quad \forall n\ (n \leq i \Rightarrow Terminal(In(n,c))) \land (n > i \Rightarrow In(n,c) = Nothing) \land$
   $\quad \forall n\ (n \leq j \Rightarrow Terminal(Out(n,c))) \land (n > j \Rightarrow Out(n,c) = Nothing)$

11. Gates, terminals, and signals are all distinct.
    $$\forall g,t,s \ \ Gate(g) \land Terminal(t) \land Signal(s) \Rightarrow g \neq t \land g \neq s \land t \neq s\,.$$
12. Gates are circuits.
    $$\forall g \ \ Gate(g) \Rightarrow Circuit(g)$$

Encode the specific problem instance

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

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

Then we show the connections between them:

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

Pose queries to the inference procedure

What combinations of inputs would cause the first output of C_1 (the sum bit) to be 0 and the second output of C_1 (the carry bit) to be 1?

$$\begin{aligned}
\exists i_1,i_2,i_3 \ \ & Signal(In(1,C_1)) = i_1 \land Signal(In(2,C_1)) = i_2 \land Signal(In(3,C_1)) = i_3\\
& \land Signal(Out(1,C_1)) = 0 \land Signal(Out(2,C_1)) = 1\,.
\end{aligned}$$

The answers are substitutions for the variables i_1, i_2, and i_3 such that the resulting sentence is entailed by the knowledge base. ASKVARS will give us three such substitutions:

$$\{i_1/1,\ i_2/1,\ i_3/0\} \quad \{i_1/1,\ i_2/0,\ i_3/1\} \quad \{i_1/0,\ i_2/1,\ i_3/1\}\,.$$

What are the possible sets of values of all the terminals for the adder circuit?

$$\begin{aligned}
\exists i_1,i_2,i_3,o_1,o_2 \ \ & Signal(In(1,C_1)) = i_1 \land Signal(In(2,C_1)) = i_2\\
& \land Signal(In(3,C_1)) = i_3 \land Signal(Out(1,C_1)) = o_1 \land Signal(Out(2,C_1)) = o_2\,.
\end{aligned}$$

This final query will return a complete input–output table for the device, which can be used to check that it does in fact add its inputs correctly. This is a simple example of **circuit verification**. We can also use the definition of the circuit to build larger digital systems, for which the same kind of verification procedure can be carried out. (See Exercise 8.ADDR.) Many domains are amenable to the same kind of structured knowledge-base development, in which more complex concepts are defined on top of simpler concepts.

Circuit verification

Debug the knowledge base

We can perturb the knowledge base in various ways to see what kinds of erroneous behaviors emerge. For example, suppose we fail to read Section 8.2.8 and hence forget to assert that $1 \neq 0$. Suddenly, the system will be unable to prove any outputs for the circuit, except for the input cases 000 and 110. We can pinpoint the problem by asking for the outputs of each gate. For example, we can ask

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

which reveals that no outputs are known at X_1 for the input cases 10 and 01. Then, we look at the axiom for XOR gates, as applied to X_1:

$$Signal(Out(1, X_1)) = 1 \;\Leftrightarrow\; Signal(In(1, X_1)) \neq Signal(In(2, X_1))\,.$$

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

$$Signal(Out(1, X_1)) = 1 \;\Leftrightarrow\; 1 \neq 0\,.$$

Now the problem is apparent: the system is unable to infer that $Signal(Out(1, X_1)) = 1$, so we need to tell it that $1 \neq 0$.

Summary

This chapter has introduced **first-order logic**, a representation language that is far more powerful than propositional logic. The important points are as follows:

- Knowledge representation languages should be declarative, compositional, expressive, context independent, and unambiguous.
- Logics differ in their **ontological commitments** and **epistemological commitments**. While propositional logic commits only to the existence of facts, first-order logic commits to the existence of objects and relations and thereby gains expressive power, appropriate for domains such as the wumpus world and electronic circuits.
- Both propositional logic and first-order logic share a difficulty in representing vague propositions. This difficulty limits their applicability in domains that require personal judgments, like politics or cuisine.
- The syntax of first-order logic builds on that of propositional logic. It adds terms to represent objects, and has universal and existential quantifiers to construct assertions about all or some of the possible values of the quantified variables.
- A **possible world**, or **model**, for first-order logic includes a set of objects and an **interpretation** that maps constant symbols to objects, predicate symbols to relations among objects, and function symbols to functions on objects.
- An atomic sentence is true only when the relation named by the predicate holds between the objects named by the terms. **Extended interpretations**, which map quantifier variables to objects in the model, define the truth of quantified sentences.
- Developing a knowledge base in first-order logic requires a careful process of analyzing the domain, choosing a vocabulary, and encoding the axioms required to support the desired inferences.

Bibliographical and Historical Notes

Although Aristotle's logic dealt with generalizations over objects, it fell far short of the expressive power of first-order logic. A major barrier to its further development was its concentration on one-place predicates to the exclusion of many-place relational predicates. The first systematic treatment of relations was given by Augustus De Morgan (1864), who cited the following example to show the sorts of inferences that Aristotle's logic could not handle: "All horses are animals; therefore, the head of a horse is the head of an animal." This inference is inaccessible to Aristotle because any valid rule that can support this inference must first analyze the sentence using the two-place predicate "*x* is the head of *y*." The logic of relations was studied in depth by Charles Sanders Peirce (Peirce, 1870; Misak, 2004).

True first-order logic dates from the introduction of quantifiers in Gottlob Frege's (1879) *Begriffschrift* ("Concept Writing" or "Conceptual Notation"). Peirce (1883) also developed first-order logic independently of Frege, although slightly later. Frege's ability to nest quantifiers was a big step forward, but he used an awkward notation. The present notation for first-order logic is due substantially to Giuseppe Peano (1889), but the semantics is virtually identical to Frege's. Oddly enough, Peano's axioms were due in large measure to Grassmann (1861) and Dedekind (1888).

Leopold Löwenheim (1915) gave a systematic treatment of model theory for first-order logic, including the first proper treatment of the equality symbol. Löwenheim's results were further extended by Thoralf Skolem (1920). Alfred Tarski (1935, 1956) gave an explicit definition of truth and model-theoretic satisfaction in first-order logic, using set theory.

John McCarthy (1958) was primarily responsible for the introduction of first-order logic as a tool for building AI systems. The prospects for logic-based AI were advanced significantly by Robinson's (1965) development of resolution, a complete procedure for first-order inference. The logicist approach took root at Stanford University. Cordell Green (1969a, 1969b) developed a first-order reasoning system, QA3, leading to the first attempts to build a logical robot at SRI (Fikes and Nilsson, 1971). First-order logic was applied by Zohar Manna and Richard Waldinger (1971) for reasoning about programs and later by Michael Genesereth (1984) for reasoning about circuits. In Europe, logic programming (a restricted form of first-order reasoning) was developed for linguistic analysis (Colmerauer *et al.*, 1973) and for general declarative systems (Kowalski, 1974). Computational logic was also well entrenched at Edinburgh through the LCF (Logic for Computable Functions) project (Gordon *et al.*, 1979). These developments are chronicled further in Chapters 9 and 10.

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

Reactions to the Whorf hypothesis (Whorf, 1956) and the problem of language and thought in general, appear in multiple books (Pullum, 1991; Pinker, 2003) including the seemingly opposing titles *Why the World Looks Different in Other Languages* (Deutscher, 2010) and *Why The World Looks the Same in Any Language* (McWhorter, 2014) (although both authors agree that there are differences and the differences are small). The "theory" theory (Gopnik and Glymour, 2002; Tenenbaum *et al.*, 2007) views children's learning about the world as analogous to the construction of scientific theories. Just as the predictions of a machine learning algorithm depend strongly on the vocabulary supplied to it, so will the child's formulation of theories depend on the linguistic environment in which learning occurs.

There are a number of good introductory texts on first-order logic, including some by leading figures in the history of logic: Alfred Tarski (1941), Alonzo Church (1956), and W.V. Quine (1982) (which is one of the most readable). Enderton (1972) gives a more mathematically oriented perspective. A highly formal treatment of first-order logic, along with many more advanced topics in logic, is provided by Bell and Machover (1977). Manna and Waldinger (1985) give a readable introduction to logic from a computer science perspective, as do Huth and Ryan (2004), who concentrate on program verification. Barwise and Etchemendy (2002) take an approach similar to the one used here. Smullyan (1995) presents results concisely, using the tableau format. Gallier (1986) provides an extremely rigorous mathematical exposition of first-order logic, along with a great deal of material on its use in automated reasoning. *Logical Foundations of Artificial Intelligence* (Genesereth and Nilsson, 1987) is both a solid introduction to logic and the first systematic treatment of logical agents with percepts and actions, and there are two good handbooks: van Bentham and ter Meulen (1997) and Robinson and Voronkov (2001). The journal of record for the field of pure mathematical logic is the *Journal of Symbolic Logic*, whereas the *Journal of Applied Logic* deals with concerns closer to those of artificial intelligence.

CHAPTER 9

INFERENCE IN FIRST-ORDER LOGIC

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

In this chapter, we describe algorithms that can answer any answerable first-order logic question. Section 9.1 introduces inference rules for quantifiers and shows how to reduce first-order inference to propositional inference, albeit at potentially great expense. Section 9.2 describes how **unification** can be used to construct inference rules that work directly with first-order sentences. We then discuss three major families of first-order inference algorithms: **forward chaining** (Section 9.3), **backward chaining** (Section 9.4), and **resolution-based theorem proving** (Section 9.5).

9.1 Propositional vs. First-Order Inference

One way to do first-order inference is to convert the first-order knowledge base to propositional logic and use propositional inference, which we already know how to do. A first step is eliminating universal quantifiers. For example, suppose our knowledge base contains the standard folkloric axiom that all greedy kings are evil:

$$\forall x \; King(x) \land Greedy(x) \;\Rightarrow\; Evil(x)\,.$$

From that we can infer any of the following sentences:

$$\begin{array}{l} King(John) \land Greedy(John) \;\Rightarrow\; Evil(John) \\ King(Richard) \land Greedy(Richard) \;\Rightarrow\; Evil(Richard) \\ King(Father(John)) \land Greedy(Father(John)) \;\Rightarrow\; Evil(Father(John))\,. \\ \quad\vdots \end{array}$$

Universal Instantiation

In general, the rule of **Universal Instantiation** (**UI** for short) says that we can infer any sentence obtained by substituting a **ground term** (a term without variables) for a universally quantified variable.[1]

To write out the inference rule formally, we use the notion of **substitutions** introduced in Section 8.3. Let $\textsc{Subst}(\theta,\alpha)$ denote the result of applying the substitution θ to the sentence α. Then the rule is written

$$\frac{\forall v \;\; \alpha}{\textsc{Subst}(\{v/g\},\alpha)}$$

for any variable v and ground term g. For example, the three sentences given earlier are obtained with the substitutions $\{x/John\}$, $\{x/Richard\}$, and $\{x/Father(John)\}$.

[1] Do not confuse these substitutions with the extended interpretations used to define the semantics of quantifiers in Section 8.2.6. The substitution replaces a variable with a term (a piece of syntax) to produce a new sentence, whereas an interpretation maps a variable to an object in the domain.

Existential Instantiation

Similarly, the rule of **Existential Instantiation** replaces an existentially quantified variable with a single *new constant symbol*. The formal statement is as follows: for any sentence α, variable v, and constant symbol k that does not appear elsewhere in the knowledge base,

$$\frac{\exists v \;\; \alpha}{\textsc{Subst}(\{v/k\}, \alpha)} \,.$$

For example, from the sentence

$$\exists x \;\; Crown(x) \land OnHead(x, John)$$

we can infer the sentence

$$Crown(C_1) \land OnHead(C_1, John)$$

as long as C_1 does not appear elsewhere in the knowledge base. Basically, the existential sentence says there is some object satisfying a condition, and applying the existential instantiation rule just gives a name to that object. Of course, that name must not already belong to another object. Mathematics provides a nice example: suppose we discover that there is a number that is a little bigger than 2.71828 and that satisfies the equation $d(x^y)/dy = x^y$ for x. We can give this number the name e, but it would be a mistake to give it the name of an existing object, such as π. In logic, the new name is called a **Skolem constant**.

Skolem constant

Whereas Universal Instantiation can be applied many times to the same axiom to produce many different consequences, Existential Instantiation need only be applied once, and then the existentially quantified sentence can be discarded. For example, we no longer need $\exists x\, Kill(x, Victim)$ once we have added the sentence $Kill(Murderer, Victim)$.

9.1.1 Reduction to propositional inference

We now show how to convert any first-order knowledge base into a propositional knowledge base. The first idea is that, just as an existentially quantified sentence can be replaced by one instantiation, a universally quantified sentence can be replaced by the set of *all possible* instantiations. For example, suppose our knowledge base contains just the sentences

$$\begin{array}{l} \forall x \;\; King(x) \land Greedy(x) \;\Rightarrow\; Evil(x) \\ King(John) \\ Greedy(John) \\ Brother(Richard, John)\,. \end{array} \qquad (9.1)$$

and that the only objects are $John$ and $Richard$. We apply UI to the first sentence using all possible substitutions, $\{x/John\}$ and $\{x/Richard\}$. We obtain

$$\begin{array}{l} King(John) \land Greedy(John) \;\Rightarrow\; Evil(John) \\ King(Richard) \land Greedy(Richard) \;\Rightarrow\; Evil(Richard)\,. \end{array}$$

Next replace ground atomic sentences, such as $King(John)$, with proposition symbols, such as $JohnIsKing$. Finally, apply any of the complete propositional algorithms in Chapter 7 to obtain conclusions such as $JohnIsEvil$, which is equivalent to $Evil(John)$.

Propositionalization

This technique of **propositionalization** can be made completely general, as we show in Section 9.5. However, there is a problem when the knowledge base includes a function symbol, the set of possible ground-term substitutions is infinite! For example, if the knowledge base mentions the $Father$ symbol, then infinitely many nested terms such as $Father(Father(Father(John)))$ can be constructed.

Fortunately, there is a famous theorem due to Jacques Herbrand (1930) to the effect that if a sentence is entailed by the original, first-order knowledge base, then there is a proof involving just a *finite* subset of the propositionalized knowledge base. Since any such subset has a maximum depth of nesting among its ground terms, we can find the subset by first generating all the instantiations with constant symbols (*Richard* and *John*), then all terms of depth 1 ($Father(Richard)$ and $Father(John)$), then all terms of depth 2, and so on, until we are able to construct a propositional proof of the entailed sentence.

We have sketched an approach to first-order inference via propositionalization that is **complete**—that is, any entailed sentence can be proved. This is a major achievement, given that the space of possible models is infinite. On the other hand, we do not know until the proof is done that the sentence *is* entailed! What happens when the sentence is *not* entailed? Can we tell? Well, for first-order logic, it turns out that we cannot. Our proof procedure can go on and on, generating more and more deeply nested terms, but we will not know whether it is stuck in a hopeless loop or whether the proof is just about to pop out. This is very much like the halting problem for Turing machines. Alan Turing (1936) and Alonzo Church (1936) both proved, in rather different ways, the inevitability of this state of affairs. *The question of entailment for first-order logic is* ***semidecidable****—that is, algorithms exist that say yes to every entailed sentence, but no algorithm exists that also says no to every nonentailed sentence.*

9.2 Unification and First-Order Inference

The sharp-eyed reader will have noticed that the propositionalization approach generates many unnecessary instantiations of universally quantified sentences. We'd rather have an approach that uses just the one rule, reasoning that $\{x/John\}$ solves the query $Evil(x)$ as follows: given the rule that greedy kings are evil, find some x such that x is a king and x is greedy, and then infer that this x is evil. More generally, if there is some substitution θ that makes each of the conjuncts of the premise of the implication identical to sentences already in the knowledge base, then we can assert the conclusion of the implication, after applying θ. In this case, the substitution $\theta = \{x/John\}$ achieves that aim. Now suppose that instead of knowing $Greedy(John)$, we know that *everyone* is greedy:

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

Then we would still like to be able to conclude that $Evil(John)$, because we know that John is a king (given) and John is greedy (because everyone is greedy). What we need for this to work is to find a substitution for both the variables in the implication sentence and the variables in the sentences that are in the knowledge base. In this case, applying the substitution $\{x/John, y/John\}$ to the implication premises $King(x)$ and $Greedy(x)$ and the knowledge-base sentences $King(John)$ and $Greedy(y)$ will make them identical. Thus, we can infer the consequent of the implication.

Generalized Modus Ponens

This inference process can be captured as a single inference rule that we call **Generalized Modus Ponens**:[2] For atomic sentences p_i, $p_i{}'$, and q, where there is a substitution θ such that

[2] Generalized Modus Ponens is more general than Modus Ponens (page 222) in the sense that the known facts and the premise of the implication need match only up to a substitution, rather than exactly. On the other hand, Modus Ponens allows any sentence α as the premise, rather than just a conjunction of atomic sentences.

$\text{SUBST}(\theta, p_i{}') = \text{SUBST}(\theta, p_i)$, for all i,

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

There are $n+1$ premises to this rule: the n atomic sentences $p_i{}'$ and the one implication. The conclusion is the result of applying the substitution θ to the consequent q. For our example:

$p_1{}'$ is $King(John)$	p_1 is $King(x)$
$p_2{}'$ is $Greedy(y)$	p_2 is $Greedy(x)$
θ is $\{x/John, y/John\}$	q is $Evil(x)$
$\text{SUBST}(\theta, q)$ is $Evil(John)$.	

It is easy to show that Generalized Modus Ponens is a sound inference rule. First, we observe that, for any sentence p (whose variables are assumed to be universally quantified) and for any substitution θ,

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

is true by Universal Instantiation. It is true in particular for a θ that satisfies the conditions of the Generalized Modus Ponens rule. Thus, from $p_1{}', \ldots, p_n{}'$ we can infer

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

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

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

Now, θ in Generalized Modus Ponens is defined so that $\text{SUBST}(\theta, p_i{}') = \text{SUBST}(\theta, p_i)$, for all i; therefore the first of these two sentences matches the premise of the second exactly. Hence, $\text{SUBST}(\theta, q)$ follows by Modus Ponens.

Generalized Modus Ponens is a **lifted** version of Modus Ponens—it raises Modus Ponens from ground (variable-free) propositional logic to first-order logic. We will see in the rest of this chapter that we can develop lifted versions of the forward chaining, backward chaining, and resolution algorithms introduced in Chapter 7. The key advantage of lifted inference rules over propositionalization is that they make only those substitutions that are required to allow particular inferences to proceed.

Lifting

9.2.1 Unification

Lifted inference rules require finding substitutions that make different logical expressions look identical. This process is called **unification** and is a key component of all first-order inference algorithms. The UNIFY algorithm takes two sentences and returns a **unifier** for them (a substitution) if one exists:

Unification

Unifier

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

Let us look at some examples of how UNIFY should behave. Suppose we have a query $AskVars(Knows(John, x))$: whom does John know? Answers to this query can be found by finding all sentences in the knowledge base that unify with $Knows(John, x)$. Here are the results of unification with four different sentences that might be in the knowledge base:

$$\text{UNIFY}(Knows(John, x), Knows(John, Jane)) = \{x/Jane\}$$
$$\text{UNIFY}(Knows(John, x), Knows(y, Bill)) = \{x/Bill, y/John\}$$
$$\text{UNIFY}(Knows(John, x), Knows(y, Mother(y))) = \{y/John, x/Mother(John)\}$$
$$\text{UNIFY}(Knows(John, x), Knows(x, Elizabeth)) = failure.$$

The last unification fails because x cannot take on the values *John* and *Elizabeth* at the same time. Now, remember that $Knows(x, Elizabeth)$ means "Everyone knows Elizabeth," so we *should* be able to infer that John knows Elizabeth. The problem arises only because the two sentences happen to use the same variable name, x. The problem can be avoided by **standardizing apart** one of the two sentences being unified, which means renaming its variables to avoid name clashes. For example, we can rename x in $Knows(x, Elizabeth)$ to x_{17} (a new variable name) without changing its meaning. Now the unification will work:

Standardizing apart

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

Exercise 9.STAN delves further into the need for standardizing apart.

There is one more complication: we said that UNIFY should return a substitution that makes the two arguments look the same. But there could be more than one such unifier. For example, $\textsc{Unify}(Knows(John, x), Knows(y, z))$ could return $\{y/John, x/z\}$ or could return $\{y/John, x/John, z/John\}$. The first unifier gives $Knows(John, z)$ as the result of unification, whereas the second gives $Knows(John, John)$. The second result could be obtained from the first by an additional substitution $\{z/John\}$; we say that the first unifier is *more general* than the second, because it places fewer restrictions on the values of the variables.

Most general unifier (MGU)

Every unifiable pair of expressions has a single **most general unifier (MGU)** that is unique up to renaming and substitution of variables. For example, $\{x/John\}$ and $\{y/John\}$ are considered equivalent, as are $\{x/John, y/John\}$ and $\{x/John, y/x\}$.

An algorithm for computing most general unifiers is shown in Figure 9.1. The process is simple: recursively explore the two expressions simultaneously "side by side," building up a unifier along the way, but failing if two corresponding points in the structures do not match. There is one expensive step: when matching a variable against a complex term, one must check whether the variable itself occurs inside the term; if it does, the match fails because no consistent unifier can be constructed. For example, $S(x)$ can't unify with $S(S(x))$. This so-called **occur check** makes the complexity of the entire algorithm quadratic in the size of the expressions being unified. Some systems, including many logic programming systems, simply omit the occur check and put the onus on the user to avoid making unsound inferences as a result. Other systems use more complex unification algorithms with linear-time complexity.

Occur check

9.2.2 Storage and retrieval

Underlying the TELL, ASK, and ASKVARS functions used to inform and interrogate a knowledge base are the more primitive STORE and FETCH functions. STORE(s) stores a sentence s into the knowledge base and FETCH(q) returns all unifiers such that the query q unifies with some sentence in the knowledge base. The problem we used to illustrate unification—finding all facts that unify with $Knows(John, x)$—is an instance of FETCHing.

The simplest way to implement STORE and FETCH is to keep all the facts in one long list and unify each query against every element of the list. Such a process is inefficient, but it works. The remainder of this section outlines ways to make retrieval more efficient.

We can make FETCH more efficient by ensuring that unifications are attempted only with sentences that have *some* chance of unifying. For example, there is no point in trying to unify $Knows(John, x)$ with $Brother(Richard, John)$. We can avoid such unifications by **indexing** the facts in the knowledge base. A simple scheme called **predicate indexing** puts all the

Indexing

Predicate indexing

```
function UNIFY($x, y, \theta$=empty) returns a substitution to make $x$ and $y$ identical, or failure
  if $\theta$ = failure then return failure
  else if $x = y$ then return $\theta$
  else if VARIABLE?($x$) then return UNIFY-VAR($x, y, \theta$)
  else if VARIABLE?($y$) then return UNIFY-VAR($y, x, \theta$)
  else if COMPOUND?($x$) and COMPOUND?($y$) then
      return UNIFY(ARGS($x$), ARGS($y$), UNIFY(OP($x$), OP($y$), $\theta$))
  else if LIST?($x$) and LIST?($y$) then
      return UNIFY(REST($x$), REST($y$), UNIFY(FIRST($x$), FIRST($y$), $\theta$))
  else return failure

function UNIFY-VAR($var, x, \theta$) returns a substitution
  if $\{var/val\} \in \theta$ for some $val$ then return UNIFY($val, x, \theta$)
  else if $\{x/val\} \in \theta$ for some $val$ then return UNIFY($var, val, \theta$)
  else if OCCUR-CHECK?($var, x$) then return failure
  else return add $\{var/x\}$ to $\theta$
```

Figure 9.1 The unification algorithm. The arguments x and y can be any expression: a constant or variable, or a compound expression such as a complex sentence or term, or a list of expressions. The argument θ is a substitution, initially the empty substitution, but with $\{var/val\}$ pairs added to it as we recurse through the inputs, comparing the expressions element by element. In a compound expression such as $F(A,B)$, OP(x) field picks out the function symbol F and ARGS(x) field picks out the argument list (A,B).

Knows facts in one bucket and all the *Brother* facts in another. The buckets can be stored in a hash table for efficient access.

Predicate indexing is useful when there are many predicate symbols but only a few clauses for each symbol. Sometimes, however, a predicate has many clauses. For example, suppose that the tax authorities want to keep track of who employs whom, using a predicate $Employs(x,y)$. This would be a very large bucket with perhaps millions of employers and tens of millions of employees. Answering a query such as $Employs(x, Richard)$ with predicate indexing would require scanning the entire bucket.

For this particular query, it would help if facts were indexed both by predicate and by second argument, perhaps using a combined hash table key. Then we could simply construct the key from the query and retrieve exactly those facts that unify with the query. For other queries, such as $Employs(IBM, y)$, we would need to have indexed the facts by combining the predicate with the first argument. Therefore, facts can be stored under multiple index keys, rendering them instantly accessible to various queries that they might unify with.

Given a sentence to be stored, it is possible to construct indices for *all possible* queries that unify with it. For the fact $Employs(IBM, Richard)$, the queries are

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

These queries form a **subsumption lattice**, as shown in Figure 9.2(a). The lattice has some Subsumption lattice

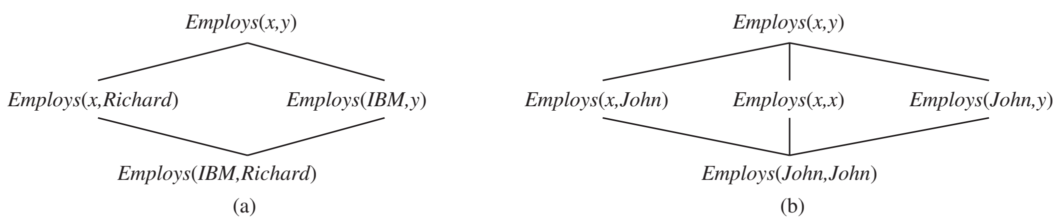


Figure 9.2 (a) The subsumption lattice whose lowest node is $Employs(IBM,Richard)$. (b) The subsumption lattice for the sentence $Employs(John,John)$.

interesting properties. The child of any node in the lattice is obtained from its parent by a single substitution; and the "highest" common descendant of any two nodes is the result of applying their most general unifier. A sentence with repeated constants has a slightly different lattice, as shown in Figure 9.2(b). Although function symbols are not shown in the figure, they too can be incorporated into the lattice structure.

For predicates with a small number of arguments, it is a good tradeoff to create an index for every point in the subsumption lattice. That requires a little more work at storage time, but speeds up retrieval time. However, for a predicate with n arguments, the lattice contains $O(2^n)$ nodes. If function symbols are allowed, the number of nodes is also exponential in the size of the terms in the sentence to be stored. This can lead to a huge number of indices.

We have to somehow limit the indices to ones that are likely to be used frequently in queries; otherwise we will waste more time in creating the indices than we save by having them. We could adopt a fixed policy, such as maintaining indices only on keys composed of a predicate plus a single argument. Or we could learn an adaptive policy that creates indices to meet the demands of the kinds of queries being asked. For commercial databases where facts number in the billions, the problem has been the subject of intensive study, technology development, and continual optimization.

9.3 Forward Chaining

In Section 7.5 we showed a forward-chaining algorithm for knowledge bases of propositional definite clauses. Here we expand that idea to cover first-order definite clauses.

Of course there are some logical sentences that cannot be stated as a definite clause, and thus cannot be handled by this approach. But rules of the form $Antecedent \Rightarrow Consequent$ are sufficient to cover a wide variety of interesting real-world systems.

9.3.1 First-order definite clauses

First-order definite clauses are disjunctions of literals of which *exactly one is positive*. That means a definite clause is either atomic, or is an implication whose antecedent is a conjunction of positive literals and whose consequent is a single positive literal. Existential quantifiers are not allowed, and universal quantifiers are left implicit: if you see an x in a definite clause, that means there is an implicit $\forall x$ quantifier. A typical first-order definite clause looks like this:

$$King(x) \land Greedy(x) \Rightarrow Evil(x)\,,$$

but the literals $King(John)$ and $Greedy(y)$ also count as definite clauses. First-order liter-