GLOBAL EDITION

Stuart **Russell**

Peter **Norvig**

Artificial Intelligence
A Modern Approach

Fourth Edition

Artificial Intelligence

A Modern Approach

Fourth Edition

Global Edition

PEARSON SERIES IN ARTIFICIAL INTELLIGENCE

Stuart Russell and Peter Norvig, Editors

FORSYTH & PONCE

Computer Vision: A Modern Approach, 2nd ed.

GRAHAM

ANSI *Common Lisp*

JURAFSKY & MARTIN

Speech and Language Processing, 2nd ed.

NEAPOLITAN

Learning Bayesian Networks

RUSSELL & NORVIG

Artificial Intelligence: A Modern Approach, 4th ed.

Artificial Intelligence
A Modern Approach
Fourth Edition
Global Edition

Stuart J. Russell and Peter Norvig

Contributing writers:
Ming-Wei Chang
Jacob Devlin
Anca Dragan
David Forsyth
Ian Goodfellow
Jitendra M. Malik
Vikash Mansinghka
Judea Pearl
Michael Wooldridge

Pearson Education Limited
KAO Two
KAO Park
Hockham Way
Harlow
CM17 9SR
United Kingdom

and Associated Companies throughout the world

Visit us on the World Wide Web at: www.pearsonglobaleditions.com

Authorized adaptation from the United States edition, entitled Artificial Intelligence: A Modern Approach, *4th Edition, ISBN 978-0-13-461099-3 byStuart J. Russell and Peter Norvig, published by Pearson Education* © *2021.*

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

ISBN 10: 1-292-40113-3 (Print)
ISBN 13: 978-1-292-40113-3 (Print)
ISBN 13: 978-1-292-41007-4(ePub)
eBook formatted by SPi Global

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 29 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.

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 pearsonglobaleditions.com. 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.
- 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 pearsonglobaleditions.com; 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);

- Michael Wooldridge (Chapter 17, Multiagent Decision Making);
- Vikash Mansinghka (Section 18.4, Programs as Probability Models);
- Ian Goodfellow (Chapter 22, Deep Learning);
- Jacob Devlin and Mei-Wing Chang (Chapter 25, Deep Learning for Natural Language Processing);
- Anca Dragan (Chapter 26, Robotics);
- Jitendra Malik and David Forsyth (Chapter 27, Computer Vision).

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

IV Uncertain knowledge and reasoning

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.

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.

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]

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 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

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 28 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** 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.

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** 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**—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 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**.

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.

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.

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 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.

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.

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 6 and 16 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.

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.

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.

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. 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.

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."

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.

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.

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.

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 of an outcome. The modern notion of rational decision making under uncertainty involves maximizing expected utility, as explained in Chapter 15.

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.

In contrast, Immanuel Kant, in 1785, 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.

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.

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.

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 London 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.

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.

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.

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.

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!

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, 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 17).

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 16 and, under the heading of **reinforcement learning**, in Chapter 23.

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 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.

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.*

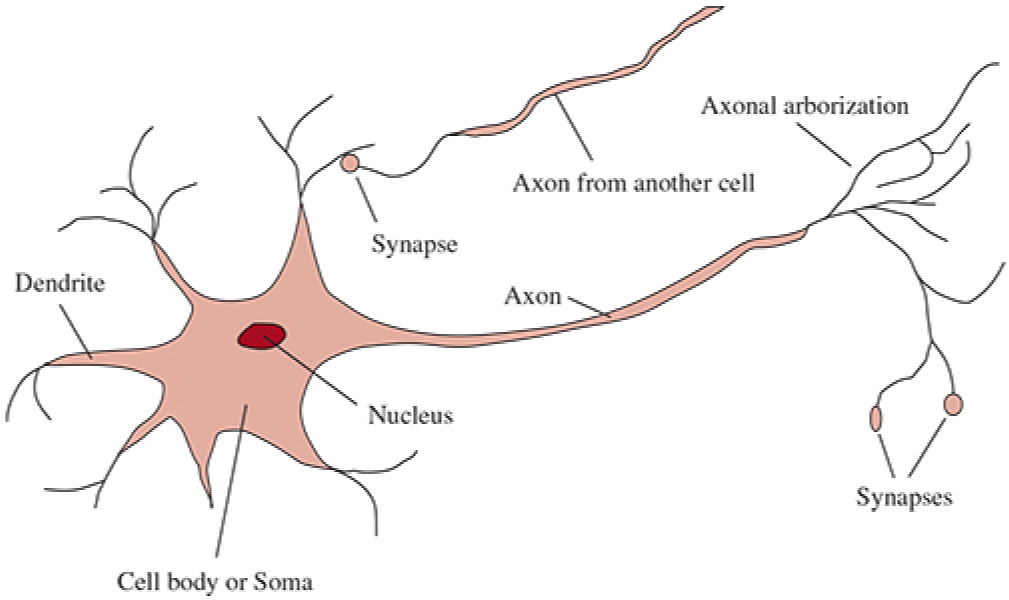


Figure 1.1 The parts of a nerve cell or neuron. Each neuron consists of a cell body, or soma, that contains a cell nucleus. Branching out from the cell body are a number of fibers called dendrites and a single long fiber called the axon. The axon stretches out for a long distance, much longer than the 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).

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.

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.

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.

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 29). Crudely put, without the right theory, faster machines just give you the wrong answer faster.

	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).

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 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 way 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.

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.

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.

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 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.

Of course, there were calculating devices before the electronic computer. The earliest automated machines, dating from the 17th century, were discussed on page 24. 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.

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.

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.

Modern control theory, especially the branch known as stochastic optimal control, has as its goal the design of systems that minimize 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.

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.

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; Allen Newell and Herbert Simon (1975) for symbolic models of problem solving and human cognition; Ed Feigenbaum and Raj Reddy (1994) for developing expert systems that encode human knowledge to solve real-world problems; Judea Pearl (2011) for developing probabilistic reasoning techniques that deal with uncertainty in a principled manner; and finally Yoshua Bengio, Geoffrey Hinton, and Yann LeCun (2019) for making “deep learning” (multilayer neural networks) a critical part of modern computing. The rest of this section goes into more detail on each phase of AI history.

1.3.1 The inception of artificial intelligence (1943–1956)

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

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 28. 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 28 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.

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.

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 23), 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 6).

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.

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 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.

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?

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).

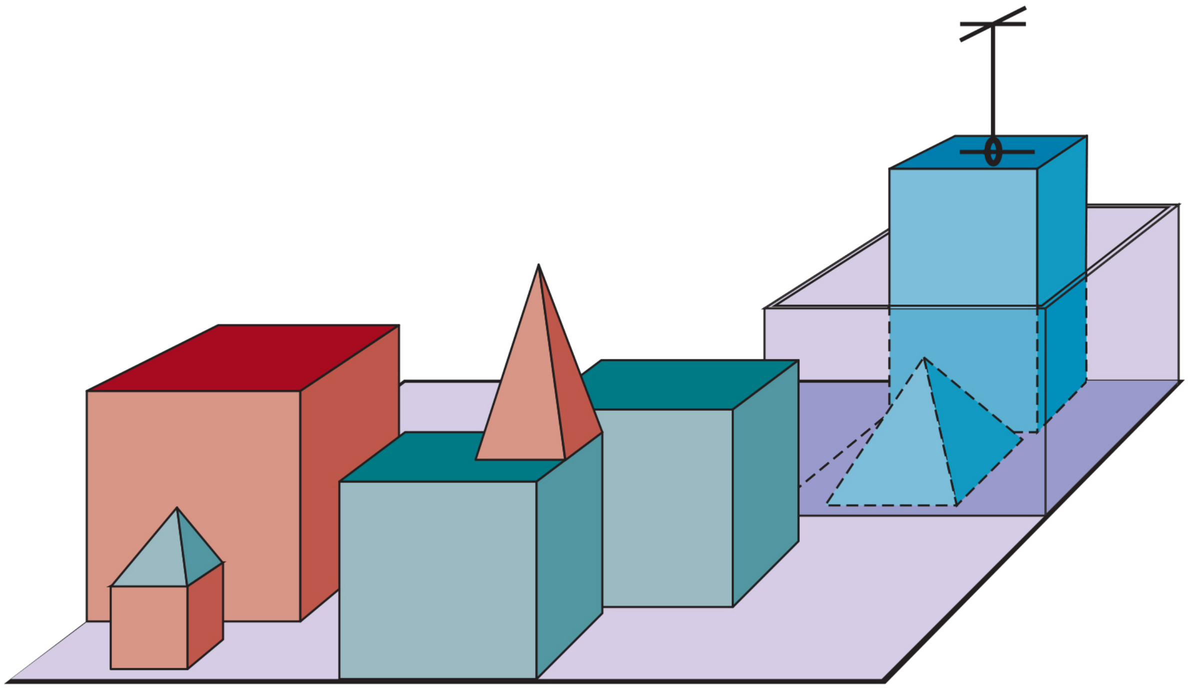


Figure 1.3 A scene from the blocks world. SHRDLU (Winograd, 1972) has just completed the command "Find a block which is taller than the one you are holding and put it in the box."

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.

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.

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 $x1$ and $x2$ 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.

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.

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.

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.

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 Competition 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.

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, 13, 14, 15, and 18 cover this area, in addition to more recent developments that have greatly increased the expressive power of probabilistic formalisms; Chapter 21 describes methods for learning Bayesian networks and related models from data.

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)

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)

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 hand-written 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 22).

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):

- 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 28 and 29.

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

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

- *Lethal autonomous weapons:* These are defined by the United Nations as weapons that can locate, select, and eliminate human targets without human intervention. A primary concern with such weapons is their *scalability:* the absence of a requirement for human supervision means that a small group can deploy an arbitrarily large number of weapons against human targets defined by any feasible recognition criterion. The technologies needed for autonomous weapons are similar to those needed for self-driving cars. Informal expert discussions on the potential risks of lethal autonomous weapons began at the UN in 2014, moving to the formal pre-treaty stage of a Group of Governmental Experts in 2017.

- *Surveillance and persuasion:* While it is expensive, tedious, and sometimes legally questionable for security personnel to monitor phone lines, video camera feeds, emails, and other messaging channels, AI (speech recognition, computer vision, and natural language understanding) can be used in a scalable fashion to perform mass surveillance of individuals and detect activities of interest. By tailoring information flows to individuals through social media, based on machine learning techniques, political behavior can be modified and controlled to some extent—a concern that became apparent in elections beginning in 2016.
- *Biased decision making:* Careless or deliberate misuse of machine learning algorithms for tasks such as evaluating parole and loan applications can result in decisions that are biased by race, gender, or other protected categories. Often, the data themselves reflect pervasive bias in society.
- *Impact on employment:* Concerns about machines eliminating jobs are centuries old. The story is never simple: machines do some of the tasks that humans might otherwise do, but they also make humans more productive and therefore more employable, and make companies more profitable and therefore able to pay higher wages. They may render some activities economically viable that would otherwise be

impractical. Their 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 28.3. 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 28.3). 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)** movement (Goertzel and Pennachin, 2007), held its first conference and organized the *Journal of Artificial General Intelligence* in 2008.

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.

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.

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.

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 15, 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 17, 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 23, 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 28, 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.

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.

[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.

[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.”

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

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

[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.

[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.

[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.

[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 held has reached a level of stability and maturity. The present emphasis on deep learning may represent a resurgence of the scruffies.

[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.

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

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.

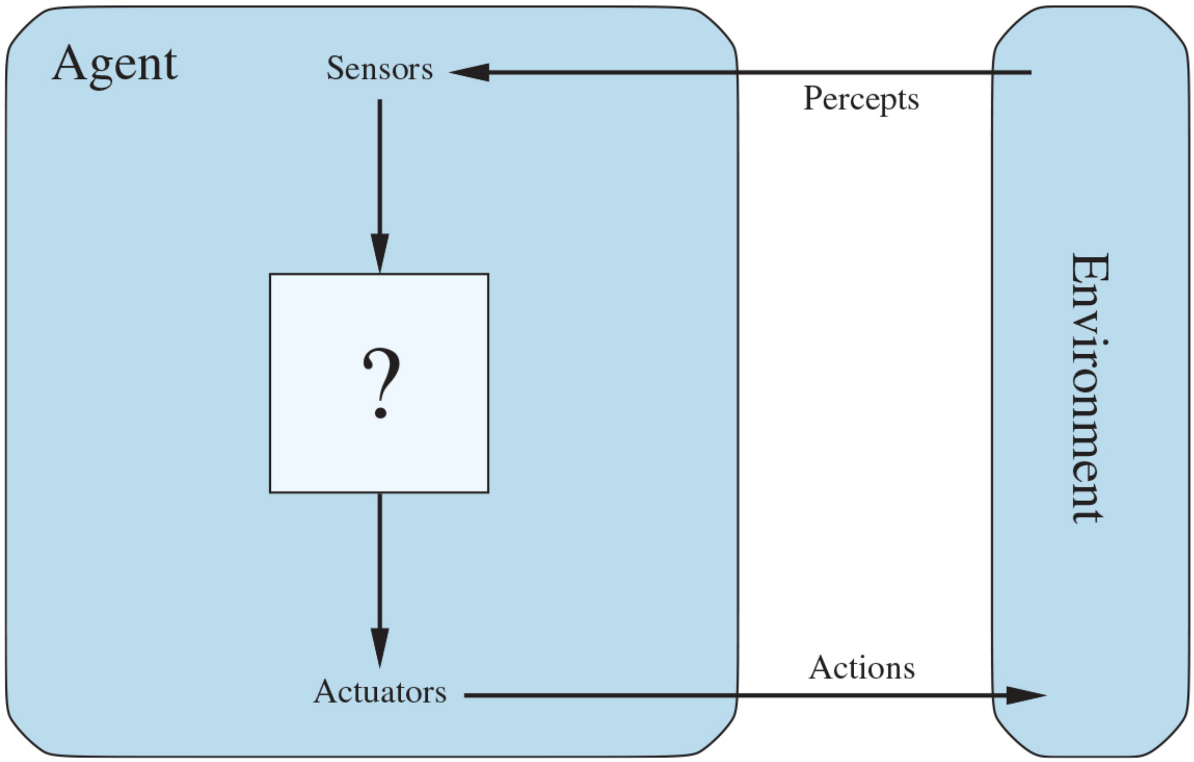


Figure 2.1 Agents interact with environments through sensors and actuators.

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.

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.

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

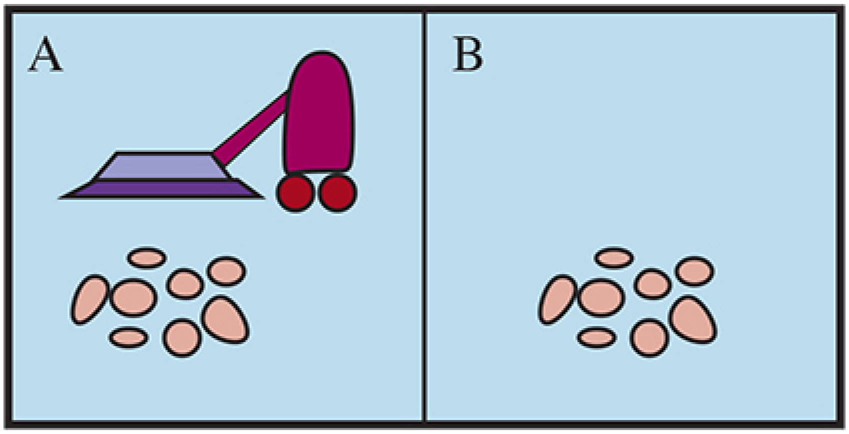


Figure 2.2 A vacuum-cleaner world with just two locations. Each location can be clean or dirty, and the agent can move left or right and can clean the square that it occupies. Different versions of the vacuum world allow for different rules about what the agent can perceive, whether its actions always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

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.

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*
$\vdots$	$\vdots$
$[A, Clean]$, $[A, Clean]$, $[A, Clean]$	*Right*
$[A, Clean]$, $[A, Clean]$, $[A, Dirty]$	*Suck*
$\vdots$	$\vdots$

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?

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.

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 51), 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 51. 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 15, 17, and 23.

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.

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

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 15. 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.

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.

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.

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

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

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.

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.

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.

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.

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.

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 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.

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.

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.

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”).

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.

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, 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.

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 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 57, 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.

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 = *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 26 and 27 deal directly with the sensors and actuators.

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.

```
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.

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 $\sum_{t=1}^{T} \mid \mathcal{P} \mid^{t}$ 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.

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

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.

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.

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 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

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.

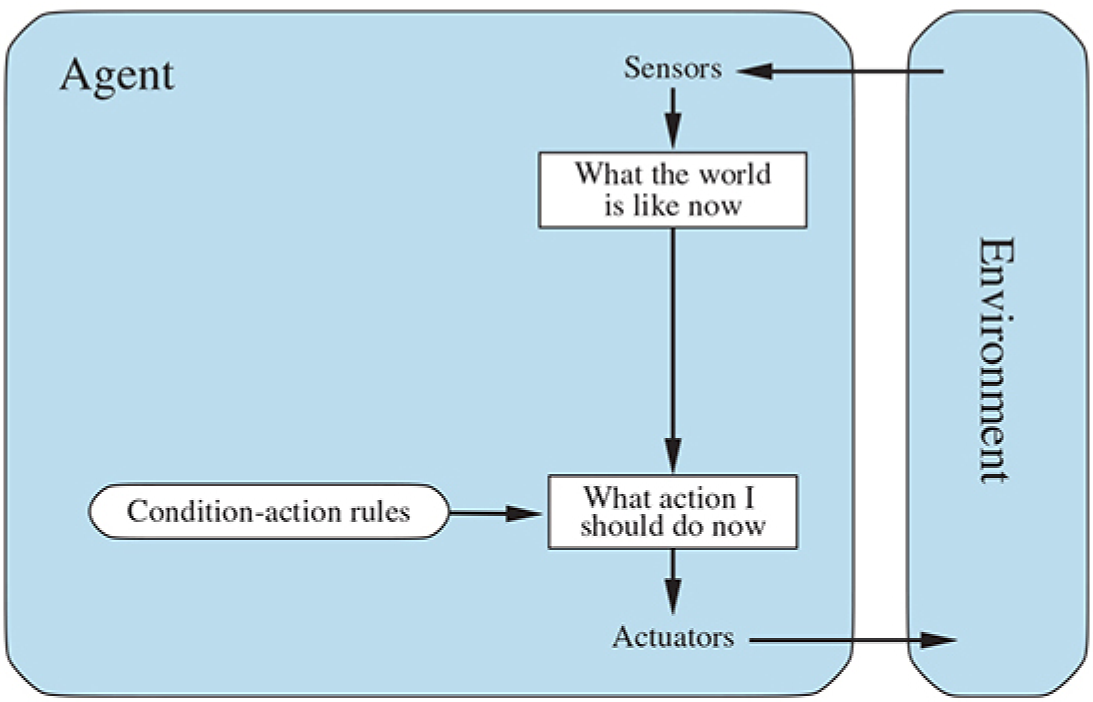


Figure 2.9 Schematic diagram of a simple reflex agent. We use rectangles to denote the current internal state of the agent's decision process, and ovals to represent the background information used in the process.

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 22).

```
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.

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.*

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.

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

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.

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.

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**.

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**.

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.

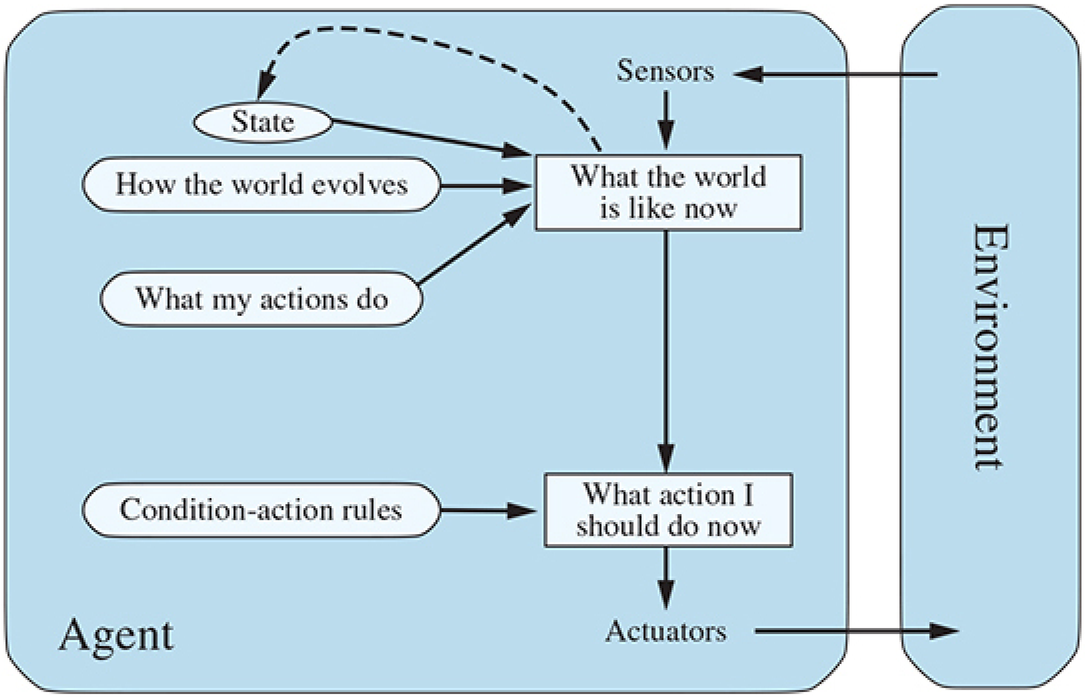


Figure 2.11 A model-based reflex agent.

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.

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 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.

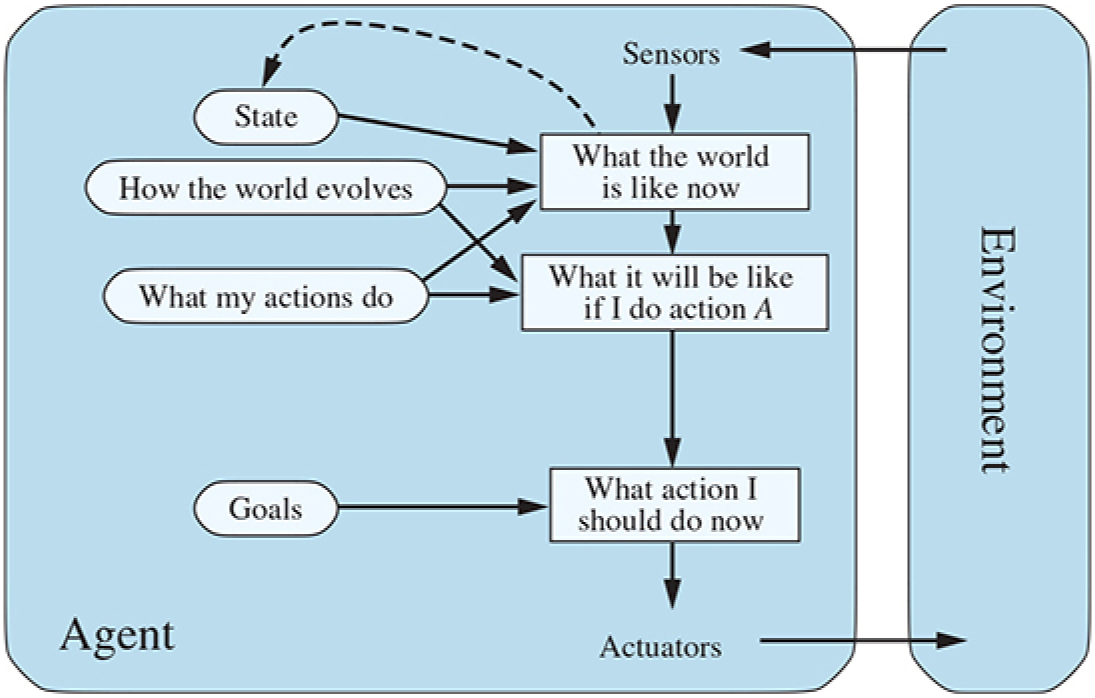


Figure 2.13 A model-based, goal-based agent. It keeps track of the world state as well as a set of goals it is trying to achieve, and chooses an action that will (eventually) lead to the achievement of its goals.

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, 4, and 6) 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, 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]

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.

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 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 15, 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.

The utility-based agent structure appears in Figure 2.14. Utility-based agent programs appear in Chapters 15 and 16, 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 17.

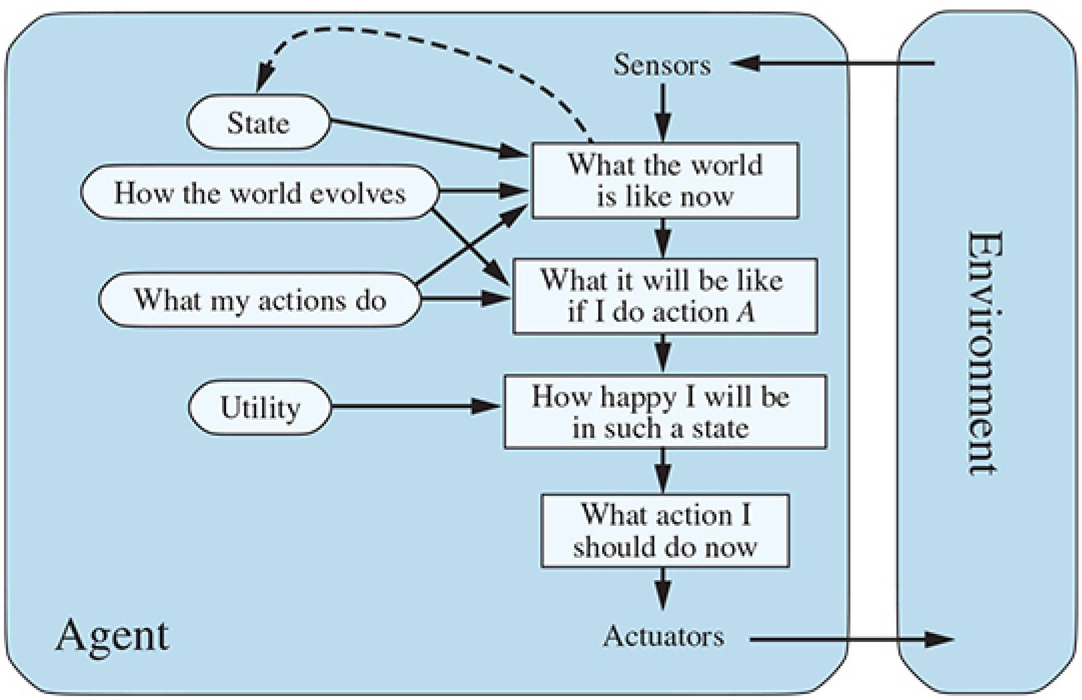


Figure 2.14 A model-based, utility-based agent. It uses a model of the world, along with a utility function that measures its preferences among states of the world. Then it chooses the action that leads to the best expected utility, where expected utility is

computed by averaging over all possible outcome states, weighted by the probability of the outcome.

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 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 23 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.

Finally, all of this assumes that the designer can specify the utility function correctly; Chapters 16, 17, and 23 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, 21, 22, and 23 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 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.

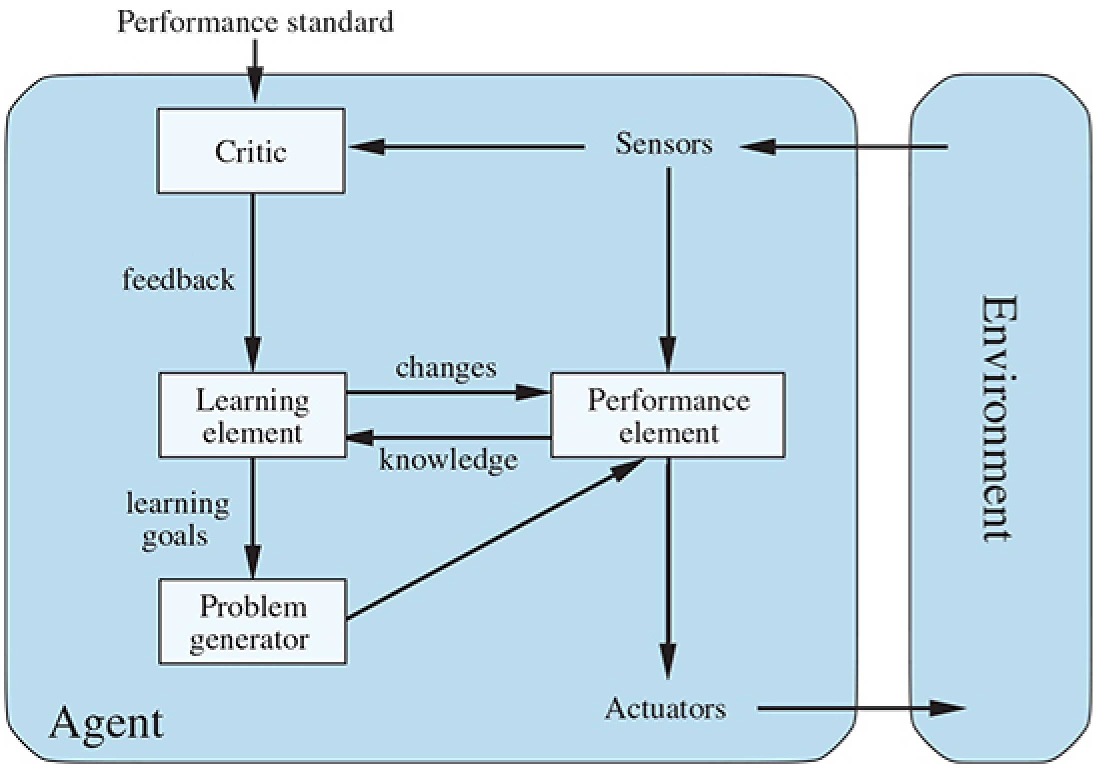


Figure 2.15 A general learning agent. The "performance element" box represents what we have previously considered to be the whole agent program. Now, the "learning element" box gets to modify that program to improve its performance.

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.

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.

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 23.

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.

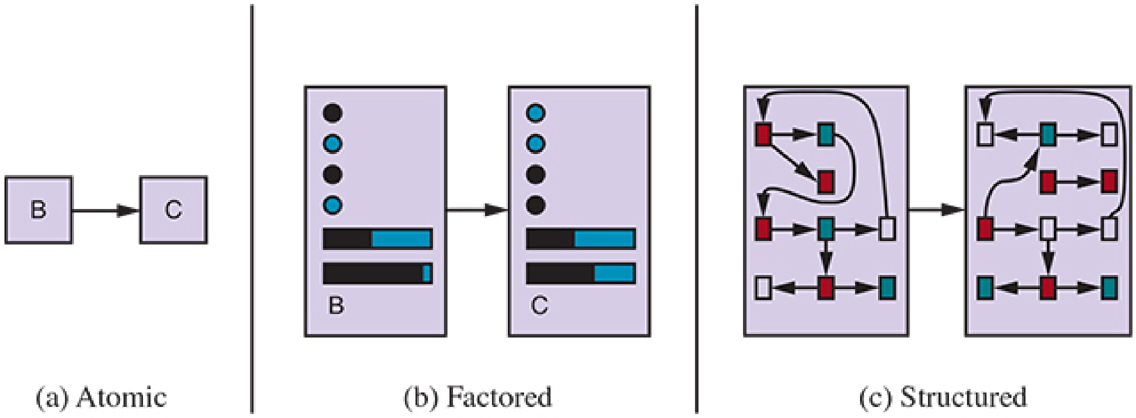


Figure 2.16 Three ways to represent states and the transitions between them. (a) Atomic representation: a state (such as B or C) is a black box with no internal structure; (b) Factored representation: a state consists of a vector of attribute values; values can be Boolean, real-valued, or one of a fixed 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.

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 82). 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, 4, and 6), hidden Markov models (Chapter 14), and Markov decision processes (Chapter 16) all work with atomic representations.

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 5), propositional logic (Chapter 7), planning (Chapter 11), Bayesian networks (Chapters 12, 13, 14, 15, and 18), and various machine learning algorithms.

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 18), and much of natural language understanding (Chapters 24 and 25). In fact, much of what humans express in natural language concerns objects and their relationships.

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.

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.

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).

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 16.

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.

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 15). 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, 21, 22, and 23.

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.

---

[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.

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

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

[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.

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

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

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 *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 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 6 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:

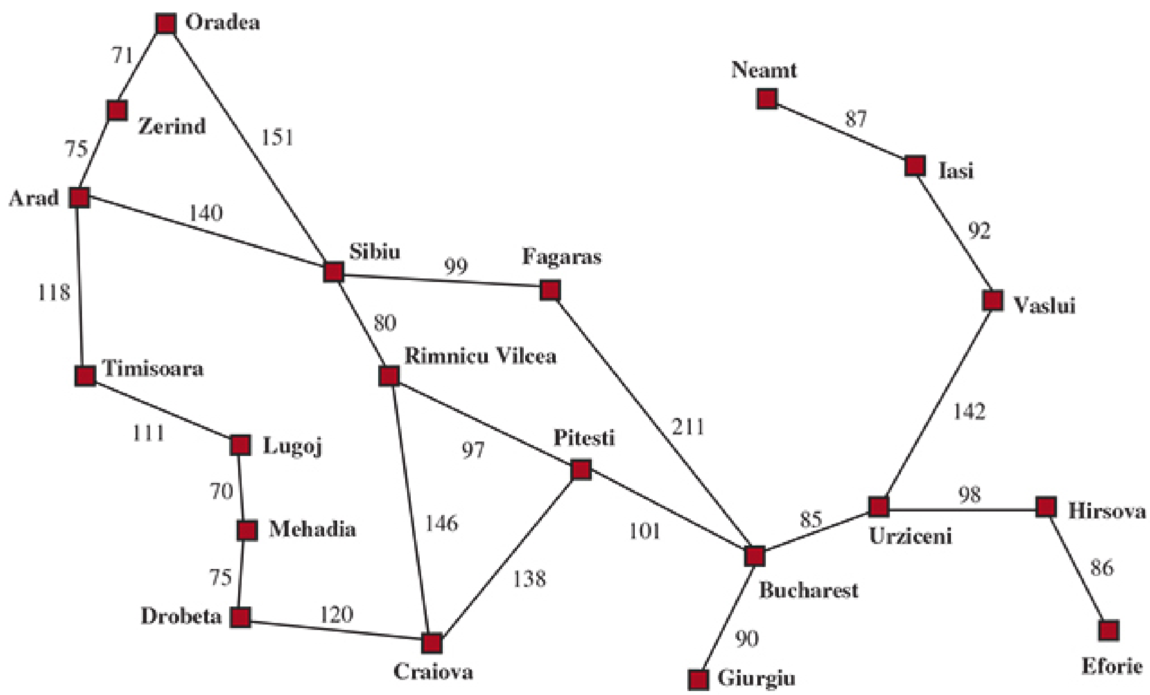


Figure 3.1 A simplified road map of part of Romania, with road distances in miles.

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

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:

- A set of possible **states** that the environment can be in. We call this the **state space**.

- The **initial state** that the agent starts in. For example: *Arad.*
- 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."
- 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:

$$\text{ACTIONS}(Arad) = \{ToSibiu, ToTimisoara, ToZerind\}.$$

- 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,

$$\text{RESULT}(Arad, ToZerind) = Zerind.$$

- 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.

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]

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.

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.

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.

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. 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. 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

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 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 cell 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.

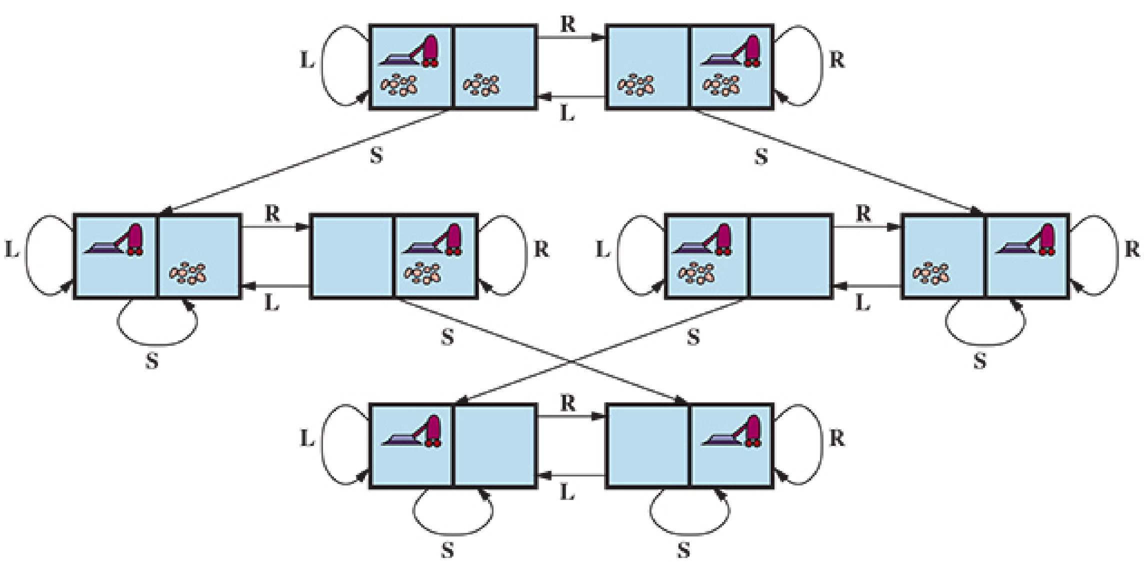


Figure 3.2 The state-space graph for the two-cell vacuum world. There are 8 states and three actions for each state: L = *Left*, R =

Right, S = *Suck.*

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.

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.

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:

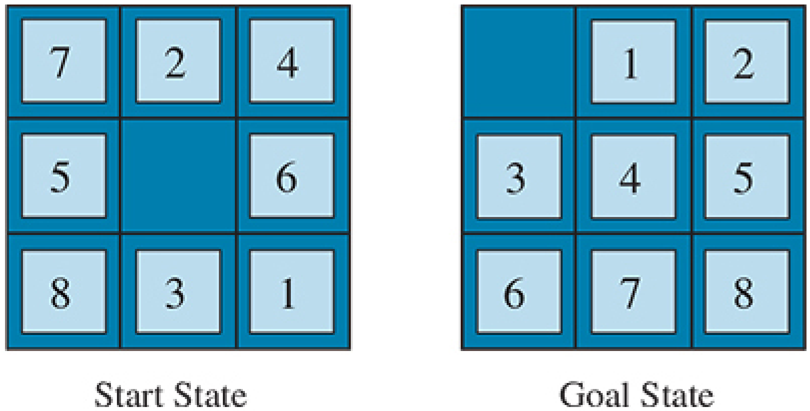


Figure 3.3 A typical instance of the 8-puzzle.

- **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 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.

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 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 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.

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.

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 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**.

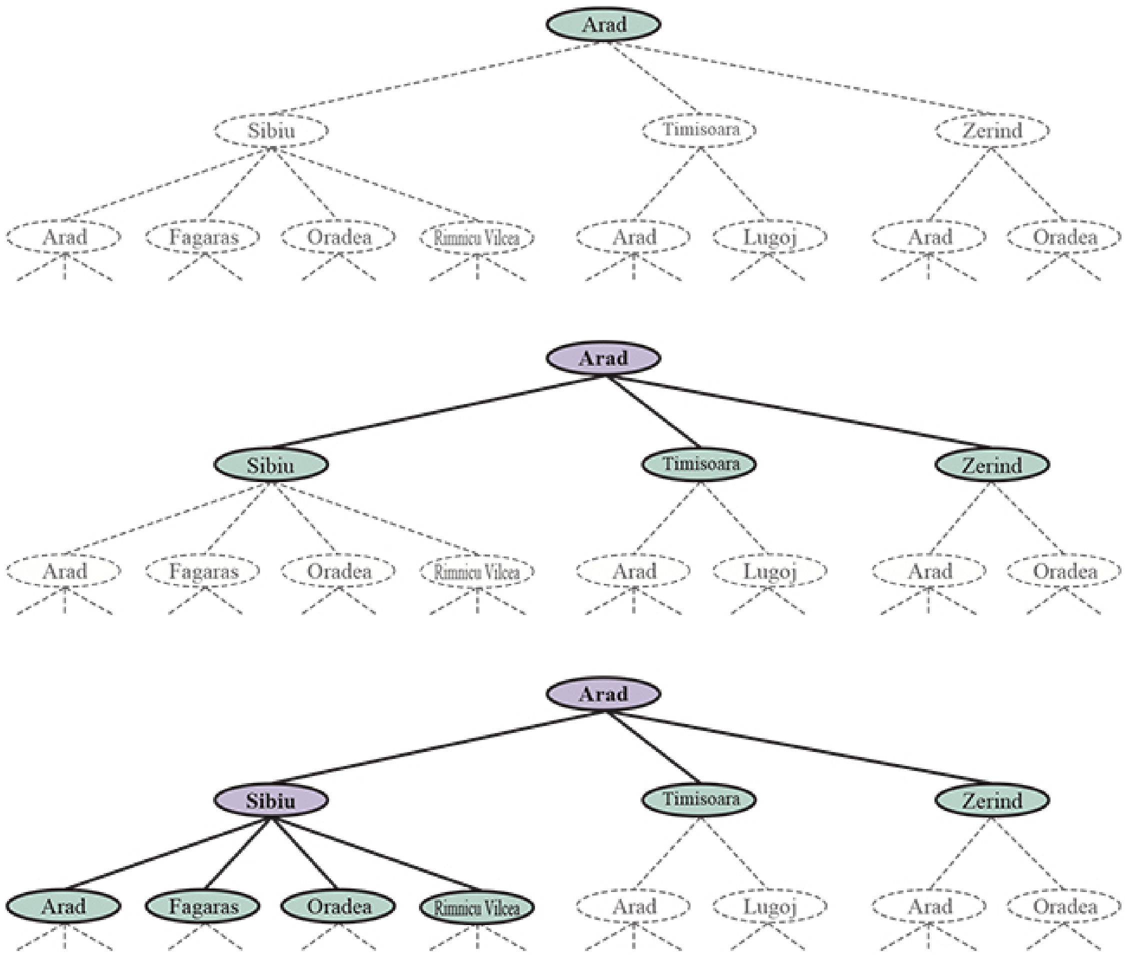


Figure 3.4 Three partial search trees for finding a route from Arad to Bucharest. Nodes that have been *expanded* are lavender with bold letters; nodes on the frontier that have been *generated* but not yet expanded are in green; the set of states corresponding to these two types of nodes are said to 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.

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.

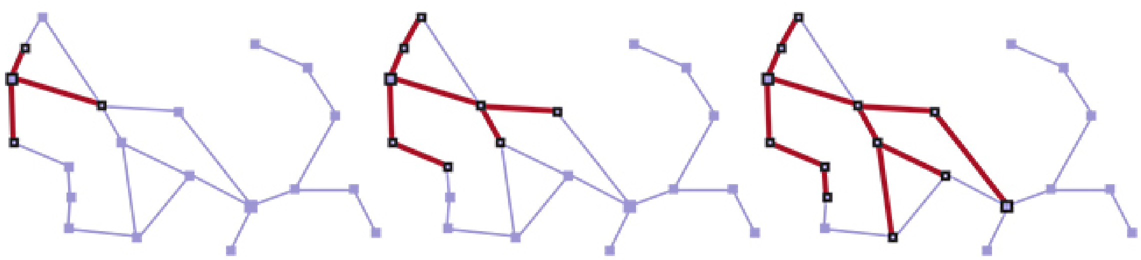


Figure 3.5 A sequence of search trees generated by a graph search on the Romania problem of Figure 3.1. At each stage, we have expanded every node on the frontier, extending every path with all applicable actions that don't result in a state that has already been reached. Notice that at the third stage,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

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.

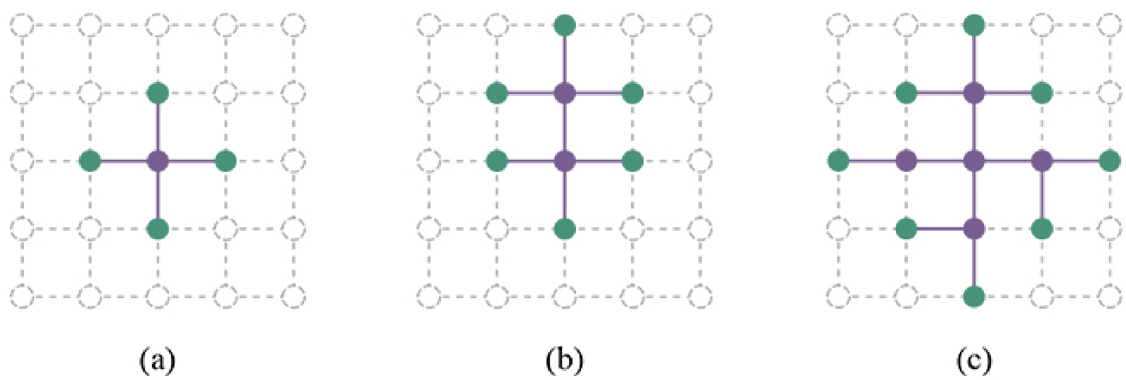


Figure 3.6 The separation property of graph search, illustrated on a rectangular-grid problem. The frontier (green) separates the interior (lavender) from the exterior (faint dashed). The frontier is the set of nodes (and corresponding states) that have been reached but not yet expanded; the interior 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.

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.

```
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 3.7 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**.

3.3.2 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.

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:

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

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.

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.

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 treelike 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?
- **Cost optimality:** Does it find a solution with the lowest path cost of all solutions?[7]
- **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.
- **Space complexity:** How much memory is needed to perform the search?

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.

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 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.

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

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.

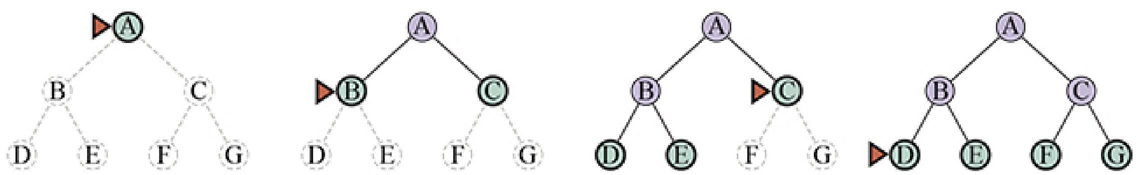


Figure 3.8 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by the triangular marker.

```
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.

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 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.

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.

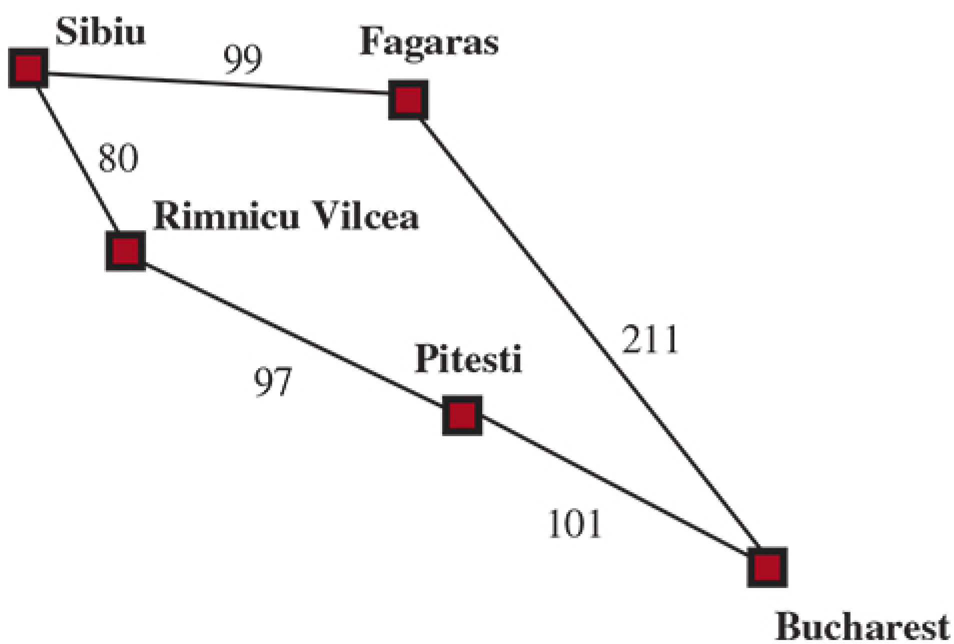


Figure 3.10 Part of the Romania state space, selected to illustrate uniform-cost search.

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

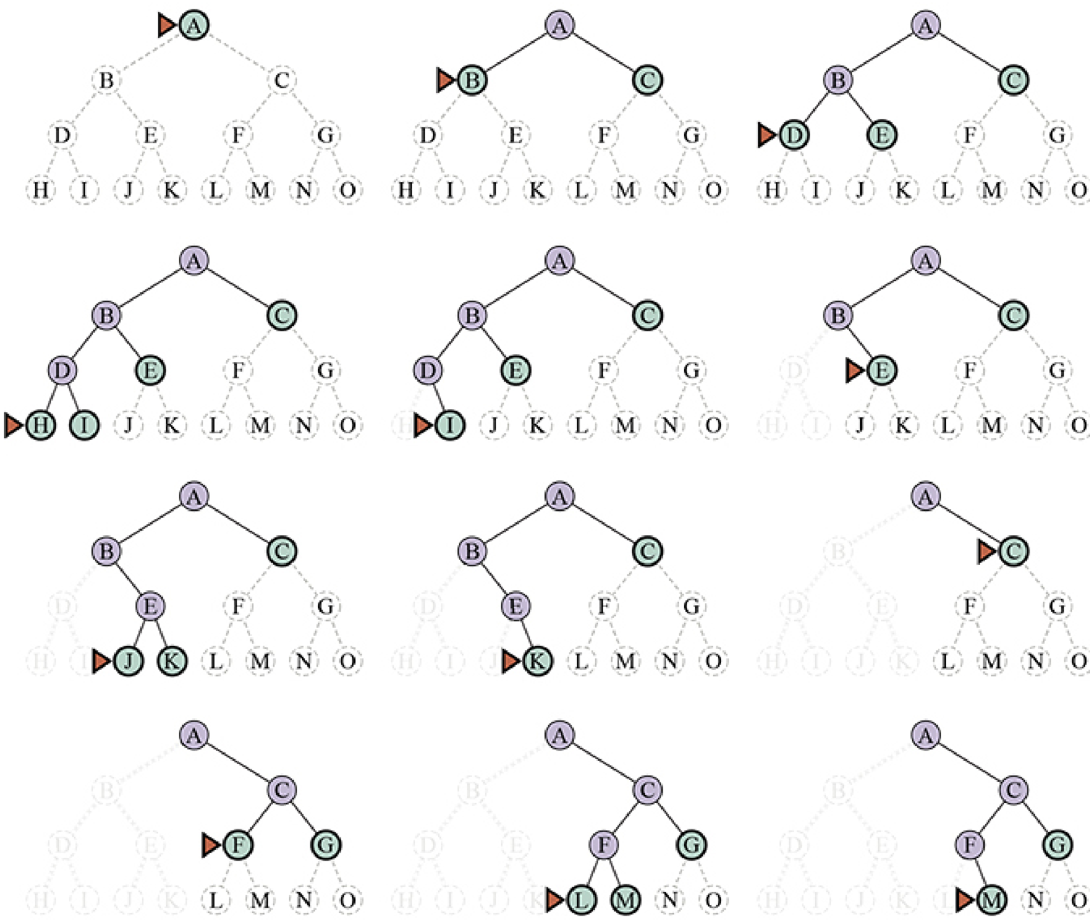


Figure 3.11 A dozen steps (left to right, top to bottom) in the progress of a depth-first search on a binary tree from start state A to goal M. The frontier is in green, with a triangle marking the node to be expanded next. Previously expanded nodes are lavender, and potential future nodes have faint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

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 5), propositional satisfiability (Chapter 7), and logic programming (Chapter 9).

A variant of depth-first search called **backtracking search** uses even less memory. (See Chapter 5 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

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 $O(b^l)$ and the space complexity is $O(bl)$. 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, $l = 19$ 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.

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.

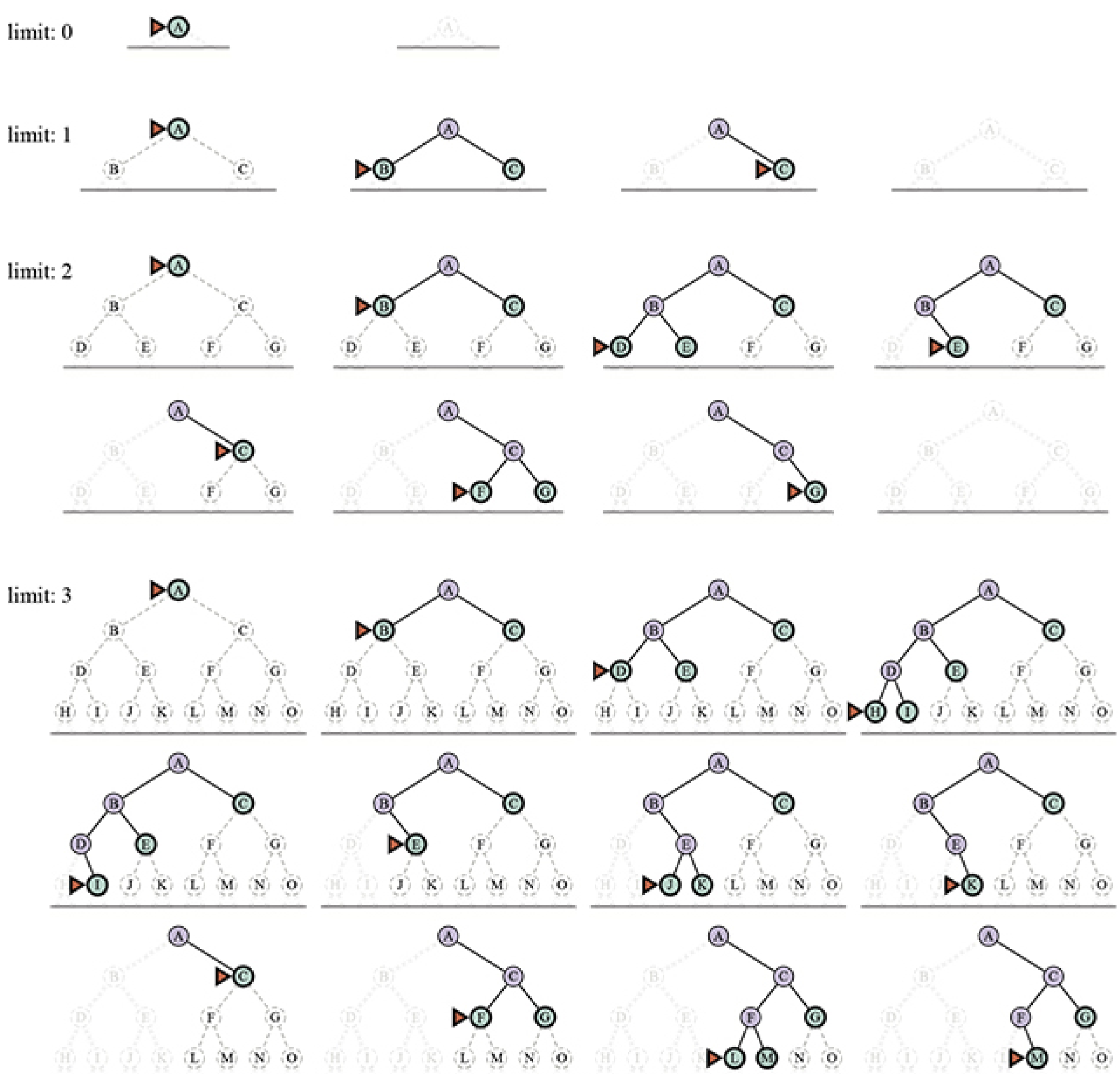


Figure 3.13 Four iterations of iterative deepening search for goal M on a binary tree, with the depth limit varying from 0 to 3. Note the interior nodes form a single path. The triangle marks the node to expand next; green nodes with dark outlines are on the frontier; the very faint nodes provably can't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

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

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

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

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 $b = d = 10$).

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 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.

```
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.

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

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]

> $h(n)$ = estimated cost of the cheapest path from the state at node n 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.

3.5.1 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 $f(n) = h(n)$.

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.

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.

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.

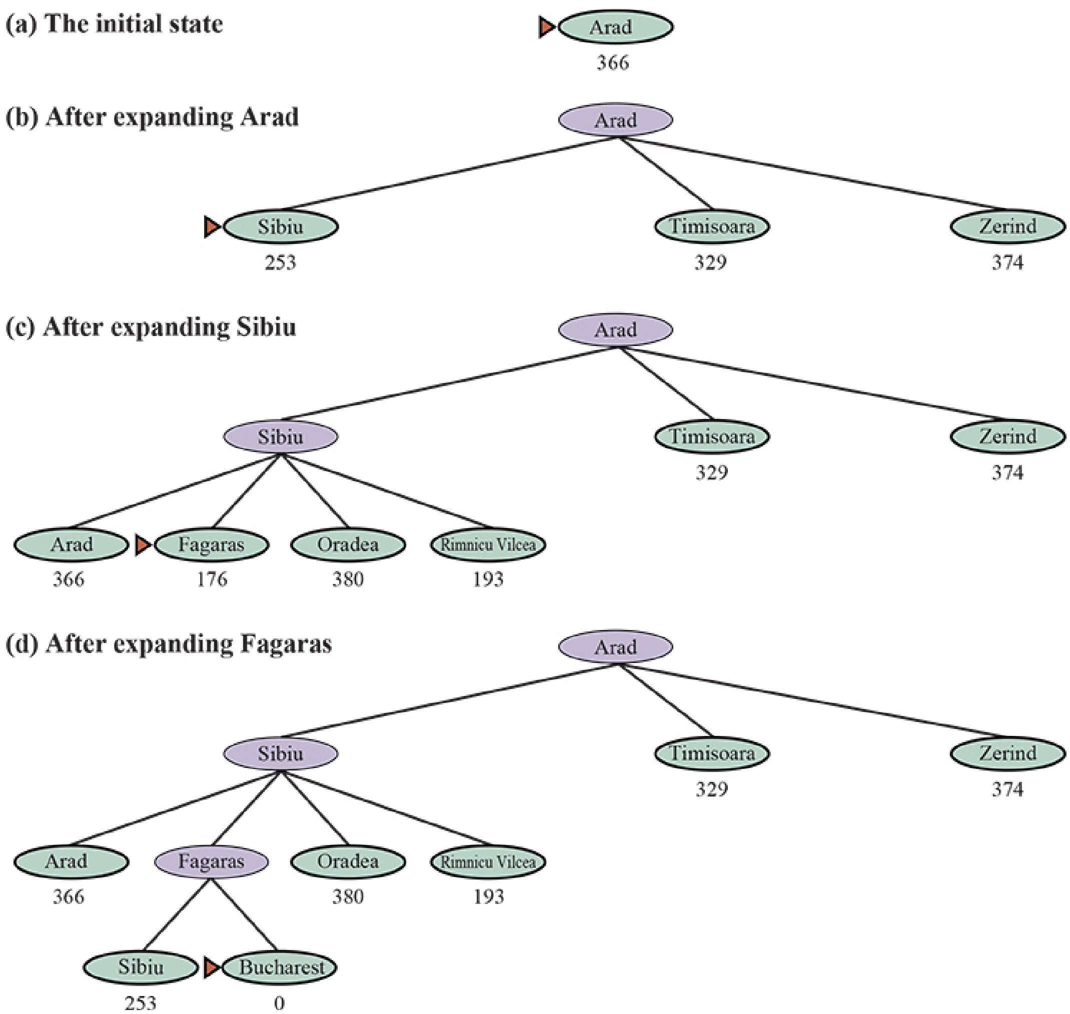


Figure 3.17 Stages in a greedy best-first tree-like search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

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

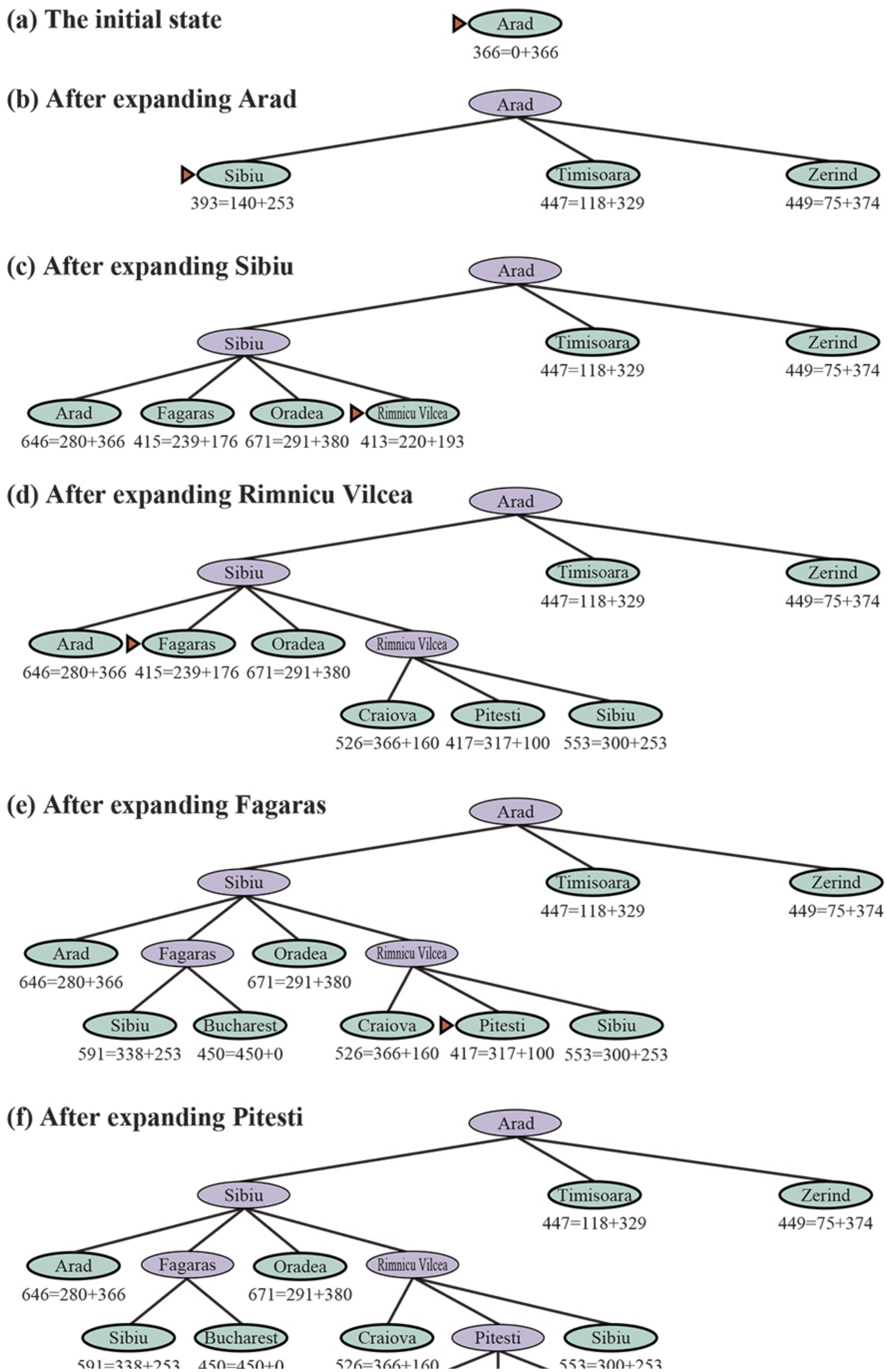
The most common informed search algorithm is **A* search** (pronounced “A-star search”), a best-first search that uses the evaluation function

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

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

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

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.


(a) The initial state
Arad
366=0+366
(b) After expanding Arad
Arad
Sibiu
393=140+253
Timisoara
447=118+329
Zerind
449=75+374
(c) After expanding Sibiu
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
413=220+193
(d) After expanding Rimnicu Vilcea
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253
(e) After expanding Fagaras
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253
(f) After expanding Pitesti
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
Bucharest
Craiova
Pitesti
Sibiu

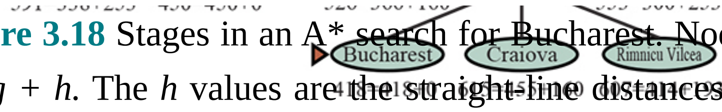


Figure 3.18 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 3.16.

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 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:

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

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.

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:

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

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.

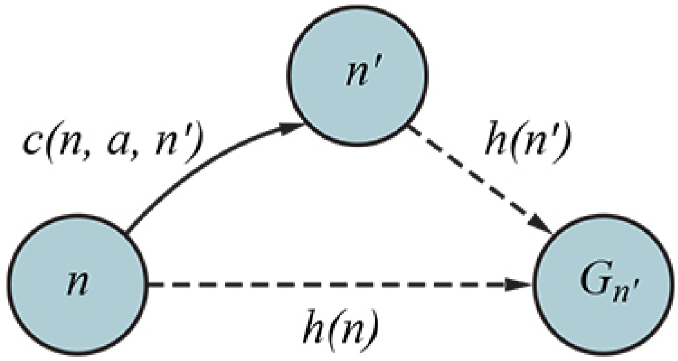


Figure 3.19 Triangle inequality: If the heuristic h is **consistent**, then the single number $h(n)$ will be less than the sum of the cost $c(n, a, a')$ of the action from n to n' plus the heuristic estimate $h(n')$.

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.

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.

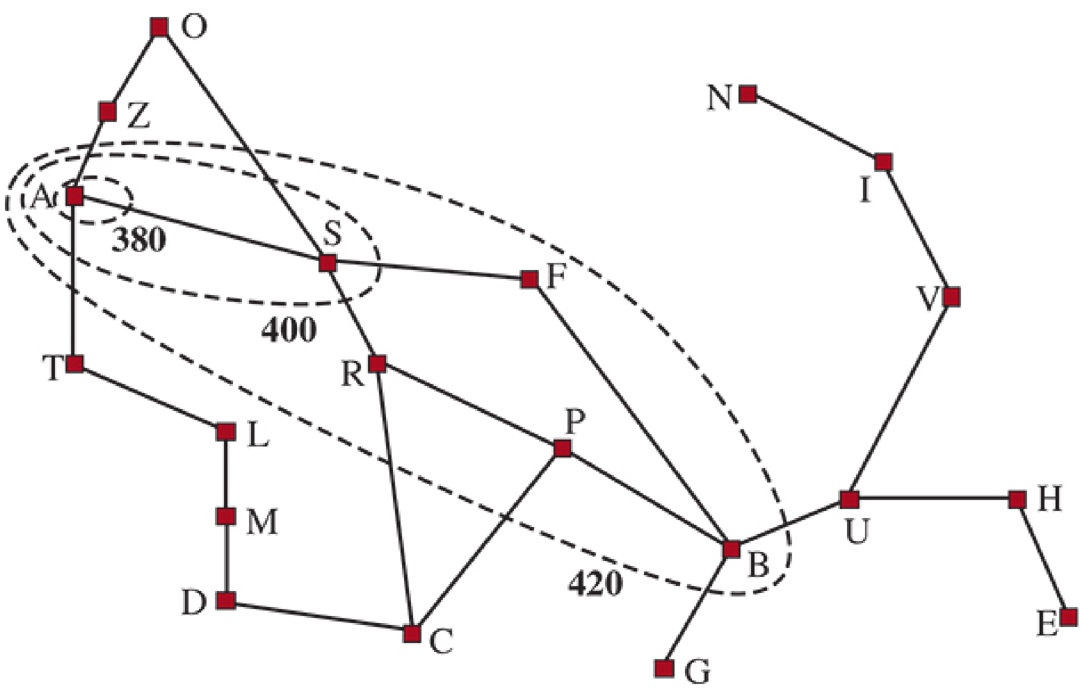


Figure 3.20 Map of Romania showing contours at $f = 380$, $f = 400$, and $f = 420$, with Arad as the start state. Nodes inside a given contour have $f =$

$g + h$ costs less than or equal to the contour value.

With 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.

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 $h(n) \leq c(n, \mathrm{a}, n') + h(n')$; 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:

- 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 $f(n) > C^*$.

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.

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 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.

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 $W > 1$.

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 $W \times C^*$.

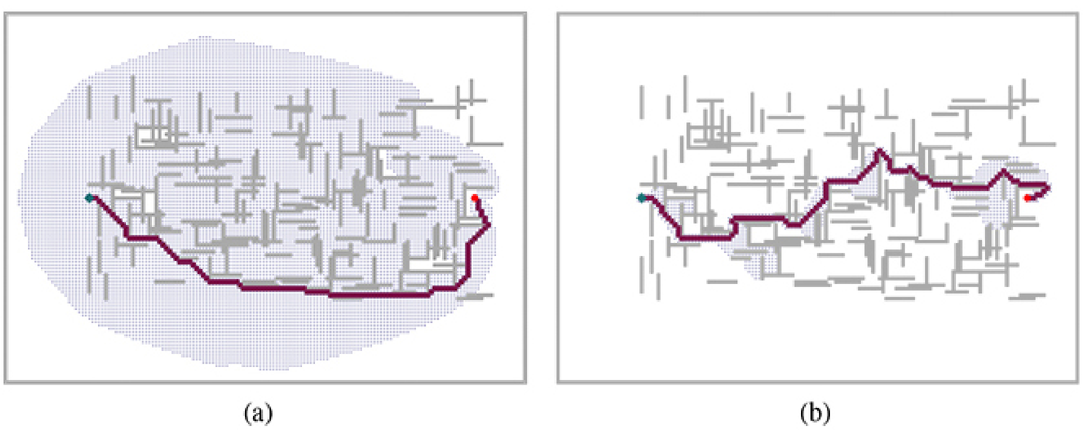


Figure 3.21 Two searches on the same grid: (a) an A* search and (b) a weighted A* search with weight $W = 2$. The gray bars are obstacles, the purple line is the path from the green start to red goal, and the small dots are states that were reached by each search. On this particular problem, weighted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

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:

$$\begin{array}{rccl} \text{A* search:} & g(n) + h(n) & (W = 1) \\ \text{Uniform-cost search:} & g(n) & (W = 0) \\ \text{Greedy best-first search:} & h(n) & (W = \infty) \\ \text{Weighted A* search:} & g(n) + W \times \ h(n) & (1 < W < \infty) \end{array}$$

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.

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.

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 90), 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.

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 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.

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.

```
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.

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

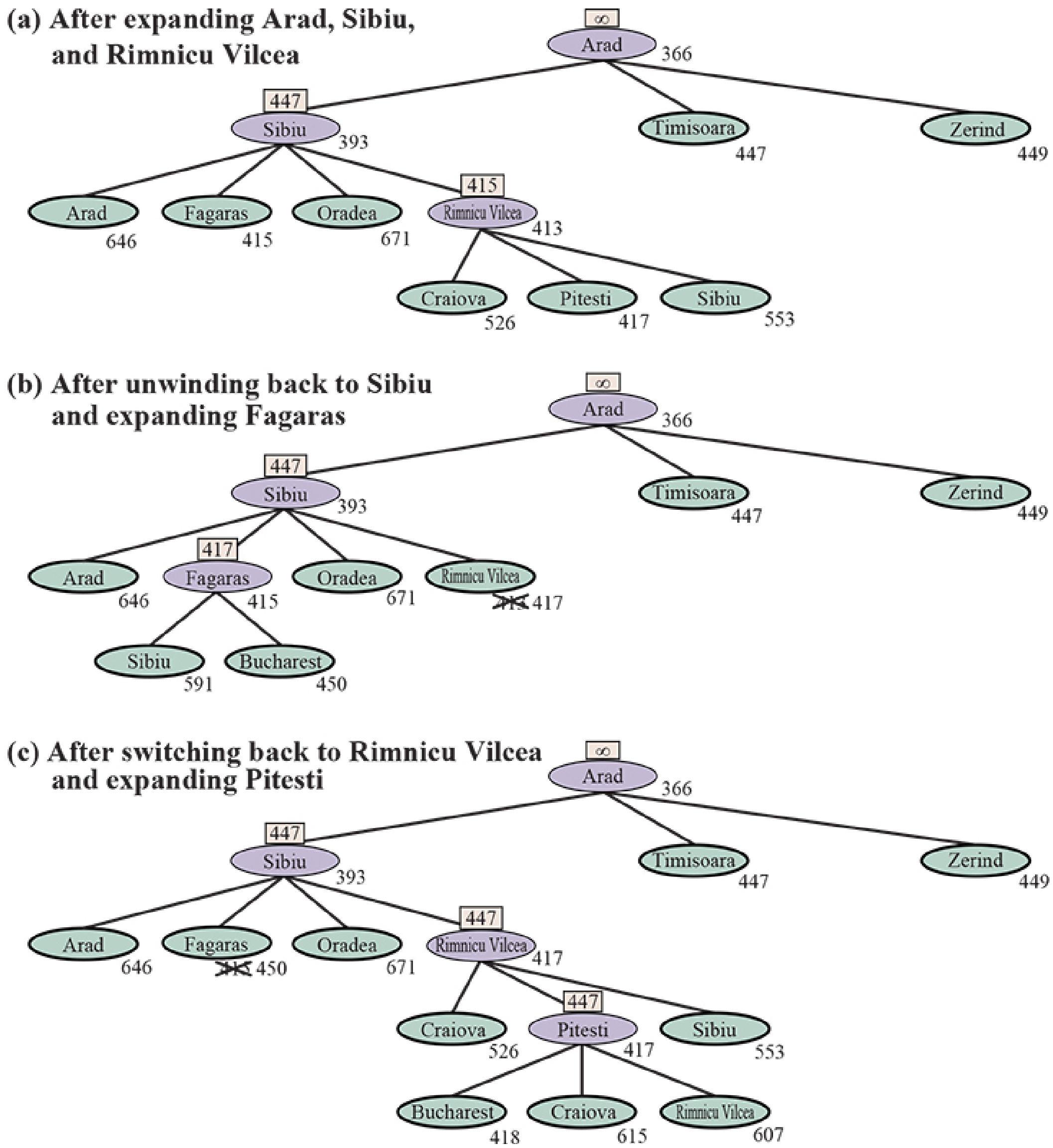


Figure 3.23 Stages in an RBFS search for the shortest route to Bucharest. The *f-limit* value for each recursive call is shown on top of each current node, and every node is labeled with its *f*-cost. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is 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.

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*.

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.

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

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

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.

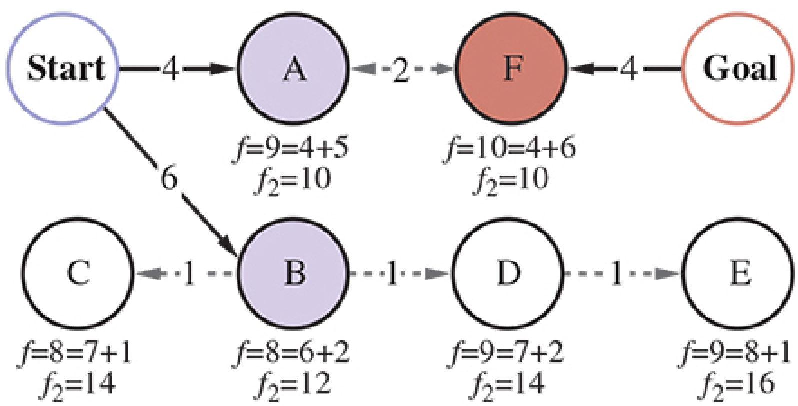


Figure 3.24 Bidirectional search maintains two frontiers: on the left, nodes A and B are successors of Start; on the right, node F is an inverse successor of Goal. Each node is labeled with $f = g + h$ values and the $f_2 = \max(2g, g + h)$ value. (The g values are the sum of the action costs as shown 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.

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 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).

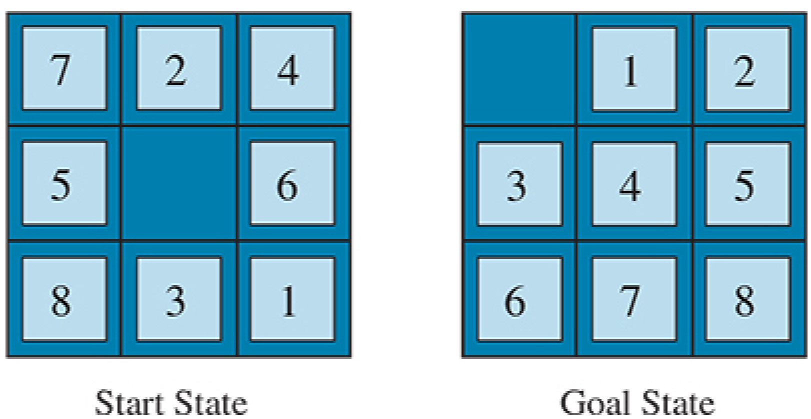


Figure 3.25 A typical instance of the 8-puzzle. The shortest solution is 26 actions long.

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.
- 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

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

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

3.6.1 The effect of heuristic accuracy on performance

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,

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

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.

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 108 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.

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.

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 106).

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

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

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

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

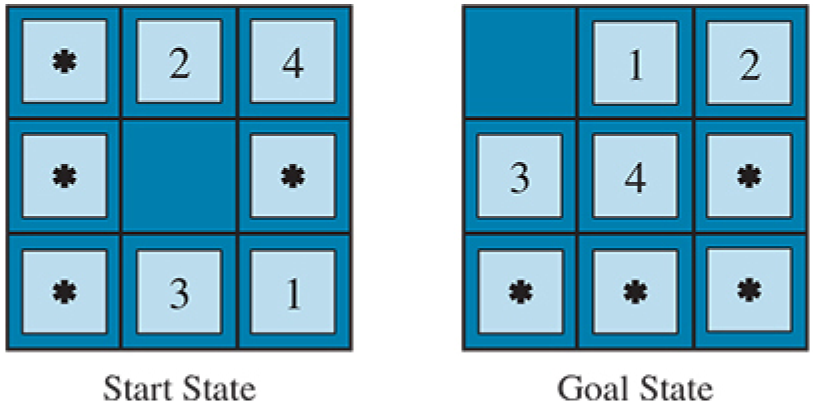


Figure 3.27 A subproblem of the 8-puzzle instance given in Figure 3.25. The task is to get tiles 1, 2, 3, 4, and the blank into their correct positions, without worrying about what happens to the other tiles.

The 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 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.

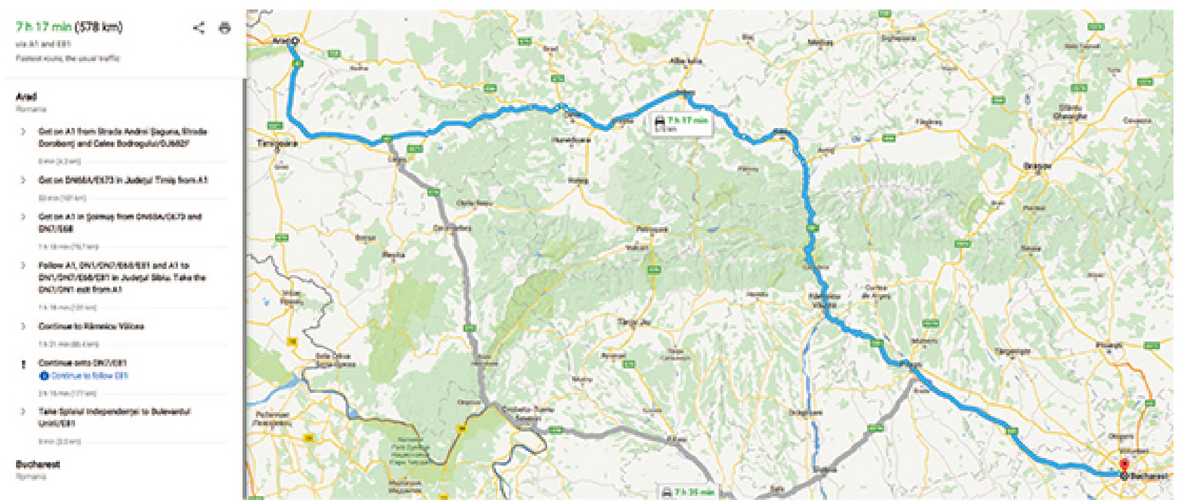


Figure 3.28 A Web service providing driving directions, computed by a search algorithm.

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

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

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

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 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.

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:

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

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 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 23. The goal of learning is to minimize the **total cost** of problem solving, trading off computational expense and path cost.

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 23 are also applicable to search.

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 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.

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.”

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 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.*

---

[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.

[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.

[4] See Section 11.4.

[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.*

[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.

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

[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.

[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 $g(n)$.

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

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

[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).

[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.

[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 16.

[16] Landmark points are sometimes called "pivots" or "anchors."

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 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.

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

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, 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**.

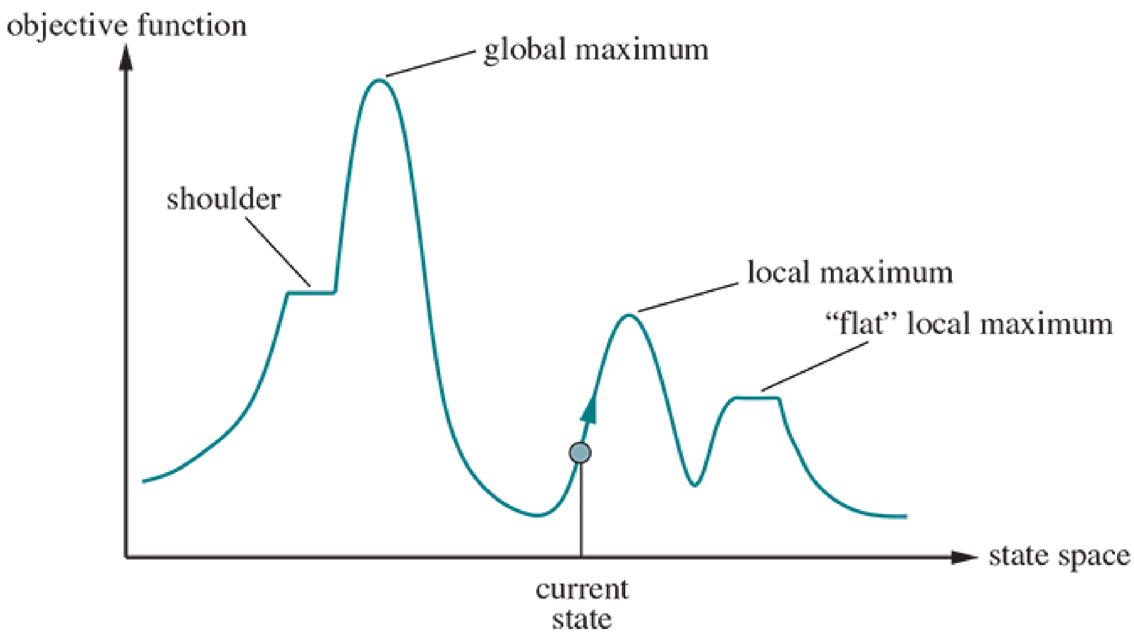


Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum.

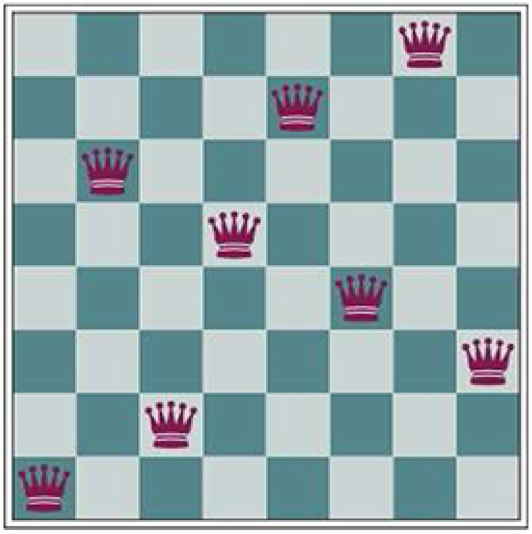
4.1.1 Hill-climbing search

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.

```
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.

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 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.



(a)

(b)

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.

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 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.
- **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.
- **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.

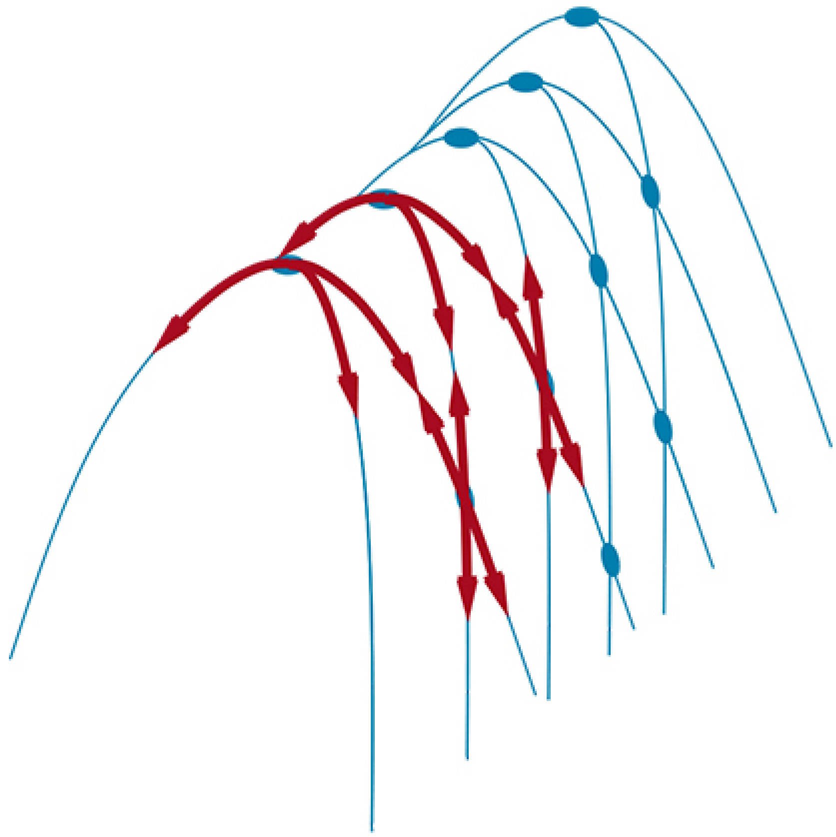


Figure 4.4 Illustration of why ridges cause difficulties for hill climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point 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.

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.

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.

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

```
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^(ΔE/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.

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.

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.

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:

- The size of the population.
- 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.
- 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.
- 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.
- 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.
- 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-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).

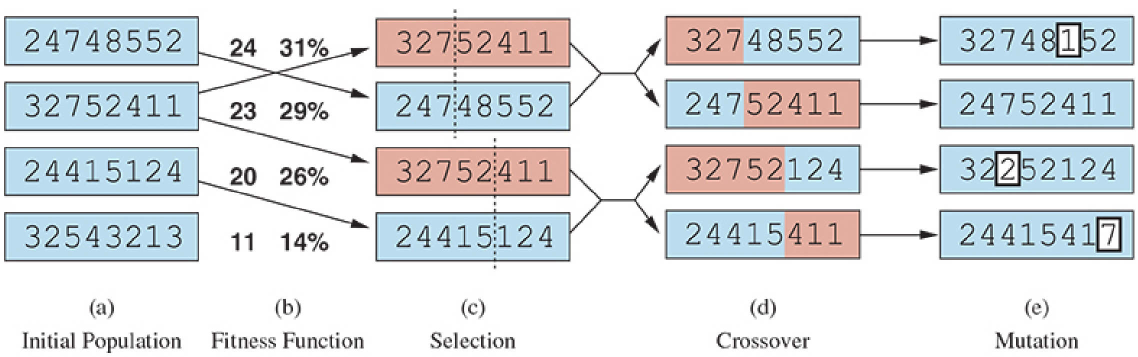


Figure 4.6 A genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by a fitness function in (b) resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

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.

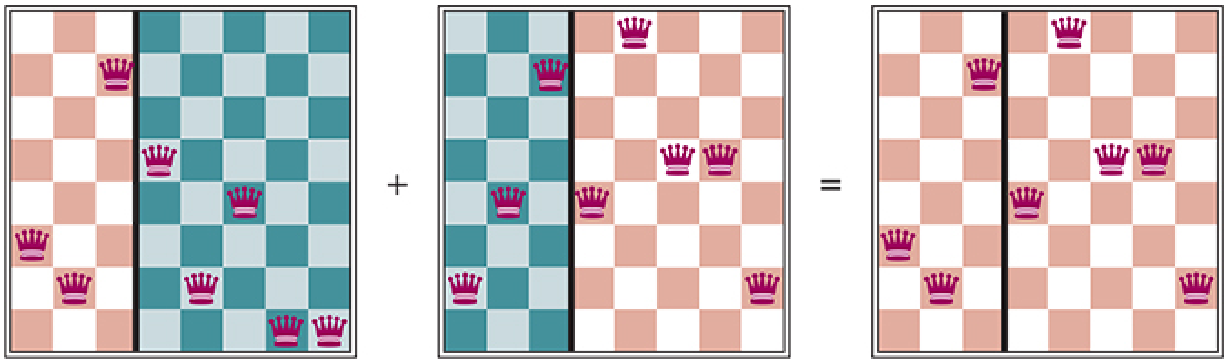


Figure 4.7 The 8-queens states corresponding to the first two parents in Figure 4.6(c) and the first offspring in Figure 4.6(d). The green columns are lost in the crossover step and the red columns are retained. (To interpret the numbers in Figure 4.6: row 1 is the bottom row, and 8 is the top row.)

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.

```
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.

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.

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).

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

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

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.

One way to deal with a continuous state space is to **discretize** it. For example, instead of allowing the $(x_\mathrm{i}, y_\mathrm{i})$ 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.

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

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

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

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

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.

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

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

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.

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 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.

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 21).

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.**

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.

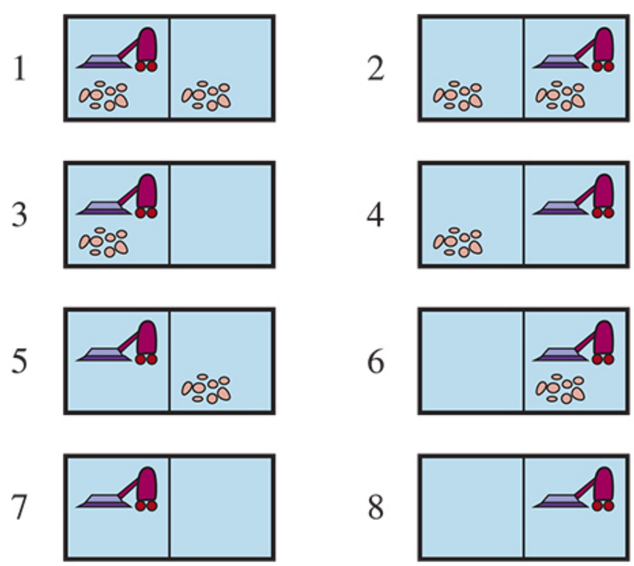


Figure 4.9 The eight possible states of the vacuum world; states 7 and 8 are goal states.

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 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:

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

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

$$[\mathit{Suck}, \mathbf{if}\ \mathit{State} = 5\ \mathbf{then}\ [\mathit{Right}, \mathit{Suck}]\ \mathbf{else}\ [\,]]\ . \tag{4.3}$$

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, 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 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.

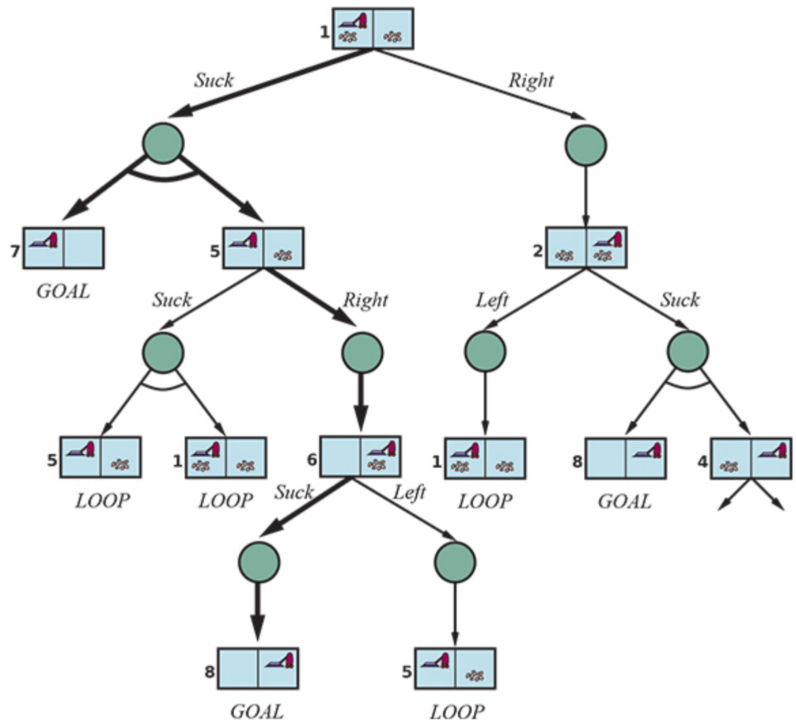


Figure 4.10 The first two levels of the search tree for the erratic vacuum world. State nodes are OR nodes where some action must be chosen. At the AND nodes, shown as circles, every outcome must be handled, as indicated by the arc linking the outgoing branches. The solution found is shown in bold lines.

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.

```
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(state, path) then return failure
  for each action in problem.ACTIONS(state) do
     plan ← AND-SEARCH(problem, RESULTS(state, action), [state] + [path])
     if plan ≠ 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.

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.)

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:

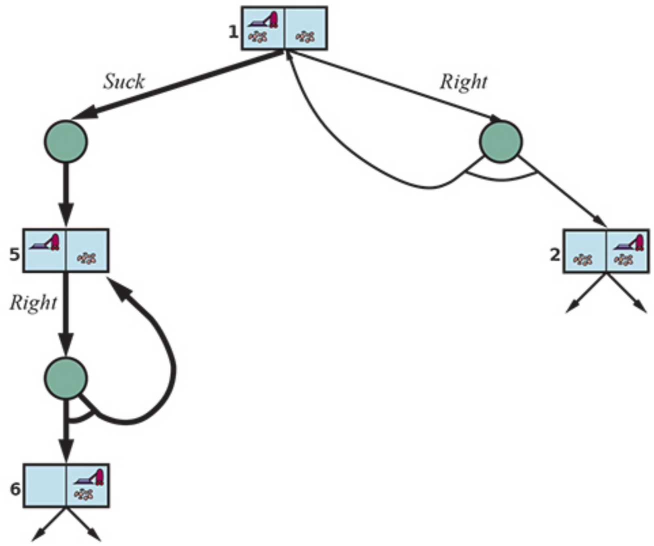


Figure 4.12 Part of the search graph for a slippery vacuum world, where we have shown (some) cycles explicitly. All solutions for this problem are cyclic plans because there is no way to move reliably.

$$[Suck, \textbf{while}\ State = 5\ \textbf{do}\ Right, Suck]$$

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.

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

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.

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 $\text{ACTIONS}_P(s_1) \neq \text{ACTIONS}_P(s_2)$; 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:

$$\text{ACTIONS}(b) = \bigcup_{s \in b} \text{ACTIONS}_P(s).$$

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):

$$b' = \text{RESULT}\,(b, a) = \{s' : s' = \text{RESULT}_P\,(s, a) \text{ and } s \in b\}.$$

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:

$$\begin{aligned} b' = \text{RESULT}\,(b, a) &= \{s' : s' \in \text{RESULTS}_P\,(s, a) \text{ and } s \in b\} \\ &= \bigcup_{s \in b} \text{RESULTS}_P(s, a)\,, \end{aligned}$$

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).

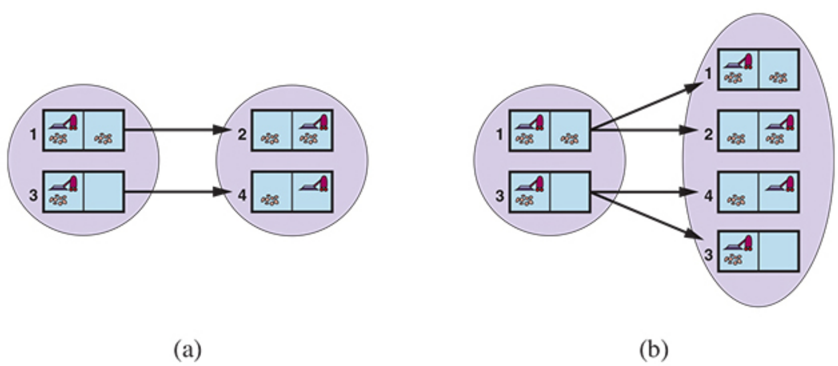


Figure 4.13 (a) Predicting the next belief state for the sensorless vacuum world with the deterministic action, *Right*. (b) Prediction for the same belief state and action in the slippery version of the sensorless vacuum world.

- **Goal test:** The agent *possibly* achieves the goal if *any* state s in the belief state satisfies the goal test of the underlying problem, Is-Goal$_P(s)$. The agent *necessarily* achieves the goal if *every* state satisfies 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 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.

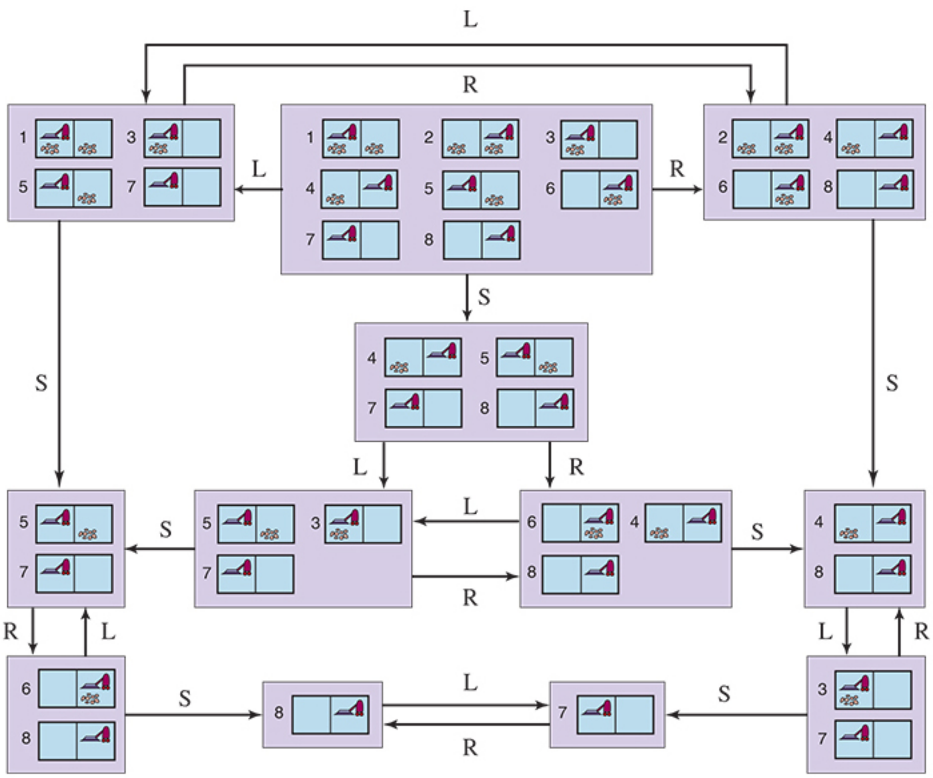


Figure 4.14 The reachable portion of the belief-state space for the deterministic, sensorless vacuum world. Each rectangular box corresponds to a single belief state. At any given point, the agent has a belief state but does not know which physical state it is in. The initial belief state (complete ignorance) is the top center box.

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 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.

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 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, PERCEPT(s) = s for every state s, and for sensorless problems 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:

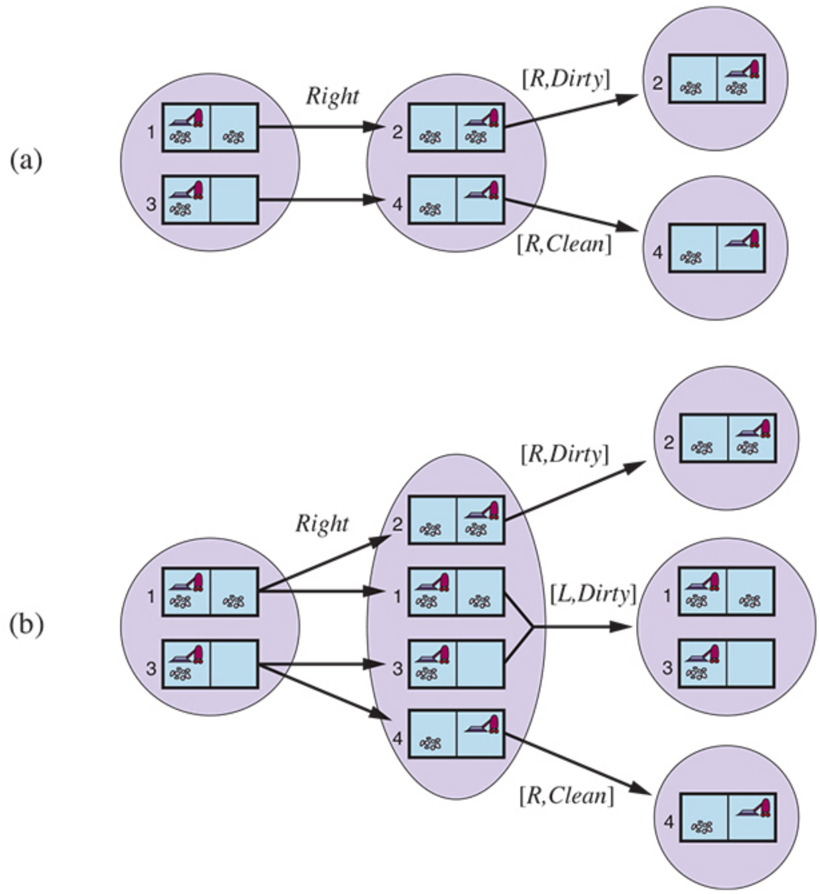


Figure 4.15 Two examples of transitions in local-sensing vacuum worlds. (a) In the deterministic world, *Right* is applied in the initial belief state, resulting in a new predicted belief state with two possible physical states; for those states, the possible percepts are [*R*, *Dirty*] and [*R*, *Clean*], 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.

- The **prediction** stage computes the belief state resulting from the action, RESULT(b, a), exactly as we did with sensorless problems. To emphasize that this is a prediction, we use the notation $\hat{b}$ = RESULT(b, a), where the "hat" over the b means "estimated," and we also use PREDICT(b, a) as a synonym for 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):

$$\text{POSSIBLE-PERCEPTS}\left(\hat{b}\right) = \left\{o\text{: } o = \text{PERCEPT}\left(s\right) \text{ and } s \in \hat{b}\right\}.$$

- 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:

$$b_o = \text{UPDATE}\,(\hat{b},\ a) = \left\{ s : o = \text{PERCEPT}\,(s) \text{ and } s\ \in\ \hat{b} \right\}.$$

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:

$$\begin{aligned}\text{RESULTS}(b,a) = \{b_o : b_o = \text{UPDATE}\,(&\text{PREDICT}\,(b, a), o) \text{ and} \\ &o\ \in\ \text{POSSIBLE-PERCEPTS}\,(\text{PREDICT}(b,a))\}. \qquad (4.5)\end{aligned}$$

4.4.3 Solving partially observable problems

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

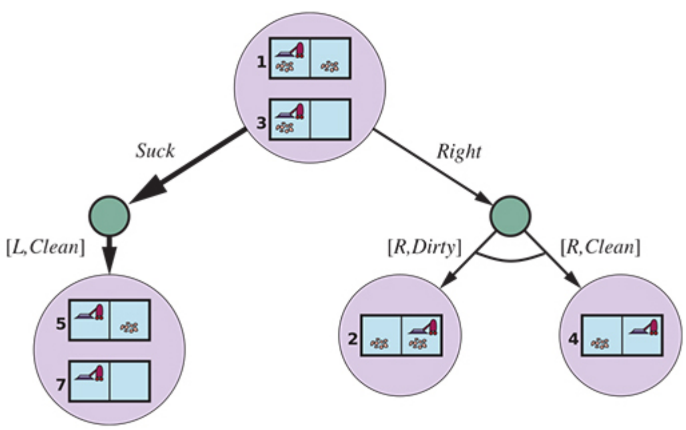


Figure 4.16 The first level of the AND–OR search tree for a problem in the local-sensing vacuum world; *Suck* is the first action in the solution.

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

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.

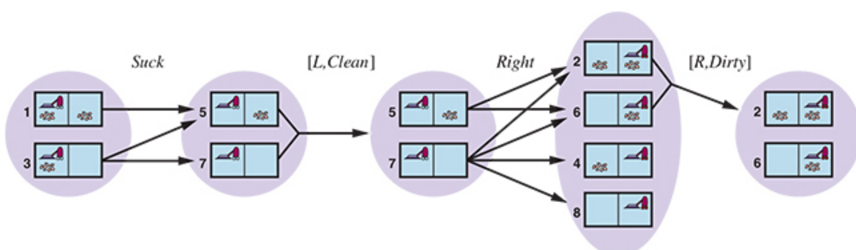
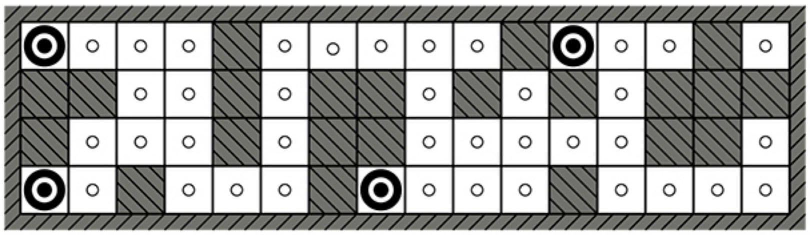


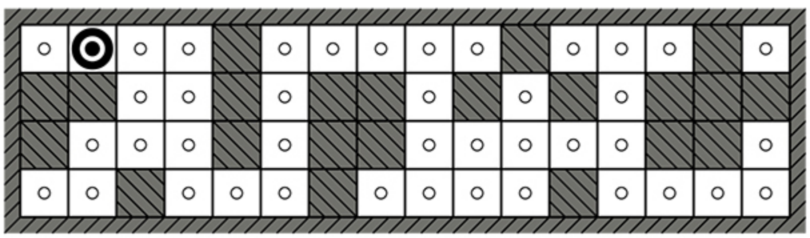
Figure 4.17 Two prediction–update cycles of belief-state maintenance in the kindergarten vacuum world with local sensing.

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 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.



(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, ʘ, (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.

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 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

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 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.

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 16 relaxes these assumptions) and stipulate that the agent knows only the following:

- 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.
- IS-GOAL(s), the goal test.

Note in particular that the agent *cannot* determine 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.

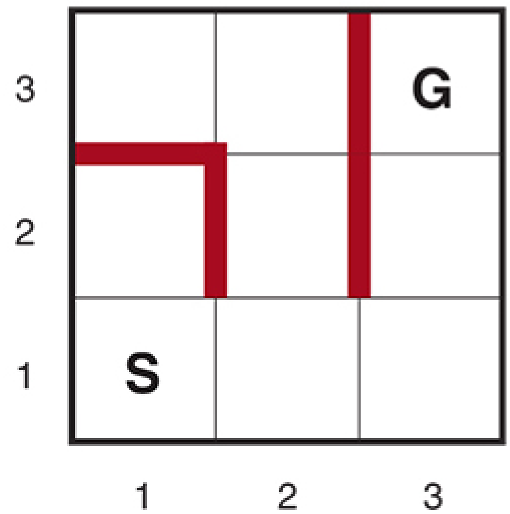


Figure 4.19 A simple maze problem. The agent starts at S and must reach G but knows nothing of the environment.

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 116).

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.

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 space or the bottom one; the two look identical based on what the agent has seen. Therefore, there is no 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).

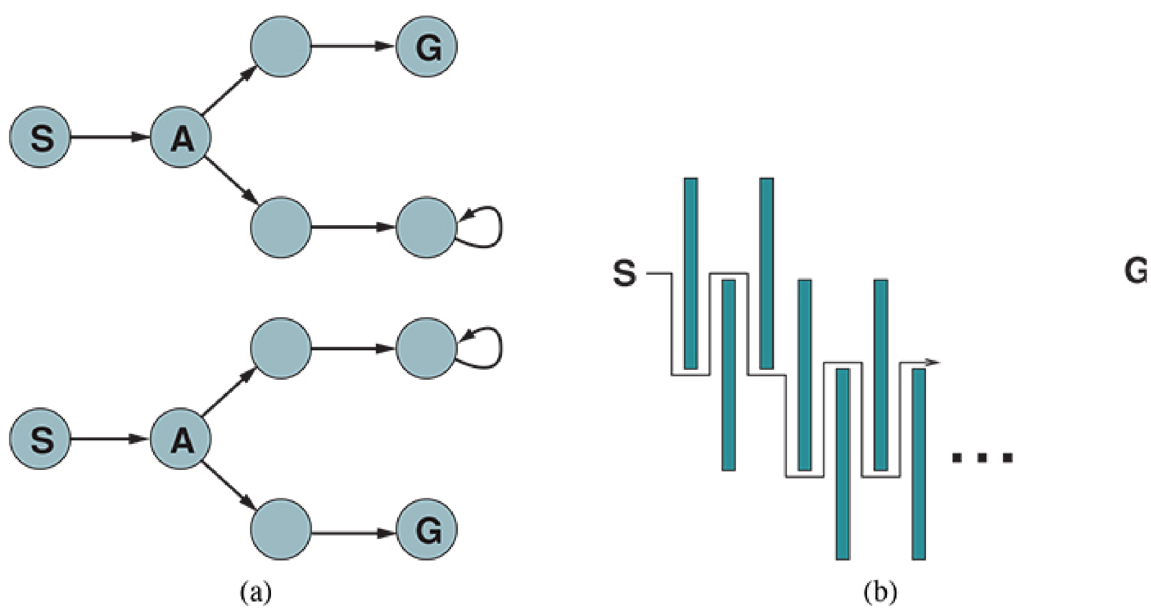


Figure 4.20 (a) Two state spaces that might lead an online search agent into a dead end. Any given agent will fail in at least one of these spaces. (b) A two-dimensional environment that can cause an online search agent to follow an arbitrarily inefficient route to the goal. Whichever choice the agent makes, the adversary blocks that route with another long, thin wall, so that the path followed is much longer than the best possible path.

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 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 23.3.2.

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 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.

```
function ONLINE-DFS-AGENT(problem, s′) returns an action
        s, a, the previous state and action, initially null
        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
       a ← an action b such that result[s′, b] = POP(unbacktracked[s′])s′ ← null
  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.

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.

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.

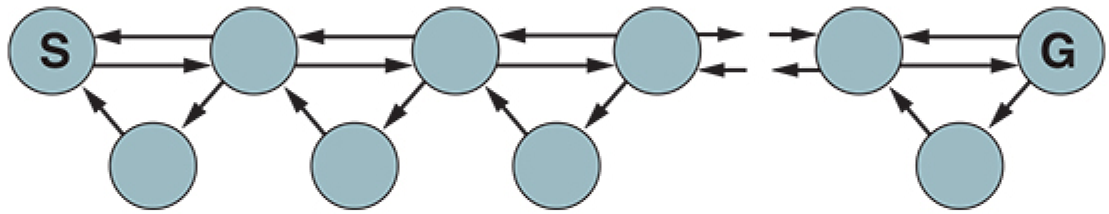


Figure 4.22 An environment in which a random walk will take exponentially many steps to find the goal.

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.

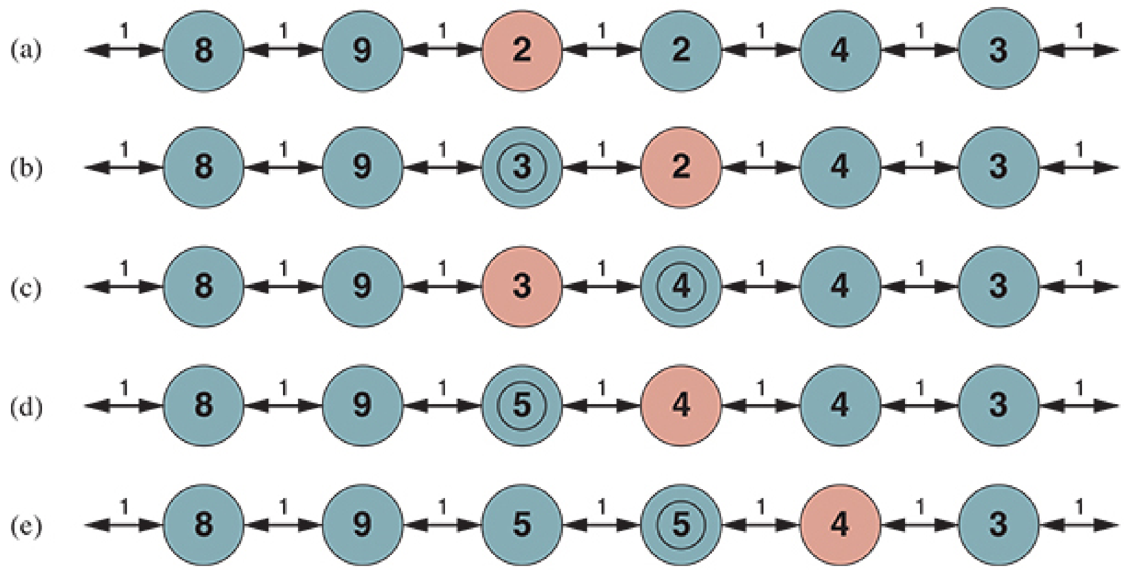


Figure 4.23 Five iterations of LRTA* on a one-dimensional state space. Each state is labeled with $H(s)$, the current cost estimate to reach a goal, and every link has an action cost of 1. The red state marks the location of the agent, and the updated cost estimates at each iteration have a double circle.

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.

---

function LRTA*-AGENT(*problem*, s', h) **returns** an action
    s, a, the previous state and action, initially null
    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** $H[s'] \leftarrow h(s')$
  if s is not null **then**
    $result[s, a] \leftarrow s'$
    $H[s] \leftarrow \min_{b \in \text{ACTIONS}(s)}$ LRTA*-COST(*problem*, $s, b, result[s, b], H$)
  $a \leftarrow \operatorname{argmin}_{b \in \text{ACTIONS}(s)}$ LRTA*-COST(*problem*, $s', b, result[s', b], H$)
  $s \leftarrow s'$
  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 23.

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 23, 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.

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).

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.

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 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 21) 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 Computation 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 16). 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 16.

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).

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 **realtime 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.

---

[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.

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

[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.

[4] A set of points S is convex if the line joining any two points in S is also contained in 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.

[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.

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

[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."

[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).

CHAPTER 5

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**.

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.

5.1 Defining Constraint Satisfaction Problems

A constraint satisfaction problem consists of three components, X, D, and C:

X is a set of variables, $(X_1,..., X_n\}$.

D is a set of domains, $\{D_1,..., D_n\}$, one for each variable.

C is a set of constraints that specify allowable combinations of values.

A domain, D_i, consists of a set of allowable values, $\{v_1,..., v_k\}$, 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 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 $\langle (X_1, X_2), X_1 > X_2 \rangle$.

CSPs deal with **assignments** of values to variables, $\{X_i = v_i, X_j = v_j,...\}$. 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.

5.1.1 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 5.1(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:

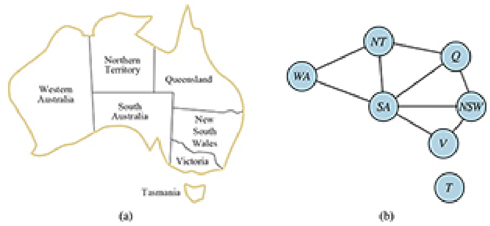


Figure 5.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

$$\chi = \{WA, NT, Q, NSW, V, SA, T\}.$$

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:

$$\begin{aligned} C = \{&SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V, \\ &WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}. \end{aligned}$$

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 5.1(b). The nodes of the graph correspond to variables of the problem, and an edge connects any two variables that participate in a constraint.

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.

5.1.2 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:

$$\begin{aligned}\chi = \{&Axle_F, Axle_B, Wheel_{RF}, Wheel_{LF}, Wheel_{RB}, Wheel_{LB}, Nuts_{RF},\\ &Nuts_{LF}, Nuts_{RB}, Nuts_{LB}, Cap_{RF}, Cap_{LF}, Cap_{RB}, Cap_{LB}, Inspect\}.\end{aligned}$$

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:

$$(Axle_F + 10 \leq Axle_B)\ \textbf{or}\ (Axle_B + 10 \leq Axle_F)\ .$$

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 jobshop scheduling problems like this with thousands of variables.

5.1.3 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.

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.

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.

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.)

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 5.1(b).

We can also define higher-order constraints. The ternary constraint *Between*(X, Y, Z), for example, can be defined as $\langle (X, Y, Z), X < Y < Z \textbf{ or } X > Y > Z \rangle$.

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 5.2.6), all variables in a row, column, or 3×3 box must satisfy an *Alldiff* constraint.

Another example is provided by **cryptarithmetic** puzzles (Figure 5.2(a)). Each letter in a cryptarithmetic puzzle represents a different digit. For the case in Figure 5.2(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:

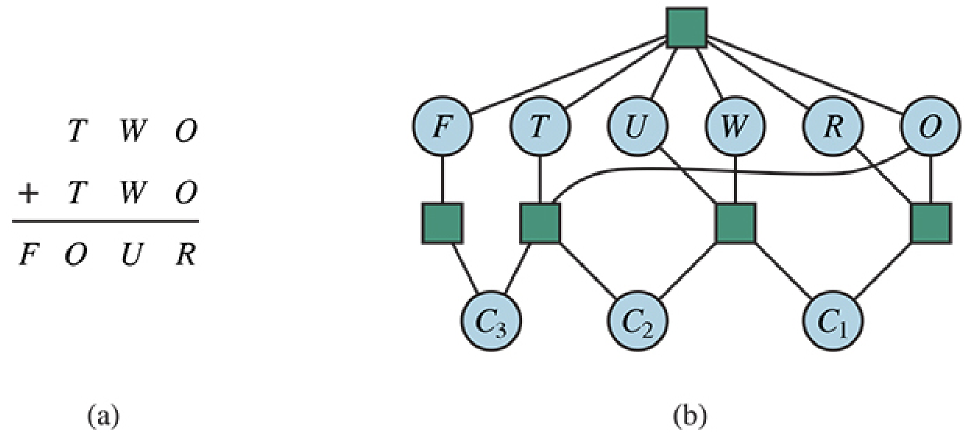


Figure 5.2 (a) A cryptarithmetic problem. Each letter stands for a distinct digit; the aim is to find a substitution of digits for letters such that the resulting sum is arithmetically correct, with the added restriction that no leading zeroes are allowed. (b) The constraint hypergraph for the cryptarithmetic 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.

$$
\begin{aligned}
&O + O = R + 10 \cdot C_1 \\
&C_1 + W + W = U + 10 \cdot C_2 \\
&C_2 + T + T = O + 10 \cdot C_3 \\
&C_3 = F\ ,
\end{aligned}
$$

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 5.2(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.

Alternatively, as Exercise 5.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 $\chi = \{X, Y, Z\}$, 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 $\chi = \{C_1, C_2\}$, 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 $R_1 = \{((1,2,3), (1,2)), ((2,3,5), (2,3))\}$.

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 5.2.5.

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 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.

5.2 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 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.

The key idea is **local consistency**. If we treat each variable as a node in a graph (see Figure 5.1(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.

5.2.1 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 5.1) 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.

5.2.2 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 5.3). 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 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).

```
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 5.3 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.

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.

5.2.3 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.

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.

5.2.4 *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.

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.

5.2.5 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 5.1. 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.

Another important higher-order constraint is the **resource constraint**, sometimes called the *Atmost* constraint. For example, in a scheduling problem, let $P_1,...,P_4$ 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.

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

$$D_1 = [0,165] \text{ and } D_2 = [0,385] \ .$$

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

$$D_1 = [35,165] \text{ and } D_2 = [255,385] \ .$$

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.

5.2.6 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 5.4). A row, column, or box is called a **unit**.

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):

$$\begin{aligned}
&Alldiff\ (A1, A2, A3, A4, A5, A6, A7, A8, A9)\\
&Alldiff\ (B1, B2, B3, B4, B5, B6, B7, B8, B9)\\
&\ldots\\
&Alldiff\ (A1, B1, C1, D1, E1, F1, G1, H1, I1)\\
&Alldiff\ (A2, B2, C2, D2, E2, F2, G2, H2, I2)\\
&\ldots\\
&Alldiff\ (A1, A2, A3, B1, B2, B3, C1, C2, C3)\\
&Alldiff\ (A4, A5, A6, B4, B5, B6, C4, C5, C6)\\
&\ldots
\end{aligned}$$

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 5.4(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 5.4(b).

	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 5.4 (a) A Sudoku puzzle and (b) its solution.

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 constraintsolving mechanisms can take over.

5.3 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 *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 5.5 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 5.6, where we have assigned variables in the order *WA, NT, Q*,....

```
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 5.5 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 generalpurpose heuristics discussed in Section 5.3.1. 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.

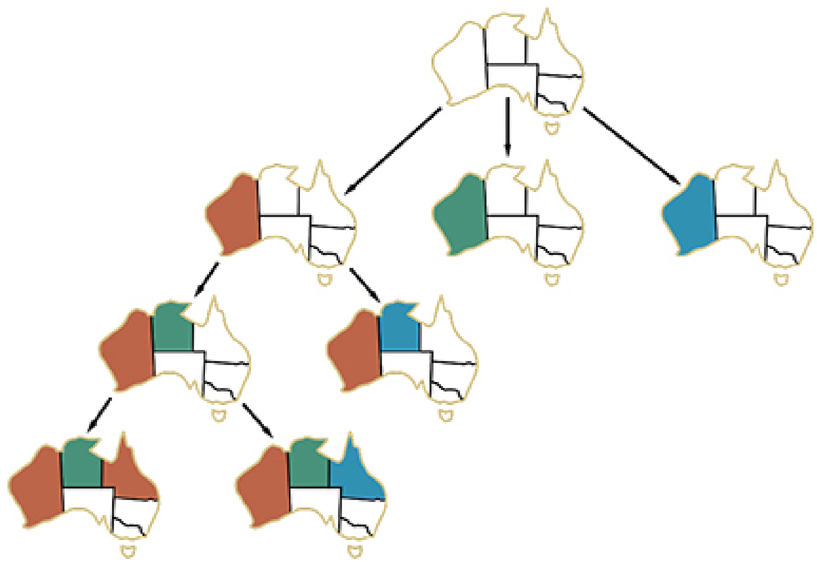


Figure 5.6 Part of the search tree for the map-coloring problem in Figure 5.1.

Notice that BACKTRACKING-SEARCH keeps only a single representation of a state (assignment) and alters that representation rather than creating new ones (see page 98).

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:

- (5.3.1) Which variable should be assigned next (SELECT-UNASSIGNED-VARIABLE), and in what order should its values be tried (ORDER-DOMAIN-VALUES)?
- (5.3.2) What inferences should be performed at each step in the search (INFERENCE)?
- (5.3.3) Can we BACKTRACK more than one step when appropriate?
- (5.3.4) Can we save and reuse partial results from the search?

5.3.1 Variable and value ordering

The backtracking algorithm contains the line

$$var \leftarrow \text{SELECT-UNASSIGNED-VARIABLE}(csp, assignment) \;.$$

The simplest strategy for SELECT-UNASSIGNED-VARIABLE is static ordering: choose the variables in order, $\{X_1,X_2,...\}$. The next simplest is to choose randomly. Neither strategy is optimal. For example, after the assignments for *WA* = *red* and *NT* = *green* in Figure 5.6, 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.

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

Once a variable has been selected, the algorithm must decide on the order in which to examine its values. The **least-constraining-value** heuristic is effective for this. It prefers the value that rules out the fewest choices for the neighboring variables in the constraint graph. For example, suppose that in Figure 5.1 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.

5.3.2 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.

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 5.7 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.

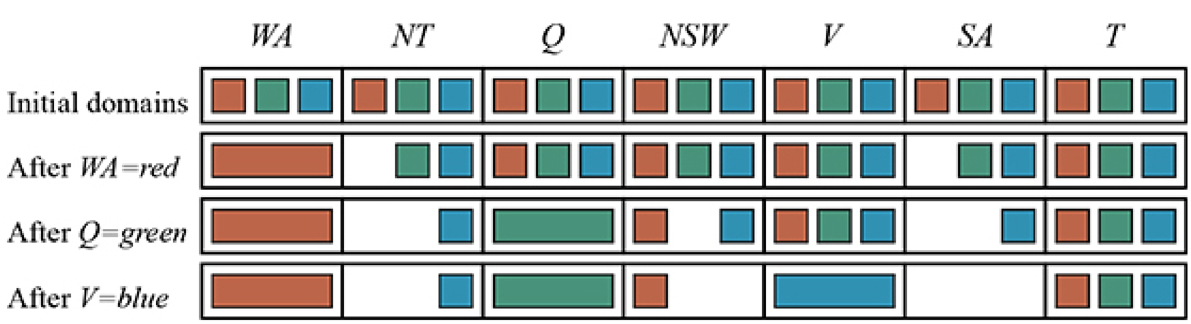


Figure 5.7 The progress of a map-coloring search with forward checking. *WA* = *red* is assigned first; then forward checking deletes *red* from the domains of the neighboring variables *NT* and *SA*. After *Q*

= *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.

For many problems the search will be more effective if we combine the MRV heuristic with forward checking. Consider Figure 5.7 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 5.7. 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.

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 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.

5.3.3 Intelligent backtracking: Looking backward

The BACKTRACKING-SEARCH algorithm in Figure 5.5 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 c**hronological backtracking** because the *most recent* decision point is revisited. in this subsection, we consider better possibilities.

Consider what happens when we apply simple backtracking in Figure 5.1 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.

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**.

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:

$$conf\ (X_i) \leftarrow\ conf\ (X_i) \cup conf\ (X_j) - \{X_i\}.$$

5.3.4 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.

For example, consider the state {*WA* = *red*, *NT* = *green*, *Q* = *blue*} in the bottom row of Figure 5.6. 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 5.6 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.

5.4 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 167. In Figure 5.8 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.

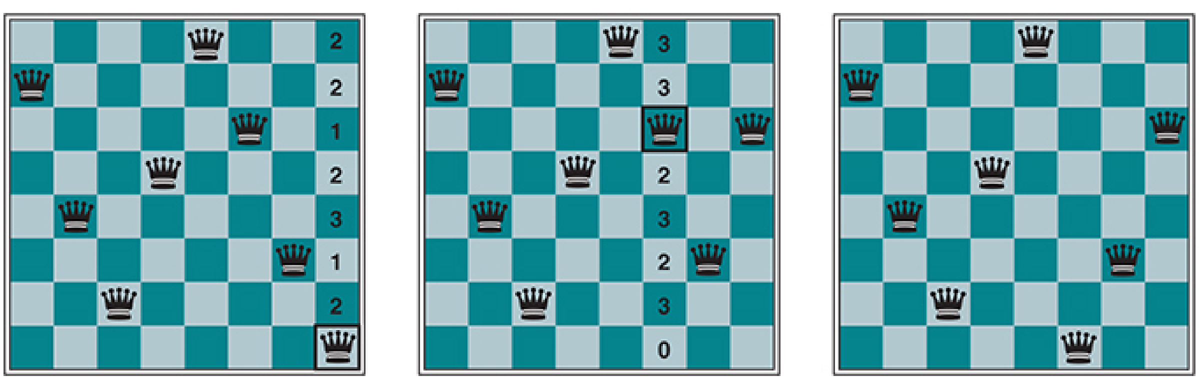


Figure 5.8 A two-step solution using min-conflicts for an 8-queens problem. At each stage, a queen is chosen for reassignment in its column. The number of conflicts (in this case, the number of attacking queens) is shown in each square. The

algorithm moves the queen to the min-conflicts square, breaking ties randomly.

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 5.9.[2]

```
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 5.9 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.

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 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.

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.

5.5 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 5.1(b), repeated as Figure 5.12(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.

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 $j > i$.

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 5.10(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 have to backtrack; we can move linearly through the variables. The complete algorithm is shown in Figure 5.11.

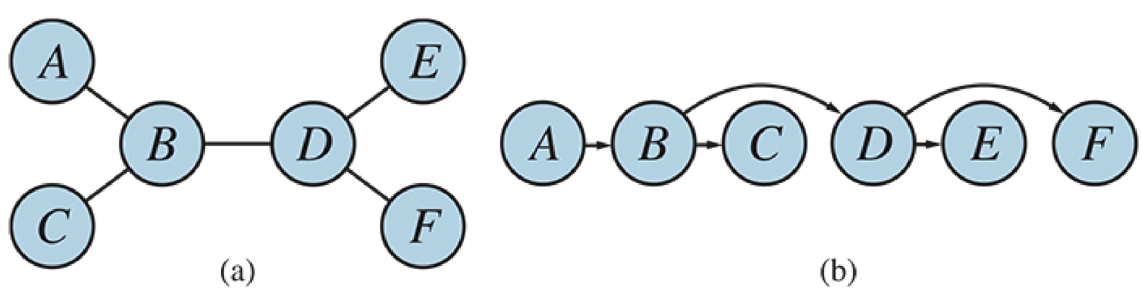


Figure 5.10 (a) The constraint graph of a tree-structured CSP. (b) A linear ordering of the variables consistent with the tree with *A* as the root. This is known as a **topological sort** of the variables.

```
function TREE-CSP-SOLVER(csp) returns a solution, or failure
   inputs: csp, a CSP with components X, D, C

   n ← number of variables in X
   assignment ← an empty assignment
   root ← any variable in X
   X ← 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 5.11 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.

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 5.5.1) or by collapsing nodes together (Section 5.5.2).

5.5.1 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 5.12(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*.

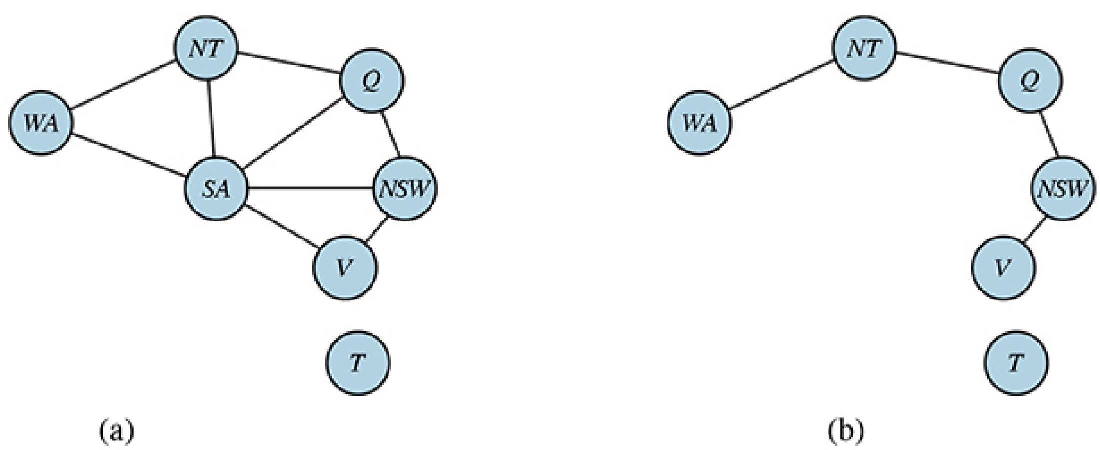


Figure 5.12 (a) The original constraint graph from Figure 5.1. (b) After the removal of *SA*, the constraint graph becomes a forest of two trees.

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 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*.

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.

5.5.2 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 5.13. A tree decomposition must satisfy these three requirements:

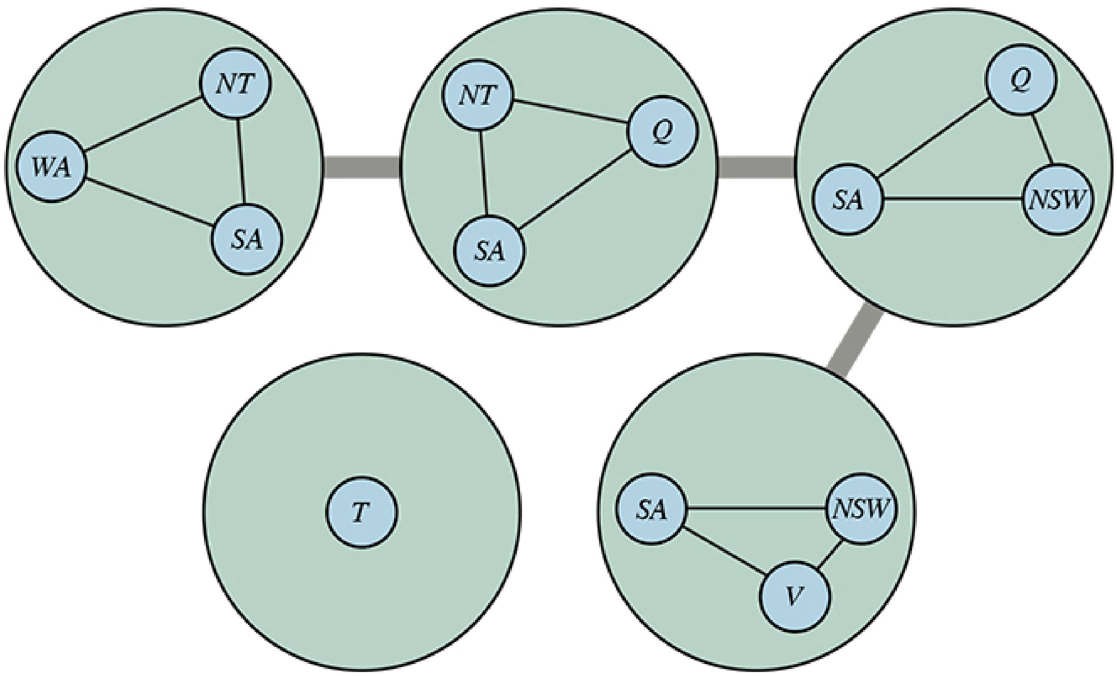


Figure 5.13 A tree decomposition of the constraint graph in Figure 5.12(a).

- 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.

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 5.13, 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 5.12 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 5.13 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 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.

5.5.3 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: $\{NT = blue, SA = green, WA = red\}$.

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

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 5.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 tradeoff 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).

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).

A very general form of intelligent backtracking was developed early on by Stallman and Sussman (1977). Their technique of **dependency-directed backtracking** combines back-jumping 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).

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 back-jumping; 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 jobshop 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 5.5.

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.*

---

[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.

[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.

[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.

CHAPTER 6

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

6.1 Game Theory

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

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

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

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

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

6.1.1 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."

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.

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.

Figure 6.1 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).

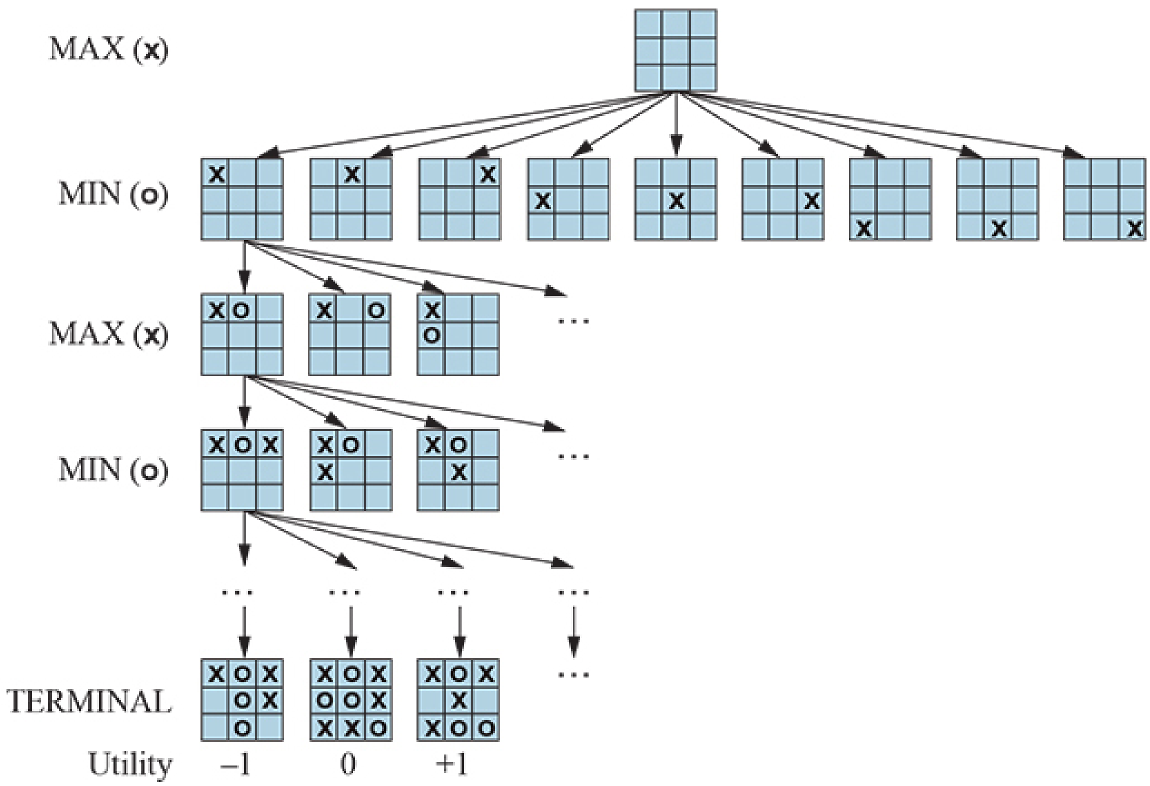


Figure 6.1 A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

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.

6.2 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 143) 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**.

Consider the trivial game in Figure 6.2. 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.

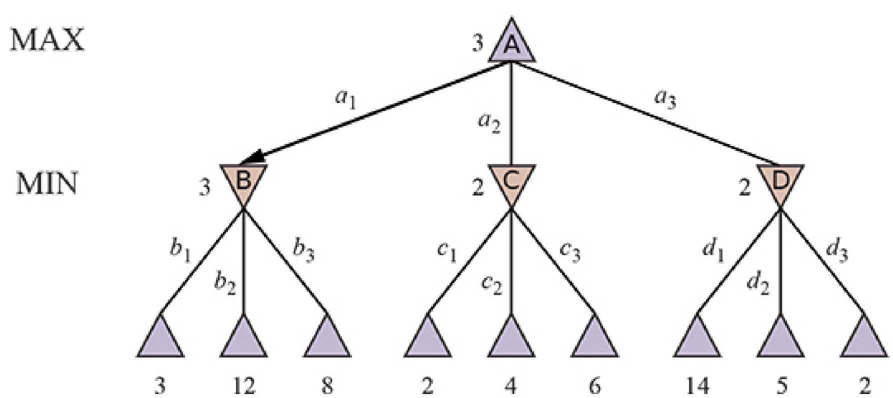


Figure 6.2 A two-ply game tree. The Δ nodes are "MAX nodes," in which it is MAX's turn to move, and the ∇ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest

minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

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 nonterminal state, MAX prefers to move to a state of maximum value when it is MAX's turn to move, and MIN prefers a state of minimum value (that is, minimum value for MAX and thus maximum value for MIN). So we have:

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s,\text{MAX}) & \text{if IS-TERMINAL}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if TO-MOVE}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if TO-MOVE}(s) = \text{MIN} \end{cases}$$

Let us apply these definitions to the game tree in Figure 6.2. 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.

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.

6.2.1 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 6.3 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 6.2, the algorithm 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.

---

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, *move* ← $-\infty$
  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, *move* ← $+\infty$
  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 6.3 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.

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 98). 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.

6.2.2 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 6.4. 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.

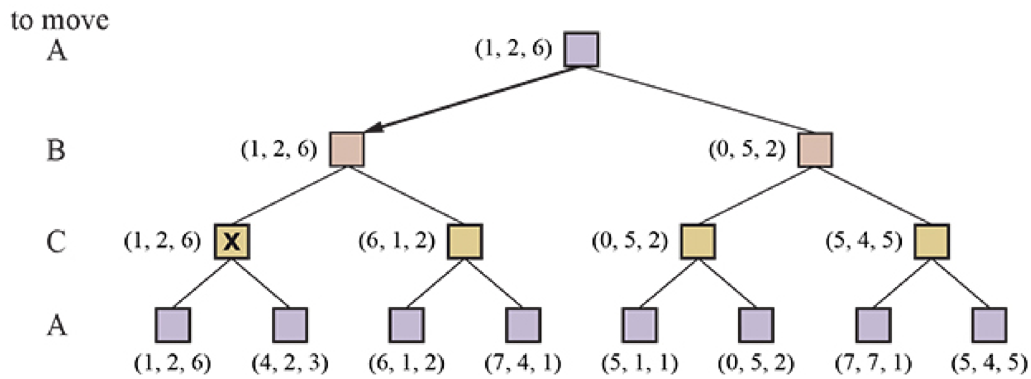


Figure 6.4 The first three ply of a game tree with three players (A, B, C). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

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.

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 17.2 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.

6.2.3 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 108) large parts of the tree that make no difference to the outcome. The particular technique we examine is called **alpha-beta pruning**.

Consider again the two-ply game tree from Figure 6.2. 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 6.5. 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 6.5 have values x and y. Then the value of the root node is given by

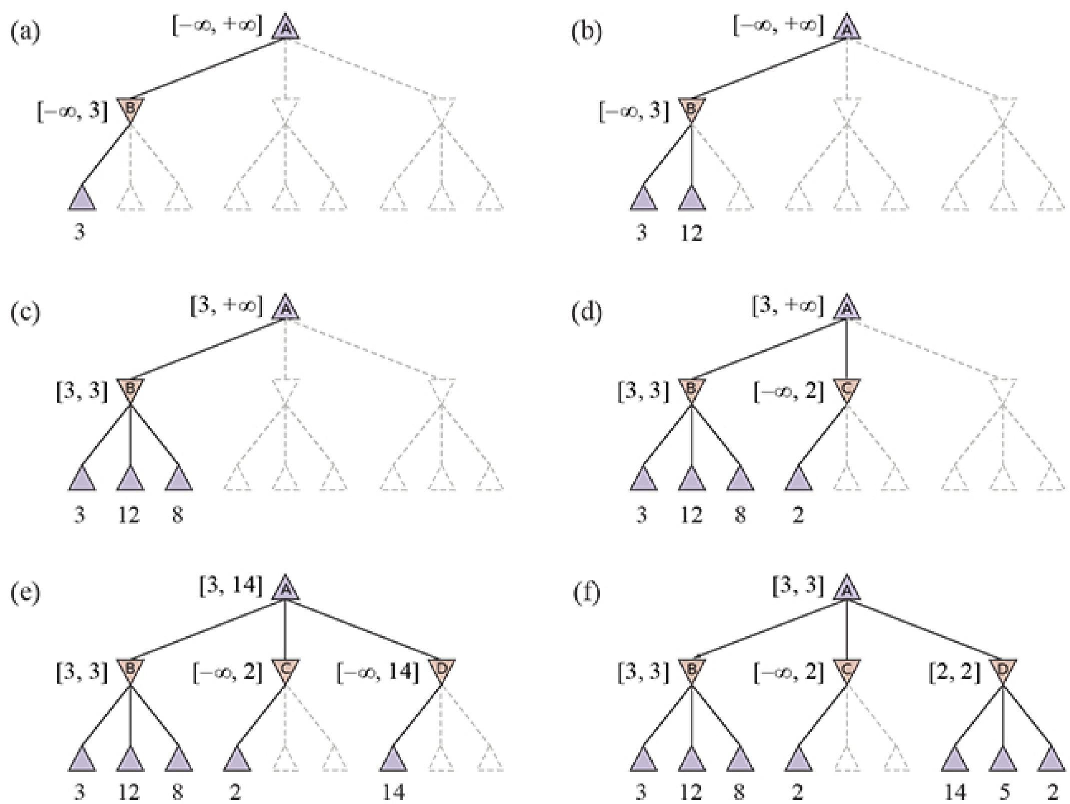


Figure 6.5 Stages in the calculation of the optimal decision for the game tree in Figure 6.2. At each point, we show the range of possible values for each node. (a) The first leaf below *B* has the value 3. Hence, *B*, which is a MIN node, has a value of *at most* 3. (b) The second leaf below *B* has 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.

$$\begin{aligned} \text{MINIMAX}(root) &= \max(\min(3, 12, 8), \min(2, x, y), \min(14, 5, 2)) \\ &= \max(3, \min(2, x, y), 2) \\ &= \max(3, z, 2) \qquad \text{where } z = \min(2, x, \mathrm{y}) \le 2 \\ &= 3. \end{aligned}$$

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 6.6), 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 6.6) or at any point higher up in the tree (e.g. m in Figure 6.6), 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.

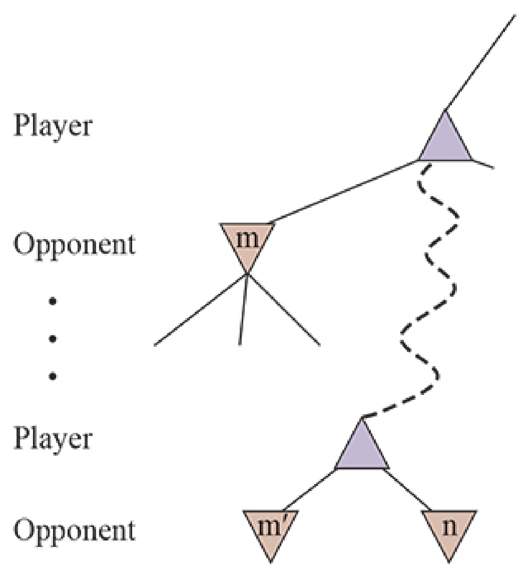


Figure 6.6 The general case for alpha–beta pruning. If m or m' is better than n for Player, we will never get to n in play.

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*, α, β) (see Figure 6.7) that describe bounds on the backed-up values that appear anywhere along the path:

```
function ALPHA-BETA-SEARCH(game, state) returns an action
   player ← game.TO-MOVE(state)
   value, move ← MAX-VALUE(game, state, −∞, +∞)
   return move

function MAX-VALUE(game, state, α, β) 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), α, β)
      if v2 > v then
         v, move ← v2, a
         α ← MAX(α, v)
      if v ≥ β then return v, move
   return v, move

function MIN-VALUE(game, state, α, β) 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), α, β)
      if v2 < v then
         v, move ← v2, a
         β ← MIN(β, v)
      if v ≤ α then return v, move
   return v, move
```

Figure 6.7 The alpha–beta search algorithm. Notice that these functions are the same as the MINIMAX-SEARCH functions in Figure 6.3, 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.

α = 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: α = "at least."

β = 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: β = "at most."

Alpha–beta search updates the values 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 6.7. Figure 6.5 traces the progress of the algorithm on a game tree.

6.2.4 Move ordering

The effectiveness of alpha–beta pruning is highly dependent on the order in which the states are examined. For example, in Figure 6.5(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 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 98). 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.

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.

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.

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 6.4), 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).

6.3 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*:

$$\text{H-MINIMAX}\,(s,d) = \begin{cases} \text{EVAL}\,(s,\text{MAX}) & \text{if IS-CUTOFF}\,(s,d) \\ \max_{a\in \text{Actions}(s)} \text{H-MINIMAX}\,(\text{RESULT}\,(s,a), d+1) & \text{if TO-MOVE}\,(s) = \text{MAX} \\ \min_{a\in \text{Actions}(s)} \text{H-MINIMAX}\,(\text{RESULT}\,(s,a), \text{d}+1) & \text{if TO-MOVE}\,(s) = \text{MIN}. \end{cases}$$

6.3.1 Evaluation functions

A heuristic evaluation function 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 EVAL(*s*, *p*) = UTILITY(*s*, *p*) and for nonterminal states, the evaluation must be somewhere between a loss and a win: UTILITY(*loss*, *p*) ≤ EVAL(*s*, *p*) ≤ 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).

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.

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 × +1)+(0.02 × 0)+ (0.16 × 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 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.

Mathematically, this kind of evaluation function is called a **weighted linear function** because it can be expressed as

$$\mathrm{EVAL}\,(s) = w_1\ f_1(s) + w_2\ f_2\ (s) + \ldots + w_n\ f_n\ (s) = \sum_{i=1}^{n} w_i\ f_i\ (s)\ ,$$

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 6.8(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').

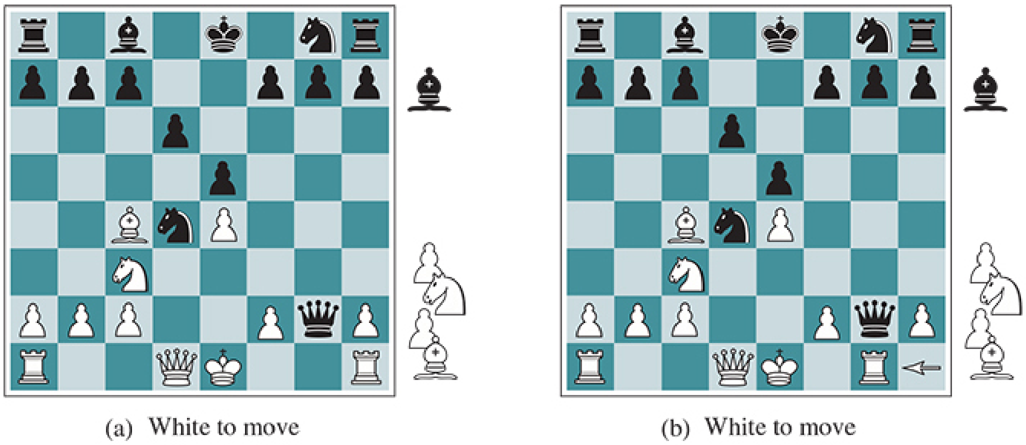


Figure 6.8 Two chess positions that differ only in the position of the rook at lower right. In (a), Black has an advantage of a knight and two pawns, which should be enough to win the game. In (b), White will capture the queen, giving it an advantage that should be strong enough to win.

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 23. 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.

6.3.2 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 6.7 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 6.8(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.

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.

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 6.9. 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.

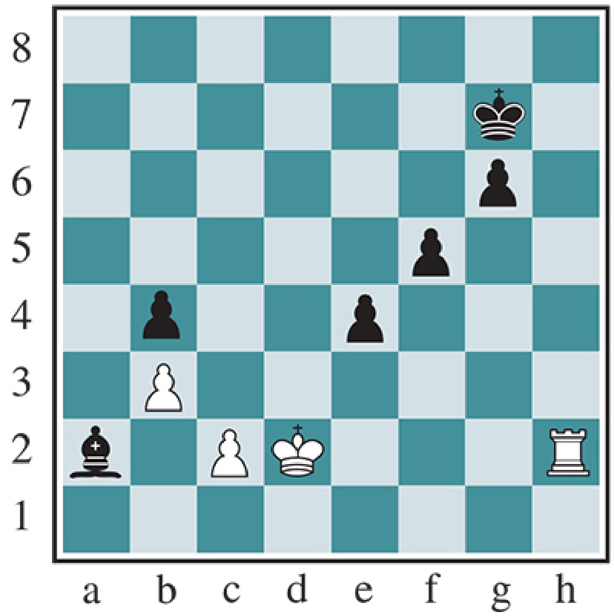


Figure 6.9 The horizon effect. With Black to move, the black bishop is surely doomed. But Black can forestall that event by checking the white king with its pawns, encouraging the king to capture the pawns. This pushes the inevitable loss of the bishop over the horizon, and thus the pawn sacrifices are seen by the search algorithm as good moves rather than bad ones.

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 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.

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.

6.3.3 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 133): 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.

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.

6.3.4 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 gameplaying 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.

6.4 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]

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, then 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.”

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.

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.

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 16.3 for more on the 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 6.10:

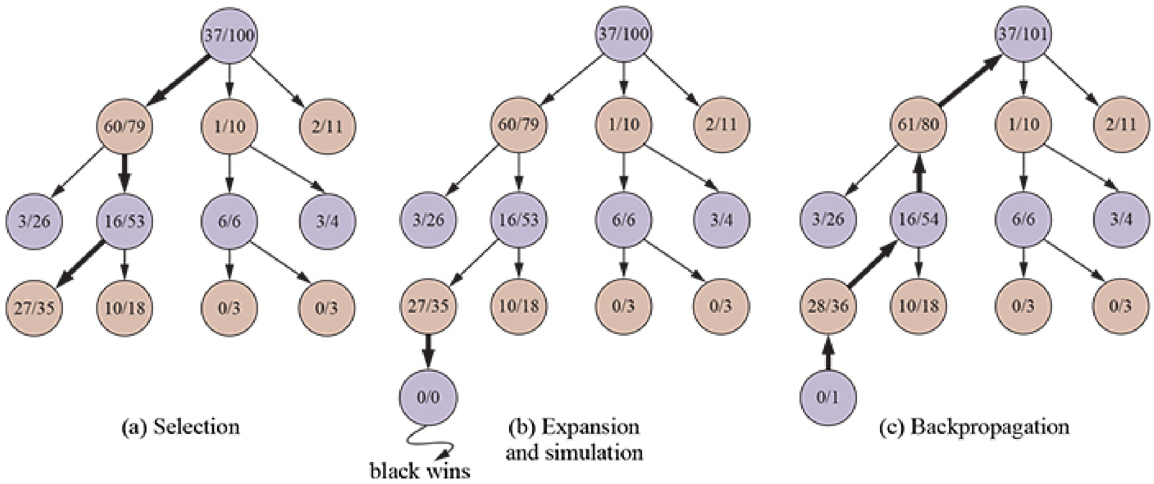


Figure 6.10 One iteration of the process of choosing a move with Monte Carlo tree search (MCTS) using the upper confidence bounds applied to trees (UCT) selection metric, shown after 100 iterations have already been done. In (a) we select moves, all the way down the tree, ending at the leaf node marked 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.

- **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 6.10(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 6.10(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 28/36 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.

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 16.3.3 for more details.) For a node n, the formula is:

$$UCBI = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(\text{PARENT}(n))}{N(n)}}$$

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 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 nonzero 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 6.10 has the highest UCB1 score, but with *C* = 1.5, it would be the 2/11 node.

Figure 6.11 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.

```
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 6.11 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.

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.

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 23.

6.5 Stochastic Games

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 6.12, Black 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).

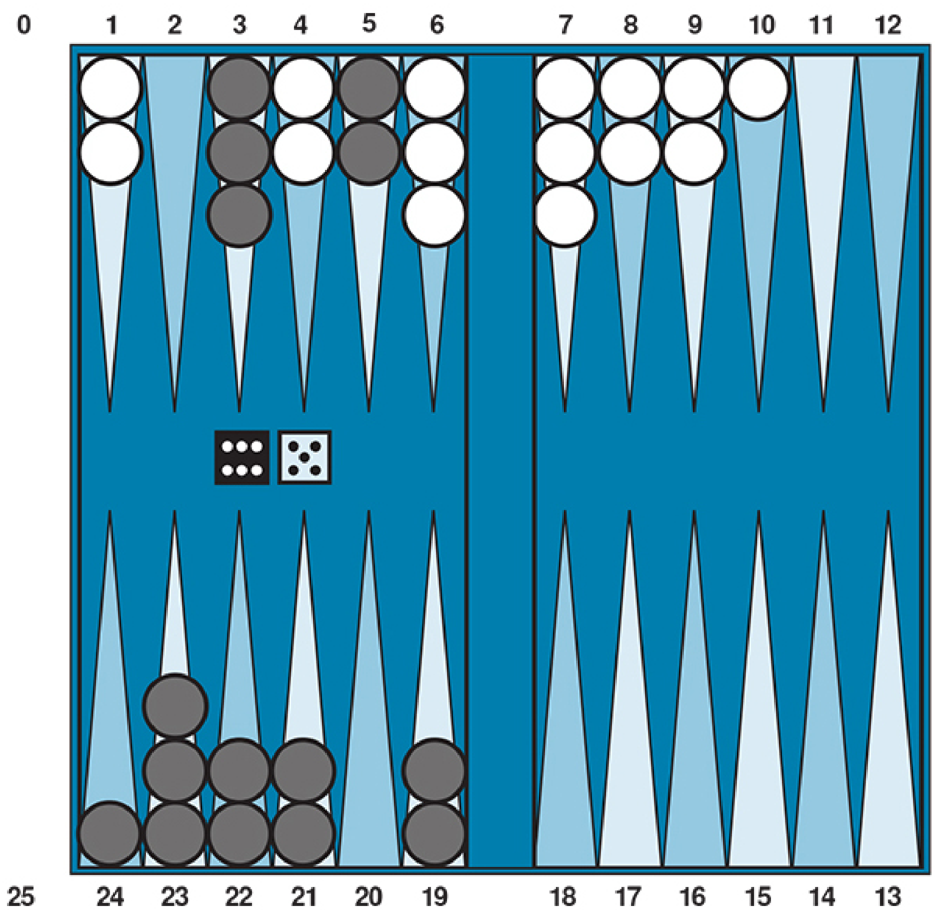


Figure 6.12 A typical backgammon position. The goal of the game is to move all one's pieces off the board. Black moves clockwise toward 25, and White moves counterclockwise toward 0. A piece can move to any position unless multiple opponent pieces are there; if there is one opponent, it is captured and 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 6.13. 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 $P(1–1) = 1/36$. The other 15 distinct rolls each have a 1/18 probability.

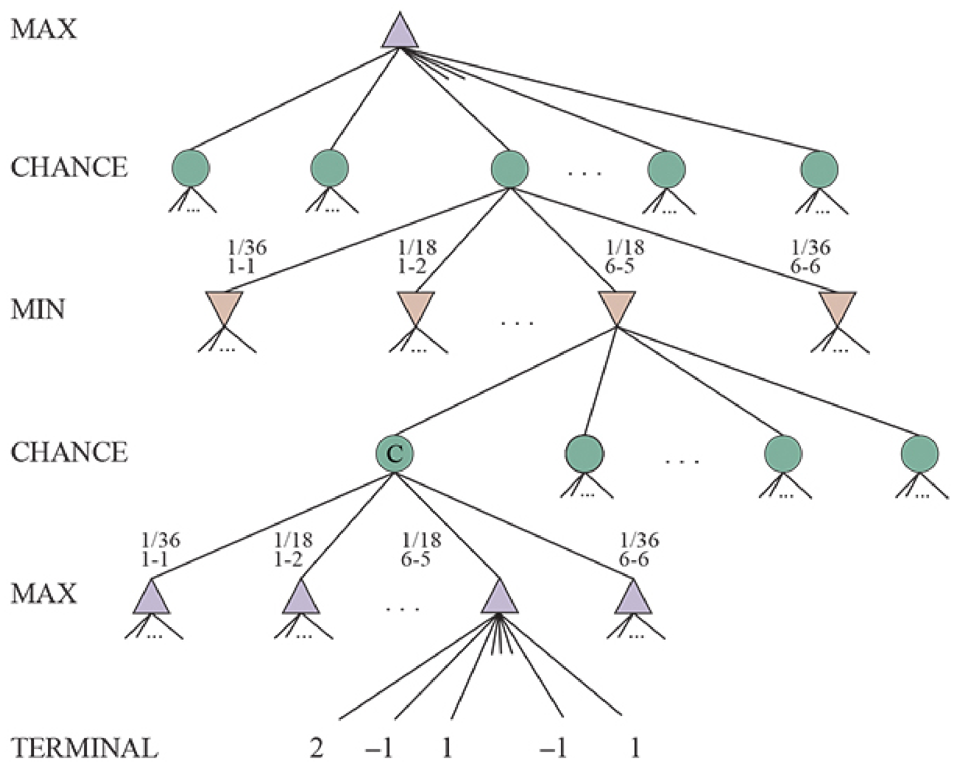


Figure 6.13 Schematic game tree for a backgammon position.

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.

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 compute the expected value, which is the sum of the value over all outcomes, weighted by the probability of each chance action:

$$
\begin{aligned}
&\text{EXPECTIMINIMAX}(s) = \\
&\quad \begin{cases}
\text{UTILITY}(s, \text{MAX}) & \text{if IS-TERMINAL}(s) \\
\max_a \text{EXPECTIMINIMAX}(\text{RESULT}(s, a)) & \text{if TO-MOVE}(s) = \text{MAX} \\
\min_a \text{EXPECTIMINIMAX}(\text{RESULT}(s, a)) & \text{if TO-MOVE}(s) = \text{MIN} \\
\sum_r P(r)\, \text{EXPECTIMINIMAX}(\text{RESULT}(s, r)) & \text{if TO-MOVE}(s) = \text{CHANCE}
\end{cases}
\end{aligned}
$$

where r represents a possible dice roll (or other chance event) and RESULT(s, r) is the same state as s, with the additional fact that the result of the dice roll is r.

6.5.1 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 6.14 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.

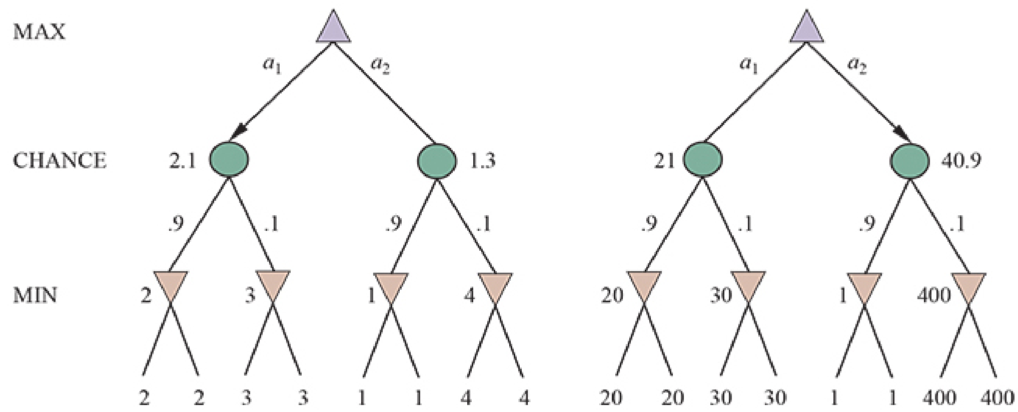


Figure 6.14 An order-preserving transformation on leaf values changes the best move.

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 and general property of situations in which uncertainty is involved, and we discuss it further in Chapter 15.

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 6.13 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.

6.6 Partially Observable Games

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

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

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

6.6.1 Kriegspiel: Partially observable chess

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

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

Kriegspiel may seem terrifyingly impossible, but humans manage it quite well and computer programs are beginning to catch up. It helps to recall the notion of a **belief state** as defined in Section 4.4 and illustrated in Figure 4.14—the set of all *logically possible* board states given the complete history of percepts to date. Initially, White’s belief state is a singleton because Black’s pieces haven’t moved yet. After White makes a move and Black responds, White’s belief state contains 20 positions, because Black has 20 replies to any 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 150. 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.

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 6.15 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.

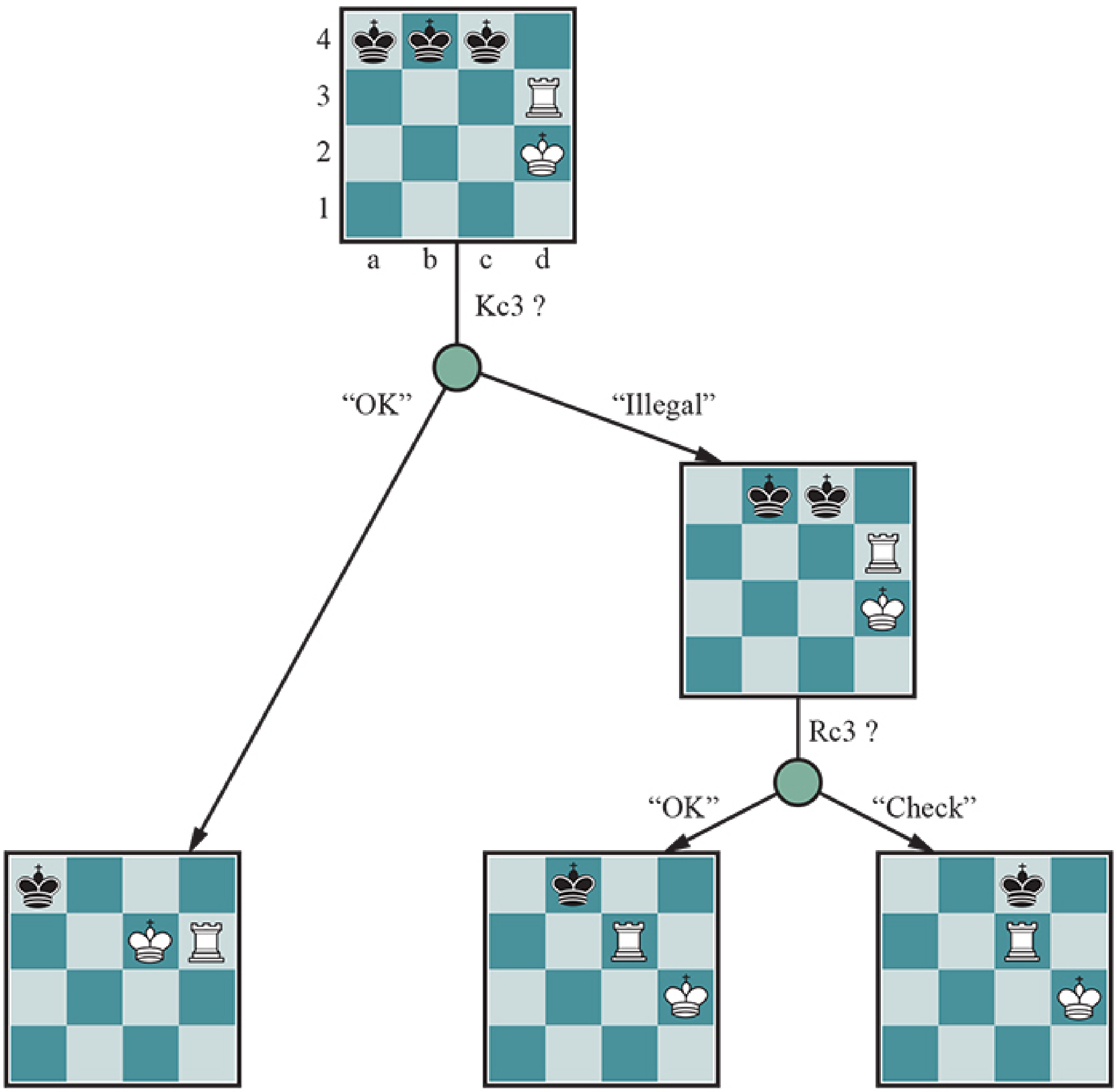


Figure 6.15 Part of a guaranteed checkmate in the KRK endgame, shown on a reduced board. In the initial belief state, Black's king is in one of three possible locations. By a combination of probing moves, the strategy narrows this down to one. Completion of the checkmate is left as an exercise.

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.

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.

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 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 16. 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 17.2.

6.6.2 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 6.15); 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 16, we show how to construct algorithms that do all these things by virtue of solving the true partially observable decision problem, resulting in an optimal equilibrium strategy (see Section 17.2).

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 nolimit 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.

6.7 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 6.16 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.

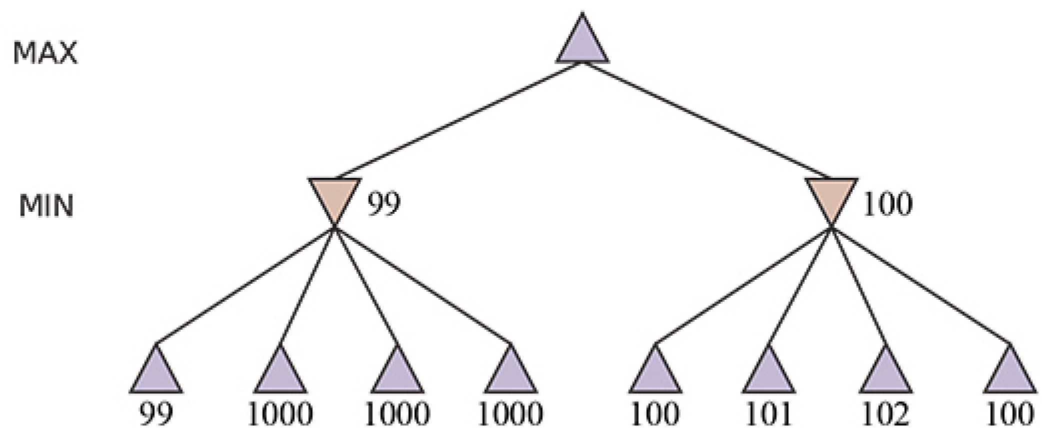


Figure 6.16 A two-ply game tree for which heuristic minimax may make an error.

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 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 specialpurpose 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 15 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.

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 23). 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 17.2) 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 17. 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.

---

[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.

[3] “Monte Carlo” algorithms are randomized algorithms named after the Casino de Monte-Carlo in Monaco.

[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 suggestmethods for compactly representing very large belief states.

[5] Bluffing—betting as if one’s hand is good, even when it’s not—is a core part of poker strategy.

[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 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.

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 5 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 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.**

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.

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.**

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.

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**, 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 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 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

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:

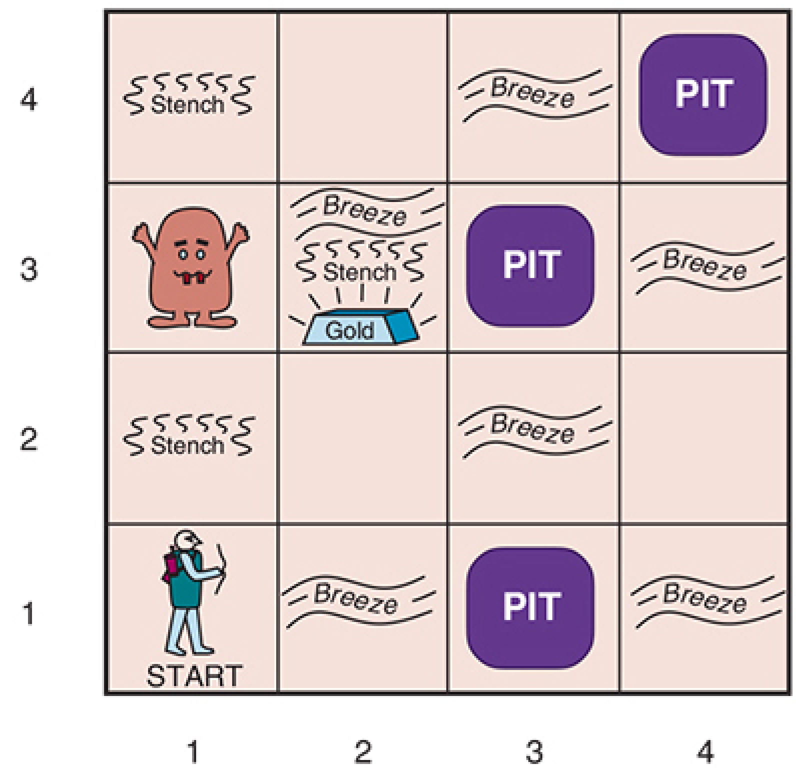


Figure 7.2 A typical wumpus world. The agent is in the bottom left comer, facing east (rightward).

- **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.
- **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 any-where 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*].

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.

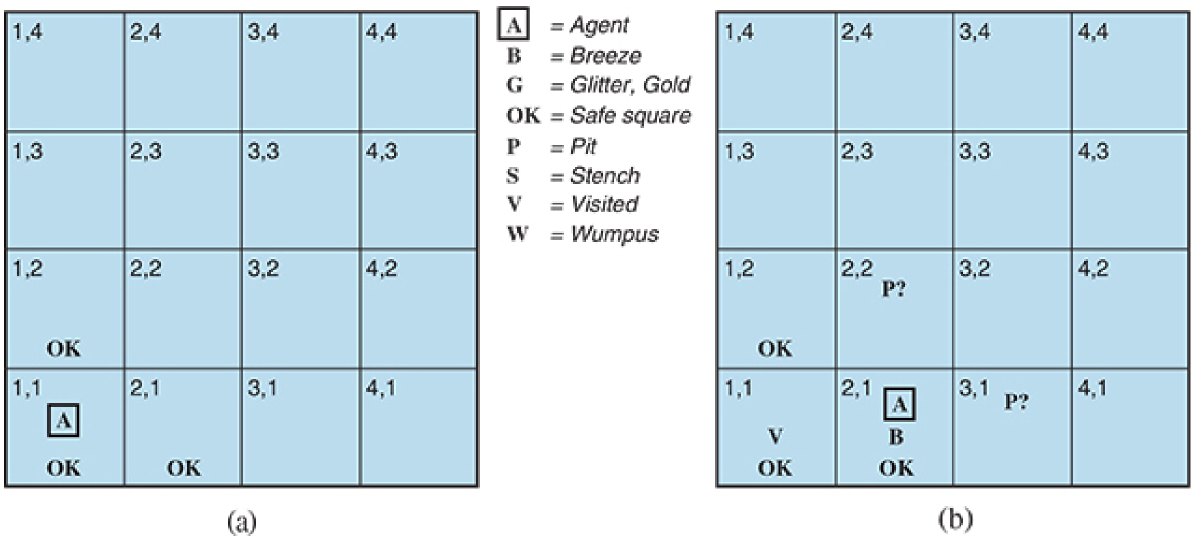


Figure 7.3 The first step taken by the agent in the wumpus world. (a) The initial situation, after percept [*None*, *None*, *None*, *None*, *None*]. (b) After moving to [2,1] and perceiving [*None*, *Breeze*, *None*, *None*, *None*].

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.

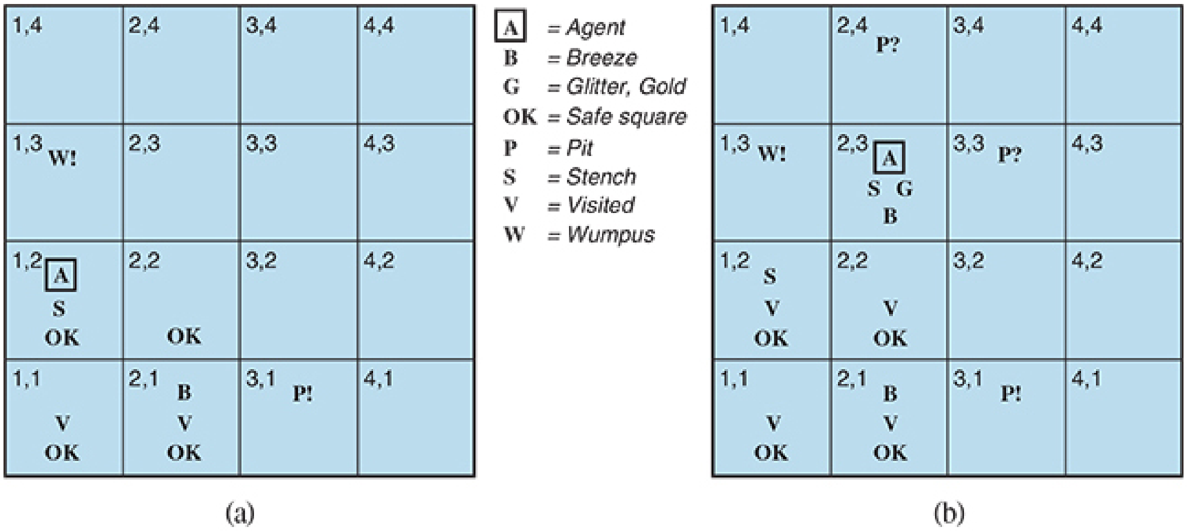


Figure 7.4 Two later stages in the progress of the agent. (a) After moving to [1,1] and then [1,2], and perceiving [*Stench*, *None*, *None*, *None*, *None*]. (b) After moving to [2,2] and then [2,3], and perceiving [*Stench*, *Breeze*, *Glitter*, *None*, *None*].

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.

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.

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]

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.

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 ⊆ here: if α ⊨ β, then α is α *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 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]

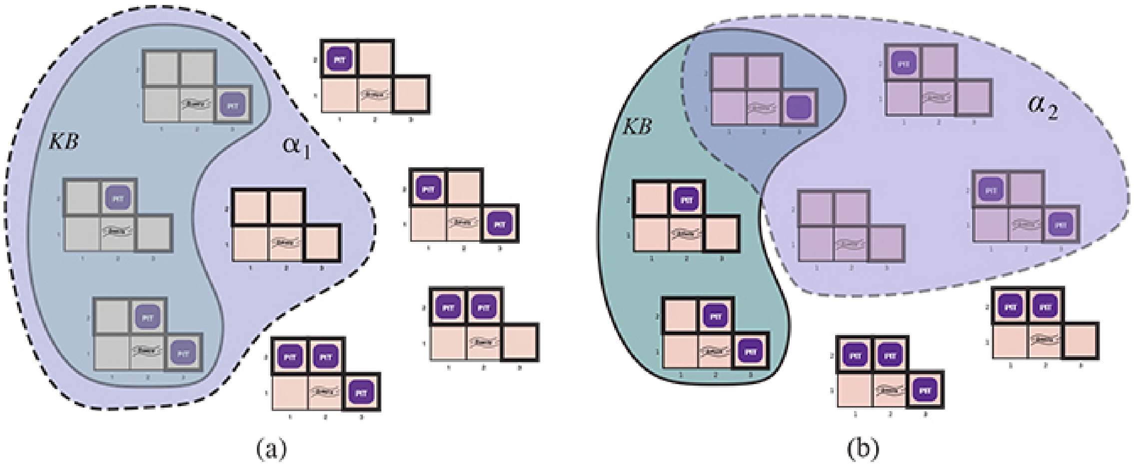


Figure 7.5 Possible models for the presence of pits in squares [1,2], [2,2], and [3,1]. The KB corresponding to the observations of nothing in [1,1] and a breeze in [2,1] is shown by the solid line. (a) Dotted line shows models of α_1 (no pit in [1,2]). (b) Dotted line shows models of α_2 (no pit in [2,2]).

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:

$$\alpha_1 = \text{"There is no pit in } [1,2]\text{."} \qquad \alpha_2 = \text{"There is no pit in } [2,2]\text{."}$$

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:

$$\text{in every model in which } KB \text{ is true, } \alpha_1 \text{ is also true.}$$

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

$$\text{in some models in which } KB \text{ is true, } \alpha_2 \text{ is false.}$$

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 $M(KB) \subseteq M(\alpha)$.

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.*"

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.

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.

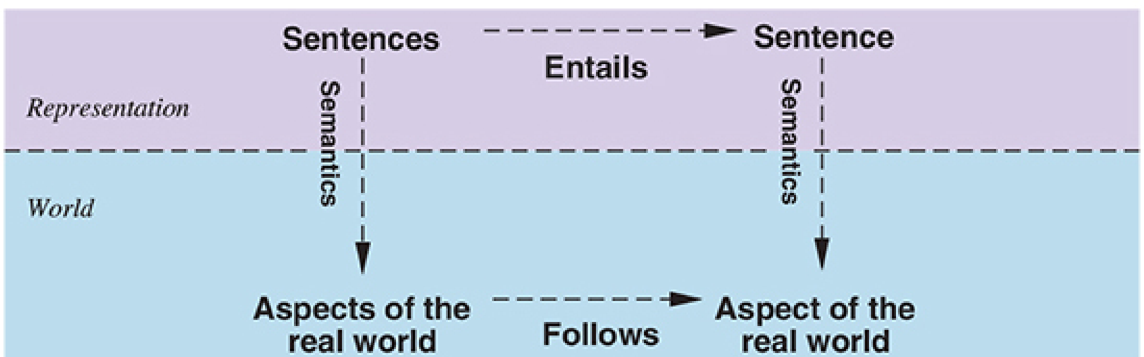


Figure 7.6 Sentences are physical configurations of the agent, and reasoning is a process of constructing new physical configurations from old ones. Logical reasoning should ensure that the new configurations represent aspects of the world that actually follow from the aspects that the old configurations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

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 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 28.) 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

We now present **propositiona 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

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** are constructed from simpler sentences, using parentheses and operators called **logical connectives.** There are five connectives in common use:

- $\neg$ (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**).
- $\wedge$ (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.")

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

Figure 7.7 gives a formal grammar of propositional logic. (BNF notation is explained on page 1081.) 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 $\neg A \wedge$ B the $\neg$ binds most tightly, giving us the equivalent of $(\neg A) \wedge B$ rather than $\neg(A \wedge B)$. (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.

$$
\begin{array}{rcl}
\textit{Sentence} & \rightarrow & \textit{AtomicSentence} \mid \textit{ComplexSentence} \\
\textit{AtomicSentence} & \rightarrow & \textit{True} \mid \textit{False} \mid P \mid Q \mid R \mid \ldots \\
\textit{ComplexSentence} & \rightarrow & (\ \textit{Sentence}\ ) \\
& \mid & \neg\, \textit{Sentence} \\
& \mid & \textit{Sentence} \land \textit{Sentence} \\
& \mid & \textit{Sentence} \lor \textit{Sentence} \\
& \mid & \textit{Sentence} \ \Rightarrow\ \textit{Sentence} \\
& \mid & \textit{Sentence} \ \Leftrightarrow\ \textit{Sentence} \\
\\
\text{OPERATOR PRECEDENCE} & : & \neg, \land, \lor, \Rightarrow, \Leftrightarrow
\end{array}
$$

Figure 7.7 A BNF (Backus–Naur Form) grammar of sentences in propositional logic, along with operator precedences, from highest to lowest.

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

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

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.

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.

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.

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.*

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 ⇒ 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 written using ⇔. 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,

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

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]:

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

- 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:

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

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

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

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 176, TT-ENTAILS? performs a recursive 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.

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	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*	*true*
false	*true*	*false*	*false*	*false*	*true*	*false*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*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 propositional logic; it takes two arguments and returns *true* or *false*.

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.

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 $P \wedge Q$ and $Q \wedge P$ 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:

$$
\begin{aligned}
(\alpha \wedge \beta) &\equiv (\beta \wedge \alpha) \quad \text{commutativity of } \wedge \\
(\alpha \vee \beta) &\equiv (\beta \vee \alpha) \quad \text{commutativity of } \vee \\
((\alpha \wedge \beta) \wedge \gamma) &\equiv (\alpha \wedge (\beta \wedge \gamma)) \quad \text{associativity of } \wedge \\
((\alpha \vee \beta) \vee \gamma) &\equiv (\alpha \vee (\beta \vee \gamma)) \quad \text{associativity of } \vee \\
\neg(\neg \alpha) &\equiv \alpha \quad \text{double-negation elimination} \\
(\alpha \Rightarrow \beta) &\equiv (\neg \beta \Rightarrow \neg \alpha) \quad \text{contraposition} \\
(\alpha \Rightarrow \beta) &\equiv (\neg \alpha \vee \beta) \quad \text{implication elimination} \\
(\alpha \Leftrightarrow \beta) &\equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)) \quad \text{biconditional elimination} \\
\neg(\alpha \wedge \beta) &\equiv (\neg \alpha \vee \neg \beta) \quad \text{De Morgan} \\
\neg(\alpha \vee \beta) &\equiv (\neg \alpha \wedge \neg \beta) \quad \text{De Morgan} \\
(\alpha \wedge (\beta \vee \gamma)) &\equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) \quad \text{distributivity of } \wedge \text{ over } \vee \\
(\alpha \vee (\beta \wedge \gamma)) &\equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma)) \quad \text{distributivity of } \vee \text{ over } \wedge
\end{aligned}
$$

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

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

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

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

(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.

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, $(R_1 \wedge R_2 \wedge R_3 \wedge R_4 \wedge R_5)$, is satisfiable because there are three models in which it is true, as shown in Figure 7.9. 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 in propositional logic—the **SAT** problem—was the first problem proved to be NP-complete. Many problems in computer science are really satisfiability problems. For example, all the constraint satisfaction problems in Chapter 5 ask whether the constraints are satisfiable by some assignment.

Validity and satisfiability are of course connected: α is valid iff $\neg\alpha$ is unsatisfiable; contrapositively, α is satisfiable iff $\neg\alpha$ is not valid. We also have the following useful result:

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

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

7.5.1 Inference and proofs

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

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

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

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

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

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

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

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

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

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

1. Apply biconditional elimination to R_2 to obtain

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

2. Apply And-Elimination to R_6 to obtain

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

3. Logical equivalence for contrapositives gives

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

4. Apply Modus Ponens with R_8 and the percept R_4 (i.e., $\neg B_{1,1}$), to obtain

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

5. Apply De Morgan's rule, giving the conclusion

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

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,

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

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

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 99) 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:

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

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):

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

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]:

$$R_{15} \;:\; P_{1,1} \;\vee P_{2,2} \;\vee P_{3,1} \,.$$

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**

$$R_{16} \;:\; P_{1,1} \;\vee P_{3,1} \,.$$

In English; if there's a pit in one of [1,1], [2,2], and [3,1] and it's notin [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

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

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

$$\frac{\ell_1 \;\vee \ldots \vee \ell_k, \quad m}{\ell_1 \;\vee \ldots \vee \ell_{i-1} \;\vee \ell_{i+1} \;\vee \ldots \vee \ell_k}$$

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**.

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

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

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

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

You can resolve only one pair of complementary literals at a time. For example, we can resolve P and $\neg P$ to deduce

$$\frac{P \vee \neg Q \vee R, \quad \neg P \vee Q}{\neg Q \vee Q \vee R}.$$

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.

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 $m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \ldots \vee m_n$ must be true, because $m_1 \vee \cdots \vee m_n$ is given. If ℓ_i is false, then $\ell_i \vee \cdots \vee \ell_{i+1} \vee \cdots \vee \ell_k$ must be true because $m_1 \vee \cdots \vee m_n$ 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 a* $\models$ *β.* 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.*

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:

$$
\begin{array}{rcl}
\mathit{CNFSentence} & \rightarrow & \mathit{Clause}_1 \land \cdots \land \mathit{Clause}_n \\
\mathit{Clause} & \rightarrow & \mathit{Literal}_1 \lor \cdots \lor \mathit{Literal}_m \\
\mathit{Fact} & \rightarrow & \mathit{Symbol} \\
\mathit{Literal} & \rightarrow & \mathit{Symbol} \mid \neg\mathit{Symbol} \\
\mathit{Symbol} & \rightarrow & P \mid Q \mid R \mid \ldots \\
\mathit{HornClauseForm} & \rightarrow & \mathit{DefiniteClauseForm} \mid \mathit{GoalClauseForm} \\
\mathit{DefiniteClauseForm} & \rightarrow & \mathit{Fact} \mid (\mathit{Symbol}_1 \land \cdots \land \mathit{Symbol}_l) \Rightarrow \mathit{Symbol} \\
\mathit{GoalClauseForm} & \rightarrow & (\mathit{Symbol}_1 \land \cdots \land \mathit{Symbol}_l) \Rightarrow \mathit{False}
\end{array}
$$

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:

   $$\neg(\neg\alpha) \equiv \alpha \quad \text{(double-negation elimination)}$$
   $$\neg(\alpha \land \beta) \equiv (\neg\alpha \lor \neg\beta) \quad \text{(De Morgan)}$$
   $$\neg(\alpha \lor \beta) \equiv (\neg\alpha \land \neg\beta) \quad \text{(De Morgan)}$$

   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.

   $$(\neg B_{1,1} \lor P_{1,2} \lor P_{2,1}) \land (\neg P_{1,2} \lor \neg B_{1,1}) \land (\neg P_{2,1} \lor B_{1,1}).$$

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 241. 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:

---

function PL-RESOLUTION(KB, α) **returns** *true* or *false*
  inputs: KB, the knowledge base, a sentence in propositional logic
    α, 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.

- 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.

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.

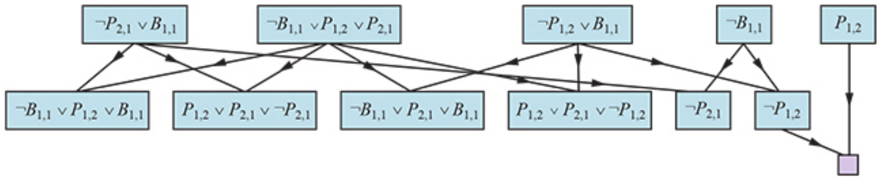


Figure 7.14 Partial application of PL-RESOLUTION to a simple inference in the wumpus world to prove the query $\neg P_{1,2}$. Each of the leftmost four clauses in the top row is paired with each of the other three, and the resolution rule is applied to yield the clauses on the bottom row. We see that 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.

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.

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

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

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 ($false \vee false \vee \ldots\ false \vee P_i$) or like ($false \vee false \vee \ldots\ false \vee \neg P_i$). 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.

Slightly more general is the **Horn clause**, which is a disjunction of literals of which *at 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.

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 $L_{1,1}$.
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.
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.

```
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 ← a table, where count[c] is initially the number of symbols in clause c's premise
  inferred ← a table, where inferred[s] is initially false for all symbols
  queue ← a queue of symbols, initially symbols known to be true in KB

  while queue is not empty do
     p ← POP(queue)
     if p = q then return true
     if inferred[p] = false then
        inferred[p] ← 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 *queue* 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 $Q \Rightarrow P$.

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.

$P \Rightarrow Q$

$L \wedge M \Rightarrow P$

$B \wedge L \Rightarrow M$

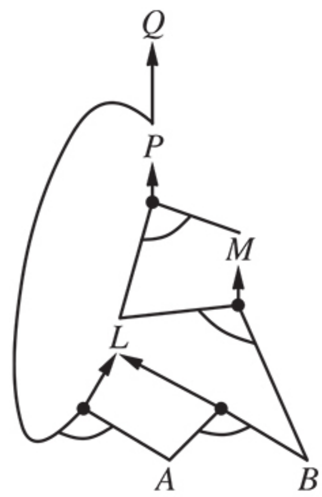
$A \wedge P \Rightarrow L$

$A \wedge B \Rightarrow L$

A

B

(a)



(b)

Figure 7.16 (a) A set of Horn clauses. (b) The corresponding AND–OR graph.

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.*

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.

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 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.

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 241 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 5.3 and the local search algorithms of Section 5.4. 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

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 $(A \lor B) \land (A \lor C)$ 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 heuristic:* A **pure symbol** is a symbol that always appears with the same "sign" in all clauses. For example, in the three clauses $(A \lor \neg B)$, $(\neg B \lor \neg C)$, and $(C \lor A)$, 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 $(\neg B \lor \neg C)$ 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 $(\neg B \lor \neg C)$ simplifies to

¬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.)

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 5.3.1 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 177) suggests choosing the variable that appears most frequently over all remaining clauses.
3. **Intelligent backtracking** (as seen in Section 5.3.3 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 131 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 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 laborious, 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 129) and SIMULATED-ANNEALING (page 133). 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 182). 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.

```
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.

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* = ∞ 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 5 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 5, we saw some surprising discoveries about certain kinds of problems. For example, the *n*-queens problem—thought to be quite tricky for backtracking search algorithms—turned out to be trivially easy for local search methods, such as min-conflicts. This is because solutions are very densely distributed in the space of assignments, and any initial assignment is guaranteed to have a solution nearby. Thus, *n*-queens is easy because it is **underconstrained**.

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:

$$(\neg D \lor \neg B \lor C) \land (B \lor \neg A \lor \neg C) \land (\neg C \lor \neg B \lor E)$$
$$\land (E \lor \neg D \lor B) \land (B \lor E \lor \neg C).$$

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.

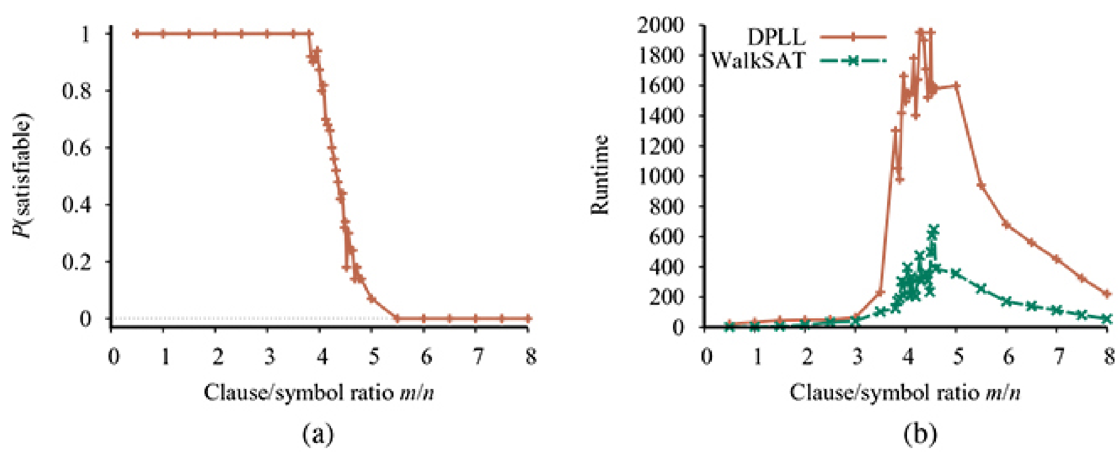


Figure 7.19 (a) Graph showing the probability that a random 3-CNF sentence with $n = 50$ symbols is satisfiable, as a function of the clause/symbol ratio m/n. (b) Graph of the median run time (measured in number of iterations) for both DPLL and WALKSAT on random 3-CNF sentences. The most difficult problems have a clause/symbol ratio of about 4.3.

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 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 wumpus 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:

$$B_{1,1} \Leftrightarrow (P_{1,2} \lor P_{2,1})$$
$$S_{1,1} \Leftrightarrow (W_{1.2} \lor W_{2,1})$$
$$\ldots$$

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:

$$W_{1,1} \lor W_{1,2} \lor \ldots \lor W_{4,3} \lor W_{4,4}.$$

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:

$$\neg W_{1,1} \lor \neg W_{1,2}$$
$$\neg W_{1,1} \lor \neg W_{1,3}$$
$$\ldots$$
$$\neg W_{4,3} \lor \neg W_{4,4}.$$

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 76. Symbols associated with permanent aspects of the world do not need a time superscript and are sometimes called **atemporal variables**.

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

$$L^t_{x,y} \Rightarrow (Breeze^t \Leftrightarrow B_{x,y})$$
$$L^t_{x,y} \Rightarrow (Stench^t \Leftrightarrow S_{x,y}).$$

Now, of course, we need axioms that allow the agent to keep track of fluents such as $L^{\mathrm{t}}_{\mathrm{x,y}}$. 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.

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.

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]:

$$L^0_{1,1} \wedge FacingEast^0 \wedge Forward^0 \Rightarrow (L^1_{2,1} \wedge \neg L^1_{1,1}). \tag{7.1}$$

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, ASK(*KB*, $L^1_{2,1}$) = *true*. So far, so good. Unfortunately, if we 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

$$Forward^t \Rightarrow (HaveArrow^t \Leftrightarrow HaveArrow^{t+1})$$
$$Forward^t \Rightarrow (WumpusAlive^t \Leftrightarrow WumpusAlive^{t+1})$$
$$\ldots$$

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.

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 $O(nt)$.

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

$$F^{t+1} \;\Leftrightarrow\; ActionCausesF^t \lor (F^t \land \neg ActionCausesNotF^t).$$

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, $L^{t+1}_{1,1}$ is true if either (a) the agent moved *Forward* from [1,2] when facing south, or from [2,1] when facing west; or (b) $L^t_{1,1}$ 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

$$\begin{aligned} L^{t+1}_{1,1} \Leftrightarrow\ & (L^t_{1,1} \land (\neg Forward^t \lor Bump^{t+1})) \\ \lor\ & (L^t_{1,2} \land (FacingSouth^t \land Forward^t)) \\ \lor\ & (L^t_{2,1} \land (FacingWest^t \land Forward^t)). \end{aligned} \tag{7.3}$$

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

$$\begin{array}{l}
\neg Stench^0 \land \neg Breeze^0 \land \neg Glitter^0 \land \neg Bump^0 \land \neg Scream^0\ ; Forward^0 \\
\neg Stench^1 \land \neg Breeze^1 \land \neg Glitter^1 \land \neg Bump^1 \land \neg Scream^1\ ; TurnRight^1 \\
\neg Stench^2 \land \neg Breeze^2 \land \neg Glitter^2 \land \neg Bump^2 \land \neg Scream^2\ ; TurnRight^2 \\
\neg Stench^3 \land \neg Breeze^3 \land \neg Glitter^3 \land \neg Bump^3 \land \neg Scream^3\ ; Forward^3 \\
\neg Stench^4 \land \neg Breeze^4 \land \neg Glitter^4 \land \neg Bump^4 \land \neg Scream^4\ ; TurnRight^4 \\
\neg Stench^5 \land \neg Breeze^5 \land \neg Glitter^5 \land \neg Bump^5 \land \neg Scream^5\ ; Forward^5 \\
\ Stench^6 \land \neg Breeze^6 \land \neg Glitter^6 \land \neg Bump^6 \land \neg Scream^6
\end{array}$$

At this point, we have ASK(KB, $L^6_{1,2}$) = *true*, so the agent knows where it is. Moreover, ASK(KB, $W_{1,3}$) = *true* and 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

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

Finally, ASK(*KB*, $OK^6_{2,2}$) = *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.

7.7.2 A hybrid agent

The ability to deduce various aspects of the state of the world can be combined fairly straight-forwardly 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.

```
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' ≤ 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.

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 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 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.

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 of the world.[13] The process of updating the belief state as new percepts arrive is called **state estimation** (see page 150). 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

$$WumpusAlive^1 \wedge L^1_{2,1} \wedge B_{2,1} \wedge (P_{3,1} \vee P_{2,2}) \qquad (7.4)$$

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 statesthat 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.

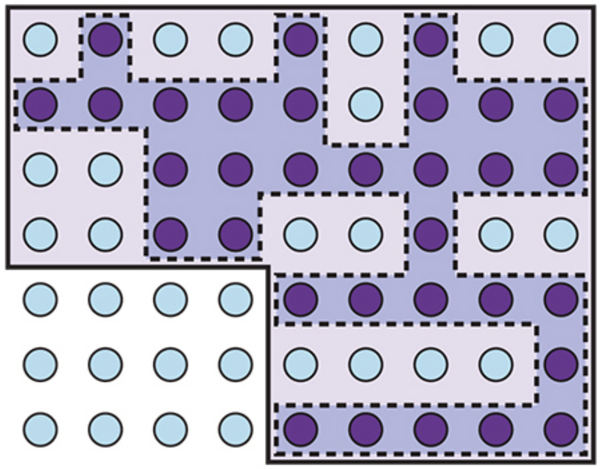


Figure 7.21 Depiction of a 1-CNF belief state (bold outline) as a simply representable, conservative approximation to the exact (wiggly) belief state (shaded region with dashed outline). Each possible world is shown as a circle; the shaded ones are consistent with all the percepts.

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.

```
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.

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

$$Shoot^t \;\Rightarrow\; HaveArrow^t.$$

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_i^t and A_j^t we add the axiom

$$\neg A_i^t \vee \neg A_j^t.$$

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 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.

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 240) 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).

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).

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).

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.

---

[1] Presumably the square containing the wampus also has a stench, but agent that square is eaten before beign able to percive anything.

[2] **Fuzzy logic**, discussed in Chapter 13, allows for degree of truth.

[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.

[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.”

[8] Latin uses two separate words: “vel” is inclusive or and “aut” is exclusive or.

[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.

[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.

[11] Section 7.4.3 conveniently glossed over this requirement.

[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.

[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.

[14] Notice that the addition of precondition axioms means that we need not include preconditions for actions in the successor-state axioms.

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.

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 x 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 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 a 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:

- 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 . . .

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.”

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 firstorder), 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 $A \wedge B$, fuzzy logic needs different rules depending on the domain. Another possibility, covered in Section 25.1, 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.”

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.

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 firstorder 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.

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.

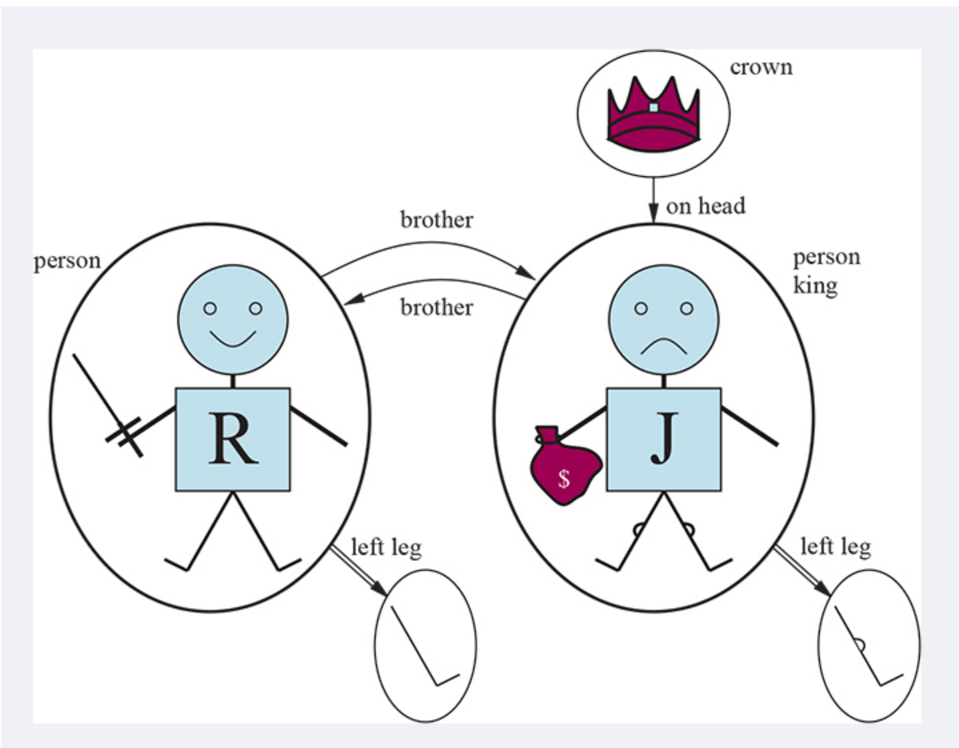


Figure 8.2 A model containing five objects, two binary relations (brother and on-head), three unary relations (person, king, and crown), and one unary function (left-leg).

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

$$\{ \langle \text{Richard the Lionheart}, \text{King John} \rangle, \langle \text{King John}, \text{Richard the Lionhearti} \rangle \}.$$

(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, (the crown, King John). 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 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.

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.

$$
\begin{array}{rcl}
\textit{Sentence} & \rightarrow & \textit{AtomicSentence} \mid \textit{ComplexSentence} \\
\textit{AtomicSentence} & \rightarrow & \textit{Predicate} \mid \textit{Predicate}(\textit{Term}, \ldots) \mid \textit{Term} = \textit{Term} \\
\textit{ComplexSentence} & \rightarrow & (\ \textit{Sentence}\ ) \\
& \mid & \neg\ \textit{Sentence} \\
& \mid & \textit{Sentence} \wedge \textit{Sentence} \\
& \mid & \textit{Sentence} \vee \textit{Sentence} \\
& \mid & \textit{Sentence}\ \Rightarrow\ \textit{Sentence} \\
& \mid & \textit{Sentence}\ \Leftrightarrow\ \textit{Sentence} \\
& \mid & \textit{Quantifier}\ \textit{Variable}, \ldots\ \textit{Sentence} \\
\\
\textit{Term} & \rightarrow & \textit{Function}(\textit{Term}, \ldots) \\
& \mid & \textit{Constant} \\
& \mid & \textit{Variable} \\
\\
\textit{Quantifier} & \rightarrow & \forall \mid \exists \\
\textit{Constant} & \rightarrow & A \mid X_1 \mid \textit{John} \mid \cdots \\
\textit{Variable} & \rightarrow & a \mid x \mid s \mid \cdots \\
\textit{Predicate} & \rightarrow & \textit{True} \mid \textit{False} \mid \textit{After} \mid \textit{Loves} \mid \textit{Raining} \mid \cdots \\
\textit{Function} & \rightarrow & \textit{Mother} \mid \textit{LeftLeg} \mid \cdots \\
\textsc{Operator Precedence} & : & \neg, =, \wedge, \vee, \Rightarrow, \Leftrightarrow
\end{array}
$$

Figure 8.3 The syntax of first-order logic with equality, specified in Backus-Naur form (see page 1081 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.

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.

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:

- *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.*

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.

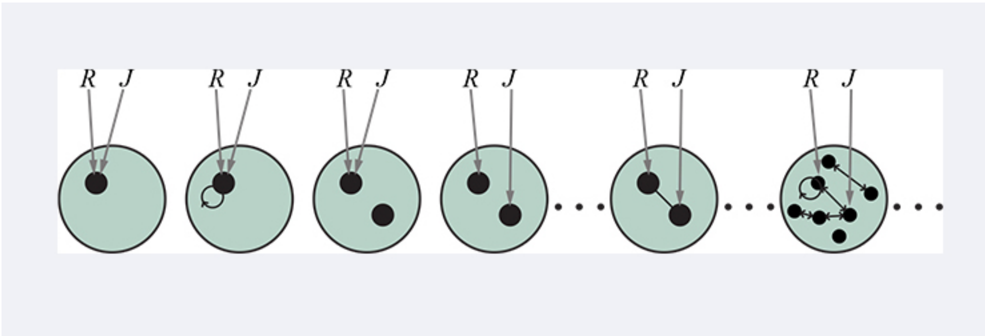


Figure 8.4 Some members of the set of all models for a language with two constant symbols, *R* and J, and one binary relation symbol. The interpretation of each constant symbol is shown by a gray arrow. Within each model, the related objects are connected by arrows.

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

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 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

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:

$$\neg Brother(LeftLeg(Richard), John)$$
$$Brother(Richard, John) \ \wedge\ Brother(John, Richard)$$
$$King(Richard) \vee King(John)$$
$$\neg King(Richard) \ \Rightarrow\ King(John)\ .$$

8.2.6 Quantifiers

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 (∀)

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 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

$$\forall x\, King(x) \;\Rightarrow\; Person(x)\,.$$

The **universal quantifier** ∀ 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**.

Intuitively, the sentence ∀x P, where P is any logical sentence, says that P is true for every object x. More precisely, ∀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.

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:

$$x \rightarrow \text{Richard the Lionheart.}$$
$$x \rightarrow \text{ King John,}$$
$$x \rightarrow \text{ Richard's left leg,}$$
$$x \rightarrow \text{ John's left leg,}$$
$$x \rightarrow \text{ the crown.}$$

The universally quantified sentence ∀x *King*(x) ⇒ *Person*(x) is true in the original model if the sentence *King*(x) ⇒ *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:

$$\text{Richard the Lionheart is a king} \Longrightarrow$$
$$\text{King John is a king} \Longrightarrow \text{King Joh}$$
$$\text{Richard's left leg is a king} \;\Rightarrow\; \text{Richard's left leg is a person.}$$
$$\text{John's left leg is a king} \;\Rightarrow\; \text{John's leftleg is a person.}$$
$$\text{The crown is a king} \Longrightarrow \text{the crow}$$

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 ⇒ (Figure 7.8 on page 237), 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 ⇒ 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

$$\forall x\, King(x) \;\wedge Person(x)$$

would be equivalent to asserting

Richard the Lionheart is a king ∧ Richard the Lionheart is a person,
King John is a king ∧ King John is a person,
Richard’s left leg is a king ∧ Richard’s left leg is a person,

and so on. Obviously, this does not capture what we want.

Existential quantification (∃)

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 ∧ Richard the Lionheart is on John's head;
King John is a crown ∧ King John is on John's head;
Richard's left leg is a crown ∧ Richard's left leg is on John's head;
John's left leg is a crown ∧ John's left leg is on John's head;
The crown is a crown ∧ 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 ⇒ appears to be the natural connective to use with ∀, ∧ is the natural connective to use with ∃. Using ∧ as the main connective with ∀ led to an overly strong statement in the example in the previous section; using ⇒ with ∃ 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 ⇒ Richard the Lionheart is on John's head;
King John is a crown ⇒ King John is on John's head;
Richard's left leg is a crown ⇒ 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.

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

$$\forall x \forall y\, Brother\,(x,y)\ \Rightarrow\ Sibling(x,y).$$

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

$$\forall x\,,y\ Sibling(x{,}y)\ \Leftrightarrow\ Sibling(y{,}x).$$

In other cases we will have mixtures. "Everybody loves somebody" means that for every person, there is someone that person loves:

$$\forall x\,\exists y\,Loves(x{,}y)\ .$$

On the other hand, to say "There is someone who is loved by everyone," we write

$$\exists y\,\forall x\,Loves(x{,}y)\ .$$

The order of quantification is therefore very important. It becomes clearer if we insert parentheses. ∀*x* (∃*y* *Loves*(*x*, *y*)) says that *everyone* has a particular property, namely, the property that they love someone. On the other hand, ∃*y* (∀*x* *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

$$\forall x\,(Crown\,(x) \vee (\exists x\ Brother\,(Richard, x)))\ .$$

Here the x in *Brother (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: ∃*x* *Brother (Richard, x)* is a sentence about Richard (that he has a brother), not about *x*; so putting a ∀*x* outside it has no effect. It could equally well have been written ∃*z* *Brother* (*Richard, z*). Because this can be a source of confusion, we will always use different variable names with nested quantifiers.

Connections between ∀ and ∃

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:

$$\forall x\,\neg Likes(x, Parsnips)\quad \text{is equivalent to}\quad \neg\exists x\ \ Likes(x, Parsnips)\ .$$

We can go one step further: "Everyone likes ice cream" means that there is no one who does not like ice cream:

$$\forall x\ Likes(x, IceCream)\quad \text{is equivalent to}\quad \neg\exists x\ \ \neg Likes(x, IceCream)\ .$$

Because ∀ is really a conjunction over the universe of objects and ∃ 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:

$$\begin{array}{llll}
\neg\exists x\,P \equiv \forall x\,\neg P & & \neg(P \vee Q) \equiv \neg P \wedge \neg Q \\
\neg\forall x\,P \equiv \exists x\,\neg P & & \neg(P \wedge Q) \equiv \neg P \vee \neg Q \\
\forall x\,P \equiv \neg\exists x\,\neg P & & P \wedge Q \equiv \neg(\neg P \vee \neg Q) \\
\exists x\,P \equiv \neg\forall x\,\neg P & & P \vee Q \equiv \neg(\neg P \wedge \neg Q).
\end{array}$$

Thus, we do not really need both ∀ and ∃, just as we do not really need both ∧ and ∀ . Still, readability is more important than parsimony, so we will keep both of the quantifiers.

8.2.7 Equality

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

$$\exists x, y\, Brother(x, Richard) \land Brother(y, Richard) \land \neg(x = y).$$

The sentence

$$\exists x, y\, Brother(x, Richard) \land Brother(y, Richard)$$

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

$$Brother(John, Richard) \land Brother(Geoffrey, Richard), \tag{8.3}$$

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* ≠ *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:

$$\begin{aligned} &Brother\,(John, Richard) \land Brother\,(Geoffrey, Richard) \land John \neq Geoffrey \\ &\qquad \land \forall x \;\; Brother\,(x, Richard) \Rightarrow (x = John \lor x = Geoffrey)\,. \end{aligned}$$

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?

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.

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.

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 277). 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.

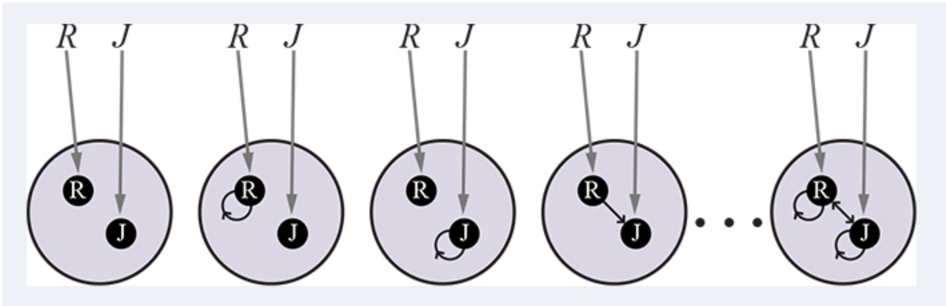


Figure 8.5 Some members of the set of all models for a language with two constant symbols, R and J, and one binary relation symbol, under database semantics. The interpretation of the constant symbols is fixed, and there is a distinct object for each constant symbol.

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 knowl edge representation, a domain is just some part of the world about which we wish to express some knowledge.

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 person, and all kings are persons:

$$\text{TELL } (KB,\ King(John))\ .$$
$$\text{TELL } (KB,\ Person(Richard))\ .$$
$$\text{TELL } (KB, \forall x\ King(x)\ \Rightarrow Person(x))\ .$$

We can ask questions of the knowledge base using ASK. For example,

$$\text{ASK } (KB,\ King(John))$$

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

$$\text{ASK } (KB,\ Person(John))$$

should also return *true***.** We can ask quantified queries, such as ASK(KB, ∃*x Person(x)).*

$$\text{ASK}(KB,\ \exists x\, Person(x)).$$

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,

$$\text{ASK VARS } (KB, Person(x))$$

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.** ASKVARSis 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*) ∨ *King*(*Richard*) there is no single binding to x that makes the query ∃*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:

$$\forall m, c \;\; Mother\,(c) = m \;\Leftrightarrow Female\,(m) \wedge Parent\,(m, c).$$

One's husband is one's male spouse:

$$\forall w, h \;\; Husband\,(h, w) \;\Leftrightarrow Male\,(h) \wedge Spouse\,(h, w).$$

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:

$$\forall\, g, c \;\; Grandparent\,(g, c) \;\Leftrightarrow\; \exists\, p\, Parent\,(g, p) \;\wedge Parent\;(p, c).$$

A sibling is another child of one's parent:

$$\forall x, y \;\; Sibling\,(x, y) \;\;\Leftrightarrow\;\; x \neq y \wedge \exists p \;\, Parent\,(p, x) \wedge Parent\,(p, y).$$

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.

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:

$$\forall x,y \quad Sibling(x,y) \;\Leftrightarrow\; Sibling(y,x).$$

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

$$\forall x \quad Person\,(x) \Leftrightarrow \ldots$$

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:

$$\begin{array}{ll} \forall x & Person(x) \;\Rightarrow\; \ldots \\ \forall x & \ldots \;\Rightarrow\; Person(x)\,. \end{array}$$

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 *Spouse(George, Laura)*; but this does not follow from the axioms given earlier—even after we add *Jim* ≠ *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

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:

$$\forall m\ NatNum\ (m)\ \Rightarrow\ +(0{,}m) = m.$$
$$\forall m, n\, NatNum\ (m) \wedge NatNum\,(n) \Rightarrow +\,(S(m), n) = S(+\,(m, n))\ .$$

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

$$\forall m, n\ NatNum\,(m)\ \wedge NatNum\,(n)\ \ \Rightarrow\ \ (m+1)+n = (m+n)+1.$$

This axiom reduces addition to repeated application of the successor function.

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 ∀*x* ∀*y* *P*(*x, y*) with ∀*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.

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.

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:

$$\forall\, s\; Set\,(s) \Leftrightarrow (s = \{\})\; \vee\; (\exists x,\; s_2\; Set\,(s_2)\; \wedge\; s\; =\; Add\,(x,\, s_2))\; .$$

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:

$$\forall x,\; s\; x \in s\;\; \Leftrightarrow\;\; \exists y, s_2\,(s = Add\,(y,\; s_2)\; \wedge\; (x = y \vee x \in s_2)).$$

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:

$$\forall s_1,\; s_2\,(s_1\; =\; s_2)\;\; \Leftrightarrow\;\; (s_1 \subseteq s_2 \wedge s_2 \subseteq s_1).$$

7. An object is in the intersection of two sets if and only if it is a member of both sets:

$$\forall x,\; s_1,\; s_2\, x \in (s_1 \cap\; s_2)\;\; \Leftrightarrow\;\; (x \in s_1 \wedge x \in s_2).$$

8. An object is in the union of two sets if and only if it is a member of either set:

$$\forall x,\; s_1,\; s_2\, x \in (s_1 \cup\; s_2)\;\; \Leftrightarrow\;\; (x \in s_1 \vee x \in s_2).$$

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.

8.3.4 The wumpus world

Some propositional logic axioms for the wumpus world were given in Chapter 7. The firstorder 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

$$\text{ASK VARS}(KB,\ BestAction(a, 5))\ .$$

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:

$$\begin{aligned}
&\forall t,\ s, g, w, c\ Percept\left([s, Breeze, g, w, c], t\right) \ \Rightarrow\ Breeze(t)\\
&\forall t,\ s, g, w, c\ Percept\left([s, None, g, w, c], t\right) \ \Rightarrow\ Breeze(t)\\
&\forall t,\ s, b, w, c\ Percept\left([s, Glitter, w, c], t\right) \ \Rightarrow\ Glitter(t)\\
&\forall t,\ s, b, w, c\ Percept\left([s, None, w, c], t\right) \ \Rightarrow\ \neg Glitter(t)
\end{aligned}$$

and so on. These rules exhibit a trivial form of the reasoning process called **perception**, which we study in depth in Chapter 27. 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 $Square_{1,2}$ $a$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

$$\begin{aligned}
&\forall x, y, a, b\ Adjacent([x, y], [a, b]) \ \Leftrightarrow\\
&\qquad (x = a \land (y = b - 1 \lor y = b + 1)) \lor (y = b \land (x = a - 1 \lor x = a + 1)).
\end{aligned}$$

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 ∀*t* *At*(*Wumpus*, [1,3], *t*). We can then say that objects can be at only one location at a time:

$$\forall x,\ s_1,\ s_2,\ t\ At\,(x, s_1, t) \wedge At(x,\ s_2, t) \Rightarrow s_1 = s_2.$$

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:

$$\forall s, t \quad At\,(Agent,\ s,\ t) \wedge Breeze\,(t) \quad \Rightarrow Breezy(s)\ .$$

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 238) and would need a different set of axioms for each geographical layout of the world, first-order logic just needs one axiom:

$$\forall s \quad Breezy(s) \Leftrightarrow \exists r\ Adjacent(r, s) \wedge Pit(r). \qquad (8.4)$$

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 258) becomes

$$\forall t \quad HaveArrow\,(t+1) \Leftrightarrow (HaveArrow(t) \wedge \neg Action(Shoot, t)).$$

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 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.

   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.

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.

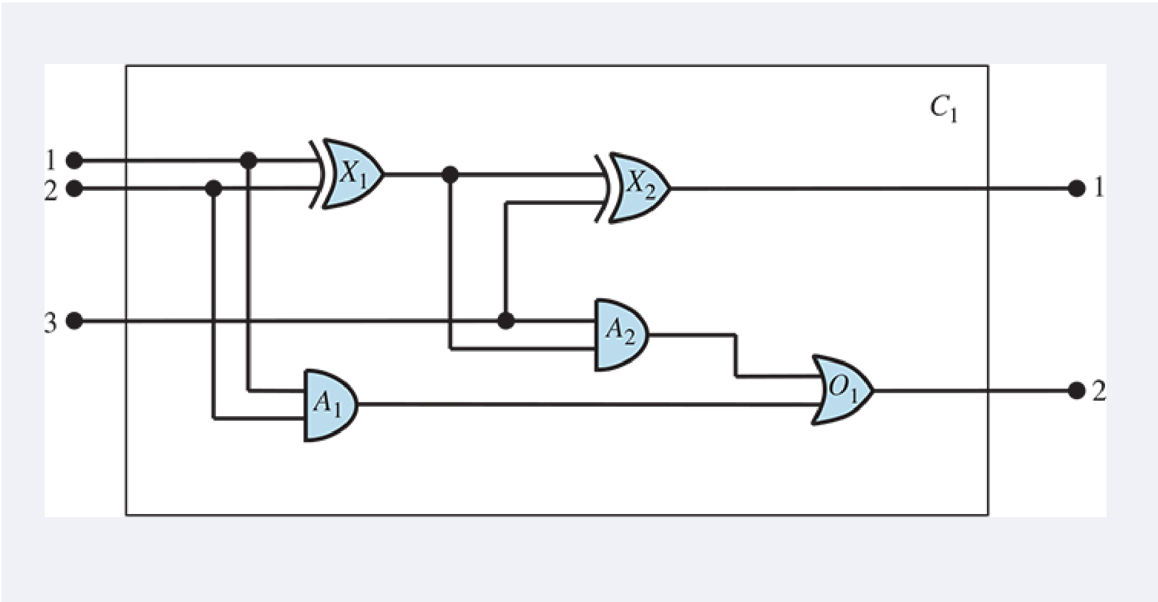


Figure 8.6 A digital circuit *Cj*, purporting to be a one-bit full adder. The first two inputs are the two bits to be added, and the third input is a carry bit. The first output is the sum, and the second output is a carry bit for the next adder. The circuit contains two XOR gates, two AND gates, and one OR gate.

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 $In(1, X_1)$ to denote the first input terminal for circuit X_1. A similar function *Out*(n, c) is used for output terminals. The predicate *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:

$$\forall g\, Gate\,(g) \wedge Type\,(g) = NOT \;\Rightarrow$$
$$Signal(Out(1, g)) \neq Signal(In(1, g)).$$

9. The gates (except for NOT) have two inputs and one output.

$$\forall g\, Gate\,(g) \wedge Type\,(g) = NOT \;\Rightarrow\; Arity(g, 1, 1)\ .$$
$$\forall g\, Gate\,(g) \wedge k = Type\,(g) \wedge (k = AND \vee k = OR \vee k = XOR) \Rightarrow$$
$$Arity(g, 2, 1)\ .$$

10. A circuit has terminals, up to its input and output arity, and nothing beyond its arity:

$$\forall c, i, j\ Circuit(c) \wedge Arity(c,\ i, j) \Rightarrow$$
$$\forall n\ (n \leq i \Rightarrow Terminal(In(n, c))) \wedge (n > i \Rightarrow In(n, c) = Nothing) \wedge$$
$$\forall n\ (n \leq j \Rightarrow Terminal(Out(n, c))) \wedge (n > j \Rightarrow Out(n, c) = Nothing)$$

11. Gates, terminals, and signals are all distinct.

$$\forall g, t, s\ Gate\,(g) \wedge Terminal\,(t) \wedge Signal\,(s) \Rightarrow$$
$$g \neq t \wedge g \neq s \wedge t \neq s.$$

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:

$$Circuit\,(C_1) \wedge Arity\,(C_1, 3, 2)$$
$$Gate\,(X_1) \wedge Type\,(X_1) = XOR$$
$$Gate\,(X_2) \wedge Type\,(X_2) = XOR$$
$$Gate\,(A_1) \wedge Type\,(A_1) = AND$$
$$Gate\,(A_2) \wedge Type\,(A_2) = AND$$
$$Gate\,(O_1) \wedge Type\,(O_1) = OR.$$

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?

$$\exists i_1,\, i_2,\, i_3\ Signal\,(In\,(1, C_1)) = i_1\ \wedge\ Signal\,(In\,(2, C_1)) = i2\ \wedge Signal\,(In\,(3, C_1)) = i_3$$
$$\wedge\, Signal\,(Out\,(1,\ C_1)) = 0\ \wedge Signal\,(Out(2, C_1)) = 1.$$

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?

$$\exists i_1,\, i_2,\, i_3, o_1,\, o_2,\ Signal\,(In(1, C_1)) = i_1 \wedge Signal\,(In(2, C_1)) = i_2$$
$$\wedge\, Signal(In(3, C_1)) = i3 \wedge Signal(Out(1, C_1)) = o_1 \wedge Signal(Out(2, C_1)) = o_2.$$

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.

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 $1 \neq 0$. Suppose we find that the system is 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.

---

[1] First-order logic is also called f**irst-order predicate calculus**; it may be abbreviated as **FOL** or **FOPC**.

[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 (λx : $x \times x$) 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 A 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.

[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.

[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.

[7] Actually he had four, the others being William and Henry.

[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.

[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.

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:

$$\forall x\, King(x) \wedge Greedy(x) \Rightarrow Evil(x).$$

From that we can infer any of the following sentences:

$$\begin{aligned}
&King(John) \wedge Greedy(John) \Rightarrow Evil(John)\\
&King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard)\\
&King(Father(John)) \wedge Greedy(Father(John)) \Rightarrow Evil(Father(John)).\\
&\quad\vdots
\end{aligned}$$

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 SUBST(θ, α) denote the result of applying the substitution θ to the sentence α. Then the rule is written

$$\frac{\forall v\ \alpha}{\text{SUBST}\left(\{v/g\}, \alpha\right)}$$

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 $\{x\ /\ Father\ (John)\}$.

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,

$$\frac{\exists v\ \alpha}{\text{SUBST}\left(\{v/k\}, \alpha\right)}$$

For example, from the sentence

$$\exists x\, Crown\,(x)\ \wedge OnHead(x, John)$$

we can infer the sentence

$$Crown(C_1) \wedge OnHead(C_1, John)$$

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**.

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

$$\begin{aligned}
&\forall x\, King(x) \wedge Greedy(x) \Rightarrow Evil(x)\\
&King\,(John) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (9.1)\\
&Greedy\,(John)\\
&Brother\,(Richard; John)\ \ :
\end{aligned}$$

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

$$King(John) \ \wedge \ Greedy(John) \Rightarrow Evil(John)$$
$$King(Richard) \ \wedge \ Greedy(Richard) \Rightarrow Evil(Richard).$$

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*).

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:

$$\forall y\ Greedy(y)\ . \qquad (9.2)$$

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 knowledgebase sentences $King(John)$ and $Greedy(y)$ will make them identical. Thus, we can infer the consequent of the implication.

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 $\text{SUBST}(\theta\ p_i') = \text{SUBST}(\theta, p_i)$, 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 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 $\{x/\text{John}, \text{y}/\text{John}\}$	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.

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:

$$\text{UNIFY}(p, q) = \theta \text{ where } \text{SUBST}(\theta, p) = \text{SUBST}(\theta, q).$$

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:

$$\text{UNIFY}\,(Knows(John, x), Knows(John, Jane)) = \{x/\text{Jane}\}$$
$$\text{UNIFY}\,(Knows(John, x), Knows(y, Bill)) = \{x/Bill, y/John\}$$
$$\text{UNIFY}(Knows(John,x), Knows(y,Mother(y))) = \{y/John,x/Mother(John)\}$$
$$\text{UNIFY}(Knows(John, x), Knows(x, Elizabeth)) = failure\ .$$

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:

$$\text{UNIFY}(Knows\,(John, x), Knows(x_{17}, Elizabeth)) = \{x/Elizabeth,\ x_{17}/John\}$$

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, 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.

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 lineartime complexity.

```
function UNIFY(x, y, θ=empty) returns a substitution to make x and y identical, or failure
   if θ = failure then return failure
   else if x = y then return θ
   else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
   else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
   else if COMPOUND?(x) and COMPOUND?(y) then
       return UNIFY(ARGS(x), ARGS(y), UNIFY(OP(x), OP(y), θ))
   else if LIST?(x) and LIST?(y) then
       return UNIFY(REST(x), REST(y), UNIFY(FIRST(x), FIRST(y), θ))
   else return failure

function UNIFY-VAR(var, x, θ) returns a substitution
   if {var/val} ∈ θ for some val then return UNIFY(val, x, θ)
   else if {x/val} ∈ θ for some val then return UNIFY(var, val, θ)
   else if OCCUR-CHECK?(var, x) then return failure
   else return add {var/x} to θ
```

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*).

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
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

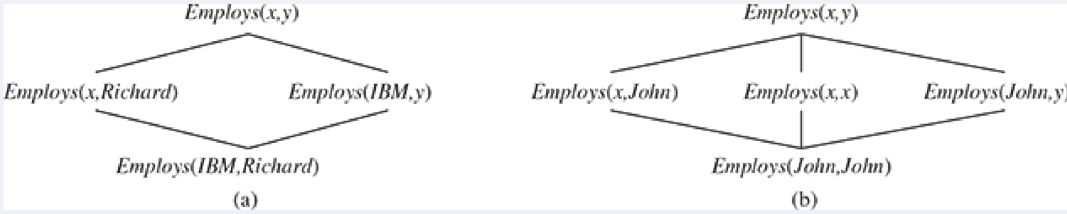


Figure 9.2 (a) The subsumption lattice whose lowest node is *Employs*(*IBM*, *Richard*). (b) The subsumption lattice for the sentence *Employs*(*John*, *John*).

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* ⇒ *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:

$$King(x) \wedge Greedy(x) \Rightarrow Evil(x),$$

but the literals $King(John)$ and $Greedy(y)$ also count as definite clauses. First-order literals can include variables, so $Greedy(y)$ is interpreted as “everyone is greedy” (the universal quantifier is implicit).

Let us put definite clauses to work in representing the following problem:

> The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.

First, we will represent these facts as first-order definite clauses:

“... it is a crime for an American to sell weapons to hostile nations”:

$$American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x).$$

“Nono ... has some missiles.” The sentence $\exists x\ Owns(Nono,x) \wedge Missile(x)$ is transformed into two definite clauses by Existential Instantiation, introducing a new constant M_1:

$$Owns(Nono, M_1) \qquad (9.4)$$
$$Missile\,(M_1) \qquad (9.5)$$

“All of its missiles were sold to it by Colonel West”:

$$Missile\,(x) \wedge Owns(Nono, x) \;\Rightarrow\; Sells(West, x, Nono). \qquad (9.6)$$

We will also need to know that missiles are weapons:

$$\text{Missile}\,(x) \;\Rightarrow Weapon(x) \qquad (9.7)$$

and we must know that an enemy of America counts as “hostile”:

$$\text{Missile}\,(x, America) \;\Rightarrow Hostile(x). \qquad (9.8)$$

“West, who is American . . .”:

$$American(West). \qquad (9.9)$$

“The country Nono, an enemy of America ...”:

$$Enemy(Nono, America). \qquad (9.10)$$

This knowledge base happens to be a **Datalog** knowledge base: Datalog is a language consisting of first-order definite clauses with no function symbols. Datalog gets its name because it can represent the type of statements typically made in relational databases. The absence of function symbols makes inference much easier.

9.3.2 A simple forward-chaining algorithm

Figure 9.3 shows a simple forward chaining inference algorithm. Starting from the known facts, it triggers all the rules whose premises are satisfied, adding their conclusions to the known facts. The process repeats until the query is answered (assuming that just one answer is required) or no new facts are added. Notice that a fact is not "new" if it is just a **renaming** of a known fact—a sentence is a renaming of another if they are identical except for the names of the variables. For example, *Likes*(*x*, *IceCream*) and *Likes*(*y*, *IceCream*) are renamings of each other. They both mean the same thing: "Everyone likes ice cream."

```
function FOL-FC-ASK(KB, α) returns a substitution or false
  inputs: KB, the knowledge base, a set of first-order definite clauses
          α, the query, an atomic sentence

  while true do
     new ← { }        // The set of new sentences inferred on each iteration
     for each rule in KB do
        (p1 ∧ . . . ∧ pn ⇒ q) ← STANDARDIZE-VARIABLES(rule)
        for each θ such that SUBST(θ, p1 ∧ . . . ∧ pn) = SUBST(θ, p′1 ∧ . . . ∧ p′n)
                 for some p′1, . . . , p′n in KB
           q′ ← SUBST(θ, q)
           if q′ does not unify with some sentence already in KB or new then
              add q′ to new
              φ ← UNIFY(q′, α)
              if φ is not failure then return φ
     if new = { } then return false
     add new to KB
```

Figure 9.3 A conceptually straightforward, but inefficient, forward-chaining algorithm. On each iteration, it adds to *KB* all the atomic sentences that can be inferred in one step from the implication sentences and the atomic sentences already in *KB*. The function STANDARDIZE-VARIABLES replaces all variables in its arguments with new ones that have not been used before.

We use our crime problem to illustrate FOL-FC-ASK. The implication sentences available for chaining are (9.3), (9.6), (9.7), and (9.8). Two iterations are required:

- On the first iteration, rule (9.3) has unsatisfied premises.
  Rule (9.6) is satisfied with $\{x/M_1\}$, and *Sells*(*West*, M_1, *Nono*) is added.
  Rule (9.7) is satisfied with $\{x/M_1\}$, and *Weapon*(M_1) is added.

Rule (9.8) is satisfied with $\{x/Nono\}$, and $Hostile(Nono)$ is added.

- On the second iteration, rule (9.3) is satisfied with $\{x/West, y/M_1, z/Nono\}$, and the inference $Criminal(West)$ is added.

Figure 9.4 shows the proof tree that is generated. Notice that no new inferences are possible at this point because every sentence that could be concluded by forward chaining is already contained explicitly in the KB. Such a knowledge base is called a **fixed point** of the inference process. Fixed points reached by forward chaining with first-order definite clauses are similar to those for propositional forward chaining (page 249); the principal difference is that a firstorder fixed point can include universally quantified atomic sentences.

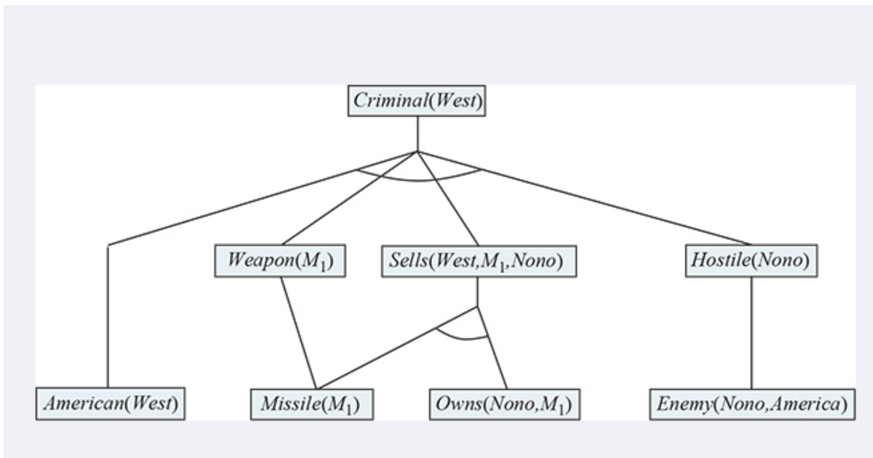


Figure 9.4 The proof tree generated by forward chaining on the crime example. The initial facts appear at the bottom level, facts inferred on the first iteration in the middle level, and facts inferred on the second iteration at the top level.

FOL-FC-ASK is easy to analyze. First, it is **sound**, because every inference is just an application of Generalized Modus Ponens, which is sound. Second, it is **complete** for definite clause knowledge bases; that is, it answers every query whose answers are entailed by any knowledge base of definite clauses.

For Datalog knowledge bases, which contain no function symbols, the proof of completeness is fairly easy. We begin by counting the number of possible facts that can be added, which determines the maximum number of iterations. Let k be the maximum **arity** (number of arguments) of any predicate, p be the number of predicates, and n be the number of constant symbols. Clearly, there can be no more than pn^k distinct ground facts, so after this many iterations the algorithm must have reached a fixed point. Then we can make an argument very similar to the proof of completeness for propositional forward chaining. (See page 249.) The details of how to make the transition from propositional to first-order completeness are given for the resolution algorithm in Section 9.5.

For general definite clauses with function symbols, FOL-FC-ASK can generate infinitely many new facts, so we need to be more careful. For the case in which an answer to the query sentence q is entailed, we must appeal to Herbrand's theorem (page 300) to establish that the algorithm will find a proof. (See Section 9.5 for the resolution case.) If the query has no answer, the algorithm could fail to terminate in some cases. For example, if the knowledge base includes the Peano axioms

$$NatNum(0)$$
$$\forall n\, NatNum(n) \Rightarrow NatNum(S(n)),$$

then forward chaining adds *NatNum*(*S*(0)), *NatNum*(*S*(*S*(0))), *NatNum*(*S*(*S*(*S*(0)))), and so on. This problem is unavoidable in general. As with general first-order logic, entailment with definite clauses is semidecidable.

9.3.3 Efficient forward chaining

The forward-chaining algorithm in Figure 9.3 is designed for ease of understanding, not efficiency. There are three sources of inefficiency. First, the inner loop of the algorithm tries to match every rule against every fact in the knowledge base. Second, the algorithm rechecks every rule on every iteration, even if very few additions have been made to the knowledge base. Third, the algorithm can generate many facts that are irrelevant to the goal. We address each of these issues in turn.

Matching rules against known facts

The problem of matching the premise of a rule against the facts in the knowledge base might seem simple enough. For example, suppose we want to apply the rule

$$Missile(x) \Rightarrow Weapon(x)\ .$$

Then we need to find all the facts that unify with *Missile* (*x*); in a suitably indexed knowledge base, this can be done in constant time per fact. Now consider a rule such as

$$Missile(x) \wedge Owns(Nono, x)\ \Rightarrow\ Sells(West, x, Nono).$$

Again, we can find all the objects owned by Nono in constant time per object; then, for each object, we could check whether it is a missile. However, if the knowledge base contains many objects owned by Nono and very few missiles, then it would be better to find all the missiles first and then check whether they are owned by Nono. This is the **conjunct ordering** problem: find an ordering to solve the conjuncts of the rule premise so that the total cost is minimized. It turns out that finding the optimal ordering is NP-hard, but good heuristics are available. For example, the **minimum-remaining-values** (MRV) heuristic used for CSPs in Chapter 5 would suggest ordering the conjuncts to look for missiles first if there are fewer missiles than there are objects owned by Nono.

The connection between this **pattern matching** and constraint satisfaction is actually very close. We can view each conjunct as a constraint on the variables that it contains—for example, *Missile*(*x*) is a unary constraint on *x*. Extending this idea, *we can express every finite-domain CSP as a single definite clause together with some associated ground facts*. Consider the map-coloring problem from Figure 5.1, shown again in Figure 9.5(a). An equivalent formulation as a single definite clause is given in Figure 9.5(b). Clearly, the conclusion *Colorable*() can be inferred only if the CSP has a solution. Because CSPs in general include 3-SAT problems as special cases, we can conclude that *matching a definite clause against a set of facts is NP-hard.*

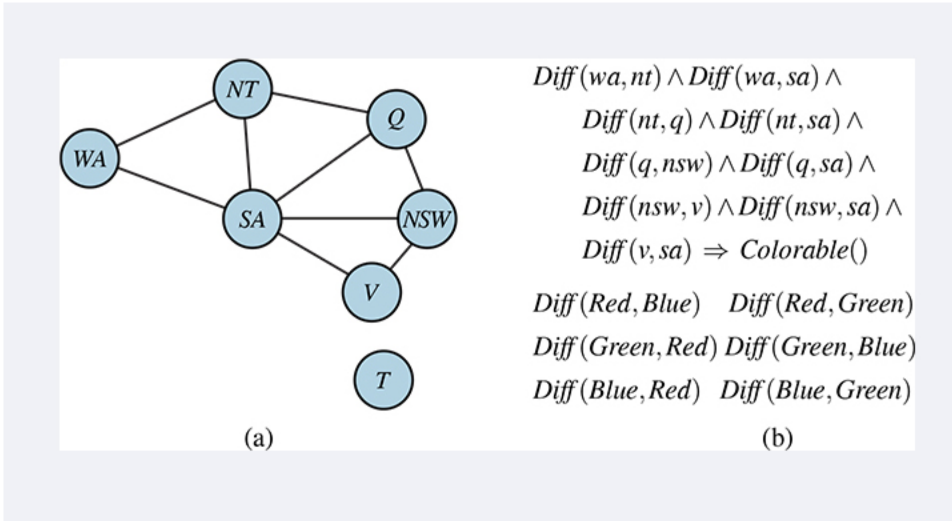


Figure 9.5 (a) Constraint graph for coloring the map of Australia. (b) The map-coloring CSP expressed as a single definite clause. Each map region is represented as a variable whose value can be one of the constants *Red, Green,* or *Blue* (which are declared *Diff).*

It might seem rather depressing that forward chaining has an NP-hard matching problem in its inner loop. There are three ways to cheer ourselves up:

- We can remind ourselves that most rules in real-world knowledge bases are small and simple (like the rules in our crime example) rather than large and complex (like the CSP formulation in Figure 9.5). It is common in the database world to assume that both the sizes of rules and the arities of predicates are bounded by a constant and to worry only about **data complexity**—that is, the complexity of inference as a function of the number of ground facts in the knowledge base. It is easy to show that the data complexity of forward chaining is polynomial, not exponential.
- We can consider subclasses of rules for which matching is efficient. Essentially every Datalog clause can be viewed as defining a CSP, so matching will be tractable just when the corresponding CSP is tractable. Chapter 5 describes several tractable families of CSPs. For example, if the constraint graph (the graph whose nodes are variables and whose links are constraints) forms a tree, then the CSP can be solved in linear time. Exactly the same result holds for rule matching. For instance, if we remove South Australia from the map in Figure 9.5, the resulting clause is

  $$Diff(wa, nt) \wedge Diff(nt, q) \wedge Diff(q, nsw) \wedge Diff(nsw, v) \Rightarrow Colorable()$$

  which corresponds to the reduced CSP shown in Figure 5.12 on page 185. Algorithms for solving tree-structured CSPs can be applied directly to the problem of rule matching.
- We can try to eliminate redundant rule-matching attempts in the forward-chaining algorithm, as described next.

Incremental forward chaining

When we showed how forward chaining works on the crime example, we cheated. In particular, we omitted some of the rule matching done by the algorithm shown in Figure 9.3. For example, on the second iteration, the rule

$$Missile(x) \Leftrightarrow Weapon(x)$$

matches against *Missile*(M_1) (again), and of course the conclusion *Weapon*(M_1) is already known so nothing happens. Such redundant rule matching can be avoided if we make the following observation: *Every new fact inferred on iteration t must be derived from at least one new fact inferred on iteration* $t - 1$. This is true because any inference that does not require a new fact from iteration $t - 1$ could have been done at iteration $t - 1$ already.

This observation leads naturally to an incremental forward-chaining algorithm where, at iteration t, we check a rule only if its premise includes a conjunct p_i that unifies with a fact p_i newly inferred at iteration $t - 1$. The rule-matching step then fixes p_i to match with p'_i, but allows the other conjuncts of the rule to match with facts from any previous iteration. This algorithm generates exactly the same facts at each iteration as the algorithm in Figure 9.3, but is much more efficient.

With suitable indexing, it is easy to identify all the rules that can be triggered by any given fact, and many real systems operate in an "update" mode wherein forward chaining occurs in response to every TELL. Inferences cascade through the set of rules until the fixed point is reached, and then the process begins again for the next new fact.

Typically, only a small fraction of the rules in the knowledge base are actually triggered by the addition of a given fact. This means that a great deal of redundant work is done in repeatedly constructing partial matches that have some unsatisfied premises. Our crime example is rather too small to show this effectively, but notice that a partial match is constructed on the first iteration between the rule

$$American(x) \land Weapon(y) \land Sells(x, y, z) \land Hostile(z) \Rightarrow Criminal(x)$$

and the fact *American*(*West*). This partial match is then discarded and rebuilt on the second iteration (when the rule succeeds). It would be better to retain and gradually complete the partial matches as new facts arrive, rather than discarding them.

The **Rete algorithm**[3] was the first to address this problem. The algorithm preprocesses the set of rules in the knowledge base to construct a dataflow network in which each node is a literal from a rule premise. Variable bindings flow through the network and are filtered out when they fail to match a literal. If two literals in a rule share a variable—for example, *Sells*(x, y, z) ∧ *Hostile*(z) in the crime example—then the bindings from each literal are filtered through an equality node. A variable binding reaching a node for an n-ary literal such as *Sells*(x,y,z) might have to wait for bindings for the other variables to be established before the process can continue. At any given point, the state of a Rete network captures all the partial matches of the rules, avoiding a great deal of recomputation.

Rete networks, and various improvements thereon, have been a key component of so- called **production systems**, which were among the earliest forward-chaining systems in widespread use.[4] The XCON system (originally called R1; McDermott, 1982) was built with a production-system architecture. XCON contained several thousand rules for designing configurations of computer components for customers of the Digital Equipment Corporation. It was one of the first clear commercial successes in the emerging field of expert systems. Many other similar systems have been built with the same underlying technology, which has been implemented in the general-purpose language OPS-5.

Production systems are also popular in **cognitive architectures**—that is, models of human reasoning—such as ACT (Anderson, 1983) and S OAR(Laird *et al.,* 1987). In such systems, the "working memory" of the system models human short-term memory, and the productions are part of long-term memory. On each cycle of operation, productions are matched against the working memory of facts. A production whose conditions are satisfied can add or delete facts in working memory. In contrast to the typical situation in databases, production systems often have many rules and relatively few facts. With suitably optimized matching technology, systems can operate in real time with tens of millions of rules.

Irrelevant facts

Another source of inefficiency is that forward chaining makes all allowable inferences based on the known facts, *even if they are irrelevant to the goal.* In our crime example, there were no rules capable of drawing irrelevant conclusions. But if there had been many rules describing the eating habits of Americans or the components and prices of missiles, then FOL-FC-ASK would have generated irrelevant conclusions.

One way to avoid drawing irrelevant conclusions is to use backward chaining, as described in Section 9.4. Another way is to restrict forward chaining to a selected subset of rules, as in PL-FC-ENTAILS? (page 249). A third approach has emerged in the field of **deductive databases**, which are large-scale databases, like relational databases, but which use forward chaining as the standard inference tool rather than SQL queries. The idea is to rewrite the rule set, using information from the goal, so that only relevant variable bindings—those belonging to a so-called **magic set**—are considered during forward inference. For example, if the goal is *Criminal*(*West*), the rule that concludes *Criminal*(*x*) will be rewritten to include an extra conjunct that constrains the value of *x*:

$$Magic(x) \land American(x) \land Weapon(y) \land Sells(x, y, z) \land Hostile(z) \Rightarrow Criminal(x).$$

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

9.4 Backward Chaining

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

9.4.1 A backward-chaining algorithm

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

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

function FOL-BC-OR(KB, goal, θ) returns a substitution
  for each rule in FETCH-RULES-FOR-GOAL(KB, goal) do
    (lhs ⇒ rhs) ← STANDARDIZE-VARIABLES(rule)
    for each θ′ in FOL-BC-AND(KB, lhs, UNIFY(rhs, goal, θ)) do
      yield θ′

function FOL-BC-AND(KB, goals, θ) returns a substitution
  if θ = failure then return
  else if LENGTH(goals) = 0 then yield θ
  else
    first,rest ← FIRST(goals), REST(goals)
    for each θ′ in FOL-BC-OR(KB, SUBST(θ, first), θ) do
      for each θ″ in FOL-BC-AND(KB, rest, θ′) do
        yield θ″
```

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

Backward chaining is a kind of AND/OR search—the OR part because the goal query can be proved by any rule in the knowledge base, and the AND part because all the conjuncts in the *lhs* of a clause must be proved. FOL-BC-OR works by fetching all clauses that might unify with the goal, standardizing the variables in the clause to be brand-new variables, and then, if the *rhs* of the clause does indeed unify with the goal, proving every conjunct in the *lhs*, using FOL-BC-AND. That function works by proving each of the conjuncts in turn, keeping track of the accumulated substitution as it goes. Figure 9.7 is the proof tree for deriving *Criminal*(*West*) from sentences (9.3) through (9.10).

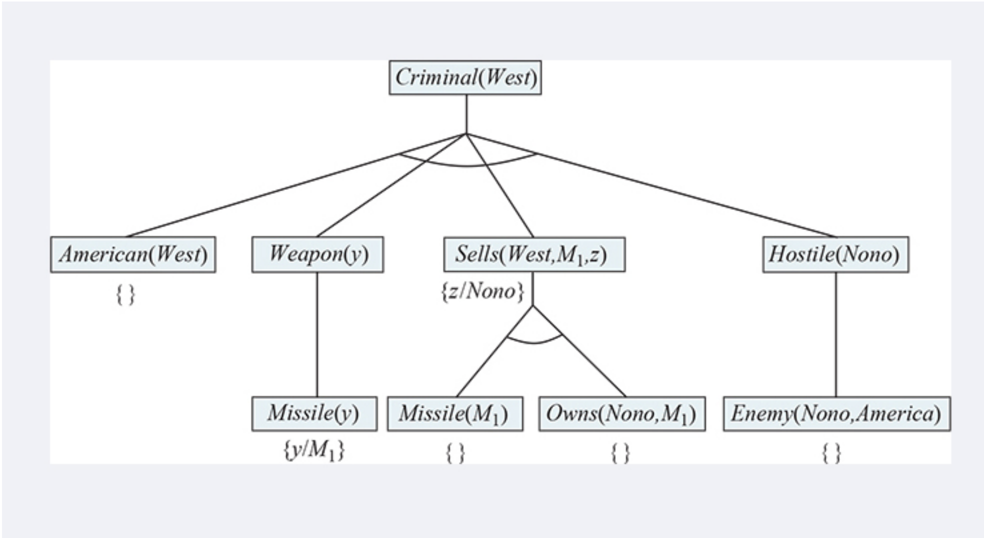


Figure 9.7 Proof tree constructed by backward chaining to prove that West is a criminal. The tree should be read depth first, left to right. To prove *Criminal*(*West*), we have to prove the four conjuncts below it. Some of these are in the knowledge base, and others require further backward chaining. Bindings for each successful unification are shown next to the corresponding subgoal. Note that once one subgoal in a conjunction succeeds, its substitution is applied to subsequent subgoals. Thus, by the time FOL-BC-ASK gets to the last conjunct, originally *Hostile* (z), z is already bound to *Nono*.

Backward chaining, as we have written it, is clearly a depth-first search algorithm. This means that its space requirements are linear in the size of the proof. It also means that backward chaining (unlike forward chaining) suffers from problems with repeated states and incompleteness. Despite these limitations, backward chaining has proven to be popular and effective in logic programming languages.

9.4.2 Logic programming

Logic programming is a technology that comes close to embodying the declarative ideal described in Chapter 7: that systems should be constructed by expressing knowledge in a formal language and that problems should be solved by running inference processes on that knowledge. The ideal is summed up in Robert Kowalski's equation,

$$Algorithm \;=\; Logic + Control\,.$$

Prolog is the most widely used logic programming language. It is used primarily as a rapidprototyping language and for symbol-manipulation tasks such as writing compilers (Van Roy, 1990) and parsing natural language (Pereira and Warren, 1980). Many expert systems have been written in Prolog for legal, medical, financial, and other domains.

Prolog programs are sets of definite clauses written in a notation somewhat different from standard first-order logic. Prolog uses uppercase letters for variables and lowercase for constants—the opposite of our convention for logic. Commas separate conjuncts in a clause, and the clause is written "backwards" from what we are used to; instead of $A \wedge B \Rightarrow C$ in Prolog we have `C :- A, B`. Here is a typical example:

```
criminal(X) :- american(X), weapon(Y), sells(X,Y,Z), hostile(Z).
```

In Prolog the notation `[E|L]` denotes a list whose first element is `E` and whose rest is `L`. Here is a Prolog program for `append(X,Y,Z)`, which succeeds if list `Z` is the result of appending lists `X` and `Y`:

$$\text{append}\ ([], \text{Y}, \text{Y}).$$
$$\text{append}\ ([\text{A}|\text{X}], \text{Y}, [\text{A}|\text{Z}]) \quad :- \quad \text{append}\ (\text{X,Y,Z}).$$

In English, we can read these clauses as (1) appending the empty list and the list `Y` produces the same list `Y`, and (2) `[A|Z]` is the result of appending `[A|X]` and `Y`, provided that `Z` is the result of appending `X` and `Y`. In most high-level languages we can write a similar recursive function that describes how to append two lists. The Prolog definition is actually more powerful, however, because it describes a *relation* that holds among three arguments, rather than a *function* computed from two arguments. For example, we can ask the query `append(X,Y,[1,2,3])`: what two lists can be appended to give `[1,2,3]`? Prolog gives us back the solutions

$$\begin{array}{ll} \text{X} = [] & \text{Y} = [1,2,3]; \\ \text{X} = [1] & \text{Y} = [2,3]; \\ \text{X} = [1,2] & \text{Y} = [3]; \\ \text{X} = [1,2,3] & \text{Y} = [] \end{array}$$

The execution of Prolog programs is done through depth-first backward chaining, where clauses are tried in the order in which they are written in the knowledge base. Prolog's design represents a compromise between declarativeness and execution efficiency. Some aspects of Prolog fall outside standard logical inference:

- Prolog uses the database semantics of Section 8.2.8 rather than first-order semantics, and this is apparent in its treatment of equality and negation (see Section 9.4.4).
- There is a set of built-in functions for arithmetic. Literals using these function symbols are "proved" by executing code rather than doing further inference. For example, the goal "`X` is 4+3" succeeds with `X` bound to 7. On the other hand, the goal "`5 is X+Y`" fails, because the built-in functions do not do arbitrary equation solving.
- There are built-in predicates that have side effects when executed. These include inputoutput predicates and the `assert/retract` predicates for modifying the knowledge base. Such predicates have no counterpart in logic and can produce confusing results— for example, if facts are asserted in a branch of the proof tree that eventually fails.
- The **occur check** is omitted from Prolog's unification algorithm. This means that some unsound inferences can be made; these are almost never a problem in practice.
- Prolog uses depth-first backward-chaining search with no checks for infinite recursion. This makes for a usable programming language that is very fast when used properly, but it means that some programs that look like valid logic will fail to terminate.

9.4.3 Redundant inference and infinite loops

We now turn to the Achilles heel of Prolog: the mismatch between depth-first search and search trees that include repeated states and infinite paths. Consider the following logic program that decides if a path exists between two points on a directed graph:

```
path(X, Z) :- link(X, Z).
path(X, Z) :- path(X,Y), link(Y, Z).
```

A simple three-node graph, described by the facts link(a,b) and link(b,c), is shown in Figure 9.8(a). With this program, the query path(a,c) generates the proof tree shown in Figure 9.9(a). On the other hand, if we put the two clauses in the order

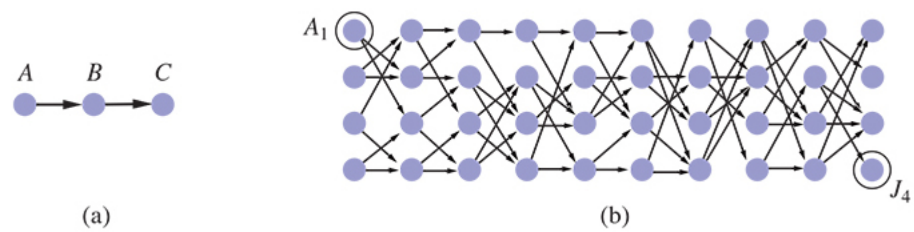


Figure 9.8 (a) Finding a path from A to C can lead Prolog into an infinite loop. (b) A graph in which each node is connected to two random successors in the next layer. Finding a path from A_1 to J_4 requires 877 inferences.

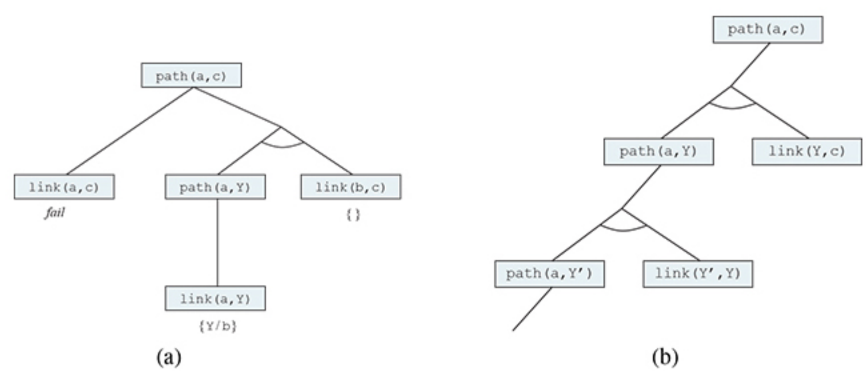


Figure 9.9 (a) Proof that a path exists from A to C. (b) Infinite proof tree generated when the clauses are in the “wrong” order.

path (X,Z) : – path (X,Y), link (Y,Z).
path (X,Z) : – link (X,Z).

then Prolog follows the infinite path shown in Figure 9.9(b). Prolog is therefore **incomplete** as a theorem prover for definite clauses—even for Datalog programs, as this example shows— because, for some knowledge bases, it fails to prove sentences that are entailed. Notice that forward chaining does not suffer from this problem: `once path(a,b), path(b,c)`, and `path(a,c)` are inferred, forward chaining halts.

Depth-first backward chaining also has problems with redundant computations. For example, when finding a path from A_1 to J_4 in Figure 9.8(b), Prolog performs 877 inferences, most of which involve finding all possible paths to nodes from which the goal is unreachable. This is similar to the repeated-state problem discussed in Chapter 3. The total amount of inferen ce can be exponential in the number of ground facts that are generated. If we apply forward chaining instead, at most n^2 `path(X,Y)` facts can be generated linking n nodes. For the problem in Figure 9.8(b), only 62 inferences are needed.

Forward chaining on graph search problems is an example of **dynamic programming**, in which the solutions to subproblems are constructed incrementally from those of smaller subproblems and are cached to avoid recomputation. We can obtain a similar effect in a backward chaining system, except that here we are breaking down large goals into smaller ones, rather than building them up.

Either way, storing intermediate results to avoid duplication is key. This is the approach taken by **tabled logic programming** systems, which use efficient storage and retrieval mechanisms. Tabled logic programming combines the goal-directedness of backward chaining with the dynamic-programming efficiency of forward chaining. It is also complete for Data- log knowledge bases, which means that the programmer need worry less about infinite loops. (It is still possible to get an infinite loop with predicates like `father(X,Y)` that refer to a potentially unbounded number of objects.)

9.4.4 Database semantics of Prolog

Prolog uses database semantics, as discussed in Section 8.2.8. The unique names assumption says that every Prolog constant and every ground term refers to a distinct object, and the closed world assumption says that the only sentences that are true are those that are entailed by the knowledge base. There is no way to assert that a sentence is false in Prolog. This makes Prolog less expressive than first-order logic, but it is part of what makes Prolog more efficient and more concise. Consider the following assertions about some course offerings:

$$Course(CS,\ 101),\ Course(CS,\ 102),\ Course(CS,\ 106),\ Course\,(EE,\ 101).$$

Under the unique names assumption, *CS* and *EE* are different (as are 101, 102, and 106), so this means that there are four distinct courses. Under the closed-world assumption there are no other courses, so there are exactly four courses. But if these were assertions in FOL rather than in database semantics, then all we could say is that there are somewhere between one and infinity courses. That's because the assertions (in FOL) do not deny the possibility that other unmentioned courses are also offered, nor do they say that the courses mentioned are different from each other. If we wanted to translate Equation (9.11) into FOL, we would get the following sentence:

$$\begin{aligned} Course(d,n) \ \Leftrightarrow\ & (d = CS \wedge n = 101) \vee (d = CS \wedge n = 102) \\ & \vee (d = CS \wedge n = 106) \vee (d = EE \wedge n = 101). \end{aligned} \tag{19.12}$$

This is called the **completion** of Equation (9.11). It expresses in FOL the idea that there are at most four courses. To express in FOL the idea that there are at least four courses, we need to write the completion of the equality predicate:

$$\begin{aligned} x = y \Leftrightarrow & (x = CS \wedge y = CS) \vee (x = EE \wedge y = EE) \vee (x = 101 \wedge y = 101) \\ & \vee (x = 102 \wedge y = 102) \vee (x = 106 \wedge y = 106). \end{aligned}$$

The completion is useful for understanding database semantics, but for practical purposes, if your problem can be described with database semantics, it is more efficient to reason with Prolog or some other database semantics system, rather than translating into FOL and reasoning with a full FOL theorem prover.

9.4.5 Constraint logic programming

In our discussion of forward chaining (Section 9.3), we showed how constraint satisfaction problems (CSPs) can be encoded as definite clauses. Standard Prolog solves such problems in exactly the same way as the backtracking algorithm given in Figure 5.5.

Because backtracking enumerates the domains of the variables, it works only for **finite- domain** CSPs. In Prolog terms, there must be a finite number of solutions for any goal with unbound variables. (For example, a map coloring problem in which each variable can take on one of four different colors.) Infinite-domain CSPs—for example, with integer- or realvalued variables—require quite different algorithms, such as bounds propagation or linear programming.

Consider the following example. We define `triangle(X,Y,Z)` as a predicate that holds if the three arguments are numbers that satisfy the triangle inequality:

$$\begin{aligned} &\text{triangle(X,Y,Z)} \ \ :- \\ &\quad \text{X>0,\ \ Y>0,\ \ Z>0,\ \ X+Y>Z,\ \ Y+Z>X,\ \ X+Z>Y.} \end{aligned}$$

If we ask Prolog the query `triangle(3,4,5)`, it succeeds. On the other hand, if we ask `triangle(3,4,Z)`, no solution will be found, because the subgoal `Z>0` cannot be handled by Prolog; we can't compare an unbound value to 0.

Constraint logic programming (CLP) allows variables to be *constrained* rather than *bound.* A CLP solution is the most specific set of constraints on the query variables that can be derived from the knowledge base. For example, the solution to the `triangle(3,4,Z)` query is the constraint `7 > Z > 1`. Standard logic programs are just a special case of CLP in which the solution constraints must be equality constraints—that is, bindings.

CLP systems incorporate various constraint-solving algorithms for the constraints allowed in the language. For example, a system that allows linear inequalities on real-valued variables might include a linear programming algorithm for solving those constraints. CLP systems also adopt a much more flexible approach to solving standard logic programming queries. For example, instead of depth-first, left-to-right backtracking, they might use any of the more efficient algorithms discussed in Chapter 5, including heuristic conjunct ordering, backjumping, cutset conditioning, and so on. CLP systems therefore combine elements of constraint satisfaction algorithms, logic programming, and deductive databases.

Several systems that allow the programmer more control over the search order for inference have been defined. The MRS language (Genesereth and Smith, 1981; Russell, 1985) allows the programmer to write **metarules** to determine which conjuncts are tried first. The user could write a rule saying that the goal with the fewest variables should be tried first or could write domain-specific rules for particular predicates.

9.5 Resolution

The last of our three families of logical systems, and the only one that works for any knowledge base, not just definite clauses, is **resolution**. We saw on page 241 that propositional resolution is a complete inference procedure for propositional logic; in this section, we extend it to first-order logic.

9.5.1 Conjunctive normal form for first-order logic

The first step is to convert sentences to **conjunctive normal form** (CNF)—that is, a conjunction of clauses, where each clause is a disjunction of literals.[5] In CNF, literals can contain variables, which are assumed to be universally quantified. For example, the sentence

$$\forall x, y, z\ American(x)\ \land\ Weapon(y)\ \land\ Sells(x, y,\ z)\ \land\ Hostile(z)\ \Rightarrow\ Criminal(x)$$

becomes, in CNF,

$$\neg American(x)\ \lor\ \neg Weapon(y)\ \lor\ \neg Sells(x, y,\ z)\ \lor\ \neg Hostile(z)\ \lor\ Criminal(x).$$

The key is that *Every sentence of first-order logic can be converted into an inferentially equivalent CNF sentence.*

The procedure for conversion to CNF is similar to the propositional case, which we saw on page 244. The principal difference arises from the need to eliminate existential quantifiers. We illustrate the procedure by translating the sentence "Everyone who loves all animals is loved by someone," or

$$\forall x\ [\forall y\ Animal(y)] \Rightarrow Loves(x, y)] \Rightarrow [\exists y\ Loves(y, x)].$$

The steps are as follows:

- **Eliminate implications** : Replace $P \Rightarrow Q$ with $\neg P \lor Q$. For our sample sentence, this needs to be done twice:

  $$\forall x\, \neg[\forall y\, Animal(y)\ \Rightarrow\ Loves(x, y)]\ \lor\ [\exists y\, Loves(y, x)]$$
  $$\forall x\, \neg[\forall y\, \neg Animal(y)\ \lor\ Loves(x, y)]\ \lor\ [\exists y\, Loves(y, x)].$$

- **Move $\neg$ inwards**: In addition to the usual rules for negated connectives, we need rules for negated quantifiers. Thus, we have

  $\neg\forall x\, p$ becomes $\exists x \neg p$
  $\neg\exists x\, p$ becomes $\forall x \neg p.$

  Our sentence goes through the following transformations:

$$\forall x\ [\exists y\ \neg(\neg Animal(y) \lor Loves(x,y))] \lor [\exists y\ Loves(y,x)].$$
$$\forall x\ [\exists y\ \neg\neg Animal(y) \land \neg Loves(x,y)] \lor [\exists y\ Loves(y,x)].$$
$$\forall x\ [\exists y\ Animal(y) \land \neg Loves(x,y)] \lor [\exists y\ Loves(y,x)].$$

Notice how a universal quantifier ($\forall y$) in the premise of the implication has become an existential quantifier. The sentence now reads "Either there is some animal that x doesn't love, or (if this is not the case) someone loves x." Clearly, the meaning of the original sentence has been preserved.

- **Standardize variables** : For sentences like $(\exists\, x P(x)) \lor (\exists\, x Q(x))$ that use the same variable name twice, change the name of one of the variables. This avoids confusion later when we drop the quantifiers. Thus, we have

  $$\forall x\ [\exists y\ Animal(y) \land \neg Loves(x,y)] \lor [\exists z\ Loves(z,x)].$$

- **Skolemize** : **Skolemization** is the process of removing existential quantifiers by elimination. In the simple case, it is just like the Existential Instantiation rule of Section 9.1: translate $\exists x\ P(x)$ into $P(A)$, where A is a new constant. However, we can't apply Existential Instantiation to our sentence above because it doesn't match the pattern $\exists v\ \alpha$; only parts of the sentence match the pattern. If we blindly apply the rule to the two matching parts we get

  $$\forall x\ [Animal(y) \land \neg Loves(x,A)]\ \lor\ Loves(B,x),$$

  which has the wrong meaning entirely: it says that everyone either fails to love a particular animal A or is loved by some particular entity B. In fact, our original sentence allows each person to fail to love a different animal or to be loved by a different person. Thus, we want the Skolem entities to depend on x:

  $$\forall x\ [Animal(F(x)) \land \neg Loves(x,F(x))]\ \lor\ Loves(G(x),x).$$

  Here F and G are **Skolem functions**. The general rule is that the arguments of the Skolem function are all the universally quantified variables in whose scope the existential quantifier appears. As with Existential Instantiation, the Skolemized sentence is satisfiable exactly when the original sentence is satisfiable.

- **Drop universal quantifiers** : At this point, all remaining variables must be universally quantified. Therefore, we don't lose any information if we drop the quantifier:

  $$[Animal(F(x)) \land \neg Loves(x,F(x))]\ \lor\ Loves(G(x),x).$$

- **Distribute $\lor$ over $\land$**:

  $$[Animal(F(x)) \lor Loves(G(x),x)]\ \land\ [\neg Loves(x,F(x))\ \lor\ Loves(G(x),x)]\ .$$

  This step may also require flattening out nested conjunctions and disjunctions.

The sentence is now in CNF and consists of two clauses. It is much more difficult to read than the original sentence with implications. (It may help to explain that the Skolem function $F(x)$ refers to the animal potentially unloved by x, whereas $G(x)$ refers to someone who might love x.) Fortunately, humans seldom need to look at CNF sentences—the translation process is easily automated.

9.5.2 The resolution inference rule

The resolution rule for first-order clauses is simply a lifted version of the propositional resolution rule given on page 244. Two clauses, which are assumed to be standardized apart so that they share no variables, can be resolved if they contain complementary literals. Propositional literals are complementary if one is the negation of the other; first-order literals are complementary if one *unifies with* the negation of the other. Thus, we have

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

where $\text{UNIFY}\ (\ell_i, \neg m_j) = \theta$. For example, we can resolve the two clauses

$$[Animal(F(x)) \vee Loves(G(x), x)] \quad \text{and} \quad [\neg Loves(u, v) \vee \neg Kills(u, v)].$$

by eliminating the complementary literals $Loves(G(x),x)$ and $\neg Loves(u,v)$, with the unifier $\theta = \{u/G(x), v/x\}$, to produce the **resolvent** clause

$$[Animal(F(x)) \vee \neg Kills(G(x), x)]\ .$$

This rule is called the **binary resolution** rule because it resolves exactly two literals. The binary resolution rule by itself does not yield a complete inference procedure. The full resolution rule resolves subsets of literals in each clause that are unifiable. An alternative approach is to extend **factoring**—the removal of redundant literals—to the first-order case. Propositional factoring reduces two literals to one if they are *identical;* first-order factoring reduces two literals to one if they are *unifiable.* The unifier must be applied to the entire clause. The combination of binary resolution and factoring is complete.

9.5.3 Example proofs

Resolution proves that $KB \models a$ by proving that $KB \wedge \neg\alpha$ unsatisfiable—that is, by deriving the empty clause. The algorithmic approach is identical to the propositional case, described in Figure 7.13, so we need not repeat it here. Instead, we give two example proofs. The first is the crime example from Section 9.3. The sentences in CNF are

$$\neg American(x) \vee \neg Weapon(y) \vee \neg Sells(x, y,\ z) \vee \neg Hostile(z) \vee Criminal(x)$$
$$\neg Missile(x) \vee \neg Owns(Nono, x) \vee Sells(West, x, Nono)$$
$$\neg Enemy(x, America) \vee Hostile(x)$$
$$\neg Missile(x) \vee Weapon(x)$$
$$Owns(Nono, M_1) \qquad Missile(M_1)$$
$$American(West) \qquad Enemy(Nono, America).$$

We also include the negated goal ¬*Criminal*(*West*). The resolution proof is shown in Figure 9.10. Notice the structure: single "spine" beginning with the goal clause, resolving against clauses from the knowledge base until the empty clause is generated. This is characteristic of resolution on Horn clause knowledge bases. In fact, the clauses along the main spine correspond *exactly* to the consecutive values of the *goals* variable in the backward-chaining algorithm of Figure 9.6. This is because we always choose to resolve with a clause whose positive literal unifies with the leftmost literal of the "current" clause on the spine; this is exactly what happens in backward chaining. Thus, backward chaining is just a special case of resolution with a particular control strategy to decide which resolution to perform next.

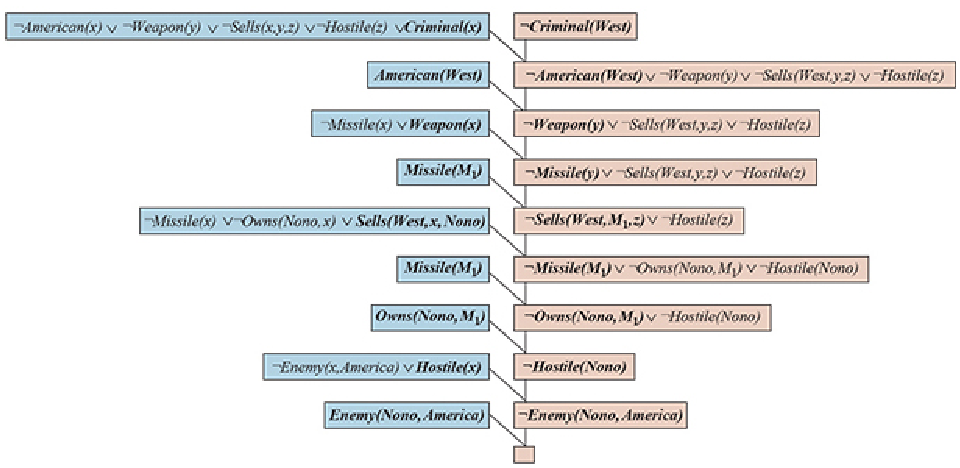


Figure 9.10 A resolution proof that West is a criminal. At each resolution step, the literals that unify are in bold and the clause with the positive literal is shaded blue.

Our second example makes use of Skolemization and involves clauses that are not definite clauses. This results in a somewhat more complex proof structure. In English:

Everyone who loves all animals is loved by someone.

Anyone who kills an animal is loved by no one.

Jack loves all animals.

Either Jack or Curiosity killed the cat, who is named Tuna.

Did Curiosity kill the cat?

First, we express the original sentences, some background knowledge, and the negated goal G in first-order logic:

A. $\forall x\, [\forall y\, Animal(y) \Rightarrow Loves(x, y)] \Rightarrow [\exists y\, Loves(y, x)]$

B. $\forall x\, [\exists z\, Animal(z) \wedge Kills(x,\ z)] \Rightarrow [\forall y\, \neg Loves(y, x)]$

C. $\forall x\, Animal(x)\ \Rightarrow\ Loves(Jack, x)$

D. $Kills(Jack, Tuna) \vee Kills(Curiosity, Tuna)$

E. $Cat(Tuna)$

F. $\forall x\, Cat(x) \Rightarrow Animal(x)$

¬G. $\neg Kills(Curiosity, Tuna)$

Now we apply the conversion procedure to convert each sentence to CNF:

A1. $Animal(F(x)) \vee Loves(G(x), x)$

A2. $\neg Loves(x, F(x)) \vee Loves(G(x), x)$

B. $\neg Loves(y, x) \vee \neg Animal(z) \vee \neg Kills(x,\ z)$

C. $\neg Animal(x) \vee Loves(Jack, x)$

D. $Kills(Jack, Tuna) \vee Kills(Curiosity, Tuna)$

E. $Cat(Tuna)$

F. $\neg Cat(x) \vee Animal(x)$

¬G. $\neg Kills(Curiosity, Tuna)$

The resolution proof that Curiosity killed the cat is given in Figure 9.11. In English, the proof could be paraphrased as follows:

Suppose Curiosity did not kill Tuna. We know that either Jack or Curiosity did; thus Jack must have. Now, Tuna is a cat and cats are animals, so Tuna is an animal. Because anyone who kills an animal is loved by no one, we know that no one loves Jack. On the other hand, Jack loves all animals, so someone loves him; so we have a contradiction. Therefore, Curiosity killed the cat.

The proof answers the question “Did Curiosity kill the cat?” but often we want to pose more general questions, such as “Who killed the cat?” Resolution can do this, but it takes a little more work to obtain the answer. The goal is ∃*w* *Kills*(*w*, *Tuna*), which, when negated, becomes ¬*Kills*(*w*, *Tuna*) in CNF. Repeating the proof in Figure 9.11 with the new negated goal, we obtain a similar proof tree, but with the substitution {*w/Curiosity*} in one of the steps. So, in this case, finding out who killed the cat is just a matter of keeping track of the bindings for the query variables in the proof. Unfortunately, resolution can sometimes produce **nonconstructive proofs** for existential goals, where we know a query is true, but there isn't a unique binding for the variable.

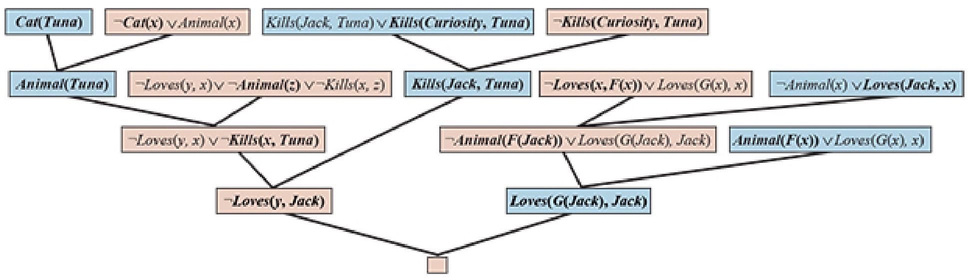


Figure 9.11 A resolution proof that Curiosity killed the cat. Notice the use of factoring in the derivation of the clause *Loves*(*G*(*Jack*), *Jack*). Notice also in the upper right, the unification of *Loves*(*x*,*F*(*x*)) and *Loves*(*Jack*,*x*) can only succeed after the variables have been standardized apart.

9.5.4 Completeness of resolution

This section gives a completeness proof of resolution. It can be safely skipped by those who are willing to take it on faith.

We show that resolution is **refutation-complete**, which means that *if* a set of sentences is unsatisfiable, then resolution will always be able to derive a contradiction. Resolution cannot be used to generate all logical consequences of a set of sentences, but it can be used to establish that a given sentence is entailed by the set of sentences. Hence, it can be used to find all answers to a given question, $Q(x)$, by proving that $KB \wedge \neg Q(x)$ is unsatisfiable.

We take it as given that any sentence in first-order logic (without equality) can be rewritten as a set of clauses in CNF. This can be proved by induction on the form of the sentence, using atomic sentences as the base case (Davis and Putnam, 1960). Our goal therefore is to prove the following: *if S is an unsatisfiable set of clauses, then the application of a finite number of resolution steps to S will yield a contradiction.*

Our proof sketch follows Robinson's original proof with some simplifications from Genesereth and Nilsson (1987). The basic structure of the proof (Figure 9.12) is as follows:

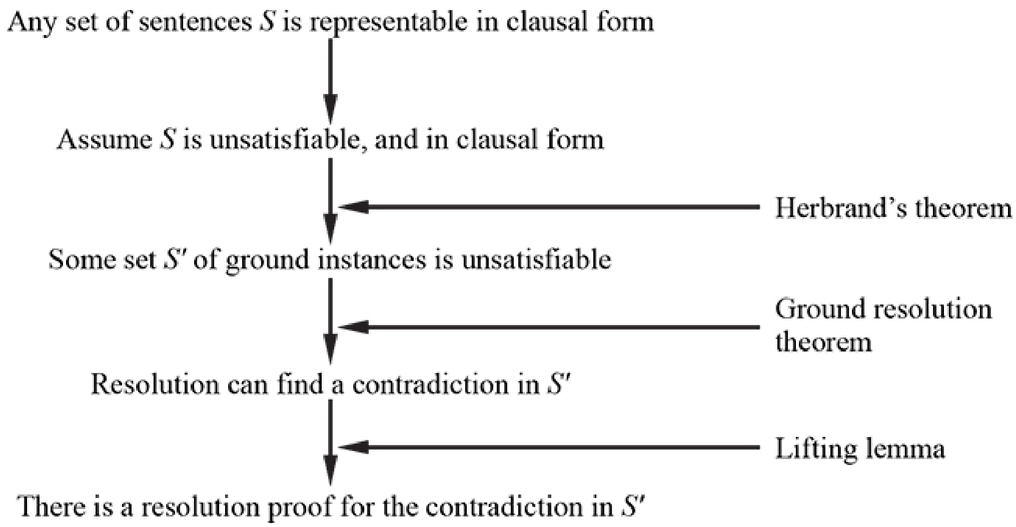


Figure 9.12 Structure of a completeness proof for resolution.

1. First, we observe that if S is unsatisfiable, then there exists a particular set of *ground instances* of the clauses of S such that this set is also unsatisfiable (Herbrand's theorem).
2. We then appeal to the **ground resolution theorem** given in Chapter 7, which states that propositional resolution is complete for ground sentences.
3. We then use a **lifting lemma** to show that, for any propositional resolution proof using the set of ground sentences, there is a corresponding first-order resolution proof using the first-order sentences from which the ground sentences were obtained.

To carry out the first step, we need three new concepts:

- **Herbrand universe**: If S is a set of clauses, then HS, the Herbrand universe of S, is the set of all ground terms constructible from the following:

  a. The function symbols in S, if any.
  b. The constant symbols in S, if any; if none, then a default constant symbol, S.

  For example, if S contains just the clause $\neg P(x,F(x,A)) \vee \neg Q(x,A) \vee R(x,B)$, then H_S is the following infinite set of ground terms:

  $$\{A, B, F(A, A), F(A, B), F(B, A), F(B, B), F(A, F(A, A)), \ldots\}.$$

- **Saturation**: If S is a set of clauses and P is a set of ground terms, then $P(S)$, the saturation of S with respect to P, is the set of all ground clauses obtained by applying all possible consistent substitutions of ground terms in P for variables in S.

- **Herbrand base**: The saturation of a set S of clauses with respect to its Herbrand universe is called the Herbrand base of S, written as $H_S(S)$. For example, if S contains solely the clause given above, then $H_S(S)$ is the infinite set of clauses

$$\begin{aligned}&\{\neg P(A, F(A, A)) \lor \neg Q(A, A) \lor R(A, B),\\ &\ \neg P(B, F(B, A)) \lor \neg Q(B, A) \lor R(B, B),\\ &\ \neg P(F(A, A), F(A, A), A)) \lor \neg Q(F(A, A), A) \lor R(F(A, A), B),\\ &\ \neg P(F(A, B), F(F(A, B), A)) \lor \neg Q(F(A, B), A) \lor R(F(A, B), B), \ldots\}\end{aligned}$$

These definitions allow us to state a form of **Herbrand's theorem** (Herbrand, 1930):

> If a set S of clauses is unsatisfiable, then there exists a finite subset of $H_S(S)$ that is also unsatisfiable.

Let S' be this finite subset of ground sentences. Now, we can appeal to the ground resolution theorem (page 246) to show that the **resolution closure** $RC(S')$ contains the empty clause. That is, running propositional resolution to completion on S' will derive a contradiction.

Now that we have established that there is always a resolution proof involving some finite subset of the Herbrand base of S, the next step is to show that there is a resolution proof using the clauses of S itself, which are not necessarily ground clauses. We start by considering a single application of the resolution rule. Robinson stated this lemma:

> Let C_1 and C_2 be two clauses with no shared variables, and let C'_1 *a* and C'_2 be ground instances of C_1 and C_2. If C' is a resolvent of C'_1 and C'_2, then there exists a clause C such that (1) C is a resolvent of C_1 and C_2 and (2) C' is a ground instance of C.

Gödel's Incompleteness Theorem

By slightly extending the language of first-order logic to allow for the **mathematical induction schema** in arithmetic, Kurt Godel was able to show, in his **incompleteness theorem**, that there are true arithmetic sentences that cannot be proved.

The proof of the incompleteness theorem is somewhat beyond the scope of this book, occupying, as it does, at least 30 pages, but we can give a hint here. We begin with the logical theory of numbers. In this theory, there is a single constant, 0, and a single function, S (the successor function). In the intended model, $S(0)$ denotes 1, $S(S(0))$ denotes 2, and so on; the language therefore has names for all the natural numbers. The vocabulary also includes the function symbols $+$, $\times$, and *Expt* (exponentiation) and the usual set of logical connectives and quantifiers.

The first step is to notice that the set of sentences that we can write in this language can be enumerated. (Imagine defining an alphabetical order on the symbols and then arranging, in alphabetical order, each of the sets of sentences of length 1, 2, and so on.) We can then number each sentence α with a unique natural number $\#\alpha$ (the **Gödel number**). This is crucial: number theory contains a name for each of its own sentences. Similarly, we can number each possible proof P with a Gödel number $G(P)$, because a proof is simply a finite sequence of sentences.

Now suppose we have a recursively enumerable set A of sentences that are true statements about the natural numbers. Recalling that A can be named by a given set of integers, we can imagine writing in our language a sentence $\alpha(j, A)$ of the following sort:

> $\forall\ i\ i$ is not the Godel number of a proof of the sentence whose Gödel number is j, where the proof uses only premises in A.

Then let σ be the sentence $\alpha(\#\sigma, A)$, that is, a sentence that states its own unprovability from A. (That this sentence always exists is true but not entirely obvious.)

Now we make the following ingenious argument: Suppose that σ *is* provable from A; then σ is false (because σ says it cannot be proved). But then we have a false sentence that is provable from A, so A cannot consist of only true sentences— a violation of our premise. Therefore, σ is *not* provable from A. But this is exactly what σ itself claims; hence σ is a true sentence.

So, we have shown (barring $29\frac{1}{2}$ pages) that for any set of true sentences of number theory, and in particular any set of basic axioms, there are other true sentences that *cannot* be proved from those axioms. This establishes, among other things, that we can never prove all the theorems of mathematics *within any given system of axioms.* Clearly, this was an important discovery for mathematics. Its significance for AI has been widely debated, beginning with speculations by Godel himself. We take up the debate in Chapter 28.

This is called a **lifting lemma**, because it lifts a proof step from ground clauses up to general first-order clauses. In order to prove his basic lifting lemma, Robinson had to invent unification and derive all of the properties of most general unifiers. Rather than repeat the proof here, we simply illustrate the lemma:

$$\begin{aligned}
C_1 &= \neg P(x, F(x, A)) \vee \neg Q(x, A) \vee R(x, B) \\
C_2 &= \neg N(G(y),\ z) \vee P(H(y),\ z) \\
C'_1 &= \neg P(H(B), F(H(B), A)) \vee \neg Q(H(B), A) \vee R(H(B), B) \\
C'_2 &= \neg N(G(B), F(H(B), A)) \vee P(H(B), F(H(B), A)) \\
C' &= \neg N(G(B), F(H(B), A)) \vee \neg Q(H(B), A) \vee R(H(B), B) \\
C &= \neg N(G(y), F(H(y), A)) \vee \neg Q(H(y), A) \vee R(H(y), B).
\end{aligned}$$

We see that indeed C' is a ground instance of C. In general, for C_1 and C_2 to have any resolvents, they must be constructed by first applying to C_1 and C_2 the most general unifier of a pair of complementary literals in C_1 and C_2. From the lifting lemma, it is easy to derive a similar statement about any sequence of applications of the resolution rule:

> For any clause C' in the resolution closure of S' there is a clause C in the resolution closure of S such that C' is a ground instance of C and the derivation of C is the same length as the derivation of C'.

From this fact, it follows that if the empty clause appears in the resolution closure of S', it must also appear in the resolution closure of S. This is because the empty clause cannot be a ground instance of any other clause. To recap: we have shown that if S is unsatisfiable, then there is a finite derivation of the empty clause using the resolution rule.

The lifting of theorem proving from ground clauses to first-order clauses provides a vast increase in power. This increase comes from the fact that the first-order proof need instantiate variables only as far as necessary for the proof, whereas the ground-clause methods were required to examine a huge number of arbitrary instantiations.

9.5.5 Equality

None of the inference methods described so far in this chapter can handle an assertion of the form $x = y$ without some additional work. Three distinct approaches can be taken. The first is to axiomatize equality—to write down sentences about the equality relation in the knowledge base. We need to say that equality is reflexive, symmetric, and transitive, and we also have to say that we can substitute equals for equals in any predicate or function. So we need three basic axioms, and then one for each predicate and function:

$$\forall x\; x = x$$

$$\forall x, y\; x = y \;\Rightarrow\; y = x$$

$$\forall x, y, z\; x = y \land y = z \Rightarrow x = z$$

$$\forall x, y\; x = y \;\Rightarrow\; (P_1(x) \;\Leftrightarrow\; P_1(y))$$

$$\forall x, y\; x = y \;\Rightarrow\; (P_2(x) \;\Leftrightarrow\; P_2(y))$$

$$\vdots$$

$$\forall w, x, y, z\; w = y \land x = z \Rightarrow (F_1(w, x) = F_1(y, z))$$

$$\forall w, x, y, z\; w = y \land x = z \Rightarrow (F_2(w, x) = F_2(y, z))$$

$$\vdots$$

Given these sentences, a standard inference procedure such as resolution can perform tasks requiring equality reasoning, such as solving mathematical equations. However, these axioms will generate a lot of conclusions, most of them not helpful to a proof. So the second approach is to add inference rules rather than axioms. The simplest rule, **demodulation**, takes a unit clause $x = y$ and some clause α that contains the term x, and yields a new clause formed by substituting y for x within α. It works if the term within α unifies with x; it need not be exactly equal to x. Note that demodulation is directional; given $x = y$, the x always gets replaced with y, never vice versa. That means that demodulation can be used for simplifying expressions using demodulators such as $z + 0 = z$ or $z^1 = z$. As another example, given

$$Father(Father(x)) = PaternalGrandfather(x)$$
$$Birthdate(Father(Father(Bella)), 1926)$$

we can conclude by demodulation

$$Birthdate(PaternalGrandfather(Bella), 1926)\ .$$

More formally, we have

- **Demodulation**: For any terms x, y, and z, where z appears somewhere in literal m, and where UNIFY$(x, z) = \theta \neq failure$,

  $$\frac{x=y, \qquad m_1 \vee \cdots \vee m_n}{\text{SUB}(\text{SUBST}(\theta,x), \text{SUBST}(\theta,y), m_1 \vee \cdots \vee m_n)}\ .$$

  where SUBST is the usual substitution of a binding list, and SUB(x,y,m) means to replace x with y somewhere within m.

The rule can also be extended to handle non-unit clauses in which an equal sign appears:

- **Paramodulation**: For any terms x, y, and z, where z appears somewhere in literal m, and where UNIFY$(x,z) = \theta$,

  $$\frac{\ell_1 \vee \cdots \vee \ell_k \vee x=y, \qquad m_1 \vee \cdots \vee m_n}{\text{SUB}(\text{SUBST}(\theta,x), \text{SUBST}\,(\theta,y), \text{SUBST}\,(\theta, \ell_k \vee m_1 \vee \cdots \vee m_n))}\ .$$

For example, from

$$P(F(x,B),x) \vee Q(x) \qquad \text{and} \qquad F(A,y) = y \vee R(y)$$

we have θ = UNIFY$(F(A,y),F(x,B))$ = $\{x/A,y/B\}$, and we can conclude by paramodulation the sentence

$$P(B,A) \vee Q(A) \vee R(B).$$

Paramodulation yields a complete inference procedure for first-order logic with equality.

A third approach handles equality reasoning entirely within an extended unification algorithm. That is, terms are unifiable if they are *provably* equal under some substitution, where “provably” allows for equality reasoning. For example, the terms 1 + 2 and 2 +1 normally are not unifiable, but a unification algorithm that knows that $x + y = y + x$ could unify them with the empty substitution. **Equational unification** of this kind can be done with efficient algorithms designed for the particular axioms used (commutativity, associativity, and so on) rather than through explicit inference with those axioms. Theorem provers using this technique are closely related to the CLP systems described in Section 9.4.

9.5.6 Resolution strategies

We know that repeated applications of the resolution inference rule will eventually find a proof if one exists. In this subsection, we examine strategies that help find proofs *efficiently.*

Unit preference: This strategy prefers to do resolutions where one of the sentences is a single literal (also known as a **unit clause**). The idea behind the strategy is that we are trying to produce an empty clause, so it might be a good idea to prefer inferences that produce shorter clauses. Resolving a unit sentence (such as P) with any other sentence (such as $\neg P \vee \neg Q \vee R$) always yields a clause (in this case, $\neg Q \vee R$) that is shorter than the other clause. When the unit preference strategy was first tried for propositional inference in 1964, it led to a dramatic speedup, making it feasible to prove theorems that could not be handled without the preference. **Unit resolution** is a restricted form of resolution in which every resolution step must involve a unit clause. Unit resolution is incomplete in general, but complete for Horn clauses. Unit resolution proofs on Horn clauses resemble forward chaining.

The Otter theorem prover (McCune, 1990), uses a form of best-first search. Its heuristic function measures the “weight” of each clause, where lighter clauses are preferred. The exact choice of heuristic is up to the user, but generally, the weight of a clause should be correlated with its size or difficulty. Unit clauses are treated as light; the search can thus be seen as a generalization of the unit preference strategy.

Set of support: Preferences that try certain resolutions first are helpful, but in general it is more effective to try to eliminate some potential resolutions altogether. For example, we can insist that every resolution step involve at least one element of a special set of clauses—the *set of support.* The resolvent is then added into the set of support. If the set of support is small relative to the whole knowledge base, the search space will be reduced dramatically.

To ensure completeness of this strategy, we can choose the set of support S so that the remainder of the sentences are jointly satisfiable. For example, one can use the negated query as the set of support, on the assumption that the original knowledge base is consistent. (After all, if it is not consistent, then the fact that the query follows from it is vacuous.) The set-of- support strategy has the additional advantage of generating goal-directed proof trees that are often easy for humans to understand.

Input resolution: In this strategy, every resolution combines one of the input sentences (from the KB or the query) with some other sentence. The proof in Figure 9.10 on page 319 uses only input resolutions and has the characteristic shape of a single "spine" with single sentences combining onto the spine. Clearly, the space of proof trees of this shape is smaller than the space of all proof graphs. In Horn knowledge bases, Modus Ponens is a kind of input resolution strategy, because it combines an implication from the original KB with some other sentences. Thus, it is no surprise that input resolution is complete for knowledge bases that are in Horn form, but incomplete in the general case. The **linear resolution** strategy is a slight generalization that allows P and Q to be resolved together either if P is in the original KB or if P is an ancestor of Q in the proof tree. Linear resolution is complete.

Subsumption: The subsumption method eliminates all sentences that are subsumed by (that is, more specific than) an existing sentence in the KB. For example, if $P(x)$ is in the KB, then there is no sense in adding $P(A)$ and even less sense in adding $P(A) \vee Q(B)$. Subsumption helps keep the KB small and thus helps keep the search space small.

Learning: We can improve a theorem prover by learning from experience. Given a collection of previously-proved theorems, train a machine learning system to answer the question: given a set of premises and a goal to prove, what proof steps are similar to steps that were successful in the past? The DEEPHOL system (Bansal *et al.*, 2019) does exactly that, using deep neural networks (see Chapter 22) to build models (called *embeddings*) of goals and premises, and using them to make selections. Training can use both human- and computer-generated proofs as examples, starting from a collection of 10,000 proofs.

Practical uses of resolution theorem provers

We have shown how first-order logic can represent a simple real-world scenario involving concepts like selling, weapons, and citizenship. But complex real-world scenarios have too much uncertainty and too many unknowns. Logic has proven to be more successful for scenarios involving formal, strictly defined concepts, such as the **synthesis** and **verification** of both hardware and software. Theorem-proving research is carried out in the fields of hardware design, programming languages, and software engineering—not just in AI.

In the case of hardware, the axioms describe the interactions between signals and circuit elements. (See Section 8.4.2 on page 291 for an example.) Logical reasoners designed specially for verification have been able to verify entire CPUs, including their timing properties (Srivas and Bickford, 1990). The AURA theorem prover has been applied to design circuits that are more compact than any previous design (Wojciechowski and Wojcik, 1983).

In the case of software, reasoning about programs is quite similar to reasoning about actions, as in Chapter 7: axioms describe the preconditions and effects of each statement. The formal synthesis of algorithms was one of the first uses of theorem provers, as outlined by Cordell Green (1969a), who built on earlier ideas by Herbert Simon (1963). The idea is to constructively prove a theorem to the effect that “there exists a program *p* satisfying a certain specification.” Although fully automated **deductive synthesis**, as it is called, has not yet become feasible for general-purpose programming, hand-guided deductive synthesis has been successful in designing several novel and sophisticated algorithms. Synthesis of specialpurpose programs, such as scientific computing code, is also an active area of research.

Similar techniques are now being applied to software verification by systems such as the SPIN model checker (Holzmann, 1997). For example, the Remote Agent spacecraft control program was verified before and after flight (Havelund *et al.*, 2000). The RSA public key encryption algorithm and the Boyer-Moore string-matching algorithm have been verified this way (Boyer and Moore, 1984).

Summary

We have presented an analysis of logical inference in first-order logic and a number of algorithms for doing it.

- A first approach uses inference rules (**universal instantiation** and **existential instantiation**) to **propositionalize** the inference problem. Typically, this approach is slow, unless the domain is small.
- The use of **unification** to identify appropriate substitutions for variables eliminates the instantiation step in first-order proofs, making the process more efficient in many cases.
- A lifted version of **Modus Ponens** uses unification to provide a natural and powerful inference rule, **generalized Modus Ponens**. The **forward-chaining** and **backwardchaining** algorithms apply this rule to sets of definite clauses.
- Generalized Modus Ponens is complete for definite clauses, although the entailment problem is **semidecidable**. For **Datalog** knowledge bases consisting of function-free definite clauses, entailment is decidable.
- Forward chaining is used in **deductive databases**, where it can be combined with relational database operations. It is also used in **production systems**, which perform efficient updates with very large rule sets. Forward chaining is complete for Datalog and runs in polynomial time.
- Backward chaining is used in **logic programming systems**, which employ sophisticated compiler technology to provide very fast inference. Backward chaining suffers from redundant inferences and infinite loops; these can be alleviated by **memoization**.

- **Prolog** , unlike first-order logic, uses a closed world with the unique names assumption and negation as failure. These make Prolog a more practical programming language, but bring it further from pure logic.
- The generalized **resolution** inference rule provides a complete proof system for firstorder logic, using knowledge bases in conjunctive normal form.
- Several strategies exist for reducing the search space of a resolution system without compromising completeness. One of the most important issues is dealing with equality; we showed how **demodulation** and **paramodulation** can be used.
- Efficient resolution-based theorem provers have been used to prove interesting mathematical theorems and to verify and synthesize software and hardware.

Bibliographical and Historical Notes

Gottlob Frege, who developed full first-order logic in 1879, based his system of inference on a collection of valid schemas plus a single inference rule, Modus Ponens. Whitehead and Russell (1910) expounded the so-called *rules of passage* (the actual term is from Herbrand (1930)) that are used to move quantifiers to the front of formulas. Skolem constants and Skolem functions were introduced, appropriately enough, by Thoralf Skolem (1920). Oddly enough, it was Skolem who introduced the Herbrand universe (Skolem, 1928).

Herbrand's theorem (Herbrand, 1930) has played a vital role in the development of automated reasoning. Herbrand is also the inventor of **unification**. Gödel (1930) built on the ideas of Skolem and Herbrand to show that first-order logic has a complete proof procedure. Alan Turing (1936) and Alonzo Church (1936) simultaneously showed, using very different proofs, that validity in first-order logic was not decidable. The excellent text by Enderton (1972) explains all of these results in a rigorous yet understandable fashion.

Abraham Robinson proposed that an automated reasoner could be built using proposition- alization and Herbrand's theorem, and Paul Gilmore (1960) wrote the first program. Davis and Putnam (1960) introduced the propositionalization method of Section 9.1. Prawitz (1960) developed the key idea of letting the quest for propositional inconsistency drive the search, and generating terms from the Herbrand universe only when they were necessary to establish propositional inconsistency. This idea led John Alan Robinson (no relation) to develop resolution (Robinson, 1965).

Resolution was adopted for question-answering systems by Cordell Green and Bertram Raphael (1968). Early AI implementations put a good deal of effort into data structures that would allow efficient retrieval of facts; this work is covered in AI programming texts (Char- niak *et al.*, 1987; Norvig, 1992; Forbus and de Kleer, 1993). By the early 1970s, **forward chaining** was well established in AI as an easily understandable alternative to resolution. AI applications typically involved large numbers of rules, so it was important to develop efficient rule-matching technology, particularly for incremental updates.

The technology for **production systems** was developed to support such applications. The production system language OPS-5 (Forgy, 1981; Brownston **et** *al.*, 1985), incorporating the efficient Rete match process (Forgy, 1982), was used for applications such as the R 1 expert system for minicomputer configuration (McDermott, 1982). Kraska **et** *al.* (2017) describe how neural nets can learn an efficient indexing scheme for specific data sets.

The SOAR cognitive architecture (Laird **et** *al.*, 1987; Laird, 2008) was designed to handle very large rule sets—up to a million rules (Doorenbos, 1994). Example applications of SOAR include controlling simulated fighter aircraft (Jones **et** *al.*, 1998), airspace management (Taylor **et** *al.*, 2007), AI characters for computer games (Wintermute **et** *al.*, 2007), and training tools for soldiers (Wray and Jones, 2005).

The field of **deductive databases** began with a workshop in Toulouse in 1977 attended by experts in logical inference and databases (Gallaire and Minker, 1978). Influential work by Chandra and Harel (1980) and Ullman (1985) led to the adoption of **Datalog** as a standard language for deductive databases. The development of the **magic sets** technique for rule rewriting by Bancilhon **et** *al.* (1986) allowed forward chaining to borrow the advantage of goal-directedness from backward chaining.

The rise of the Internet led to increased availability of massive online databases. This drove increased interest in integrating multiple databases into a consistent dataspace (Halevy, 2007). Kraska *et al.* (2017) showed speedups of up to 70% by using machine learning to create **learned index structures** for efficient data lookup.

Backward chaining for logical inference originated in the PLANNER language (Hewitt, 1969). Meanwhile, in 1972, Alain Colmerauer had developed and implemented **Prolog** for the purpose of parsing natural language—Prolog's clauses were intended initially as context- free grammar rules (Roussel, 1975; Colmerauer *et al.*, 1973).

Much of the theoretical background for logic programming was developed by Robert Kowalski at Imperial College London, working with Colmerauer; see Kowalski (1988) and Colmerauer and Roussel (1993) for a historical overview. Efficient Prolog compilers are generally based on the Warren Abstract Machine (WAM) model of computation developed by David H. D. Warren (1983). Van Roy (1990) showed that Prolog programs can be competitive with C programs in terms of speed.

Methods for avoiding unnecessary looping in recursive logic programs were developed independently by Smith *et al.* (1986) and Tamaki and Sato (1986). The latter paper also included memoization for logic programs, a method developed extensively as **tabled logic programming** by David S. Warren. Swift and Warren (1994) show how to extend the WAM to handle tabling, enabling Datalog programs to execute an order of magnitude faster than forward-chaining deductive database systems.

Early work on constraint logic programming was done by Jaffar and Lassez (1987). Jaffar *et al.* (1992) developed the CLP(R) system for handling real-valued constraints. There are now commercial products for solving large-scale configuration and optimization problems with constraint programming; one of the best known is ILOG (Junker, 2003). Answer set programming (Gelfond, 2008) extends Prolog, allowing disjunction and negation.

Texts on logic programming and Prolog include Shoham (1994), Bratko (2009), Clocksin (2003), and Clocksin and Mellish (2003). Prior to 2000, the *Journal of Logic Programming* was the journal of record; it has been replaced by *Theory and Practice of Logic Programming.* Logic programming conferences include the International Conference on Logic Programming (ICLP) and the International Logic Programming Symposium (ILPS).

Research into **mathematical theorem proving** began even before the first complete firstorder systems were developed. Herbert Gelernter's Geometry Theorem Prover (Gelernter, 1959) used heuristic search methods combined with diagrams for pruning false subgoals and was able to prove some quite intricate results in Euclidean geometry. The **demodulation** and **paramodulation** rules for equality reasoning were introduced by Wos *et al.* (1967) and Wos and Robinson (1968), respectively. These rules were also developed independently in the context of term-rewriting systems (Knuth and Bendix, 1970). The incorporation of equality reasoning into the unification algorithm is due to Gordon Plotkin (1972). Jouannaud and Kirchner (1991) survey equational unification from a term-rewriting perspective. An overview of unification is given by Baader and Snyder (2001).

A number of control strategies have been proposed for resolution, beginning with the unit preference strategy (Wos *et al.*, 1964). The set-of-support strategy was proposed by Wos *et al.* (1965) to provide a degree of goal-directedness in resolution. Linear resolution first appeared in Loveland (1970). Genesereth and Nilsson (1987, Chapter 5) provide an analysis of a wide variety of control strategies. Alemi *et al.* (2017) show how the DEEPMATH system uses deep neural nets to select the axioms that are most likely to lead to a proof when handed to a traditional theorem prover. In a sense, the neural net plays the role of the mathematician's intuition, and the theorem prover plays the role of the mathematician's technical expertise. (Loos *et al.*, 2017) show that this approach can be extended to help guide the search, allowing more theorems to be proved.

A Computational Logic (Boyer and Moore, 1979) is the basic reference on the Boyer- Moore theorem prover. Stickel (1992) describes the Prolog Technology Theorem Prover (PTTP), which combines Prolog compilation and model elimination. SETHEO (Letz *et al.*,1992) is another widely used theorem prover based on this approach. LEANTAP (Beckert and Posegga, 1995) is an efficient theorem prover implemented in only 25 lines of Prolog. Weidenbach (2001) describes SPASS, one of the strongest current theorem provers. The most successful theorem prover in recent annual competitions has been VAMPIRE (Riazanov and Voronkov, 2002). The COQ system (Bertot *et al.*, 2004) and the E equational solver (Schulz, 2004) have also proven to be valuable tools for proving correctness.

Theorem provers have been used to automatically synthesize and verify software. Examples include the control software for NASA's Orion capsule (Lowry, 2008) and other spacecraft (Denney *et al.*, 2006). The design of the FM9001 32-bit microprocessor was proved correct by the NQTHM theorem proving system (Hunt and Brock, 1992).

The Conference on Automated Deduction (CADE) runs an annual contest for automated theorem provers. Sutcliffe (2016) describes the 2016 competition; top-scoring systems include VAMPIRE (Riazanov and

Voronkov, 2002), PROVER9 (Sabri, 2015), and an updated version of E (Sehulz, 2013). Wiedijk (2003) compares the strength of 15 mathematical provers. TPTP (Thousands of Problems for Theorem Provers) is a library of theorem-proving problems, useful for comparing the performance of systems (Sutcliffe and Suttner, 1998; Sutcliffe *et al.*, 2006).

Theorem provers have come up with novel mathematical results that eluded human mathematicians for decades, as detailed in the book *Automated Reasoning and the Discovery of Missing Elegant Proofs* (Wos and Pieper, 2003). The SAM (Semi-Automated Mathematics) program was the first, proving a lemma in lattice theory (Guard *et al.*, 1969). The AURA program has also answered open questions in several areas of mathematics (Wos and Winker, 1983). The Boyer-Moore theorem prover (Boyer and Moore, 1979) was used by Natarajan Shankar to construct a formal proof of Gödel's Incompleteness Theorem (Shankar, 1986). The NUPRL system proved Girard's paradox (Howe, 1987) and Higman's Lemma (Murthy and Russell, 1990).

In 1933, Herbert Robbins proposed a simple set of axioms—the **Robbins algebra—** that appeared to define Boolean algebra, but no proof could be found (despite serious work by Alfred Tarski and others) until EQP (a version of OTTER) computed a proof (McCune, 1997). Benzmuller and Paleo (2013) used a higher-order theorem prover to verify Gödel's proof of the existence of "God." The Kepler sphere-packing theorem was proved by Thomas Hales (2005) with the help of some complicated computer calculations, but the proof was not completely accepted until a formal proof was generated with the help of the HOL Light and Isabelle proof assistants (Hales *et al.*, 2017).

Many early papers in mathematical logic are collected in *From Frege to Gödel: A Source Book in Mathematical Logic* (van Heijenoort, 1967).

Textbooks geared toward automated deduction include the classic *Symbolic Logic and Mechanical Theorem Proving* (Chang and Lee, 1973), as well as more recent works by Duffy (1991), Wos *et al.* (1992), Bibel (1993), and Kaufmann *et al.* (2000). The principal journal for theorem proving is the *Journal of Automated Reasoning*; the main conferences are the annual Conference on Automated Deduction (CADE) and the International Joint Conference on Automated Reasoning (IJCAR). The *Handbook of Automated Reasoning* (Robinson and Voronkov, 2001) collects papers in the field. MacKenzie's *Mechanizing Proof* (2004) covers the history and technology of theorem proving for the popular audience.

---

[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.

[2] Generalized Modus Ponens is more general than Modus Ponens (page 241) 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.

[3] Rete is Latin for net. It rhymes with *treaty.*

[4] The word **production** in **production systems** denotes a condition-action rule.

[5] A clause can also be represented as an implication with a conjunction of atoms in the premise and a disjunction of atoms in the conclusion (Exercise 9.STAN). This is called **implicative normal form** or **Kowalski form** (especially when written with a right-to-left implication symbol (Kowalski, 1979)) and is generally much easier to read than a disjunction with many negated literals.

CHAPTER 10

KNOWLEDGE REPRESENTATION

In which we show how to represent diverse facts about the real world in a form that can be used to reason and solve problems.

The previous chapters showed how an agent with a knowledge base can make inferences that enable it to act appropriately. In this chapter we address the question of what *content* to put into such an agent's knowledge base—how to represent facts about the world. We will use first-order logic as the representation language, but later chapters will introduce different representation formalisms such as hierarchical task networks for reasoning about plans (Chapter 11), Bayesian networks for reasoning with uncertainty (Chapter 13), Markov models for reasoning over time (Chapter 16), and deep neural networks for reasoning about images, sounds, and other data (Chapter 22). But no matter what representation you use, the facts about the world still need to be handled, and this chapter gives you a feeling for the issues.

Section 10.1 introduces the idea of a general ontology, which organizes everything in the world into a hierarchy of categories. Section 10.2 covers the basic categories of objects, substances, and measures; Section 10.3

covers events; and Section 10.4 discusses knowledge about beliefs. We then return to consider the technology for reasoning with this content: Section 10.5 discusses reasoning systems designed for efficient inference with categories, and Section 10.6 discusses reasoning with default information.

10.1 Ontological Engineering

In “toy” domains, the choice of representation is not that important; many choices will work. Complex domains such as shopping on the Internet or driving a car in traffic require more general and flexible representations. This chapter shows how to create these representations, concentrating on general concepts—such as *Events*, *Time*, *Physical Objects*, and *Beliefs*—that occur in many different domains. Representing these abstract concepts is sometimes called **ontological engineering.**

We cannot hope to represent *everything* in the world, even a 1000-page textbook, but we will leave placeholders where new knowledge for any domain can fit in. For example, we will define what it means to be a physical object, and the details of different types of objects—robots, televisions, books, or whatever—can be filled in later. This is analogous to the way that designers of an object-oriented programming framework (such as the Java Swing graphical framework) define general concepts like *Window*, expecting users to use these to define more specific concepts like *SpreadsheetWindow.* The general framework of concepts is called an **upper ontology** because of the convention of drawing graphs with the general concepts at the top and the more specific concepts below them, as in Figure 10.1.

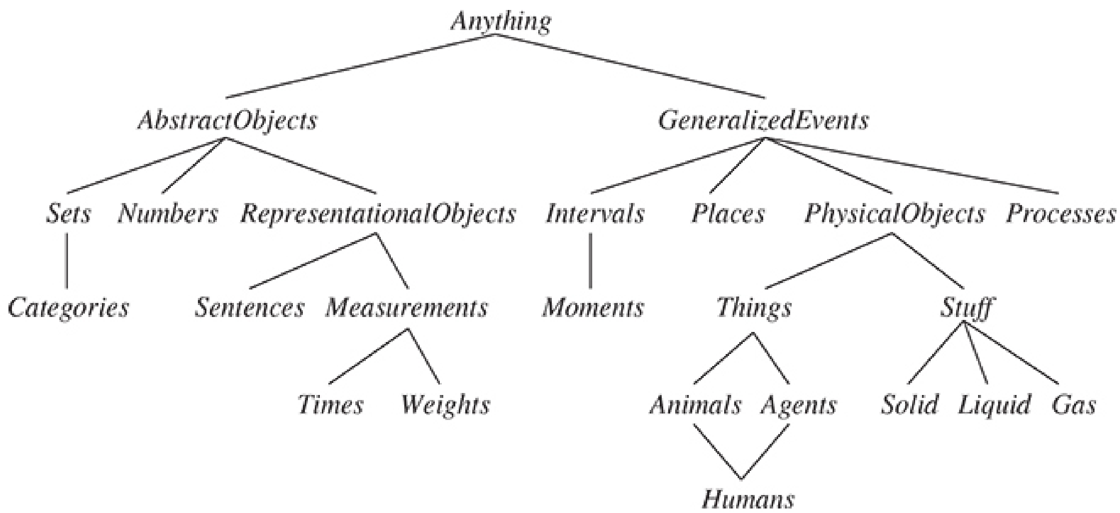


Figure 10.1 The upper ontology of the world, showing the topics to be covered later in the chapter. Each link indicates that the lower concept is a specialization of the upper one. Specializations are not necessarily disjoint—a human is both an animal and an agent. We will see in Section 10.3.2 why physical objects come under generalized events.

Before considering the ontology further, we should state one important caveat. We have elected to use first-order logic to discuss the content and organization of knowledge, although certain aspects of the real world are hard to capture in FOL. The principal difficulty is that most generalizations have exceptions or hold only to a degree. For example, although "tomatoes are red" is a useful rule, some tomatoes are green, yellow, or orange. Similar exceptions can be found to almost all the rules in this chapter. The ability to handle exceptions and uncertainty is extremely important, but is orthogonal to the task of understanding the general ontology. For this reason, we delay the discussion of exceptions until Section 10.5 of this chapter, and the more general topic of reasoning with uncertainty until Chapter 12.

Of what use is an upper ontology? Consider the ontology for circuits in Section 8.4.2. It makes many simplifying assumptions: time is omitted completely; signals are fixed and do not propagate; the structure of the circuit remains constant. A more general ontology would consider signals at particular times, and would include the wire lengths and propagation delays. This would allow us to simulate the timing properties of the circuit, and indeed such simulations are often carried out by circuit designers.

We could also introduce more interesting classes of gates, for example, by describing the technology (TTL, CMOS, and so on) as well as the input–output specification. If we wanted to discuss reliability or diagnosis, we would include the possibility that the structure of the circuit or the properties of the gates might change spontaneously. To account for stray capacitances, we would need to represent where the wires are on the board.

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

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

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

We should say up front that the enterprise of general ontological engineering has so far had only limited success. None of the top AI applications (as listed in Chapter 1) make use of a general ontology—they all use special-purpose knowledge engineering and machine learning. Social/political considerations can make it difficult for competing parties to agree on an ontology. As Tom Gruber (2004) says, “Every ontology is a treaty—a social agreement—among people with some common motive in sharing.” When competing concerns outweigh the motivation for sharing, there can be no common ontology. The smaller the number of stakeholders, the easier it is to create an ontology, and thus it is harder to create a generalpurpose ontology than a limited-purpose one, such as the Open Biomedical Ontology (Smith *et al.*, 2007). Those ontologies that do exist have been created along four routes:

1. By a team of trained ontologists or logicians, who architect the ontology and write axioms. The CYC system was mostly built this way (Lenat and Guha, 1990).
2. By importing categories, attributes, and values from an existing database or databases. DBPEDIA was built by importing structured facts from Wikipedia (Bizer *et al.*, 2007).
3. By parsing text documents and extracting information from them. TEXTRUNNER was built by reading a large corpus of Web pages (Banko and Etzioni, 2008).
4. By enticing unskilled amateurs to enter commonsense knowledge. The OPENMIND system was built by volunteers who proposed facts in English (Singh *et al.*, 2002; Chklovski and Gil, 2005).

As an example, the Google Knowledge Graph uses semistructured content from Wikipedia, combining it with other content gathered from across the web under human curation. It contains over 70 billion facts and provides answers for about a third of Google searches (Dong *et al.*, 2014).

10.2 Categories and Objects

The organization of objects into **categories** is a vital part of knowledge representation. Although interaction with the world takes place at the level of individual objects, *much reasoning takes place at the level of categories*. For example, a shopper would normally have the goal of buying a basketball, rather than a *particular* basketball such as BB_9. Categories also serve to make predictions about objects once they are classified. One infers the presence of certain objects from perceptual input, infers category membership from the perceived properties of the objects, and then uses category information to make predictions about the objects. For example, from its green and yellow mottled skin, one-foot diameter, ovoid shape, red flesh, black seeds, and presence in the fruit aisle, one can infer that an object is a watermelon; from this, one infers that it would be useful for fruit salad.

There are two choices for representing categories in first-order logic: predicates and objects. That is, we can use the predicate *Basketball*(b), or we can **reify**[1] the category as an object, *Basketballs*. We could then say *Member*(b, *Basketballs*), which we will abbreviate as $b \in$ *Basketballs*, to say that b is a member of the category of basketballs. We say *Subset*(*Basketballs*,*Balls*), abbreviated as *Basketballs* $\subset$ *Balls*, to say that *Basketballs* is a **subcategory** of *Balls*. We will use subcategory, subclass, and subset interchangeably.

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

Subclass relations organize categories into a **taxonomic hierarchy** or **taxonomy** . Taxonomies have been used explicitly for centuries in technical fields. The largest such taxonomy organizes about 10 million living and extinct species, many of them beetles,[2] into a single hierarchy; library science has developed a taxonomy of all fields of knowledge, encoded as the Dewey Decimal system; and tax authorities and other government departments have developed extensive taxonomies of occupations and commercial products.

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

- An object is a member of a category.

  $BB_9 \in Basketballs$

- A category is a subclass of another category.

  $Basketballs \subset Balls$

- All members of a category have some properties.

  $(x \in Basketballs) \Rightarrow Spherical(x)$

- Members of a category can be recognized by some properties.

  $Orange(x) \wedge Round(x) \wedge Diameter(x) = 9.5'' \wedge x \in Balls \Rightarrow x \in Basketballs$

- A category as a whole has some properties.

  $Dogs \in DomesticatedSpecies$

Notice that because *Dogs* is a category and is a member of *DomesticatedSpecies*, the latter must be a category of categories. Of course there are exceptions to many of the above rules (punctured basketballs are not spherical); we deal with these exceptions later.

Although subclass and member relations are the most important ones for categories, we also want to be able to state relations between categories that are not subclasses of each other. For example, if we just say that *Undergraduates* and *GraduateStudents* are subclasses of *Students*, then we have not said that an undergraduate cannot also be a graduate student. We say that two or more categories are **disjoint** if they have no members in common. We may also want to say that the classes undergrad and graduate student form an **exhaustive decomposition** of university students. A exhaustive decomposition of disjoint sets is known as a partition. Here are some more examples of these three concepts:

$$\begin{aligned}
&Disjoint(\{Animals, Vegetables\})\\
&ExhaustiveDecomposition(\{Americans, Canadians, Mexicans\},\\
&\qquad\qquad NorthAmericans)\\
&Partition(\{Animals, Plants, Fungi, Protista, Monera\},\\
&\qquad\qquad LivingThings)\ .
\end{aligned}$$

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

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

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

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

As we discuss in the sidebar on natural kinds on page 338, strict logical definitions for categories are usually possible only for artificial formal terms, not for ordinary objects. But definitions are not always necessary.

10.2.1 Physical composition

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

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

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

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

Therefore, we can conclude *PartOf (Bucharest, Earth)*. Categories of **composite objects** are often characterized by structural relations among parts. For example, a biped is an object

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

The notation for “exactly two” is a little awkward; we are forced to say that there are two legs, that they are not the same, and that if anyone proposes a third leg, it must be the same as one of the other two. In Section 10.5.2, we describe a formalism called description logic that makes it easier to represent constraints like "exactly two."

We can define a *PartPartition* relation analogous to the *Partition* relation for categories. (See Exercise 10.DECM.) An object is composed of the parts in its *PartPartition* and can be viewed as deriving some properties from those parts. For example, the mass of a composite object is the sum of the masses of the parts. Notice that this is not the case with categories, which have no mass, even though their elements might.

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

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

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

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

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

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

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

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

10.2.2 Measurements

In both scientific and commonsense theories of the world, objects have height, mass, cost, and so on. The values that we assign for these properties are called **measures.** Ordinary quantitative measures are quite easy to represent. We imagine that the universe includes abstract "measure objects," such as the *length* that is the length of this line segment:

. We can call this length 1.5 inches or 3.81 centimeters. Thus, the same length has different names in our language. We represent the length with a **units function** that takes a number as argument. (An alternative is explored in Exercise 10.ALTM.)

Natural Kinds

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

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

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

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

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

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

Thus, we can write down useful facts about categories without exact definitions. The difficulty of providing exact definitions for most natural categories was explained in depth by Wittgenstein (1953). He used the example of *games* to show that members of a category shared “family resemblances” rather than necessary and sufficient characteristics: what strict definition encompasses chess, tag, solitaire, and dodgeball?

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

If the line segment is called L_1,, we can write

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

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

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

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

$$Diameter(Basketball_{12}) = Inches(9.5)$$
$$Listprice(Basketball_{12}) = \$(19)$$
$$Weight(BunchOf\ (\{Apple_1, Apple_2, Apple_3\}))\ =\ Pounds(2)$$
$$d \in Days \Rightarrow Duration(d) = Hours(24)\ .$$

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

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

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

$$e_1 \in Exercises \land e_2 \in \ Exercises \land Wrote(Norvig, e_1) \land Wrote(Russell, e_2) \Rightarrow \\ \quad Difficulty(e_1) \ > \ Difficulty(e_2).$$

$$e_1 \in Exercises \land e_2 \in \ Exercises \land Difficulty(e_1) \ > Difficulty(\ e_2) \Rightarrow \\ \quad ExpectedScore(e_1) \ < \ ExpectedScore(e_2).$$

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

10.2.3 Objects: Things and stuff

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

The English language distinguishes clearly between *stuff* and *things*. We say "an aardvark," but, except in pretentious California restaurants, one cannot say "a butter." Linguists distinguish between **count nouns**, such as aardvarks, holes, and theorems, and **mass nouns**, such as butter, water, and energy. Several competing ontologies claim to handle this distinction. Here we describe just one; the others are covered in the historical notes section.

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

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

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

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

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

What is actually going on is this: some properties are **intrinsic**: they belong to the very substance of the object, rather than to the object as a whole. When you cut an instance of *stuff*in half, the two pieces retain the intrinsic properties—things like density, boiling point, flavor, color, ownership, and so on. On the other hand, their **extrinsic** properties—weight, length, shape, and so on—are not retained under subdivision. A category of objects that includes in its definition only *intrinsic* properties is then a substance, or mass noun; a class that includes *any* extrinsic properties in its definition is a count noun. *Stuff* and *Thing* are the most general substance and object categories, respectively.

10.3 Events

In Section 7.7.1 we discussed actions: things that happen, such as $Shoot_t$; and fluents: aspects of the world that change, such as $HaveArrow_t$. Both were represented as propositions, and we used successor-state axioms to say that a fluent will be true at time $t + 1$ if the action at time t caused it to be true, or if it was already true at time t and the action did not cause it to be false. That was for a world in which actions are discrete, instantaneous, happen one at a time, and have no variation in how they are performed (that is, there is only one kind of *Shoot*action, there is no distinction between shooting quickly, slowly, nervously, etc.).

But as we move from simplistic domains to the real world, there is a much richer range of actions or events[3] to deal with. Consider a continuous action, such as filling a bathtub. A successor-state axiom can say that the tub is empty before the action and full when the action is done, but it can't talk about what happens *during* the action. It also can't easily describe two actions happening at the same time—such as brushing one's teeth while waiting for the tub to fill. To handle such cases we introduce an approach known as e**vent calculus.**

The objects of event calculus are events, fluents, and time points. *At*(*Shankar, Berkeley*) is a fluent: an object that refers to the fact of Shankar being in Berkeley. The event E_1 of Shankar flying from San Francisco to Washington, D.C., is described as

$$E_1 \in Flyings \wedge Flyer(E_1, Shankar) \wedge Origin(E_1, SF) \wedge Destination(E_1, DC) \; .$$

where *Flyings* is the category of all flying events. By reifying events we make it possible to add any amount of arbitrary information about them. For example, we can say that Shankar's flight was bumpy with $Bumpy\,(E_1)$. In an ontology where events are n-ary predicates, there would be no way to add extra information like this; moving to an $n + 1$-ary predicate isn't a scalable solution.

To assert that a fluent is actually true starting at some point in time t_1 and continuing to time t_2, we use the predicate T, as in $T(At(Shankar,Berkeley),t_1,t_2$. Similarly, we use $Happens(E_1,t_1,t_2)$ to say that the event E_1 actually happened, starting at time t_1 and ending at time t_2. The complete set of predicates for one version of the event calculus[4] is:

$T(f, t_1, t_2)$	Fluent f is true for all times between t_1 and t_2
$Happens(e, t_1, t_2)$	Event e starts at time t_1 and ends at t_2
$Initiates(e, f, t)$	Event e causes fluent f to become true at time t
$Ter\min ates(e, f, t)$	Event e causes fluent f to cease to be true at time t
$Initiated(f, t_1, t_2)$	Fluent f become true at some point between t_1 and t_2
$Terminated(f, t_1, t_2)$	Fluent f cease to be true at some point between t_1 and t_2
$t_1 < t_2$	Time point t_1 occurs before time t_2

We can describe the effects of a flying event:

$$E{=}Flyings(a, here, there) \wedge Happens(E, t_1, t_2) \Rightarrow$$
$$Terminates(E, At(a, here), t_1) \wedge Initiates(E, At(a, there), t_2)$$

We assume a distinguished event, *Start*, that describes the initial state by saying which fluents are true (using *Initiates*) or false (using *Terminated*) at the start time. We can then describe what fluents are true at what points in time with a pair of axioms for T and $\neg T$ that follow the same general format as the successor-state axioms: Assume an event happens between time t_1 and t_3, and at t_2 somewhere in that time interval the event changes the value of fluent f, either initiating it (making it true) or terminating it (making it false). Then at time t_4 in the future, if no other intervening event has changed the fluent (either terminated or initiated it, respectively), then the fluent will have maintained its value. Formally, the axioms are:

$$Happens(e, t_1, t_3) \wedge Initiates(e, f, t_2) \wedge \neg Terminated\,(f, t_2, t_4) \wedge t_1 \leq t_2 \leq t_3 \leq t_4 \Rightarrow$$
$$T(f, t_2, t_4)$$
$$Happens(e, t_1, t_3) \wedge Terminated(e, f, t_2) \wedge \neg Initiates\,(f, t_2, t_4) \wedge t_1 \leq t_2 \leq t_3 \leq t_4 \Rightarrow$$
$$\neg T(f, t_2, t_4)$$

where *Terminated* and *Initiated* are defined by:

$$Terminated(f, t_1, t_5) \Leftrightarrow$$
$$\exists e, t_2, t_3, t_4\, Happens(e, t_2, t_4) \wedge Terminated\,(e, f, t_3) \wedge t_1 \leq t_2 \leq t_3 \leq t_4 \leq t_5$$
$$Initiated(f, t_1, t_5) \Leftrightarrow$$
$$\exists e, t_2, t_3, t_4\, Happens(e, t_2, t_4) \wedge Initiates\,(e, f, t_3) \wedge t_1 \leq t_2 \leq t_3 \leq t_4 \leq t_5$$

We can extend event calculus to represent simultaneous events (such as two people being necessary to ride a seesaw), exogenous events (such as the wind moving an object), continuous events (such as the rising of the tide), nondeterministic events (such as flipping a coin and having it come up heads or tails), and other complications.

10.3.1 Time

Event calculus opens us up to the possibility of talking about time points and time intervals. We will consider two kinds of time intervals: moments and extended intervals. The distinction is that only moments have zero duration:

$$Partition(\{Moments,\ ExtendedIntervals\},\ Intervals)$$
$$i \in Moments \Leftrightarrow Duration(i) = \mathrm{Sec}\,onds(0)\ .$$

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

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

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

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

Two intervals *Meet* if the end time of the first equals the start time of the second. The complete set of interval relations (Allen, 1983) is shown below and in Figure 10.2:

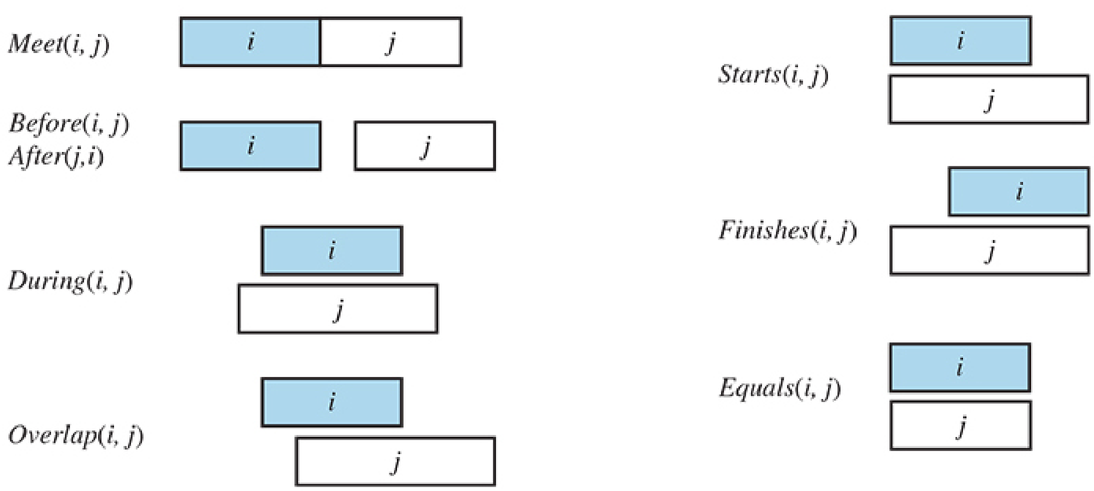


Figure 10.2 Predicates on time intervals.

$$
\begin{array}{lcl}
Meet(i,\ j) & \Leftrightarrow & End(i) = Begin(\ j) \\
Before(i,\ j) & \Leftrightarrow & End(i)\ <\ Begin(\ j) \\
After(j,\ i) & \Leftrightarrow & Before(i,\ j) \\
During(i,\ j) & \Leftrightarrow & Begin(j)\ <\ Begin(i)\ <\ End(i)\ <\ End(j) \\
Overlap(i,\ j) & \Leftrightarrow & Begin(i)\ <\ Begin(j)\ <\ End(i)\ <\ End(j) \\
Starts(i,\ j) & \Leftrightarrow & Begin(i)\ =\ Begin(j) \\
Finishes(i,\ j) & \Leftrightarrow & End(i)\ =\ End(j) \\
Equals(i,\ j) & \Leftrightarrow & Begin(i) = Begin(j) \land End(i) = End(j)
\end{array}
$$

These all have their intuitive meaning, with the exception of *Overlap*: we tend to think of overlap as symmetric (if *i* overlaps *j* then *j* overlaps *i*), but in this definition, *Overlap*(*i, j*) only is true if *i* begins before *j*. Experience has shown that this definition is more useful for writing axioms. To say that the reign of Elizabeth II immediately followed that of George VI, and the reign of Elvis overlapped with the 1950s, we can write the following:

$$
\begin{array}{l}
Meets(ReignOf\ (GeorgeVI), ReignOf\ (ElizabethII)). \\
Overlap(Fifties, ReignOf\ (Elvis)). \\
Begin(Fifties) = Begin(AD1950). \\
End(Fifties) = End(AD1959).
\end{array}
$$

10.3.2 Fluents and objects

Physical objects can be viewed as generalized events, in the sense that a physical object is a chunk of space–time. For example, *USA* can be thought of as an event that began in 1776 as a union of 13 states and is still in progress today as a union of 50. We can describe the changing properties of *USA* using state fluents, such as *Population*(*USA*). A property of USA that changes every four or eight years, barring mishaps, is its president. One might propose that *President* (*USA*) is a logical term that denotes a different object at different times.

Unfortunately, this is not possible, because a term denotes exactly one object in a given model structure. (The term *President*(*USA, t*) can denote different objects, depending on the value of *t*, but our ontology keeps time indices separate from fluents.) The only possibility is that *President* (*USA*) denotes a single object that consists of different people at different times. It is the object that is George Washington from 1789 to 1797, John Adams from 1797 to 1801, and so on, as in Figure 10.3. To say that George Washington was president throughout 1790, we can write

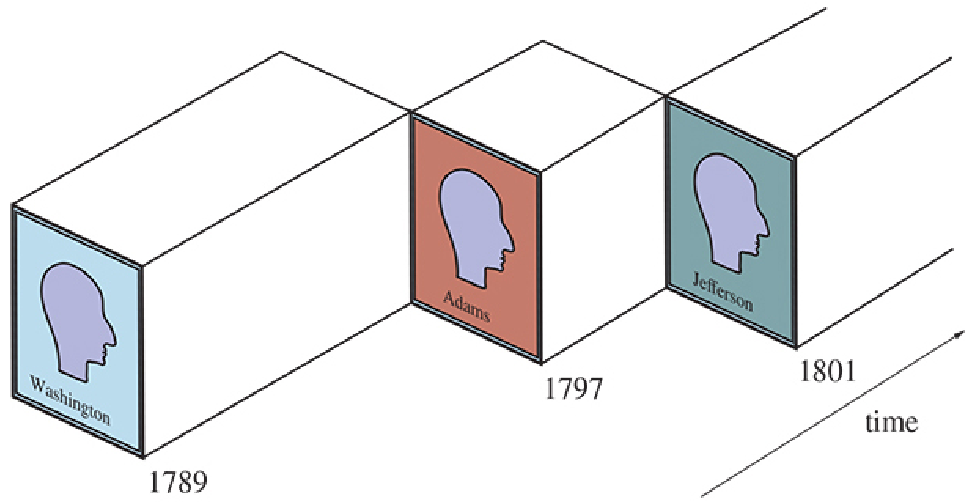


Figure 10.3 A schematic view of the object *President*(*USA*) for the early years

$$T(Equals(President(USA); GeorgeWashington), Begin(AD1790), End(AD1790)).$$

We use the function symbol *Equals* rather than the standard logical predicate =, because we cannot have a predicate as an argument to *T*, and because the interpretation is *not* that *GeorgeWashington* and *President(USA)* are logically identical in 1790; logical identity is not something that can change over time. The identity is between the subevents of the objects *President(USA)* and *GeorgeWashington* that are defined by the period 1790.

10.4 Mental Objects and Modal Logic

The agents we have constructed so far have beliefs and can deduce new beliefs. Yet none of them has any knowledge *about* beliefs or *about* deduction. Knowledge about one's own knowledge and reasoning processes is useful for controlling inference. For example, suppose Alice asks "what is the square root of 1764" and Bob replies "I don't know." If Alice insists "think harder," Bob should realize that with some more thought, this question can in fact be answered. On the other hand, if the question were "Is the president sitting down right now?" then Bob should realize that thinking harder is unlikely to help. Knowledge about the knowledge of other agents is also important; Bob should realize that the president does know.

What we need is a model of the mental objects that are in someone's head (or something's knowledge base) and of the mental processes that manipulate those mental objects. The model does not have to be detailed. We do not have to be able to predict how many milliseconds it will take for a particular agent to make a deduction. We will be happy just to be able to conclude that mother knows whether or not she is sitting.

We begin with the **propositional attitudes** that an agent can have toward mental objects: attitudes such as *Believes, Knows, Wants,* and *Informs.* The difficulty is that these attitudes do not behave like "normal" predicates. For example, suppose we try to assert that Lois knows that Superman can fly:

$$Knows(Lois, CanFly(Superman)).$$

One minor issue with this is that we normally think of *CanFly*(*Superman*) as a sentence, but here it appears as a term. That issue can be patched up by reifying *CanFly* (*Superman*); making it a fluent. A more serious problem is that, if it is true that Superman is Clark Kent, then we must conclude that Lois knows that Clark can fly, which is wrong because (in most versions of the story) Lois does *not* know that Clark is Superman.

$$\begin{aligned}&(Superman = Clark) \land Knows(Lois, CanFly(Superman))\\ &\qquad \models Knows(Lois, CanFly(Clark))\end{aligned}$$

This is a consequence of the fact that equality reasoning is built into logic. Normally that is a good thing; if our agent knows that $2 + 2 = 4$ and $4 < 5$, then we want our agent to know that $2 + 2 < 5$. This property is called **referential transparency**—it doesn't matter what term a logic uses to refer to an object, what matters is the object that the term names. But for propositional attitudes like *believes* and *knows*, we would like to have referential opacity—the terms used *do* matter, because not all agents know which terms are co-referential.

We could patch this up with even more reification: we could have one object to represent Clark/Superman, another object to represent the person that Lois knows as Clark, and yet another for the person Lois knows as Superman. However, this proliferation of objects means that the sentences we want to write quickly become verbose and clumsy.

Modal logic is designed to address this problem. Regular logic is concerned with a single modality, the modality of truth, allowing us to express "*P* is true" or "*P* is false." Modal logic includes special **modal operators** that take sentences (rather than terms) as arguments. For example, "*A* knows *P*" is represented with the notation $\mathbf{K}_A P$, where **K** is the modal operator for knowledge. It takes two arguments, an agent (written as the subscript) and a sentence. The syntax of modal logic is the same as first-order logic, except that sentences can also be formed with modal operators.

The semantics of modal logic is more complicated. In first-order logic a **model** contains a set of objects and an interpretation that maps each name to the appropriate object, relation, or function. In modal logic we want to be able to consider both the possibility that Superman's secret identity is Clark and the possibility that it isn't.

Therefore, we will need a more complicated model, one that consists of a collection of **possible worlds** rather than just one true world. The worlds are connected in a graph by **accessibility relations**, one relation for each modal operator. We say that world w_1 is accessible from world w_0 with respect to the modal operator $\mathbf{K}_A$ if everything in w_1 is consistent with what *A* knows in w_0. As an example, in the real world, Bucharest is the capital of Romania, but for an agent that did not know that, a world where the capital of Romania is, say, Sofia is accessible. Hopefully a world where $2 + 2 = 5$ would not be accessible to any agent.

In general, a knowledge atom $\mathbf{K}_A P$ is true in world *w* if and only if *P* is true in every world accessible from *w*. The truth of more complex sentences is derived by recursive application of this rule and the normal rules of first-order logic. That means that modal logic can be used to reason about nested knowledge sentences: what one agent knows about another agent's knowledge. For example, we can say that even though Lois doesn't know whether Superman's secret identity is Clark Kent, she does know that Clark knows:

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

Modal logic solves some tricky issues with the interplay of quantifiers and knowledge. The English sentence "Bond knows that someone is a spy" is ambiguous. The first reading is that there is a particular someone who Bond knows is a spy; we can write this as

$$\exists x \ \mathbf{K}_{Bond}Spy(x),$$

which in modal logic means that there is an $\boldsymbol{x}$ that, in all accessible worlds, Bond knows to be a spy. The second reading is that Bond just knows that there is at least one spy:

$$\mathbf{K}_{Bond}\exists x \ Spy(x).$$

The modal logic interpretation is that in each accessible world there is an $\boldsymbol{x}$ that is a spy, but it need not be the same x in each world.

Now that we have a modal operator for knowledge, we can write axioms for it. First, we can say that agents are able to draw conclusions; if an agent knows P and knows that P implies Q, then the agent knows Q:

$$(\mathbf{K}_a P \wedge \mathbf{K}_a(P \Rightarrow Q)) \Rightarrow \mathbf{K}_a Q.$$

From this (and a few other rules about logical identities) we can establish that $\mathbf{K}_A(P \vee \neg P)$ is a tautology; every agent knows every proposition P is either true or false. On the other hand, $(\mathbf{K}_A P) \vee (\mathbf{K}_A \neg P)$ is not a tautology; in general, there will be lots of propositions that an agent does not know to be true and does not know to be false.

It is said (going back to Plato) that knowledge is justified true belief. That is, if it is true, if you believe it, and if you have an unassailably good reason, then you know it. That means that if you know something, it must be true, and we have the axiom:

$$\mathbf{K}_a P \Rightarrow P.$$

Furthermore, logical agents (but not all people) are able to introspect on their own knowledge. If they know something, then they know that they know it:

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

We can define similar axioms for belief (often denoted by **B**) and other modalities. However, one problem with the modal logic approach is that it assumes **logical omniscience** on the part of agents. That is, if an agent knows a set of axioms, then it knows all consequences of those axioms. This is on shaky ground even for the somewhat abstract notion of knowledge, but it seems even worse for belief, because belief has more connotation of referring to things that are physically represented in the agent, not just potentially derivable.

There have been attempts to define a form of limited rationality for agents—to say that agents believe only those assertions that can be derived with the application of no more than *k* reasoning steps, or no more than s seconds of computation. These attempts have been generally unsatisfactory.

10.4.1 Other modal logics

Many modal logics have been proposed, for different modalities besides knowledge. One proposal is to add modal operators for *possibility* and *necessity:* it is possibly true that one of the authors of this book is sitting down right now, and it is necessarily true that 2 + 2 = 4.

As mentioned in Section 8.1.2, some logicians favor modalities related to time. In **linear temporal logic,** we add the following modal operators:

- **X** *P*: "*P* will be true in the next time step"
- **F** *P*: "*P* will eventually (**F**inally) be true in some future time step"
- **G** *P*: "*P* is always (**G**lobally) true"
- *P* **U** *Q*: "*P* remains true until *Q* occurs"

Sometimes there are additional operators that can be derived from these. Adding these modal operators makes the logic itself more complex (and thus makes it harder for a logical inference algorithm to find a proof). But the operators also allow us to state certain facts in a more succinct form (which makes logical inference faster). The choice of which logic to use is similar to the choice of which programming language to use: pick one that is appropriate to your task, that is familiar to you and the others who will share your work, and that is efficient enough for your purposes.

10.5 Reasoning Systems for Categories

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

10.5.1 Semantic networks

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

There are many variants of semantic networks, but all are capable of representing individual objects, categories of objects, and relations among objects. A typical graphical notation displays object or category names in ovals or boxes, and connects them with labeled links. For example, Figure 10.4 has a *MemberOf* link between *Mary* and *FemalePersons*, corresponding to the logical assertion $Mary \in FemalePersons$; similarly, the *SisterOf* link between *Mary* and *John* corresponds to the assertion *SisterOf* (*Mary, John*). We can connect categories using *SubsetOf* links, and so on. It is such fun drawing bubbles and arrows that one can get carried away. For example, we know that persons have female persons as mothers, so can we draw a *HasMother* link from *Persons* to *FemalePersons?* The answer is no, because *HasMother* is a relation between a person and his or her mother, and categories do not have mothers.[5]

For this reason, we have used a special notation—the double-boxed link—in Figure 10.4. This link asserts that

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

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

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

As before, we need to be careful not to assert that a category has legs; the single-boxed link in Figure 10.4 is used to assert properties of every member of a category.

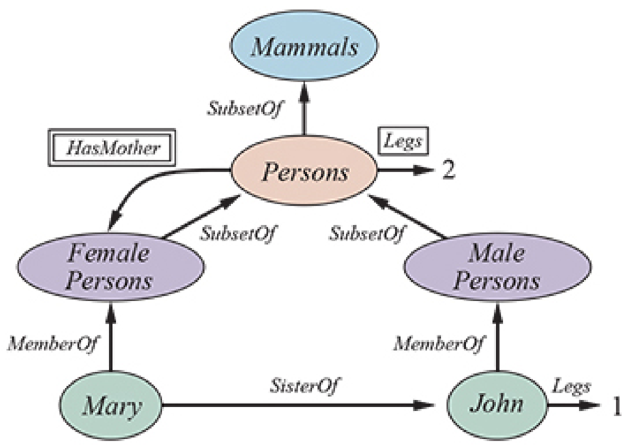


Figure 10.4 A semantic network with four objects (John, Mary, 1, and 2) and four categories. Relations are denoted by labeled links.

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

Inheritance becomes complicated when an object can belong to more than one category or when a category can be a subset of more than one other category; this is called **multiple inheritance**. In such cases, the inheritance algorithm might find two or more conflicting values answering the query. For this reason, multiple inheritance is banned in some **object-oriented programming** (OOP) languages, such as Java, that use inheritance in a class hierarchy. It is usually allowed in semantic networks, but we defer discussion of that until Section 10.6.

The reader might have noticed an obvious drawback of semantic network notation, compared to first-order logic: the fact that links between bubbles represent only *binary* relations. For example, the sentence *Fly* (*Shankar, NewYork, NewDelhi, Yesterday*) cannot be asserted directly in a semantic network. Nonetheless, we *can* obtain the effect of *n*-ary assertions by reifying the proposition itself as an event belonging to an appropriate event category. Figure 10.5 shows the semantic network structure for this particular event. Notice that the restriction to binary relations forces the creation of a rich ontology of reified concepts.

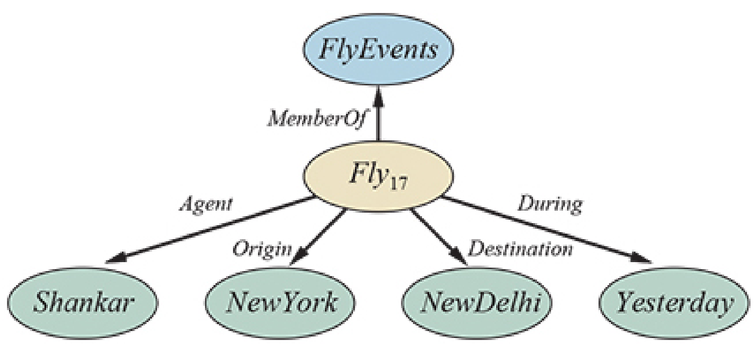


Figure 10.5 A fragment of a semantic network showing the representation of the logical assertion *Fly(Shankar, NewYork, NewDelhi, Yesterday).*

Reification of propositions makes it possible to represent every ground, function-free atomic sentence of first-order logic in the semantic network notation. Certain kinds of universally quantified sentences can be asserted using inverse links and the singly boxed and doubly boxed arrows applied to categories, but that still leaves us a long way short of full first-order logic. Negation, disjunction, nested function symbols, and existential quantification are all missing. Now it is *possible* to extend the notation to make it equivalent to first-order logic—as in Peirce's existential graphs—but doing so negates one of the main advantages of semantic networks, which is the simplicity and transparency of the inference processes. Designers can build a large network and still have a good idea about what queries will be efficient, because (a) it is easy to visualize the steps that the inference procedure will go through and (b) in some cases the query language is so simple that difficult queries cannot be posed.

In cases where the expressive power proves to be too limiting, many semantic network systems provide for **procedural attachment** to fill in the gaps. Procedural attachment is a technique whereby a query about (or sometimes an assertion of) a certain relation results in a call to a special procedure designed for that relation rather than a general inference algorithm.

One of the most important aspects of semantic networks is their ability to represent **default values** for categories. Examining Figure 10.4 carefully, one notices that John has one leg, despite the fact that he is a person and all persons have two legs. In a strictly logical KB, this would be a contradiction, but in a semantic network, the assertion that all persons have two legs has only default status; that is, a person is assumed to have two legs unless this is contradicted by more specific information. The default semantics is enforced naturally by the inheritance algorithm, because it follows links upwards from the object itself (John in this case) and stops as soon as it finds a value. We say that the default is **overridden** by the more specific value. Notice that we could also override the default number of legs by creating a category of *OneLeggedPersons*, a subset of *Persons* of which *John* is a member.

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

$$\forall x \; x \in Persons \land x \neq John \Rightarrow Leg(x, 2).$$

For a *fixed* network, this is semantically adequate but will be much less concise than the network notation itself if there are lots of exceptions. For a network that will be updated with more assertions, however, such an approach fails—we really want to say that any persons as yet unknown with one leg are exceptions too. Section 10.6 goes into more depth on this issue and on default reasoning in general.

10.5.2 Description logics

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

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

The CLASSIC language (Borgida *et al.*, 1989) is a typical description logic. The syntax of CLASSIC descriptions is shown in Figure 10.6.[6] For example, to say that bachelors are unmarried adult males we would write

$$
\begin{array}{rcl}
Concept & \rightarrow & \textbf{Thing} \mid ConceptName \\
& \mid & \textbf{And}(Concept, \ldots) \\
& \mid & \textbf{All}(RoleName, Concept) \\
& \mid & \textbf{AtLeast}(Integer, RoleName) \\
& \mid & \textbf{AtMost}(Integer, RoleName) \\
& \mid & \textbf{Fills}(RoleName, IndividualName, \ldots) \\
& \mid & \textbf{SameAs}(Path, Path) \\
& \mid & \textbf{OneOf}(IndividualName, \ldots) \\
Path & \rightarrow & [RoleName, \ldots] \\
ConceptName & \rightarrow & Adult \mid Female \mid Male \mid \ldots \\
RoleName & \rightarrow & Spouse \mid Daughter \mid Son \mid \ldots
\end{array}
$$

Figure 10.6 The syntax of descriptions in a subset of the CLASSIC language.

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

The equivalent in first-order logic would be

$$Bachelor(x) \quad \Leftrightarrow \; Unmarried(x) \land Adult(x) \land Male(x).$$

Notice that the description logic has an algebra of operations on predicates, which of course we can't do in first-order logic. Any description in CLASSIC can be translated into an equivalent first-order sentence, but some descriptions are more straightforward in CLASSIC. For example, to describe the set of men with at least three sons who are all unemployed and married to doctors, and at most two daughters who are all professors in physics or math departments, we would use

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

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

Perhaps the most important aspect of description logics is their emphasis on tractability of inference. A problem instance is solved by describing it and then asking if it is subsumed by one of several possible solution categories. In standard first-order logic systems, predicting the solution time is often impossible. It is frequently left to the user to engineer the representation to detour around sets of sentences that seem to be causing the system to take several weeks to solve a problem. The thrust in description logics, on the other hand, is to ensure that subsumption-testing can be solved in time polynomial in the size of the descriptions.[7]

This sounds wonderful in principle, until one realizes that it can only have one of two consequences: either hard problems cannot be stated at all, or they require exponentially large descriptions! However, the tractability results do shed light on what sorts of constructs cause problems and thus help the user to understand how different representations behave. For example, description logics usually lack *negation* and *disjunction.* Each forces firstorder logical systems to go through a potentially exponential case analysis in order to ensure completeness. CLASSIC allows only a limited form of disjunction in the *Fills* and *OneOf* constructs, which permit disjunction over explicitly enumerated individuals but not over descriptions. With disjunctive descriptions, nested definitions can lead easily to an exponential number of alternative routes by which one category can subsume another.

10.6 Reasoning with Default Information

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

10.6.1 Circumscription and default logic

We have seen two examples of reasoning processes that violate the **monotonicity** property of logic that was proved in Chapter 7.[8] In this chapter we saw that a property inherited by all members of a category in a semantic network could be overridden by more specific information for a subcategory. In Section 9.4.4, we saw that under the closed-world assumption, if a proposition *a* is not mentioned in *KB* then $KB \models \neg\alpha$, but $KB \wedge \alpha \models \alpha$.

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

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

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

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

Circumscription can be viewed as an example of a **model preference** logic. In such logics, a sentence is entailed (with default status) if it is true in all *preferred* models of the KB, as opposed to the requirement of truth in *all* models in classical logic. For circumscription, one model is preferred to another if it has fewer abnormal objects.[9] Let us see how this idea works in the context of multiple inheritance in semantic networks. The standard example for which multiple inheritance is problematic is called the "Nixon diamond." It arises from the observation that Richard Nixon was both a

Quaker (and hence by default a pacifist) and a Republican (and hence by default not a pacifist). We can write this as follows:

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

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

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

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

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

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

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

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

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

Since 1980, when nonmonotonic logics were first proposed, a great deal of progress has been made in understanding their mathematical properties. There are still unresolved questions, however. For example, if "Cars have four wheels" is false, what does it mean to have it in one's knowledge base? What is a good set of default rules to have? If we cannot decide, for each rule separately, whether it belongs in our knowledge base, then we have a serious problem of nonmodularity. Finally, how can beliefs that have default status be used to make decisions? This is probably the hardest issue for default reasoning.

Decisions often involve tradeoffs, and one therefore needs to compare the *strengths* of belief in the outcomes of different actions, and the ***costs*** of making a wrong decision. In cases where the same kinds of decisions are being made repeatedly, it is possible to interpret default rules as "threshold probability" statements. For example, the default rule "My brakes are always OK" really means "The probability that my brakes are OK, given no other information, is sufficiently high that the optimal decision is for me to drive without checking them." When the decision context changes—for example, when one is driving a heavily laden truck down a steep mountain road—the default rule suddenly becomes inappropriate, even though there is no new evidence of faulty brakes. These considerations have led researchers to consider how to embed default reasoning within probability theory or utility theory.

10.6.2 Truth maintenance systems

We have seen that many of the inferences drawn by a knowledge representation system will have only default status, rather than being absolutely certain. Inevitably, some of these inferred facts will turn out to be wrong and will have to be retracted in the face of new information. This process is called **belief revision**.[10] Suppose that a knowledge base *KB* contains a sentence *P*—perhaps a default conclusion recorded by a forward-chaining algorithm, or perhaps just an incorrect assertion—and we want to execute TELL(*KB*, $\neg P$). To avoid creating a contradiction, we must first execute RETRACT(*KB*, *P*). This sounds easy enough. Problems arise, however, if any *additional* sentences were inferred from *P* and asserted in the KB. For example, the implication $P \Rightarrow Q$ might have been used to add *Q*. The obvious "solution"—retracting all sentences inferred from *P*—fails because such sentences may have other justifications besides *P*. For example, if *R* and $R \Rightarrow Q$ are also in the KB, then *Q* does not have to be removed after all. **Truth maintenance systems**, or TMSs, are designed to handle exactly these kinds of complications.

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

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

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

In addition to handling the retraction of incorrect information, TMSs can be used to speed up the analysis of multiple hypothetical situations. Suppose, for example, that the Romanian Olympic Committee is choosing sites for the swimming, athletics, and equestrian events at the 2048 Games to be held in Romania. For example, let the first hypothesis be *Site*(*Swimming, Pitesti*), *Site* (*Athletics, Bucharest*), and *Site*(*Equestrian, Arad*).

A great deal of reasoning must then be done to work out the logistical consequences and hence the desirability of this selection. If we want to consider *Site*(*Athletics, Sibiu*) instead, the TMS avoids the need to start again from scratch. Instead, we simply retract *Site*(*Athletics, Bucharest*) and assert *Site*(*Athletics, Sibiu*) and the TMS takes care of the necessary revisions. Inference chains generated from the choice of Bucharest can be reused with Sibiu, provided that the conclusions are the same.

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

Truth maintenance systems also provide a mechanism for generating **explanations**. Technically, an explanation of a sentence *P* is a set of sentences *E* such that *E* entails *P*. If the sentences in *E* are already known to be true, then *E* simply provides a sufficient basis for proving that *P* must be the case. But explanations can also include **assumptions**—sentences that are not known to be true, but would suffice to prove *P* if they were true. For example, if your car won’t start, you probably don’t have enough information to definitively prove the reason for the problem. But a reasonable explanation might include the assumption that the battery is dead. This, combined with knowledge of how cars operate, explains the observed nonbehavior. In most cases, we will prefer an explanation *E* that is minimal, meaning that there is no proper subset of *E* that is also an explanation. An ATMS can generate explanations for the “car won’t start” problem by making assumptions (such as “no gas in car” or “battery dead”) in any order we like, even if some assumptions are contradictory. Then we look at the label for the sentence “car won’t start” to read off the sets of assumptions that would justify the sentence.

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

Summary

By delving into the details of how one represents a variety of knowledge, we hope we have given the reader a sense of how real knowledge bases are constructed and a feeling for the interesting philosophical issues that arise. The major points are as follows:

- Large-scale knowledge representation requires a general-purpose ontology to organize and tie together the various specific domains of knowledge.
- A general-purpose ontology needs to cover a wide variety of knowledge and should be capable, in principle, of handling any domain.
- Building a large, general-purpose ontology is a significant challenge that has yet to be fully realized, although current frameworks seem to be quite robust.
- We presented an **upper ontology** based on categories and the event calculus. We covered categories, subcategories, parts, structured objects, measurements, substances, events, time and space, change, and beliefs.
- Natural kinds cannot be defined completely in logic, but properties of natural kinds can be represented.
- Actions, events, and time can be represented with the event calculus. Such representations enable an agent to construct sequences of actions and make logical inferences about what will be true when these actions happen.
- Special-purpose representation systems, such as **semantic networks** and **description logics**, have been devised to help in organizing a

hierarchy of categories. **Inheritance** is an important form of inference, allowing the properties of objects to be deduced from their membership in categories.

- The **closed-world assumption**, as implemented in logic programs, provides a simple way to avoid having to specify lots of negative information. It is best interpreted as a default that can be overridden by additional information.
- **Nonmonotonic logics**, such as **circumscription** and **default logic**, are intended to capture default reasoning in general.
- **Truth maintenance systems** handle knowledge updates and revisions efficiently.
- It is difficult to construct large ontologies by hand; extracting knowledge from text makes the job easier.

Bibliographical and Historical Notes

Briggs (1985) claims that knowledge representation research began with first millennium BCE Indian theorizing about the grammar of Shastric Sanskrit. Western philosophers trace their work on the subject back to c. 300 BCE in Aristotle's *Metaphysics* (literally, what comes after the book on physics). The development of technical terminology in any field can be regarded as a form of knowledge representation.

Early discussions of representation in AI tended to focus on "*problem* representation" rather than "*knowledge* representation." (See, for example, Amarel's (1968) discussion of the "Missionaries and Cannibals" problem.) In the 1970s, AI emphasized the development of "expert systems" (also called "knowledge-based systems") that could, if given the appropriate domain knowledge, match or exceed the performance of human experts on narrowly defined tasks. For example, the first expert system, DENDRAL (Feigenbaum *et al.*, 1971; Lindsay *et al.*, 1980), interpreted the output of a mass spectrometer (a type of instrument used to analyze the structure of organic chemical compounds) as accurately as expert chemists. Although the success of DENDRAL was instrumental in convincing the AI research community of the importance of knowledge representation, the representational formalisms used in DENDRAL are highly specific to the domain of chemistry.

Over time, researchers became interested in standardized knowledge representation formalisms and ontologies that could assist in the creation of new expert systems. This brought them into territory previously explored by philosophers of science and of language. The discipline imposed in AI by the need for one's theories to "work" has led to more rapid and deeper progress than when these problems were the exclusive domain of philosophy (although it has at times also led to the repeated reinvention of the wheel).

But to what extent can we trust expert knowledge? As far back as 1955, Paul Meehl (see also Grove and Meehl, 1996) studied the decision-making processes of trained experts at subjective tasks such as predicting the success of a student in a training program or the recidivism of a criminal. In 19 out of the 20 studies he looked at, Meehl found that simple statistical learning algorithms (such as linear regression or naive Bayes) predict better than the experts. Tetlock (2017) also studies expert knowledge and finds it lacking in difficult cases. The Educational Testing Service has used an automated program to grade millions of essay questions on the GMAT exam since 1999. The program agrees with human graders 97% of the time, about the same level that two human graders agree (Burstein *et al.*, 2001). (This does not mean the program understands essays, just that it can distinguish good ones from bad ones about as well as human graders can.)

The creation of comprehensive taxonomies or classifications dates back to ancient times. Aristotle (384–322 BCE) strongly emphasized classification and categorization schemes. His *Organon*, a collection of works on logic assembled by his students after his death, included a treatise called ***Categories*** in which he attempted to construct what we would now call an upper ontology. He also introduced the notions of genus and species for lower-level classification. Our present system of biological classification, including the use of “binomial nomenclature” (classification via genus and species in the technical sense), was invented by the Swedish biologist Carolus Linnaeus, or Carl von Linne (1707–1778). The problems associated with natural kinds and inexact category boundaries have been addressed by Wittgenstein (1953), Quine (1953), Lakoff (1987), and Schwartz (1977), among others.

See Chapter 25 for a discussion of deep neural network representations of words and concepts that escape some of the problems of a strict ontology, but also sacrifice some of the precision. We still don't know the best way to combine the advantages of neural networks and logical semantics for representation.

Interest in larger-scale ontologies is increasing, as documented by the *Handbook on Ontologies* (Staab, 2004). The OPENCYC project (Lenat and Guha, 1990; Matuszek *et al.*,2006) has released a 150,000-concept ontology, with an upper ontology similar to the one in Figure 10.1 as well as specific concepts like "OLED Display" and "iPhone," which is a type of "cellular phone," which in turn is a type of "consumer electronics," "phone," "wireless communication device," and other concepts. The NEXTKB project extends CYC and other resources including FrameNet and WordNet into a knowledge base with almost 3 million facts, and provides a reasoning engine, FIRE to go with it (Forbus *et al.*, 2010).

The DBPEDIA project extracts structured data from Wikipedia, specifically from In- foboxes: the attribute/value pairs that accompany many Wikipedia articles (Wu and Weld, 2008; Bizer *et al.*, 2007). As of 2015, DBPEDIAcontained 400 million facts about 4 million objects in the English version alone; counting all 110 languages yields 1.5 billion facts (Lehmann *et al.*, 2015).

The IEEE working group P1600.1 created SUMO, the Suggested Upper Merged Ontology (Niles and Pease, 2001; Pease and Niles, 2002), with about 1000 terms in the upper ontology and links to over 20,000 domain-specific terms. Stoffel *et al.* (1997) describe algorithms for efficiently managing a very large ontology. A survey of techniques for extracting knowledge from Web pages is given by Etzioni *et al.* (2008).

On the Web, representation languages are emerging. RDF (Brickley and Guha, 2004) allows for assertions to be made in the form of relational triples and provides some means for evolving the meaning of names over time. OWL (Smith *et al.*, 2004) is a description logic that supports inferences over these triples. So far, usage seems to be inversely proportional to representational complexity: the traditional HTML and CSS formats account for over 99% of Web content, followed by the simplest representation schemes, such as RDFa (Adida and Birbeck, 2008), and microformats (Khare, 2006; Patel-Schneider, 2014) which use HTML and XHTML markup to add attributes to text on web pages. Usage of sophisticated RDF and OWL ontologies is not yet widespread, and the full vision of the Semantic Web (Berners- Lee *et al.*, 2001) has not been realized. The conferences on *Formal Ontology in Information Systems* **(** FOIS) covers both general and domain-specific ontologies.

The taxonomy used in this chapter was developed by the authors and is based in part on their experience in the CYC project and in part on work by Hwang and Schubert (1993) and Davis (1990, 2005). An inspirational discussion of the general project of commonsense knowledge representation appears in Hayes's (1978, 1985b) "Naive Physics Manifesto."

Successful deep ontologies within a specific field include the Gene Ontology project (Gene Ontology Consortium, 2008) and the Chemical Markup Language (Murray-Rust *et al.*, 2003). Doubts about the feasibility of a single ontology for *all* knowledge are expressed by Doctorow (2001), Gruber (2004), Halevy *et al.* (2009), and Smith (2004).

The event calculus was introduced by Kowalski and Sergot (1986) to handle continuous time, and there have been several variations (Sadri and

Kowalski, 1995; Shanahan, 1997) and overviews (Shanahan, 1999; Mueller, 2006). James Allen introduced time intervals for the same reason (Allen, 1984), arguing that intervals were much more natural than situations for reasoning about extended and concurrent events. In van Lambalgen and Hamm (2005) we see how the logic of events maps onto the language we use to talk about events. An alternative to the event and situation calculi is the fluent calculus (Thielscher, 1999), which reifies the facts out of which states are composed.

Peter Ladkin (1986a, 1986b) introduced “concave” time intervals (intervals with gaps—essentially, unions of ordinary “convex” time intervals) and applied the techniques of mathematical abstract algebra to time representation. Allen (1991) systematically investigates the wide variety of techniques available for time representation; van Beek and Manchak (1996) analyze algorithms for temporal reasoning. There are significant commonalities between the event-based ontology given in this chapter and an analysis of events due to the philosopher Donald Davidson (1980). The histories in Pat Hayes’s (1985a) ontology of liquids and the chronicles in McDermott’s (1985) theory of plans were also important influences on the field and on this chapter.

The question of the ontological status of substances has a long history. Plato proposed that substances were abstract entities entirely distinct from physical objects; he would say $MadeOf(Butter_3, Butter)$ rather than $Butter_3 \in Butter$. This leads to a substance hierarchy in which, for example, *UnsaltedButter* is a more specific substance than *Butter*. The position adopted in this chapter, in which substances are categories of objects, was championed by Richard Montague (1973). It has also been adopted in the CYC project. Copeland (1993) mounts a serious, but not invincible, attack.

The alternative approach mentioned in the chapter, in which butter is one object consisting of all buttery objects in the universe, was proposed originally by the Polish logician Leśniewski (1916). His mereology (the name is derived from the Greek word for "part") used the part–whole relation as a substitute for mathematical set theory, with the aim of eliminating abstract entities such as sets. A more readable exposition of these ideas is given by Leonard and Goodman (1940), and Goodman's *The Structure of Appearance* (1977) applies the ideas to various problems in knowledge representation.

While some aspects of the mereological approach are awkward—for example, the need for a separate inheritance mechanism based on part-whole relations—the approach gained the support of Quine (1960). Harry Bunt (1985) has provided an extensive analysis of its use in knowledge representation. Casati and Varzi (1999) cover parts, wholes, and a general theory of spatial locations.

There are three main approaches to the study of mental objects. The one taken in this chapter, based on modal logic and possible worlds, is the classical approach from philosophy (Hintikka, 1962; Kripke, 1963; Hughes and Cresswell, 1996). The book *Reasoning about Knowledge* (Fagin *et al.*, 1995) provides a thorough introduction, and Gordon and Hobbs (2017) provide *A Formal Theory of Commonsense Psychology.*

The second approach is a first-order theory in which mental objects are fluents. Davis (2005) and Davis and Morgenstern (2005) describe this approach. It relies on the possible-worlds formalism, and builds on work by Robert Moore (1980, 1985).

The third approach is a syntactic theory, in which mental objects are represented by character strings. A string is just a complex term denoting a list of symbols, so *CanFly(Clark)*can be represented by the list of symbols

[*C, a,n, F, l,y,*(*, C, l, a, r, k,*)]. The syntactic theory of mental objects was first studied in depth by Kaplan and Montague (1960), who showed that it led to paradoxes if not handled carefully. Ernie Davis (1990) provides an excellent comparison of the syntactic and modal theories of knowledge. Pnueli (1977) describes a temporal logic used to reason about programs, work that won him the Turing Award and which was expanded upon by Vardi (1996). Littman *et al.* (2017) show that a temporal logic can be a good language for specifying goals to a reinforcement learning robot in a way that is easy for a human to specify, and generalizes well to different environments.

The Greek philosopher Porphyry (c. 234–305 CE), commenting on Aristotle's *Categories,* drew what might qualify as the first semantic network. Charles S. Peirce (1909) developed existential graphs as the first semantic network formalism using modern logic. Ross Quillian (1961), driven by an interest in human memory and language processing, initiated work on semantic networks within AI. An influential paper by Marvin Minsky (1975) presented a version of semantic networks called **frames**; a frame was a representation of an object or category, with attributes and relations to other objects or categories.

The question of semantics arose quite acutely with respect to Quillian's semantic networks (and those of others who followed his approach), with their ubiquitous and very vague "IS-A links." Bill Woods's (1975) famous article "What's In a Link?" drew the attention of AI researchers to the need for precise semantics in knowledge representation formalisms. Ron Brachman (1979) elaborated on this point and proposed solutions. Patrick Hayes's (1979) "The Logic of Frames" cut even deeper, claiming that "Most of 'frames' is just a new syntax for parts of first-order logic." Drew McDermott's (1978b) "Tarskian Semantics, or, No Notation without

Denotation!" argued that the model-theoretic approach to semantics used in first-order logic should be applied to all knowledge representation formalisms. This remains a controversial idea; notably, McDermott himself has reversed his position in "A Critique of Pure Reason" (McDermott, 1987). Selman and Levesque (1993) discuss the complexity of inheritance with exceptions, showing that in most formulations it is NP-complete.

Description logics were developed as a useful subset of first-order logic for which inference is computationally tractable. Hector Levesque and Ron Brachman (1987) showed that certain uses of disjunction and negation were primarily responsible for the intractability of logical inference. This led to a better understanding of the interaction between complexity and expressiveness in reasoning systems. Calvanese *et al.* (1999) summarize the state of the art, and Baader *et al.* (2007) present a comprehensive handbook of description logic.

The three main formalisms for dealing with nonmonotonic inference—circumscription (McCarthy, 1980), default logic (Reiter, 1980), and modal nonmonotonic logic (McDermott and Doyle, 1980)—were all introduced in one special issue of the AI Journal. Delgrande and Schaub (2003) discuss the merits of the variants, given 25 years of hindsight. Answer set programming can be seen as an extension of negation as failure or as a refinement of circumscription; the underlying theory of stable model semantics was introduced by Gelfond and Lifschitz (1988), and the leading answer set programming systems are DLV (Eiter *et al.*,1998) and S MODELS (Niemela *et al.,* 2000). Brewka *et al.* (1997) give a good overview of the various approaches to nonmonotonic logic. Clark (1978) covers the negation-as-failure approach to logic programming and Clark completion. Lifschitz (2001) discusses the application of answer set programming to planning. A variety of nonmonotonic reasoning systems based on logic programming are documented in the proceedings of the conferences on *Logic Programming and Nonmonotonic Reasoning* (LPNMR).

The study of truth maintenance systems began with the TMS (Doyle, 1979) and RUP (McAllester, 1980) systems, both of which were essentially JTMSs. Forbus and de Kleer (1993) explain in depth how TMSs can be used in AI applications. Nayak and Williams (1997) show how an efficient incremental TMS called an ITMS makes it feasible to plan the operations of a NASA spacecraft in real time.

This chapter could not cover *every* area of knowledge representation in depth. The three principal topics omitted are the following:

Qualitative physics: Qualitative physics is a subfield of knowledge representation concerned specifically with constructing a logical, nonnumeric theory of physical objects and processes. The term was coined by Johan de Kleer (1975), although the enterprise could be said to have started in Fahlman's (1974) BUILD, a sophisticated planner for constructing complex towers of blocks. Fahlman discovered in the process of designing it that most of the effort (80%, by his estimate) went into modeling the physics of the blocks world to calculate the stability of various subassemblies of blocks, rather than into planning per se. He sketches a hypothetical naive-physics-like process to explain why young children can solve BUILD-like problems without access to the high-speed floating-point arithmetic used in BUILD's physical modeling. Hayes (1985a) uses "histories"—four-dimensional slices of space-time similar to Davidson's events—to construct a fairly complex naive physics of liquids. Davis (2008) gives an update to the ontology of liquids that describes the pouring of liquids into containers.

De Kleer and Brown (1985), Ken Forbus (1985), and Benjamin Kuipers (1985) independently and almost simultaneously developed systems that can reason about a physical system based on qualitative abstractions of the underlying equations. Qualitative physics soon developed to the point where it became possible to analyze an impressive variety of complex physical systems (Yip, 1991). Qualitative techniques have been used to construct novel designs for clocks, windshield wipers, and six-legged walkers (Subramanian and Wang, 1994). The collection *Readings in Qualitative Reasoning about Physical Systems* (Weld and de Kleer, 1990), an encyclopedia article by Kuipers (2001), and a handbook article by Davis (2007) provide good introductions to the field.

Spatial reasoning: The reasoning necessary to navigate in the wumpus world is trivial in comparison to the rich spatial structure of the real world. The earliest serious attempt to capture commonsense reasoning about space appears in the work of Ernest Davis (1986, 1990). The region connection calculus of Cohn *et al.* (1997) supports a form of qualitative spatial reasoning and has led to new kinds of geographical information systems; see also (Davis, 2006). As with qualitative physics, an agent can go a long way, so to speak, without resorting to a full metric representation.

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

Minker (2001) collects papers by leading researchers in knowledge representation, summarizing 40 years of work in the field. The proceedings of the international conferences on *Principles of Knowledge Representation and Reasoning* provide the most up-to-date sources for work in this area. *Readings in Knowledge Representation* (Brachman and Levesque, 1985) and *Formal Theories of the Commonsense World* (Hobbs and Moore, 1985) are excellent anthologies on knowledge representation; the former focuses more on historically important papers in representation languages and formalisms, the latter on the accumulation of the knowledge itself. Davis (1990), Stefik (1995), and Sowa (1999) provide textbook introductions to knowledge representation, van Harmelen *et al.* (2007) contributes a handbook, and Davis and Morgenstern (2004) edited a special issue of the AI Journal on the topic. Davis (2017) gives a survey of logic for commonsense reasoning. The biennial conference on *Theoretical Aspects of Reasoning About Knowledge* (TARK) covers applications of the theory of knowledge in AI, economics, and distributed systems.

---

[1] Turning a proposition into an object is called **reification** , from the Latin word *res,* or thing. John McCarthy proposed the term “thingiflcation,” but it never caught on.

[2] When asked what one could deduce about the Creator from the study of nature, biologist J. B. S. Haldane said “An inordinate fondness for beetles.”

[3] The terms “event” and “action” may be used interchangeably—they both mean “something that can happen.”

[4] Our version is based on Shanahan (1999), but with some alterations.

[5] Several early systems failed to distinguish between properties of members of a category and properties of the category as a whole. This can lead directly to inconsistencies, as pointed out by Drew McDermott (1976) in his article “Artificial Intelligence Meets Natural Stupidity.” Another

common problem was the use of *IsA* links for both subset and membership relations, in correspondence with English usage: “a cat is a mammal” and “Fifi is a cat.” See Exercise 10.NATS for more on these issues.

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

[7] CLASSIC provides efficient subsumption testing in practice, but the worst-case run time is exponential.

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

[9] For the closed-world assumption, one model is preferred to another if it has fewer true atoms—that is, preferred models are **minimal** models. There is a natural connection between the closed-world assumption and definite- clause KBs, because the fixed point reached by forward chaining on definite-clause KBs is the unique minimal model. See page 249 for more on this point.

[10] Belief revision is often contrasted with **belief update**, which occurs when a knowledge base is revised to reflect a change in the world rather than new information about a fixed world. Belief update combines belief revision with reasoning about time and change; it is also related to the process of **filtering** described in Chapter 14.

CHAPTER 11

AUTOMATED PLANNING

In which we see how an agent can take advantage of the structure of a problem to efficiently construct complex plans of action.

Planning a course of action is a key requirement for an intelligent agent. The right representation for actions and states and the right algorithms can make this easier. In Section 11.1 we introduce a general **factored** representation language for planning problems that can naturally and succinctly represent a wide variety of domains, can efficiently scale up to large problems, and does not require ad hoc heuristics for a new domain. Section 11.4 extends the representation language to allow for hierarchical actions, allowing us to tackle more complex problems. We cover efficient algorithms for planning in Section 11.2, and heuristics for them in Section 11.3. In Section 11.5 we account for partially observable and nondeterministic domains, and in Section 11.6 we extend the language once again to cover scheduling problems with resource constraints. This gets us closer to planners that are used in the real world for planning and scheduling the operations of spacecraft, factories, and military campaigns. Section 11.7 analyzes the effectiveness of these techniques.

11.1 Definition of Classical Planning

Classical planning is defined as the task of finding a sequence of actions to accomplish a goal in a discrete, deterministic, static, fully observable environment. We have seen two approaches to this task: the problem-solving agent of Chapter 3 and the hybrid propositional logical agent of Chapter 7. Both share two limitations. First, they both require ad hoc heuristics for each new domain: a heuristic evaluation function for search, and hand-written code for the hybrid wumpus agent. Second, they both need to explicitly represent an exponentially large state space. For example, in the propositional logic model of the wumpus world, the axiom for moving a step forward had to be repeated for all four agent orientations, T time steps, and n^2 current locations.

In response to these limitations, planning researchers have invested in a **factored representation** using a family of languages called **PDDL**, the Planning Domain Definition Language (Ghallab *et al.*, 1998), which allows us to express all $4Tn^2$ actions with a single action schema, and does not need domain-specific knowledge. Basic PDDL can handle classical planning domains, and extensions can handle non-classical domains that are continuous, partially observable, concurrent, and multi-agent. The syntax of PDDL is based on Lisp, but we will translate it into a form that matches the notation used in this book.

In PDDL, a **state** is represented as a conjunction of ground atomic fluents. Recall that “ground” means no variables, “fluent” means an aspect of the world that changes over time, and “ground atomic” means there is a single predicate, and if there are any arguments, they must be constants. For example, $Poor \wedge Unknown$ might represent the state of a hapless agent, and $At(Truck_1, Melbourne) \wedge At(Truck_2, Sydney)$ could represent a state in a package delivery problem. PDDL uses **database semantics**: the closed-world assumption means that any fluents that are not mentioned are false, and the unique names assumption means that $Truck_1$ and $Truck_2$ are distinct.

The following fluents are *not* allowed in a state: $At(x,y)$ (because it has variables), $\neg Poor$(because it is a negation), and $At\ (Spouse\ (Ali), Sydney)$ (because it uses a function symbol, *Spouse*). When convenient, we can think of the conjunction of fluents as a *set* of fluents.

An **action schema** represents a family of ground actions. For example, here is an action schema for flying a plane from one location to another:

$$
\begin{aligned}
&Action(Fly(p, from, to),\\
&\quad \text{PRECOND}: At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)\\
&\quad \text{EFFECT}: \neg At(p, from) \wedge At(p,\ to))
\end{aligned}
$$

The schema consists of the action name, a list of all the variables used in the schema, a **precondition** and an **effect**. The precondition and the effect are each conjunctions of literals (positive or negated atomic sentences). We can choose constants to instantiate the variables, yielding a ground (variable-free) action:

$$\begin{aligned}&Action(Fly(P_1, SFO, JFK),\\ &\quad \text{PRECOND}: At(P_1, SFO) \wedge Plane(P_1) \wedge Airport(SFO) \wedge Airport(JFK)\\ &\quad \text{EFFECT}: \neg At(P_1, SFO) \wedge At(P_1, JFK))\end{aligned}$$

A ground action *a* is **applicable** in state *s* if *s* entails the precondition of *a*; that is, if every positive literal in the precondition is in *s* and every negated literal is not.

The **result** of executing applicable action *a* in state *s* is defined as a state *s′* which is represented by the set of fluents formed by starting with *s*, removing the fluents that appear as negative literals in the action's effects (what we call the **delete list** or DEL(*a*)), and adding the fluents that are positive literals in the action's effects (what we call the **add list** or ADD(*a*)):

$$\text{RESULT}(s,\ a) = (s - \text{DEL}(a)) \cup \text{ADD}(a). \qquad (11.1)$$

For example, with the action $Fly(P_1, SFO, JFK)$, we would remove the fluent $At(P_1, SFO)$ and add the fluent $At(P_1, JFK)$.

A set of action schemas serves as a definition of a planning *domain.* A specific *problem* within the domain is defined with the addition of an initial state and a goal. The **initial state** is a conjunction of ground fluents (introduced with the keyword *Init* in Figure 11.1). As with all states, the closed-world assumption is used, which means that any atoms that are not mentioned are false. The **goal** (introduced with *Goal)* is just like a precondition: a conjunction of literals (positive or negative) that may contain variables. For example, the goal $At(C_1, SFO) \wedge \neg At(C_2, SFO) \wedge At(p, SFO)$, refers to any state in which cargo C_1 is at *SFO* but C_2 is not, and in which there is a plane at *SFO*.

$$
\begin{aligned}
&\mathit{Init}(\mathit{At}(C_1,\ \mathit{SFO}) \land \mathit{At}(C_2,\ \mathit{JFK}) \land \mathit{At}(P_1,\ \mathit{SFO}) \land \mathit{At}(P_2,\ \mathit{JFK})\\
&\qquad \land \mathit{Cargo}(C_1) \land \mathit{Cargo}(C_2) \land \mathit{Plane}(P_1) \land \mathit{Plane}(P_2)\\
&\qquad \land \mathit{Airport}(\mathit{JFK}) \land \mathit{Airport}(\mathit{SFO}))\\
&\mathit{Goal}(\mathit{At}(C_1,\ \mathit{JFK}) \land \mathit{At}(C_2,\ \mathit{SFO}))\\
&\mathit{Action}(\mathit{Load}(c,\ p,\ a),\\
&\quad \text{PRECOND: } \mathit{At}(c,\ a) \land \mathit{At}(p,\ a) \land \mathit{Cargo}(c) \land \mathit{Plane}(p) \land \mathit{Airport}(a)\\
&\quad \text{EFFECT: } \neg \mathit{At}(c,\ a) \land \mathit{In}(c,\ p))\\
&\mathit{Action}(\mathit{Unload}(c,\ p,\ a),\\
&\quad \text{PRECOND: } \mathit{In}(c,\ p) \land \mathit{At}(p,\ a) \land \mathit{Cargo}(c) \land \mathit{Plane}(p) \land \mathit{Airport}(a)\\
&\quad \text{EFFECT: } \mathit{At}(c,\ a) \land \neg \mathit{In}(c,\ p))\\
&\mathit{Action}(\mathit{Fly}(p,\ \mathit{from},\ \mathit{to}),\\
&\quad \text{PRECOND: } \mathit{At}(p,\ \mathit{from}) \land \mathit{Plane}(p) \land \mathit{Airport}(\mathit{from}) \land \mathit{Airport}(\mathit{to})\\
&\quad \text{EFFECT: } \neg \mathit{At}(p,\ \mathit{from}) \land \mathit{At}(p,\ \mathit{to}))
\end{aligned}
$$

Figure 11.1 A PDDL description of an air cargo transportation planning problem.

11.1.1 Example domain: Air cargo transport

Figure 11.1 shows an air cargo transport problem involving loading and unloading cargo and flying it from place to place. The problem can be defined with three actions: *Load*, *Unload*, and *Fly*. The actions affect two predicates: *In*(c, p) means that cargo c is inside plane p, and *At*(x, a) means that object x (either plane or cargo) is at airport a. Note that some care must be taken to make sure the *At* predicates are maintained properly. When a plane flies from one airport to another, all the cargo inside the plane goes with it. In first-order logic it would be easy to quantify over all objects that are inside the plane. But PDDL does not have a universal quantifier, so we need a different solution. The approach we use is to say that a piece of cargo ceases to be *At* anywhere when it is *In* a plane; the cargo only becomes At the new airport when it is unloaded. So *At* really means “available for use *at* a given location.” The following plan is a solution to the problem:

$$
\begin{aligned}
&[\mathit{Load}(C_1, P_1, \mathit{SFO}), \mathit{Fly}(P_1, \mathit{SFO}, \mathit{JFK}), \mathit{Unload}(C_1, P_1, \mathit{JFK}),\\
&\ \mathit{Load}(C_2, P_2, \mathit{JFK}), \mathit{Fly}(P_2, \mathit{JFK}, \mathit{SFO}), \mathit{Unload}(C_2, P_2, \mathit{SFO})].
\end{aligned}
$$

11.1.2 Example domain: The spare tire problem

Consider the problem of changing a flat tire (Figure 11.2). The goal is to have a good spare tire properly mounted onto the car’s axle, where the initial state has a flat tire on the axle and a good spare tire in the trunk. To keep it simple, our version of the problem is an abstract one, with no sticky lug nuts or other complications. There are just four actions: removing the spare from the trunk, removing the flat tire from the axle, putting the spare on the axle, and leaving the car unattended overnight. We assume that the car is parked in a particularly bad neighborhood, so that the effect of leaving it overnight is that the tires disappear. [*Remove*(*Flat, Axle*), *Remove*(*Spare, Trunk*), *PutOn*(*Spare*,*Axle*)] is a solution to the problem.

$$
\begin{array}{l}
Init(Tire(Flat) \land Tire(Spare) \land At(Flat, Axle) \land At(Spare, Trunk)) \\
Goal(At(Spare, Axle)) \\
Action(Remove(obj, loc), \\
\quad \textsc{Precond}: At(obj, loc) \\
\quad \textsc{Effect}: \neg At(obj, loc) \land At(obj, Ground)) \\
Action(PutOn(t,\ Axle), \\
\quad \textsc{Precond}: Tire(t) \land At(t, Ground) \land \neg At(Flat, Axle) \land \neg At(Spare, Axle) \\
\quad \textsc{Effect}: \neg At(t, Ground) \land At(t, Axle)) \\
Action(LeaveOvernight, \\
\quad \textsc{Precond}: \\
\quad \textsc{Effect}: \neg At(Spare, Ground) \land \neg At(Spare, Axle) \land \neg At(Spare, Trunk) \\
\qquad\qquad \land \neg At(Flat, Ground) \land \neg At(Flat, Axle) \land \neg At(Flat,\ Trunk))
\end{array}
$$

Figure 11.2 The simple spare tire problem.

11.1.3 Example domain: The blocks world

One of the most famous planning domains is the **blocks world**. This domain consists of a set of cube-shaped blocks sitting on an arbitrarily-large table.[1] The blocks can be stacked, but only one block can fit directly on top of another. A robot arm can pick up a block and move it to another position, either on the table or on top of another block. The arm can pick up only one block at a time, so it cannot pick up a block that has another one on top of it. A typical goal to get block *A* on *B* and block *B* on *C* (see Figure 11.3).

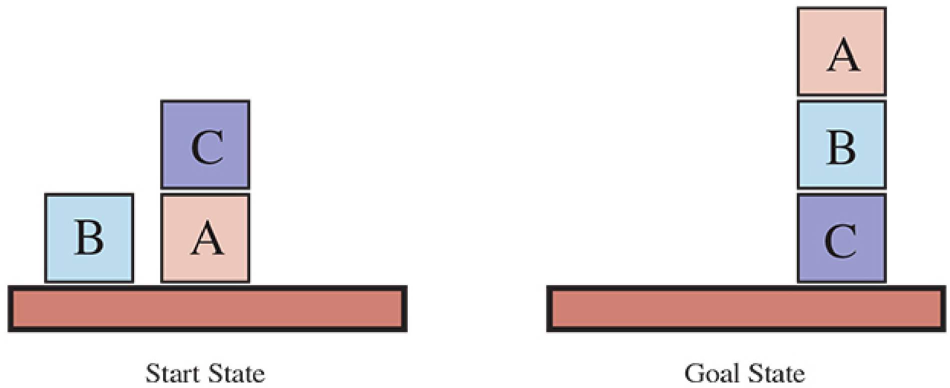


Figure 11.3 Diagram of the blocks-world problem in Figure 11.4.

$Init(On(A, Table) \land On(B, Table) \land On(C, A)$
$\quad \land Block(A) \land Block(B) \land Block(C) \land Clear(B) \land Clear(C) \land Clear(Table))$
$Goal(On(A, B) \land On(B, C))$
$Action(Move(b, x, y),$
$\quad$ PRECOND: $On(b, x) \land Clear(b) \land Clear(y) \land Block(b) \land Block(y) \land$
$\qquad\qquad (b \neq x) \land (b \neq y) \land (x \neq y),$
$\quad$ EFFECT: $On(b, y) \land Clear(x) \land \neg On(b, x) \land \neg Clear(y))$
$Action(MoveToTable(b, x),$
$\quad$ PRECOND: $On(b, x) \land Clear(b) \land Block(b) \land Block(x),$
$\quad$ EFFECT: $On(b, Table) \land Clear(x) \land \neg On(b, x))$

Figure 11.4 A planning problem in the blocks world: building a three-block tower. One solution is the sequence [*MoveToTable(C,A)*,*Move(B, Table,C)*,*Move(A, Table, B)*].

We use *On*(*b*, *x*) to indicate that block *b* is on *x*, where *x* is either another block or the table. The action for moving block *b* from the top of *x* to the top of *y* will be *Move*(*b*, *x*, *y*). Now, one of the preconditions on moving *b* is that no other block be on it. In first-order logic, this would be ¬∃x *On*(*x*, *b*) or, alternatively, ∀*x* ¬*On*(*x*, *b*). Basic PDDL does not allow quantifiers, so instead we introduce a predicate *Clear*(*x*) that is true when nothing is on *x*. (The complete problem description is in Figure 11.4.)

The action *Move* moves a block *b* from *x* to *y* if both *b* and *y* are clear. After the move is made, *b* is still clear but *y* is not. A first attempt at the *Move* schema is

$$\begin{aligned}&Action(Move(b, x, y),\\&\quad \text{PRECOND}: On(b, x) \land Clear(b) \land Clear(y),\\&\quad \text{EFFECT}: On(b, y) \land Clear(x) \land \neg On(b, x) \land \neg Clear(y)).\end{aligned}$$

Unfortunately, this does not maintain *Clear* properly when *x* or *y* is the table. When *x* is the *Table,* this action has the effect *Clear*(*Table*), but the table should not become clear; and when *y* = *Table,* it has the precondition *Clear*(*Table*), but the table does not have to be clear for us to move a block onto it. To fix this, we do two things. First, we introduce another action to move a block *b* from *x* to the table:

$$\begin{aligned}&Action(MoveToTable(b, x),\\&\quad \text{PRECOND}: On(b, x) \land Clear(b),\\&\quad \text{EFFECT}: On(b, Table) \land Clear(x) \land \neg On(b, x)).\end{aligned}$$

Second, we take the interpretation of *Clear*(*x*) to be "there is a clear space on *x* to hold a block." Under this interpretation, *Clear*(*Table*) will always be true. The only problem is that nothing prevents the planner from using *Move*(*b, x, Table*) instead of *MoveToTable*(*b, x*). We could live with this problem—it will lead to a larger-than-necessary search space, but will not lead to incorrect answers—or we could introduce the predicate *Block* and add *Block*(*b*) ∧ *Block*(*y*) to the precondition of *Move,* as shown in Figure 11.4.

11.2 Algorithms for Classical Planning

The description of a planning problem provides an obvious way to search from the initial state through the space of states, looking for a goal. A nice advantage of the declarative representation of action schemas is that we can also search backward from the goal, looking for the initial state (Figure 11.5 compares forward and backward searches). A third possibility is to translate the problem description into a set of logic sentences, to which we can apply a logical inference algorithm to find a solution.

(a)

(b)

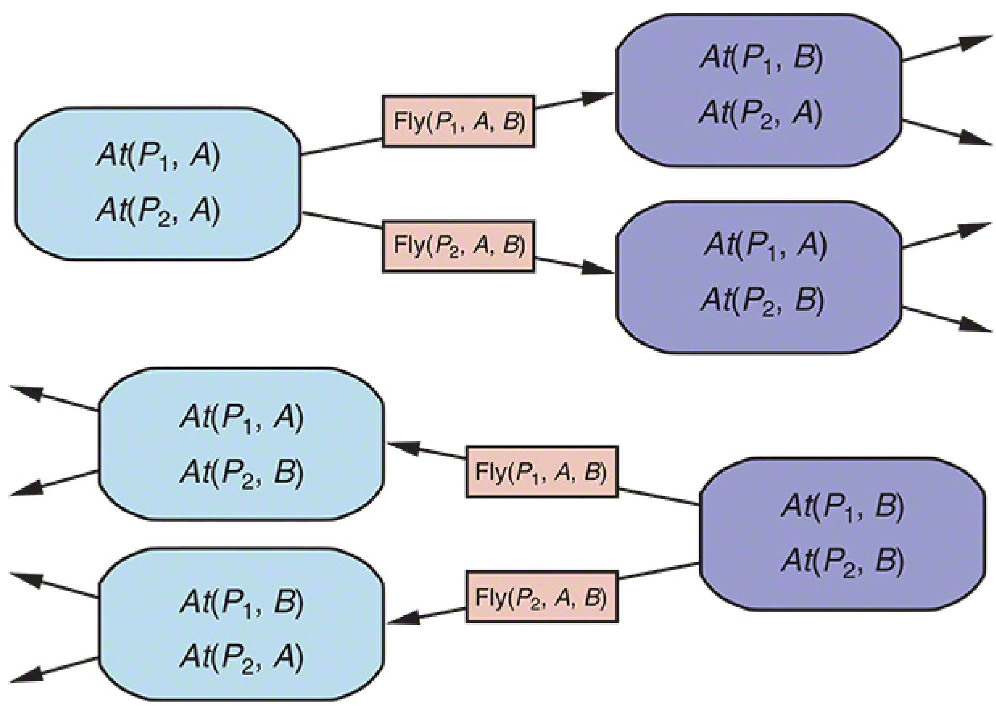


Figure 11.5 Two approaches to searching for a plan. (a) Forward (progression) search through the space of ground states, starting in the initial state and using the problem's actions to search forward for a member of the set of goal states. (b) Backward (regression) search through state descriptions, starting at the goal and using the inverse of the actions to search backward for the initial state.

11.2.1 Forward state-space search for planning

We can solve planning problems by applying any of the heuristic search algorithms from Chapter 3 or Chapter 4. The states in this search state space are ground states, where every fluent is either true or not. The goal is a state that has all the positive fluents in the problem's goal and none of the negative fluents. The applicable actions in a state, *Actions*(s), are grounded instantiations of the action schemas—that is, actions where the variables have all been replaced by constant values.

To determine the applicable actions we unify the current state against the preconditions of each action schema. For each unification that successfully results in a substitution, we apply the substitution to the action schema to yield a ground action with no variables. (It is a requirement of action schemas that any variable in the effect must also appear in the precondition; that way, we are guaranteed that no variables remain after the substitution.)

Each schema may unify in multiple ways. In the spare tire example (page 364), the *Remove* action has the precondition *At*(*obj*, *loc*), which matches against the initial state in two ways, resulting in the two substitutions {*obj/Flat*, *loc/Axle*} and {*obj/Spare*,*loc/Trunk*}; applying these substitutions yields two ground actions. If an action has multiple literals in the precondition, then each of them can potentially be matched against the current state in multiple ways.

At first, it seems that the state space might be too big for many problems. Consider an air cargo problem with 10 airports, where each airport initially has 5 planes and 20 pieces of cargo. The goal is to move all the cargo at airport *A* to airport *B*. There is a 41-step solution to the problem: load the 20 pieces of cargo into one of the planes at *A*, fly the plane to *B*, and unload the 20 pieces.

Finding this apparently straightforward solution can be difficult because the average branching factor is huge: each of the 50 planes can fly to 9 other airports, and each of the 200 packages can be either unloaded (if it is loaded) or loaded into any plane at its airport (if it is unloaded). So in any state there is a minimum of 450 actions (when all the packages are at airports with no planes) and a maximum of 10,450 (when all packages and planes are at the same airport). On average, let's say there are about 2000 possible actions per state, so the search graph up to the depth of the 41-step solution has about 2000^{41} nodes.

Clearly, even this relatively small problem instance is hopeless without an accurate heuristic. Although many real-world applications of planning have relied on domain-specific heuristics, it turns out (as we see in Section 11.3) that strong domain-independent heuristics can be derived automatically; that is what makes forward search feasible.

11.2.2 Backward search for planning

In backward search (also called **regression search**) we start at the goal and apply the actions backward until we find a sequence of steps that reaches the initial state. At each step we consider **relevant actions** (in contrast to forward search, which considers actions that are **applicable**). This reduces the branching factor significantly, particularly in domains with many possible actions.

A relevant action is one with an effect that **unifies** with one of the goal literals, but with no effect that negates any part of the goal. For example, with the goal ¬*Poor* ∧ *Famous*, an action with the sole effect *Famous* would be relevant, but one with the effect *Poor* ∧ *Famous* is not considered relevant: even though that action might be used at some point in the plan (to establish *Famous*), it cannot appear at *this* point in the plan because then *Poor* would appear in the final state.

What does it mean to apply an action in the backward direction? Given a goal g and an action a, the **regression** from g over a gives us a state description g' whose positive and negative literals are given by

$$\begin{aligned} \mathrm{POS}(g\prime) &= (\mathrm{POS}(g) - \mathrm{ADD}(a)) \cup \mathrm{POS}(Precond(a)) \\ \mathrm{NEG}(g\prime) &= (\mathrm{NEG}(g) - \mathrm{DEL}(a)) \cup \mathrm{NEG}(Precond(a)). \end{aligned}$$

That is, the preconditions must have held before, or else the action could not have been executed, but the positive/negative literals that were added/deleted by the action need not have been true before.

These equations are straightforward for ground literals, but some care is required when there are variables in g and a. For example, suppose the goal is to deliver a specific piece of cargo to SFO: $At(C_2, SFO)$. The *Unload* action schema has the effect $At(c, a)$. When we unify that with the goal, we get the substitution $\{c/C_2, a/SFO\}$; applying that substitution to the schema gives us a new schema which captures the idea of using any plane that is at SFO:

$$\begin{aligned} &Action(Unload(C_2,\ p\prime, SFO). \\ &\quad \mathrm{PRECOND}: In(C_2, p\prime) \land At(p\prime, SFO) \land Cargo(C_2) \land Plane(p\prime) \land Airport(SFO) \\ &\quad \mathrm{EFFECT}: At(C_2, SFO) \land \neg In(C_2, p\prime)). \end{aligned}$$

Here we replaced p with a new variable named p'. This is an instance of **standardizing apart** variable names so there will be no conflict between different variables that happen to have the same name (see page 302). The regressed state description gives us a new goal:

$$g\prime = In(C_2, p\prime) \land At(p\prime, SFO) \land Cargo(C_2) \land Plane(p\prime) \land Airport(SFO).$$

As another example, consider the goal of owning a book with a specific ISBN number: *Own*(9780134610993). Given a trillion 13-digit ISBNs and the single action schema

$$A = Action(Buy(i), \text{PRECOND}: ISBN(i), \text{EFFECT}: Own(i)).$$

a forward search without a heuristic would have to start enumerating the 10 billion ground *Buy* actions. But with backward search, we would unify the goal *Own*(9780134610993) with the effect *Own(i')*, yielding the substitution $\theta = \{i'/9780134610993\}$. Then we would regress over the action *Subst*(θ, *A*) to yield the predecessor state description *ISBN*(9780134610993). This is part of the initial state, so we have a solution and we are done, having considered just one action, not a trillion.

More formally, assume a goal description g that contains a goal literal g_i and an action schema A. If A has an effect literal e'_j where $Unify(g_i, e'_j) = \theta$ and where we define $A' = \text{SUBST}(\theta, A)$ and if there is no effect in A' that is the negation of a literal in g, then A' is a relevant action towards g.

For most problem domains backward search keeps the branching factor lower than forward search. However, the fact that backward search uses states with variables rather than ground states makes it harder to come up with good heuristics. That is the main reason why the majority of current systems favor forward search.

11.2.3 Planning as Boolean satisfiability

In Section 7.7.4 we showed how some clever axiom-rewriting could turn a wumpus world problem into a propositional logic satisfiability problem that could be handed to an efficient satisfiability solver. SAT-based planners such as SATPLAN operate by translating a PDDL problem description into propositional form. The translation involves a series of steps:

- Propositionalize the actions: for each action schema, form ground propositions by substituting constants for each of the variables. So instead of a single *Unload*(c, p, a) schema, we would have separate action propositions for each combination of cargo, plane, and airport (here written with subscripts), and for each time step (here written as a superscript).
- Add action exclusion axioms saying that no two actions can occur at the same time, e.g. $\neg(FlyP_1SFOJFK^1 \wedge FlyP_1SFOBUH^1)$.
- Add precondition axioms: For each ground action A^t, add the axiom $A^t \Rightarrow \text{PRE}(A)^t$, that is, if an action is taken at time t, then the preconditions must have been true. For example, $FlyP_1SFOJFK^1 \Rightarrow At(P_1, SFO) \wedge Plane(P_1) \wedge Airport(SFO) \wedge Airport(JFK)$.

- Define the initial state: assert F^0 for every fluent F in the problem's initial state, and $\neg F^0$ for every fluent not mentioned in the initial state.
- Propositionalize the goal: the goal becomes a disjunction over all of its ground instances, where variables are replaced by constants. For example, the goal of having block A on another block, $On(A, x) \land Block(x)$ in a world with objects A, B and C, would be replaced by the goal

$$(On(A,A) \land Block(A)) \lor (On(A,B) \land Block(B)) \lor (On(A,C) \land Block(C)).$$

- Add successor-state axioms: For each fluent F, add an axiom of the form

$$F^{t+1} \Leftrightarrow ActionCausesF^t \lor (F^t \land \neg ActionCausesNotF^t).$$

  where *ActionCausesF* stands for a disjunction of all the ground actions that add F, and *ActionCausesNotF* stands for a disjunction of all the ground actions that delete F.

The resulting translation is typically much larger than the original PDDL, but the efficiency of modern SAT solvers often more than makes up for this.

11.2.4 Other classical planning approaches

The three approaches we covered above are not the only ones tried in the 50-year history of automated planning. We briefly describe some others here.

An approach called Graphplan uses a specialized data structure, a **planning graph**, to encode constraints on how actions are related to their preconditions and effects, and on which things are mutually exclusive.

Situation calculus is a method of describing planning problems in first-order logic. It uses successor-state axioms just as SATPLAN does, but first-order logic allows for more flexibility and more succinct axioms. Overall the approach has contributed to our theoretical understanding of planning, but has not made a big impact in practical applications, perhaps because first-order provers are not as well developed as propositional satisfiability programs.

It is possible to encode a bounded planning problem (i.e., the problem of finding a plan of length k) as a **constraint satisfaction problem** (CSP). The encoding is similar to the encoding to a SAT problem (Section 11.2.3), with one important simplification: at each time step we need only a single variable, $Action^t$, whose domain is the set of possible actions. We no longer need one variable for every action, and we don't need the action exclusion axioms.

All the approaches we have seen so far construct *totally ordered* plans consisting of strictly linear sequences of actions. But if an air cargo problem has 30 packages being loaded onto one plane and 50 packages being loaded onto another, it seems pointless to decree a specific linear ordering of the 80 load actions.

An alternative called **partial-order planning** represents a plan as a graph rather than a linear sequence: each action is a node in the graph, and for each precondition of the action there is an edge from another action (or from the initial state) that indicates that the predecessor action establishes the precondition. So we could have a partial-order plan that says that actions *Remove*(*Spare, Trunk*) and *Remove*(*Flat, Axle*) must come before *PutOn*(*Spare, Axle*), but without saying which of the two *Remove* actions should come first. We search in the space of plans rather than world-states, inserting actions to satisfy conditions.

In the 1980s and 1990s, partial-order planning was seen as the best way to handle planning problems with independent subproblems. By 2000, forward-search planners had developed excellent heuristics that allowed them to efficiently discover the independent subproblems that partial-order planning was designed for. Moreover, SATP LAN was able to take advantage of Moore's law: a propositionalization that was hopelessly large in 1980 now looks tiny, because computers have 10,000 times more memory today. As a result, partial-order planners are not competitive on fully automated classical planning problems.

Nonetheless, partial-order planning remains an important part of the field. For some specific tasks, such as operations scheduling, partial-order planning with domain-specific heuristics is the technology of choice. Many of these systems use libraries of high-level plans, as described in Section 11.4.

Partial-order planning is also often used in domains where it is important for humans to understand the plans. For example, operational plans for spacecraft and Mars rovers are generated by partial-order planners and are then checked by human operators before being uploaded to the vehicles for execution. The plan refinement approach makes it easier for the humans to understand what the planning algorithms are doing and to verify that the plans are correct before they are executed.

11.3 Heuristics for Planning

Neither forward nor backward search is efficient without a good heuristic function. Recall from Chapter 3 that a heuristic function $h(s)$ estimates the distance from a state s to the goal, and that if we can derive an **admissible** heuristic for this distance—one that does not overestimate—then we can use A* search to find optimal solutions.

By definition, there is no way to analyze an atomic state, and thus it requires some ingenuity by an analyst (usually human) to define good domain-specific heuristics for search problems with atomic states. But planning uses a factored representation for states and actions, which makes it possible to define good domain-independent heuristics.

Recall that an admissible heuristic can be derived by defining a **relaxed problem** that is easier to solve. The exact cost of a solution to this easier problem then becomes the heuristic for the original problem. A search problem is a graph where the nodes are states and the edges are actions. The problem is to find a path connecting the initial state to a goal state. There are two main ways we can relax this problem to make it easier: by adding more edges to the graph, making it strictly easier to find a path, or by grouping multiple nodes together, forming an abstraction of the state space that has fewer states, and thus is easier to search.

We look first at heuristics that add edges to the graph. Perhaps the simplest is the **ignore-preconditions heuristic**, which drops all preconditions from actions. Every action becomes applicable in every state, and any single goal fluent can be achieved in one step (if there are any applicable actions—if not, the problem is impossible). This almost implies that the number of steps required to solve the relaxed problem is the number of unsatisfied goals— almost but not quite, because (1) some action may achieve multiple goals and (2) some actions may undo the effects of others.

For many problems an accurate heuristic is obtained by considering (1) and ignoring (2). First, we relax the actions by removing all preconditions and all effects except those that are literals in the goal. Then, we count the minimum number of actions required such that the union of those actions' effects satisfies the goal. This is an instance of the **set-cover problem**. There is one minor irritation: the set-cover problem is NP-hard. Fortunately a simple greedy algorithm is guaranteed to return a set covering whose size is within a factor of log *n* of the true minimum covering, where *n* is the number of literals in the goal. Unfortunately, the greedy algorithm loses the guarantee of admissibility.

It is also possible to ignore only *selected* preconditions of actions. Consider the sliding- tile puzzle (8-puzzle or 15-puzzle) from Section 3.2. We could encode this as a planning problem involving tiles with a single schema *Slide:*

$$\begin{aligned}&Action(Slide(t,\ s_1,\ s_2).\\&\quad \text{PRECOND}: On(t, s_1) \wedge Tile(t) \wedge Blank(s_2) \wedge Adjacent(s_1, s_2)\\&\quad \text{EFFECT}: On(t, s_2) \wedge Blank(s_1) \wedge \neg On(t, s_1) \wedge \neg Blank(s_2))\end{aligned}$$

As we saw in Section 3.6, if we remove the preconditions $Blank(s_2) \wedge Adjacent(s_1, s_2)$ then any tile can move in one action to any space and we get the number-of-misplaced-tiles heuristic. If we remove only the $Blank(s_2)$ precondition then we get the Manhattan-distance heuristic. It is easy to see how these heuristics could be derived automatically from the action schema description. The ease of manipulating the action schemas is the great advantage of the factored representation of planning problems, as compared with the atomic representation of search problems.

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

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

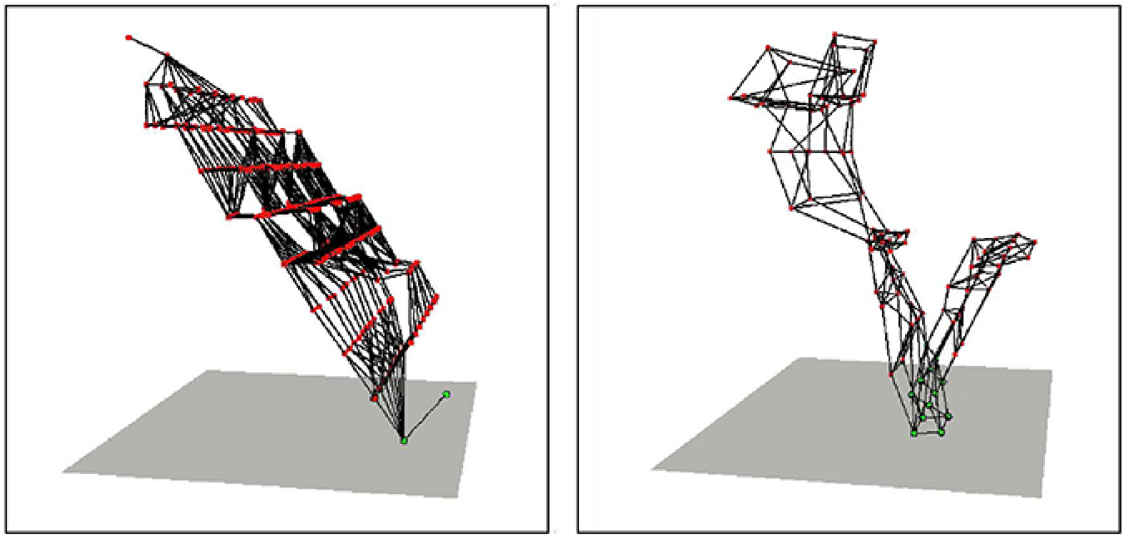


Figure 11.6 Two state spaces from planning problems with the ignore-delete-lists heuristic. The height above the bottom plane is the heuristic score of a state; states on the bottom plane are goals. There are no local minima, so search for the goal is straightforward. From Hoffmann (2005).

11.3.1 Domain-independent pruning

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

Another possibility is to do forward pruning, accepting the risk that we might prune away an optimal solution, in order to focus the search on promising branches. We can define a **preferred action** as follows: First, define a relaxed version of the problem, and solve it to get a **relaxed plan**. Then a preferred action is either a step of the relaxed plan, or it achieves some precondition of the relaxed plan.

Sometimes it is possible to solve a problem efficiently by recognizing that negative interactions can be ruled out. We say that a problem has **serializable subgoals** if there exists an order of subgoals such that the planner can achieve them in that order without having to undo any of the previously achieved subgoals. For example, in the blocks world, if the goal is to build a tower (e.g., *A* on *B*, which in turn is on *C*, which in turn is on the *Table*, as in Figure 11.3 on page 365), then the subgoals are serializable bottom to top: if we first achieve *C* on *Table*, we will never have to undo it while we are achieving the other subgoals. A planner that uses the bottom-to-top trick can solve any problem in the blocks world without backtracking (although it might not always find the shortest plan). As another example, if there is a room with *n* light switches, each controlling a separate light, and the goal is to have them all on, then we don't have to consider permutations of the order; we could arbitrarily restrict ourselves to plans that flip switches in, say, ascending order.

For the Remote Agent planner that commanded NASA's Deep Space One spacecraft, it was determined that the propositions involved in commanding a spacecraft are serializable. This is perhaps not too surprising, because a spacecraft is *designed* by its engineers to be as easy as possible to control (subject to other constraints). Taking advantage of the serialized ordering of goals, the Remote Agent planner was able to eliminate most of the search. This meant that it was fast enough to control the spacecraft in real time, something previously considered impossible.

11.3.2 State abstraction in planning

A relaxed problem leaves us with a simplified planning problem just to calculate the value of the heuristic function. Many planning problems have 10^{100} states or more, and relaxing the *actions* does nothing to reduce the number of states, which means that it may still be expensive to compute the heuristic. Therefore, we now look at relaxations that decrease the number of states by forming a **state abstraction**—a many-to-one mapping from states in the ground representation of the problem to the abstract representation.

The easiest form of state abstraction is to ignore some fluents. For example, consider an air cargo problem with 10 airports, 50 planes, and 200 pieces of cargo. Each plane can be at one of 10 airports and each package can be either in one of the planes or unloaded at one of the airports. So there are $10^{50} \times (50 + 10)^{200} \approx 10^{405}$ states. Now consider a particular problem in that domain in which it happens that all the packages are at just 5 of the airports, and all packages at a given airport have the same destination. Then a useful abstraction of the problem is to drop all the *At* fluents except for the ones involving one plane and one package at each of the 5 airports. Now there are only $10^5 \times (5 + 10)^5 \approx 10^{11}$ states. A solution in this abstract state space will be shorter than a solution in the original space (and thus will be an admissible heuristic), and the abstract solution is easy to extend to a solution to the original problem (by adding additional *Load* and *Unload* actions).

A key idea in defining heuristics is **decomposition**: dividing a problem into parts, solving each part independently, and then combining the parts. The **subgoal independence** assumption is that the cost of solving a conjunction of subgoals is approximated by the sum of the costs of solving each subgoal *independently.* The subgoal independence assumption can be optimistic or pessimistic. It is optimistic when there are negative interactions between the subplans for each subgoal—for example, when an action in one subplan deletes a goal achieved by another subplan. It is pessimistic, and therefore inadmissible, when subplans contain redundant actions—for instance, two actions that could be replaced by a single action in the merged plan.

Suppose the goal is a set of fluents G, which we divide into disjoint subsets $G_1,\ldots, G_n$. We then find optimal plans $P_1,\ldots, P_n$ that solve the respective subgoals. What is an estimate of the cost of the plan for achieving all of G? We can think of each $\text{COST}(P_i)$ as a heuristic estimate, and we know that if we combine estimates by taking their maximum value, we always get an admissible heuristic. So $\max_i \text{COST}(P_i)$ is admissible, and sometimes it is exactly correct: it could be that P_1 serendipitously achieves all the G_i. But usually the estimate is too low. Could we sum the costs instead? For many problems that is a reasonable estimate, but it is not admissible. The best case is when G_i and G_j are independent, in the sense that plans for one cannot reduce the cost of plans for the other. In that case, the estimate $\text{COST}(P_i) + \text{COST}(P_j)$ is admissible, and more accurate than the max estimate.

It is clear that there is great potential for cutting down the search space by forming abstractions. The trick is choosing the right abstractions and using them in a way that makes the total cost—defining an abstraction, doing an abstract search, and mapping the abstraction back to the original problem—less than the cost of solving the original problem. The techniques of **pattern databases** from Section 3.6.3 can be useful, because the cost of creating the pattern database can be amortized over multiple problem instances.

A system that makes use of effective heuristics is FF, or FASTFORWARD (Hoffmann, 2005), a forward state-space searcher that uses the ignore-delete-lists heuristic, estimating the heuristic with the help of a planning graph. FF then uses hill climbing search (modified to keep track of the plan) with the heuristic to find a solution. FF's hill climbing algorithm is nonstandard: it avoids local maxima by running a breadth-first search from the current state until a better one is found. If this fails, FF switches to a greedy best-first search instead.

11.4 Hierarchical Planning

The problem-solving and planning methods of the preceding chapters all operate with a fixed set of atomic actions. Actions can be strung together, and state-of-the-art algorithms can generate solutions containing thousands of actions. That's fine if we are planning a vacation and the actions are at the level of "fly from san Francisco to Honolulu," but at the motorcontrol level of "bend the left knee by 5 degrees" we would need to string together millions or billions of actions, not thousands.

Bridging this gap requires planning at higher levels of abstraction. A high-level plan for a Hawaii vacation might be "Go to san Francisco airport; take flight HA 11 to Honolulu; do vacation stuff for two weeks; take HA 12 back to san Francisco; go home." Given such a plan, the action "Go to san Francisco airport" can be viewed as a planning task in itself, with a solution such as "Choose a ride-hailing service; order a car; ride to airport." Each of these actions, in turn, can be decomposed further, until we reach the low-level motor control actions like a button-press.

In this example, planning and acting are interleaved; for example, one would defer the problem of planning the walk from the curb to the gate until after being dropped off. Thus, that particular action will remain at an abstract level prior to the execution phase. We defer discussion of this topic until Section 11.5. Here, we concentrate on the idea of **hierarchical decomposition**, an idea that pervades almost all attempts to manage complexity. For example, complex software is created from a hierarchy of subroutines and classes; armies, governments and corporations have organizational hierarchies. The key benefit of hierarchical structure is that at each level of the hierarchy, a computational task, military mission, or administrative function is reduced to a *small* number of activities at the next lower level, so the computational cost of finding the correct way to arrange those activities for the current problem is small.

11.4.1 High-level actions

The basic formalism we adopt to understand hierarchical decomposition comes from the area of **hierarchical task networks** or HTN planning. For now we assume full observability and determinism and a set of actions, now called **primitive actions**, with standard precondition-effect schemas. The key additional concept is the **high-level action** or HLA—for example, the action "Go to San Francisco airport." Each HLA has one or more possible **refinements**, into a sequence of actions, each of which may be an HLA or a primitive action. For example, the action “Go to San Francisco airport,” represented formally as *Go*(*Home, SFO*), might have two possible refinements, as shown in Figure 11.7. The same figure shows a **recursive** refinement for navigation in the vacuum world: to get to a destination, take a step, and then go to the destination.

$Refinement(Go(Home, SFO),$
  $\text{STEPS: } [Drive(Home, SFOLongTermParking),$
      $Shuttle(SFOLongTermParking, SFO)]\ )$
$Refinement(Go(Home, SFO),$
  $\text{STEPS: } [Taxi(Home, SFO)]\ )$

$Refinement(Navigate([a, b], [x, y]),$
  $\text{PRECOND: } a = x \wedge b = y$
  $\text{STEPS: } [\,]\ )$
$Refinement(Navigate([a, b], [x, y]),$
  $\text{PRECOND:} Connected([a, b], [a-1, b])$
  $\text{STEPS: } [Left, Navigate([a-1, b], [x, y])]\ )$
$Refinement(Navigate([a, b], [x, y]),$
  $\text{PRECOND:} Connected([a, b], [a+1, b])$
  $\text{STEPS: } [Right, Navigate([a+1, b], [x, y])]\ )$
...

Figure 11.7 Definitions of possible refinements for two high-level actions: going to San Francisco airport and navigating in the vacuum world. In the latter case, note the recursive nature of the refinements and the use of preconditions.

These examples show that high-level actions and their refinements embody knowledge about *how to do things.* For instance, the refinements for *Go*(*Home, SFO*) say that to get to the airport you can drive or take a ride-hailing service; buying milk, sitting down, and moving the knight to e4 are not to be considered.

An HLA refinement that contains only primitive actions is called an **implementation** of the HLA. In a grid world, the sequences [*Right, Right, Down*] and [*Down, Right, Right*] both implement the HLA *Navigate*([1,3], [3,2]). An implementation of a high-level plan (a sequence of HLAs) is the concatenation of implementations of each HLA in the sequence. Given the precondition-effect definitions of each primitive action, it is straightforward to determine whether any given implementation of a high-level plan achieves the goal.

We can say, then, that *a high-level plan achieves the goal from a given state if at least one of its implementations achieves the goal from that state.* The "at least one" in this definition is crucial—not *all* implementations need to achieve the goal, because the agent gets to decide which implementation it will execute. Thus, the set of possible implementations in HTN planning—each of which may have a different outcome—is not the same as the set of possible outcomes in nondeterministic planning. There, we required that a plan work for *all* outcomes because the agent doesn't get to choose the outcome; nature does.

The simplest case is an HLA that has exactly one implementation. In that case, we can compute the preconditions and effects of the HLA from those of the implementation (see Exercise 11.HLAU) and then treat the HLA exactly as if it were a primitive action itself. It can be shown that the right collection of HLAs can result in the time complexity of blind search dropping from exponential in the solution depth to linear in the solution depth, although devising such a collection of HLAs may be a nontrivial task in itself. When HLAs have multiple possible implementations, there are two options: one is to search among the implementations for one that works, as in Section 11.4.2; the other is to reason directly about the HLAs—despite the multiplicity of implementations—as explained in Section 11.4.3. The latter method enables the derivation of provably correct abstract plans, without the need to consider their implementations.

11.4.2 Searching for primitive solutions

HTN planning is often formulated with a single "top level" action called *Act*, where the aim is to find an implementation of *Act* that achieves the goal. This approach is entirely general. For example, classical planning problems can be defined as follows: for each primitive action a_i, provide one refinement of *Act* with steps $[a_i, Act]$. That creates a recursive definition of *Act* that lets us add actions. But we need some way to stop the recursion; we do that by providing one more refinement for *Act*, one with an empty list of steps and with a precondition equal to the goal of the problem. This says that if the goal is already achieved, then the right implementation is to do nothing.

The approach leads to a simple algorithm: repeatedly choose an HLA in the current plan and replace it with one of its refinements, until the plan achieves the goal. One possible implementation based on breadth-first tree search is shown in Figure 11.8. Plans are considered in order of depth of nesting of the refinements, rather than number of primitive steps. It is straightforward to design a graph-search version of the algorithm as well as depth-first and iterative deepening versions.

```
function HIERARCHICAL-SEARCH(problem, hierarchy) returns a solution or failure
  frontier ← a FIFO queue with [Act] as the only element
  while true do
     if IS-EMPTY(frontier) then return failure
     plan ← POP(frontier)          // chooses the shallowest plan in frontier
     hla ← the first HLA in plan, or null if none
     prefix,suffix ← the action subsequences before and after hla in plan
     outcome ← RESULT(problem.INITIAL, prefix)
     if hla is null then          // so plan is primitive and outcome is its result
        if problem.IS-GOAL(outcome)  then return plan
     else for each sequence in REFINEMENTS(hla, outcome, hierarchy) do
        add APPEND(prefix, sequence, suffix) to frontier
```

Figure 11.8 A breadth-first implementation of hierarchical forward planning search. The initial plan supplied to the algorithm is [*Act*]. The REFINEMENTS function returns a set of action sequences, one for each refinement of the HLA whose preconditions are satisfied by the specified state, *outcome.*

In essence, this form of hierarchical search explores the space of sequences that conform to the knowledge contained in the HLA library about how things are to be done. A great deal of knowledge can be encoded, not just in the action sequences specified in each refinement but also in the preconditions for the refinements. For some domains, HTN planners have been able to generate huge plans with very little search. For example, O-PLAN (Bell and Tate, 1985), which combines HTN planning with scheduling, has been used to develop production plans for Hitachi. A typical problem involves a product line of 350 different products, 35 assembly machines, and over 2000 different operations. The planner generates a 30-day schedule with three 8-hour shifts a day, involving tens of millions of steps. Another important aspect of HTN plans is that they are, by definition, hierarchically structured; usually this makes them easy for humans to understand.

The computational benefits of hierarchical search can be seen by examining an idealized case. Suppose that a planning problem has a solution with d primitive actions. For a nonhierarchical, forward state-space planner with b allowable actions at each state, the cost is $O(b^d)$, as explained in Chapter 3. For an HTN planner, let us suppose a very regular refinement structure: each nonprimitive action has r possible refinements, each into k actions at the next lower level. We want to know how many different refinement trees there are with this structure. Now, if there are d actions at the primitive level, then the number of levels below the root is $\log_k{}^d$, so the number of internal refinement nodes is $1 + k + k^2 + \cdots + k^{\log_k d-1} = (d - 1)/(k - 1)$. Each internal node has r possible refinements, so $r^{(d-1)/(k-1)}$ possible decomposition trees could be constructed.

Examining this formula, we see that keeping r small and k large can result in huge savings: we are taking the k th root of the nonhierarchical cost, if b and r are comparable. Small r and large k means a library of HLAs with a small number of refinements each yielding a long action sequence. This is not always possible: long action sequences that are usable across a wide range of problems are extremely rare.

The key to HTN planning is a plan library containing known methods for implementing complex, high-level actions. One way to construct the library is to *learn* the methods from problem-solving experience. After the excruciating experience of constructing a plan from scratch, the agent can save the plan in the library as a method for implementing the high-level action defined by the task. In this way, the agent can become more and more competent over time as new methods are built on top of old methods. One important aspect of this learning process is the ability to *generalize* the methods that are constructed, eliminating detail that is specific to the problem instance (e.g., the name of the builder or the address of the plot of land) and keeping just the key elements of the plan. It seems to us inconceivable that humans could be as competent as they are without some such mechanism.

11.4.3 Searching for abstract solutions

The hierarchical search algorithm in the preceding section refines HLAs all the way to primitive action sequences to determine if a plan is workable. This contradicts common sense: one should be able to determine that the two-HLA high-level plan

$$[Drive(Home, SFOLongTermParking),\ Shuttle(SFOLongTermParking, SFO)]$$

gets one to the airport without having to determine a precise route, choice of parking spot, and so on. The solution is to write precondition-effect descriptions of the HLAs, just as we do for primitive actions. From the descriptions, it ought to be easy to prove that the high-level plan achieves the goal. This is the holy grail, so to speak, of hierarchical planning, because if we derive a high-level plan that provably achieves the goal, working in a small search space of high-level actions, then we can commit to that plan and work on the problem of refining each step of the plan. This gives us the exponential reduction we seek.

For this to work, it has to be the case that every high-level plan that "claims" to achieve the goal (by virtue of the descriptions of its steps) does in fact achieve the goal in the sense defined earlier: it must have at least one implementation that does achieve the goal. This property has been called the **downward refinement property** for HLA descriptions.

Writing HLA descriptions that satisfy the downward refinement property is, in principle, easy: as long as the descriptions are *true*, then any high-level plan that claims to achieve the goal must in fact do so—otherwise, the descriptions are making some false claim about what the HLAs do. We have already seen how to write true descriptions for HLAs that have exactly one implementation (Exercise 11.HLAU); a problem arises when the HLA has *multiple* implementations. How can we describe the effects of an action that can be implemented in many different ways?

One safe answer (at least for problems where all preconditions and goals are positive) is to include only the positive effects that are achieved by *every* implementation of the HLA and the negative effects of *any* implementation. Then the downward refinement property would be satisfied. Unfortunately, this semantics for HLAs is much too conservative.

Consider again the HLA *Go*(*Home*, *SFO*), which has two refinements, and suppose, for the sake of argument, a simple world in which one can always drive to the airport and park, but taking a taxi requires *Cash* as a precondition. In that case, *Go*(*Home*, *SFO*) doesn't always get you to the airport. In particular, it fails if *Cash* is false, and so we cannot assert *At*(*Agent*, *SFO*) as an effect of the HLA. This makes no sense, however; if the agent didn't have *Cash*, it would drive itself. Requiring that an effect hold for *every* implementation is equivalent to assuming that *someone else*—an adversary—will choose the implementation. It treats the HLA's multiple outcomes exactly as if the HLA were a **nondeterministic** action, as in Section 4.3. For our case, the agent itself will choose the implementation.

The programming languages community has coined the term **demonic nondeterminism** for the case where an adversary makes the choices, contrasting this with **angelic nondeterminism**, where the agent itself makes the choices. We borrow this term to define **angelic semantics** for HLA descriptions. The basic concept required for understanding angelic semantics is the **reachable set** of an HLA: given a state *s*, the reachable set for an HLA *h*, written as REACH(*s*, *h*) is the set of states reachable by any of the HLA's implementations.

The key idea is that the agent can choose *which* element of the reachable set it ends up in when it executes the HLA; thus, an HLA with multiple refinements is more "powerful" than the same HLA with fewer refinements. We can also define the reachable set of a sequence of HLAs. For example, the reachable set of a sequence $[h_1, h_2]$ is the union of all the reachable sets obtained by applying h_2 in each state in the reachable set of h_1:

$$\text{REACH}(s, [h_1, h_2]) = \bigcup_{s' \in \text{ REACH}(s,\, h_1)} \text{REACH}(s', h_2).$$

Given these definitions, a high-level plan—a sequence of HLAs—achieves the goal if its reachable set *intersects* the set of goal states. (Compare this to the much stronger condition for demonic semantics, where every member of the reachable set has to be a goal state.) Conversely, if the reachable set doesn't intersect the goal, then the plan definitely doesn't work. Figure 11.9 illustrates these ideas.

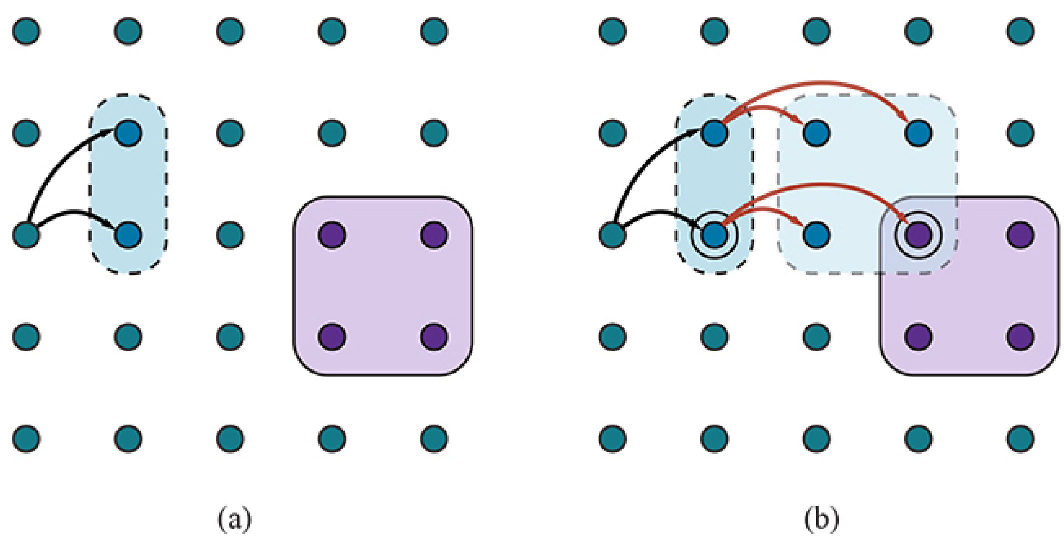


Figure 11.9 Schematic examples of reachable sets. The set of goal states is shaded in purple. Black and red arrows indicate possible implementations of h_1 and h_2, respectively. (a) The reachable set of an HLA h_1 in a state s. (b) The reachable set

for the sequence $[h_1, h_2]$. Because this intersects the goal set, the sequence achieves the goal.

The notion of reachable sets yields a straightforward algorithm: search among high- level plans, looking for one whose reachable set intersects the goal; once that happens, the algorithm can *commit* to that abstract plan, knowing that it works, and focus on refining the plan further. We will return to the algorithmic issues later; for now consider how the effects of an HLA—the reachable set for each possible initial state—are represented. A primitive action can set a fluent to *true* or *false* or leave it *unchanged.* (With conditional effects (see Section 11.5.1) there is a fourth possibility: flipping a variable to its opposite.)

An HLA under angelic semantics can do more: it can *control* the value of a fluent, setting it to true or false depending on which implementation is chosen. That means that an HLA can have nine different effects on a fluent: if the variable starts out true, it can always keep it true, always make it false, or have a choice; if the fluent starts out false, it can always keep it false, always make it true, or have a choice; and the three choices for both cases can be combined arbitrarily, making nine.

Notationally, this is a bit challenging. We'll use the language of add lists and delete lists (rather than true/false fluents) along with the $\tilde{}$ symbol to mean "possibly, if the agent so chooses." Thus, the effect $\tilde{+}A$ means "possibly add A," that is, either leave A unchanged or make it true. Similarly, $\tilde{-}A$ means "possibly delete A" and $\tilde{\pm}A$ means "possibly add or delete A." For example, the HLA *Go*(*Home*, *SFO*), with the two refinements shown in Figure 11.7, possibly deletes *Cash* (if the agent decides to take a taxi), so it should have the effect $\tilde{-}$*Cash.* Thus, we see that the descriptions of HLAs are *derivable* from the descriptions of their refinements. Now, suppose we have the following schemas for the HLAs h_1 and h_2:

$$Action(h_1, \text{PRECOND:} \neg A, \text{EFFECT} : A \wedge \simeq B),$$

$$Action(h_2, \text{ PRECOND:}\neg B, \text{ EFFECT} : \tilde{+}A \wedge \tilde{\pm}C).$$

That is, h_1 adds A and possibly deletes B, while h_2 possibly adds A and has full control over C. Now, if only B is true in the initial state and the goal is $A \wedge C$ then the sequence $[h_1, h_2]$ achieves the goal: we choose an implementation of h_1 that makes B false, then choose an implementation of h_2 that leaves A true and makes C true.

The preceding discussion assumes that the effects of an HLA—the reachable set for any given initial state—can be described exactly by describing the effect on each fluent. It would be nice if this were always true, but in many cases we can only approximate the effects because an HLA may have infinitely many implementations and may produce arbitrarily wiggly reachable sets—rather like the wiggly-belief-state problem illustrated in Figure 7.21 on page 261. For example, we said that *Go*(*Home, SFO*) possibly deletes *Cash;* it also possibly adds *At*(*Car, SFOLongTermParking*); but it cannot do both—in fact, it must do exactly one. As with belief states, we may need to write *approximate* descriptions. We will use two kinds of approximation: an **optimistic description** $\text{REACH}^{+}(s, h)$ of an HLA *h* may overstate the reachable set, while a **pessimistic description** $\text{REACH}^{-}(s, h)$ may understate the reachable set. Thus, we have

$$\text{REACH}^{-}(s, h) \subseteq \text{REACH}(s, h) \subseteq \text{REACH}^{+}(s, h) .$$

For example, an optimistic description of *Go*(*Home, SFO*) says that it possibly deletes *Cash and* possibly adds *At*(*Car,SFOLongTermParking*). Another good example arises in the 8-puzzle, half of whose states are unreachable from any given state (see Exercise 11.PART): the optimistic description of *Act* might well include the whole state space, since the exact reachable set is quite wiggly.

With approximate descriptions, the test for whether a plan achieves the goal needs to be modified slightly. If the optimistic reachable set for the plan doesn't intersect the goal, then the plan doesn't work; if the pessimistic reachable set intersects the goal, then the plan does work (Figure 11.10(a)). With exact descriptions, a plan either works or it doesn't, but with approximate descriptions, there is a middle ground: if the optimistic set intersects the goal but the pessimistic set doesn't, then we cannot tell if the plan works (Figure 11.10(b)). When this circumstance arises, the uncertainty can be resolved by refining the plan. This is a very common situation in human reasoning. For example, in planning the aforementioned two-week Hawaii vacation, one might propose to spend two days on each of seven islands. Prudence would indicate that this ambitious plan needs to be refined by adding details of inter-island transportation.

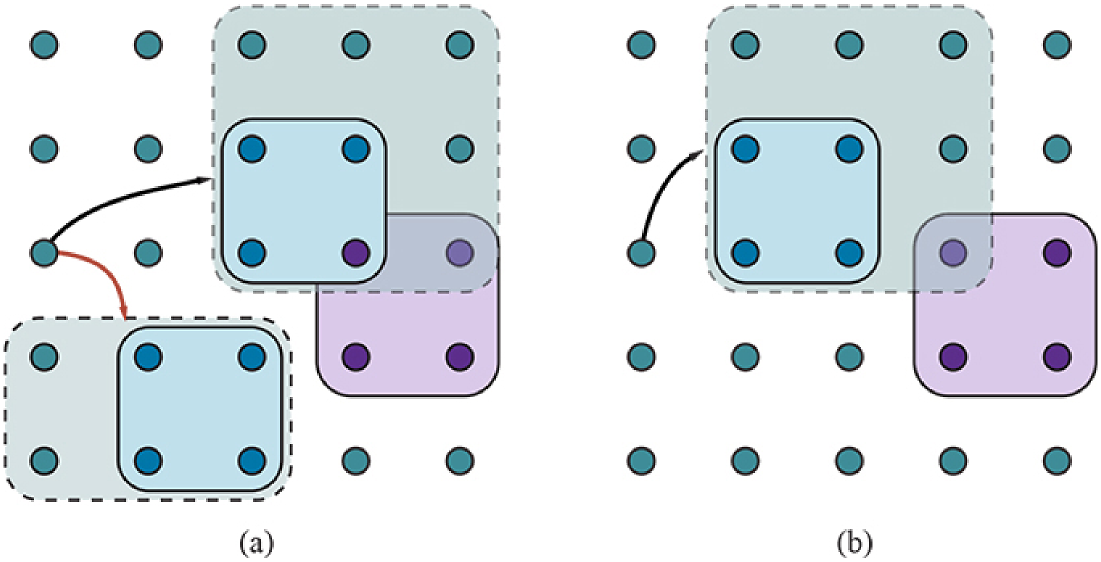


Figure 11.10 Goal achievement for high-level plans with approximate descriptions. The set of goal states is shaded in purple. For each plan, the pessimistic (solid lines, light blue) and optimistic (dashed lines, light green) reachable sets are shown. (a) The plan indicated by the black arrow definitely achieves the goal, while the plan indicated by the red arrow definitely doesn't. (b) A plan that *possibly* achieves the goal (the optimistic reachable set intersects the goal) but does not *necessarily* achieve the goal (the pessimistic reachable set does not intersect the goal). The plan would need to be refined further to determine if it really does achieve the goal.

An algorithm for hierarchical planning with approximate angelic descriptions is shown in Figure 11.11. For simplicity, we have kept to the same overall scheme used previously in Figure 11.8, that is, a breadth-first search in the space of refinements. As just explained, the algorithm can detect plans that will and won't work by checking the intersections of the optimistic and pessimistic reachable sets with the goal. (The details of how to compute the reachable sets of a plan, given approximate descriptions of each step, are covered in Exercise 11.HLAP.)

```
function ANGELIC-SEARCH(problem, hierarchy, initialPlan) returns a solution or fail
  frontier ← a FIFO queue with initialPlan as the only element
  while true do
     if IS-EMPTY?(frontier) then return fail
     plan ← POP(frontier)        // chooses the shallowest node in frontier
     if REACH⁺(problem.INITIAL, plan) intersects problem.GOAL then
        if plan is primitive then return plan        // REACH⁺ is exact for primitive plans
        guaranteed ← REACH⁻(problem.INITIAL, plan) ∩ problem.GOAL
        if guaranteed≠{ } and MAKING-PROGRESS(plan, initialPlan) then
           finalState ← any element of guaranteed
           return DECOMPOSE(hierarchy, problem.INITIAL, plan, finalState)
        hla ← some HLA in plan
        prefix,suffix ← the action subsequences before and after hla in plan
        outcome ← RESULT(problem.INITIAL, prefix)
        for each sequence in REFINEMENTS(hla, outcome, hierarchy) do
           add APPEND(prefix, sequence, suffix) to frontier

function DECOMPOSE(hierarchy, s_0, plan, s_f) returns a solution
  solution ← an empty plan
  while plan is not empty do
    action ← REMOVE-LAST(plan)
    s_i ← a state in REACH⁻(s_0, plan) such that s_f∈REACH⁻(s_i, action)
    problem ← a problem with INITIAL = s_i and GOAL = s_f
    solution ← APPEND(ANGELIC-SEARCH(problem, hierarchy, action), solution)
    s_f ← s_i
  return solution
```

Figure 11.11 A hierarchical planning algorithm that uses angelic semantics to identify and commit to high-level plans that work while avoiding high-level plans that don't. The predicate MAKING-PROGRESS checks to make sure that we aren't stuck in an infinite regression of refinements. At top level, call ANGELIC-SEARCH with [*Act*] as the *initialPlan*.

When a workable abstract plan is found, the algorithm *decomposes* the original problem into subproblems, one for each step of the plan. The initial state and goal for each subproblem are obtained by regressing a guaranteed-reachable goal state through the action schemas for each step of the plan. (See Section 11.2.2 for a discussion of how regression works.) Figure 11.9(b) illustrates the basic idea: the right-hand circled state is the guaranteed-reachable goal state, and the left-hand circled state is the intermediate goal obtained by regressing the goal through the final action.

The ability to commit to or reject high-level plans can give ANGELIC-SEARCH a significant computational advantage over HIERARCHICAL-SEARCH, which in turn may have a large advantage over plain old BREADTH-FIRST-SEARCH. Consider, for example, cleaning up a large vacuum world consisting of an arrangement of rooms connected by narrow corridors, where each room is a $w \times h$ rectangle of squares. It makes sense to have an HLA for *Navigate* (as shown in Figure 11.7) and one for *CleanWholeRoom.* (Cleaning the room could be implemented with the repeated application of another HLA to clean each row.) Since there are five primitive actions, the cost for BREADTH-FIRST-S EARCH grows as 5^d, where d is the length of the shortest solution (roughly twice the total number of squares); the algorithm cannot manage even two 3×3 rooms. HIERARCHICAL-SEARCH is more efficient, but still suffers from exponential growth because it tries all ways of cleaning that are consistent with the hierarchy. ANGELIC-SEARCH scales approximately linearly in the number of squares—it commits to a good high-level sequence of room-cleaning and navigation steps and prunes away the other options.

Cleaning a set of rooms by cleaning each room in turn is hardly rocket science: it is easy for humans because of the hierarchical structure of the task. When we consider how difficult humans find it to solve small puzzles such as the 8-puzzle, it seems likely that the human capacity for solving complex problems derives not from considering combinatorics, but rather from skill in abstracting and decomposing problems to eliminate combinatorics.

The angelic approach can be extended to find least-cost solutions by generalizing the notion of reachable set. Instead of a state being reachable or not, each state will have a cost for the most efficient way to get there. (The cost is infinite for unreachable states.) The optimistic and pessimistic descriptions bound these costs. In this way, angelic search can find provably optimal abstract plans without having to consider their implementations. The same approach can be used to obtain effective **hierarchical look-ahead** algorithms for online search, in the style of LRTA* (page 158).

In some ways, such algorithms mirror aspects of human deliberation in tasks such as planning a vacation to Hawaii—consideration of alternatives is done initially at an abstract level over long time scales; some parts of the plan are left quite abstract until execution time, such as how to spend two lazy days on Moloka‘i, while others parts are planned in detail, such as the flights to be taken and lodging to be reserved—without these latter refinements, there is no guarantee that the plan would be feasible.

11.5 Planning and Acting in Nondeterministic Domains

In this section we extend planning to handle partially observable, nondeterministic, and unknown environments. The basic concepts mirror those in Chapter 4, but there are differences arising from the use of factored representations rather than atomic representations. This affects the way we represent the agent's capability for action and observation and the way we represent **belief states**—the sets of possible physical states the agent might be in—for partially observable environments. We can also take advantage of many of the domainindependent methods given in Section 11.3 for calculating search heuristics.

We will cover **sensorless planning** (also known as **conformant planning**) for environments with no observations; **contingency planning** for partially observable and nondeterministic environments; and **online planning** and **replanning** for unknown environments. This will allow us to tackle sizable real-world problems.

Consider this problem: given a chair and a table, the goal is to have them match—have the same color. In the initial state we have two cans of paint, but the colors of the paint and the furniture are unknown. Only the table is initially in the agent's field of view:

$$Init(Object(Table) \land Object(Chair) \land Can(C_1) \land Can(C_2) \land InView(Table))$$
$$Goal(Color(Chair, c) \land Color(Table, c))$$

There are two actions: removing the lid from a paint can and painting an object using the paint from an open can.

$$\begin{aligned}
&\mathrm{Action}(RemoveLid(can),\\
&\quad \mathrm{PRECOND:}Can(can)\\
&\quad \mathrm{EFFECT:}Open(can))\\
&Action(Paint\,(x, can),\\
&\quad \mathrm{PRECOND}: Object(x) \land Can(can) \land Color(can, c) \land Open(can)\\
&\quad \mathrm{EFFECT:}Color(x, c))
\end{aligned}$$

The action schemas are straightforward, with one exception: preconditions and effects now may contain variables that are not part of the action's variable list. That is, *Paint(x, can)* does not mention the variable *c*, representing the color of the paint in the can. In the fully observable case, this is not allowed—we would have to name the action *Paint(x, can,* *c)*. But in the partially observable case, we might or might not know what color is in the can.

To solve a partially observable problem, the agent will have to reason about the percepts it will obtain when it is executing the plan. The percept will be supplied by the agent's sensors when it is actually acting, but when it is planning it will need a model of its sensors. In Chapter 4, this model was given by a function, PERCEPT(*s*). For planning, we augment PDDL with a new type of schema, the **percept schema**:

$$
\begin{aligned}
&Percept(Color(x, c), \\
&\quad \text{PRECOND}: Object(x) \land InView(x) \\
&Percept(Color(can, c), \\
&\quad \text{PRECOND}: Can(can) \land InView(can) \land Open(can))
\end{aligned}
$$

The first schema says that whenever an object is in view, the agent will perceive the color of the object (that is, for the object *x*, the agent will learn the truth value of *Color*(*x*, *c*) for all *c*). The second schema says that if an open can is in view, then the agent perceives the color of the paint in the can. Because there are no exogenous events in this world, the color of an object will remain the same, even if it is not being perceived, until the agent performs an action to change the object's color. Of course, the agent will need an action that causes objects (one at a time) to come into view:

$$
\begin{aligned}
&Action(LookAt(x), \\
&\quad \text{PRECOND}: InView(y) \land (x \neq y) \\
&\quad \text{EFFECT}: InView(x) \land \neg InView(y))
\end{aligned}
$$

For a fully observable environment, we would have a *Percept* schema with no preconditions for each fluent. A sensorless agent, on the other hand, has no *Percept* schemas at all. Note that even a sensorless agent can solve the painting problem. One solution is to open any can of paint and apply it to both chair and table, thus **coercing** them to be the same color (even though the agent doesn't know what the color is).

A contingent planning agent with sensors can generate a better plan. First, look at the table and chair to obtain their colors; if they are already the same then the plan is done. If not, look at the paint cans; if the paint in a can is the same color as one piece of furniture, then apply that paint to the other piece. Otherwise, paint both pieces with any color.

Finally, an online planning agent might generate a contingent plan with fewer branches at first—perhaps ignoring the possibility that no cans match any of the furniture—and deal with problems when they arise by replanning. It could also deal with incorrectness of its action schemas. Whereas a contingent planner simply assumes that the effects of an action always succeed—that painting the chair does the job—a replanning agent would check the result and make an additional plan to fix any unexpected failure, such as an unpainted area or the original color showing through.

In the real world, agents use a combination of approaches. Car manufacturers sell spare tires and air bags, which are physical embodiments of contingent plan branches designed to handle punctures or crashes. On the other hand, most car drivers never consider these possibilities; when a problem arises they respond as replanning agents. In general, agents plan only for contingencies that have important consequences and a nonnegligible chance of happening. Thus, a car driver contemplating a trip across the Sahara desert should make explicit contingency plans for breakdowns, whereas a trip to the supermarket requires less advance planning. We next look at each of the three approaches in more detail.

11.5.1 Sensorless planning

Section 4.4.1 (page 144) introduced the basic idea of searching in belief-state space to find a solution for sensorless problems. Conversion of a sensorless planning problem to a belief- state planning problem works much the same way as it did in Section 4.4.1; the main differences are that the underlying physical transition model is represented by a collection of action schemas, and the belief state can be represented by a logical formula instead of by an explicitly enumerated set of states. We assume that the underlying planning problem is deterministic.

The initial belief state for the sensorless painting problem can ignore *InView* fluents because the agent has no sensors. Furthermore, we take as given the unchanging facts $Object(Table) \wedge Object(Chair) \wedge Can(C_1) \wedge Can(C_2)$ because these hold in every belief state. The agent doesn't know the colors of the cans or the objects, or whether the cans are open or closed, but it does know that objects and cans have colors: $\forall x\ \exists c\ Color(x, c)$. After Skolemizing (see Section 9.5.1), we obtain the initial belief state:

$$b_0 = Color(x, C(x)).$$

In classical planning, where the **closed-world assumption** is made, we would assume that any fluent not mentioned in a state is false, but in sensorless (and partially observable) planning we have to switch to an **open-world assumption** in which states contain both positive and negative fluents, and if a fluent does not appear, its value is unknown. Thus, the belief state corresponds exactly to the set of possible worlds that satisfy the formula. Given this initial belief state, the following action sequence is a solution:

$$[RemoveLid(Can_1), Paint(Chair, Can_1), Paint(Table, Can_1)] \ .$$

We now show how to progress the belief state through the action sequence to show that the final belief state satisfies the goal.

First, note that in a given belief state *b*, the agent can consider any action whose preconditions are satisfied by *b*. (The other actions cannot be used because the transition model doesn't define the effects of actions whose preconditions might be unsatisfied.) According to Equation (4.4) (page 145), the general formula for updating the belief state *b* given an applicable action *a* in a deterministic world is as follows:

$$b\prime \ = \ \mathrm{RESULT}(b,\ a) \ = \ \{s\prime \ : \ s\prime = \mathrm{RESULT}_P(s, a) \text{ and } s \ \in \ b\}$$

where $\textsc{Result}_P$ defines the physical transition model. For the time being, we assume that the initial belief state is always a conjunction of literals, that is, a 1-CNF formula. To construct the new belief state b', we must consider what happens to each literal ℓ in each physical state *s* in *b* when action *a* is applied. For literals whose truth value is already known in *b*, the truth value in b' is computed from the current value and the add list and delete list of the action. (For example, if ℓ is in the delete list of the action, then $\neg\ell$ is added to b'.) What about a literal whose truth value is unknown in *b*? There are three cases:

1. If the action adds ℓ, then ℓ will be true in b' regardless of its initial value.
2. If the action deletes ℓ, then ℓ will be false in b' regardless of its initial value.
3. If the action does not affect ℓ, then ℓ will retain its initial value (which is unknown) and will not appear in b'.

Hence, we see that the calculation of b' is almost identical to the observable case, which was specified by Equation (11.1) on page 363:

$$b\prime = \mathrm{RESULT}(b,\ a) = (b - \mathrm{DEL}(a)) \cup \mathrm{ADD}\,(a).$$

We cannot quite use the set semantics because (1) we must make sure that b' does not contain both ℓ and $\neg\ell$, and (2) atoms may contain unbound variables. But it is still the case that $\textsc{Result}(b, a)$ is computed by starting with *b*, setting any atom that appears in DEL(*a*) to false, and setting any atom that appears in ADD(*a*) to true. For example, if we apply $RemoveLid(Can_1)$ to the initial belief state b_0, we get

$$b_1 = Color(x, C(x)) \wedge Open(Can_1).$$

When we apply the action *Paint*(*Chair,*Can_1), the precondition *Color*(Can_1,*c*) is satisfied by the literal *Color*(*x*, *C*(*x*)) with binding {*x/Cam*, *c/C*(Can_1)} and the new belief state is

$$b_2 = Color(x, C(x)) \wedge Open(Can_1) \wedge Color(Chair, C(Can_1)).$$

Finally, we apply the action *Paint*(*Table*, Can_1) to obtain

$$b_3 = Color(x, C(x)) \wedge Open(Can_1) \wedge Color(Chair, C(Can_1)) \\ \wedge Color(Table, C(Can_1)).$$

The final belief state satisfies the goal, *Color*(*Table, c*) ∧ *Color*(*Chair, c*), with the variable *c* bound to *C*(Can_1).

The preceding analysis of the update rule has shown a very important fact: *the family of belief states defined as conjunctions of literals is closed under updates defined by PDDL action schemas.* That is, if the belief state starts as a conjunction of literals, then any update will yield a conjunction of literals. That means that in a world with *n* fluents, any belief state can be represented by a conjunction of size *O*(*n*). This is a very comforting result, considering that there are 2^n states in the world. It says we can compactly represent all the subsets of those 2^n states that we will ever need. Moreover, the process of checking for belief states that are subsets or supersets of previously visited belief states is also easy, at least in the propositional case.

The fly in the ointment of this pleasant picture is that it only works for action schemas that have the *same effects* for all states in which their preconditions are satisfied. It is this property that enables the preservation of the 1-CNF belief-state representation. As soon as the effect can depend on the state, dependencies are introduced between fluents, and the 1-CNF property is lost.

Consider, for example, the simple vacuum world defined in Section 3.2.1. Let the fluents be *AtL* and *AtR* for the location of the robot and *CleanL* and *CleanR* for the state of the squares. According to the definition of the problem, the *Suck* action has no precondition—it can always be done. The difficulty is that its effect depends on the robot's location: when the robot is *AtL*, the result is *CleanL,* but when it is *AtR,* the result is *CleanR.* For such actions, our action schemas will need something new: a **conditional effect**. These have the syntax "**when** *condition: effect,*" where *condition* is a logical formula to be compared against the current state, and *effect* is a formula describing the resulting state. For the vacuum world:

$$
\begin{aligned}
&Action(Suck,\\
&\quad \text{EFFECT}: \mathbf{when}\ AtL : CleanL \wedge \mathbf{When}\ AtR : CleanR).
\end{aligned}
$$

When applied to the initial belief state *True*, the resulting belief state is $(AtL \wedge CleanL) \vee (AtR \wedge CleanR)$, which is no longer in 1-CNF. (This transition can be seen in Figure 4.14 on page 147.) In general, conditional effects can induce arbitrary dependencies among the fluents in a belief state, leading to belief states of exponential size in the worst case.

It is important to understand the difference between preconditions and conditional effects. *All* conditional effects whose conditions are satisfied have their effects applied to generate the resulting belief state; if none are satisfied, then the resulting state is unchanged. On the other hand, if a *precondition* is unsatisfied, then the action is inapplicable and the resulting state is undefined. From the point of view of sensorless planning, it is better to have conditional effects than an inapplicable action. For example, we could split *Suck* into two actions with unconditional effects as follows:

$$
\begin{aligned}
&Action(SuckL,\\
&\quad \text{PRECOND:}\ AtL;\ \text{EFFECT}: CleanL)\\
&Action(SuckR,\\
&\quad \text{PRECOND:}\ AtR;\ \text{EFFECT}: CleanR).
\end{aligned}
$$

Now we have only unconditional schemas, so the belief states all remain in 1-CNF; unfortunately, we cannot determine the applicability of *SuckL* and *SuckR* in the initial belief state.

It seems inevitable, then, that nontrivial problems will involve wiggly belief states, just like those encountered when we considered the problem of state estimation for the wumpus world (see Figure 7.21 on page 261). The solution suggested then was to use a **conservative approximation** to the exact belief state; for example, the belief state can remain in 1-CNF if it contains all literals whose truth values can be determined and treats all other literals as unknown. While this approach is *sound*, in that it never generates an incorrect plan, it is *incomplete* because it may be unable to find solutions to problems that necessarily involve interactions among literals. To give a trivial example, if the goal is for the robot to be on a clean square, then [*Suck*] is a solution but a sensorless agent that insists on 1-CNF belief states will not find it.

Perhaps a better solution is to look for action sequences that keep the belief state as simple as possible. In the sensorless vacuum world, the action sequence [*Right*, *Suck*, *Left*, *Suck*] generates the following sequence of belief states:

$$
\begin{aligned}
b_0 &= True \\
b_1 &= AtR \\
b_2 &= AtR \wedge CleanR \\
b_3 &= AtL \wedge CleanR \\
b_4 &= AtL \wedge CleanR \wedge CleanL
\end{aligned}
$$

That is, the agent *can* solve the problem while retaining a 1-CNF belief state, even though some sequences (e.g., those beginning with *Suck*) go outside 1-CNF. The general lesson is not lost on humans: we are always performing little actions (checking the time, patting our pockets to make sure we have the car keys, reading street signs as we navigate through a city) to eliminate uncertainty and keep our belief state manageable.

There is another, quite different approach to the problem of unmanageably wiggly belief states: don't bother computing them at all. Suppose the initial belief state is b_0 and we would like to know the belief state resulting from the action sequence $[a_1, \ldots, a_m]$. Instead of computing it explicitly, just represent it as "b_0 then $[a_1, \ldots, a_m]$." This is a lazy but unambiguous representation of the belief state, and it's quite concise—$O(n + m)$ where n is the size of the initial belief state (assumed to be in 1-CNF) and m is the maximum length of an action sequence. As a belief-state representation, it suffers from one drawback, however: determining whether the goal is satisfied, or an action is applicable, may require a lot of computation.

The computation can be implemented as an entailment test: if A_m represents the collection of successor-state axioms required to define occurrences of the actions $a_1, \ldots, a_m$—as explained for SATPLAN in Section 11.2.3—and G_m asserts that the goal is true after m steps, then the plan achieves the goal if $b_0 \wedge A_m \models G_m$—that is, if $b_0 \wedge A_m \wedge \neg G_m$ is unsatisfiable. Given a modern SAT solver, it may be possible to do this much more quickly than computing the full belief state. For example, if none of the actions in the sequence has a particular goal fluent in its add list, the solver will detect this immediately. It also helps if partial results about the belief state—for example, fluents known to be true or false—are cached to simplify subsequent computations.

The final piece of the sensorless planning puzzle is a heuristic function to guide the search. The meaning of the heuristic function is the same as for classical planning: an estimate (perhaps admissible) of the cost of achieving the goal from the given belief state. With belief states, we have one additional fact: solving any subset of a belief state is necessarily easier than solving the belief state:

$$\text{if } b_1 \subseteq b_2 \textit{ then } h^*(b_1) \le h^*(b_2).$$

Hence, any admissible heuristic computed for a subset is admissible for the belief state itself. The most obvious candidates are the singleton subsets, that is, individual physical states. We can take any random collection of states $s_1,\ldots, s_N$ that are in the belief state b, apply any admissible heuristic h, and return

$$H(b) = \max\{h(s_1), \ldots, h(s_N)\}$$

as the heuristic estimate for solving b. We can also use inadmissible heuristics such as the ignore-delete-lists heuristic (page 372), which seems to work quite well in practice.

11.5.2 Contingent planning

We saw in Chapter 4 that contingency planning—the generation of plans with conditional branching based on percepts—is appropriate for environments with partial observability, nondeterminism, or both. For the partially observable painting problem with the percept schemas given earlier, one possible conditional solution is as follows:

$$\begin{array}{l}
[\mathit{LookAt}(\mathit{Table}), \mathit{LookAt}(\mathit{Chair}), \\
\;\;\mathbf{if}\ \mathit{Color}(\mathit{Table}, c) \wedge \mathit{Color}(\mathit{Chair}, c)\,\mathbf{then}\ \mathit{NoOp} \\
\quad\;\; \mathbf{else}[\mathit{RemoveLid}(\mathit{Can}_1), \mathit{LookAt}(\mathit{Can}_1), \mathit{RemoveLid}(\mathit{Can}_2), \mathit{LookAt}(\mathit{Can}_2), \\
\quad\;\; \mathbf{if}\ \mathit{Color}(\mathit{Table}, c) \wedge \mathit{Color}(\mathit{can}, c)\ \mathbf{then}\ \mathit{Paint}(\mathit{Chair}; \mathit{can}) \\
\quad\;\; \mathbf{else\ if}\ \mathit{Color}(\mathit{Chair}, c) \wedge \mathit{Color}(\mathit{can}, c)\,\mathbf{then}\ \mathit{Paint}(\mathit{Table}, \mathit{can}) \\
\quad\;\; \mathbf{else}\ [\mathit{Paint}(\mathit{Chair}, \mathit{Can}_1), \mathit{Paint}(\mathit{Table}, \mathit{Can}_1)]]]
\end{array}$$

Variables in this plan should be considered existentially quantified; the second line says that if there exists some color c that is the color of the table and the chair, then the agent need not do anything to achieve the goal. When executing this plan, a contingent-planning agent can maintain its belief state as a logical formula and evaluate each branch condition by determining if the belief state entails the condition formula or its negation. (It is up to the contingent-planning algorithm to make sure that the agent will never end up in a belief state where the condition formula's truth value is unknown.) Note that with first-order conditions, the formula may be satisfied in more than one way; for example, the condition $\mathit{Color}(\mathit{Table}, c) \wedge \mathit{Color}(\mathit{can}, c)$ might be satisfied by $\{\mathit{can}/\mathit{Can}_1\}$ and by $\{\mathit{can}/\mathit{Can}_2\}$ if both cans are the same color as the table. In that case, the agent can choose any satisfying substitution to apply to the rest of the plan.

As shown in Section 4.4.2, calculating the new belief state $\hat{b}$ after an action a and subsequent percept is done in two stages. The first stage calculates the belief state after the action, just as for the sensorless agent:

$$\hat{b} = (b - \mathrm{DEL}(a)) \cup \mathrm{ADD}(a)$$

where, as before, we have assumed a belief state represented as a conjunction of literals. The second stage is a little trickier. Suppose that percept literals $p_1,\ldots, p_k$ are received. One might think that we simply need to add these into the belief state; in fact, we can also infer that the preconditions for sensing are satisfied. Now, if a percept p has exactly one percept schema, *Percept(p,* PRECOND:*c*), where c is a conjunction of literals, then those literals can be thrown into the belief state along with p. On the other hand, if p has more than one percept schema whose preconditions might hold according to the predicted belief state $\hat{b}$ then we have to add in the *disjunction* of the preconditions. Obviously, this takes the belief state outside 1-CNF and brings up the same complications as conditional effects, with much the same classes of solutions.

Given a mechanism for computing exact or approximate belief states, we can generate contingent plans with an extension of the AND-OR forward search over belief states used in Section 4.4. Actions with nondeterministic effects—which are defined simply by using a disjunction in the EFFECT of the action schema—can be accommodated with minor changes to the belief-state update calculation and no change to the search algorithm.[3] For the heuristic function, many of the methods suggested for sensorless planning are also applicable in the partially observable, nondeterministic case.

11.5.3 Online planning

Imagine watching a spot-welding robot in a car plant. The robot's fast, accurate motions are repeated over and over again as each car passes down the line. Although technically impressive, the robot probably does not seem at all *intelligent* because the motion is a fixed, preprogrammed sequence; the robot obviously doesn't "know what it's doing" in any meaningful sense. Now suppose that a poorly attached door falls off the car just as the robot is about to apply a spot-weld. The robot quickly replaces its welding actuator with a gripper, picks up the door, checks it for scratches, reattaches it to the car, sends an email to the floor supervisor, switches back to the welding actuator, and resumes its work. All of a sudden, the robot's behavior seems *purposive* rather than rote; we assume it results not from a vast, precomputed contingent plan but from an online replanning process—which means that the robot *does* need to know what it's trying to do.

Replanning presupposes some form of **execution monitoring** to determine the need for a new plan. One such need arises when a contingent planning agent gets tired of planning for every little contingency, such as whether the sky might fall on its head.[4] This means that the contingent plan is left in an incomplete form. For example, some branches of a partially constructed contingent plan can simply say *Replan*; if such a branch is reached during execution, the agent reverts to planning mode. As we mentioned earlier, the decision as to how much of the problem to solve in advance and how much to leave to replanning is one that involves tradeoffs among possible events with different costs and probabilities of occurring. Nobody wants to have a car break down in the middle of the Sahara desert and only then think about having enough water.

Replanning may be needed if the agent's model of the world is incorrect. The model for an action may have a **missing precondition**—for example, the agent may not know that removing the lid of a paint can often requires a screwdriver. The model may have a **missing effect**—painting an object may get paint on the floor as well. Or the model may have a **missing fluent** that is simply absent from the representation altogether—for example, the model given earlier has no notion of the amount of paint in a can, of how its actions affect this amount, or of the need for the amount to be nonzero. The model may also lack provision for **exogenous events** such as someone knocking over the paint can. Exogenous events can also include changes in the goal, such as the addition of the requirement that the table and chair not be painted black. Without the ability to monitor and replan, an agent's behavior is likely to be fragile if it relies on absolute correctness of its model.

The online agent has a choice of (at least) three different approaches for monitoring the environment during plan execution:

- **Action monitoring** : before executing an action, the agent verifies that all the preconditions still hold.
- **Plan monitoring** : before executing an action, the agent verifies that the remaining plan will still succeed.
- **Goal monitoring** : before executing an action, the agent checks to see if there is a better set of goals it could be trying to achieve.

In Figure 11.12 we see a schematic of action monitoring. The agent keeps track of both its original plan, *whole plan*, and the part of the plan that has not been executed yet, which is denoted by *plan*. After executing the first few steps of the plan, the agent expects to be in state E. But the agent observes that it is actually in state O. It then needs to repair the plan by finding some point P on the original plan that it can get back to. (It may be that P is the goal state, G.) The agent tries to minimize the total cost of the plan: the repair part (from O to P) plus the continuation (from P to G).

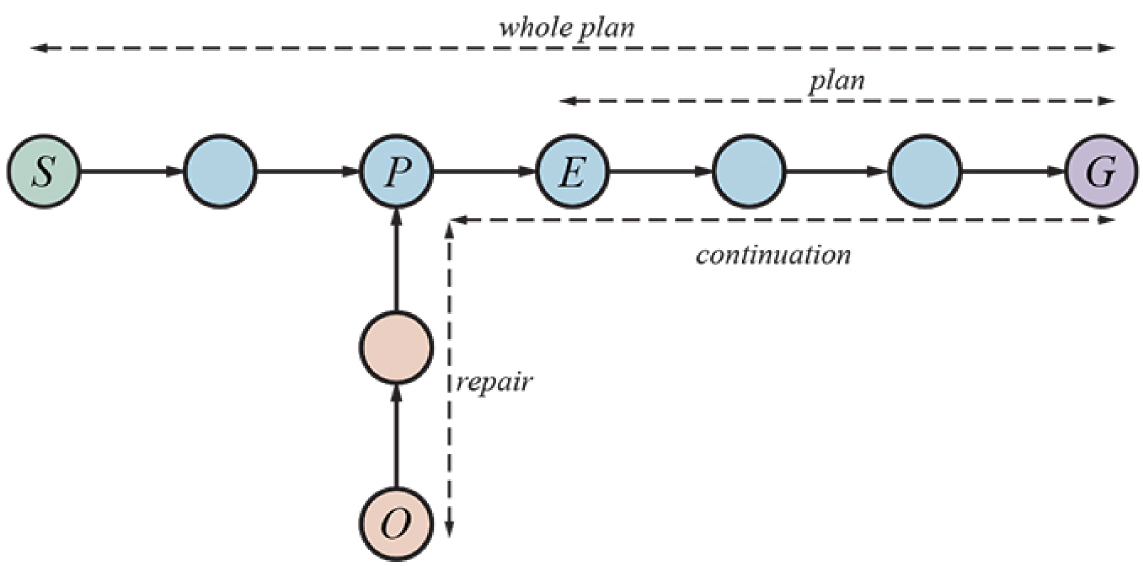


Figure 11.12 At first, the sequence “whole plan” is expected to get the agent from S to G. The agent executes steps of the plan until it expects to be in state E, but observes that it is actually in O. The agent then replans for the minimal *repair* plus *continuation* to reach G.

Now let’s return to the example problem of achieving a chair and table of matching color. Suppose the agent comes up with this plan:

$$
\begin{aligned}
&[LookAt(Table), LookAt(Chair),\\
&\quad \textbf{if } Color(Table,\ c) \wedge Color(Chair, c) \textbf{ then } NoOp\\
&\qquad \textbf{else } [RemoveLid(Can_1), LookAt(Can_1),\\
&\qquad\quad \textbf{if } Color(Table, c) \wedge Color(Can_1, c) \textbf{then } Paint(Chair, Can_1)\\
&\qquad\quad \textbf{else } \text{REPLAN}]]\ .
\end{aligned}
$$

Now the agent is ready to execute the plan. The agent observes that the table and can of paint are white and the chair is black. It then executes *Paint* (*Chair, Can_1*). At this point a classical planner would declare victory; the plan has been executed. But an online execution monitoring agent needs to check that the action succeeded.

Suppose the agent perceives that the chair is a mottled gray because the black paint is showing through. The agent then needs to figure out a recovery position in the plan to aim for and a repair action sequence to get there. The agent notices that the current state is identical to the precondition before the *Paint* (*Chair, Can_1*) action, so the agent chooses the empty sequence for *repair* and makes its *plan* be the same [*Paint*] sequence that it just attempted. With this new plan in place, execution monitoring resumes, and the *Paint* action is retried. This behavior will loop until the chair is perceived to be completely painted. But notice that the loop is created by a process of plan-execute-replan, rather than by an explicit loop in a plan. Note also that the original plan need not cover every contingency. If the agent reaches the step marked REPLAN, it can then generate a new plan (perhaps involving Can_2).

Action monitoring is a simple method of execution monitoring, but it can sometimes lead to less than intelligent behavior. For example, suppose there is no black or white paint, and the agent constructs a plan to solve the painting problem by painting both the chair and table red. Suppose that there is only enough red paint for the chair. With action monitoring, the agent would go ahead and paint the chair red, then notice that it is out of paint and cannot paint the table, at which point it would replan a repair—perhaps painting both chair and table green. A plan-monitoring agent can detect failure whenever the current state is such that the remaining plan no longer works. Thus, it would not waste time painting the chair red.

Plan monitoring achieves this by checking the preconditions for success of the entire remaining plan—that is, the preconditions of each step in the plan, except those preconditions that are achieved by another step in the remaining plan. Plan monitoring cuts off execution of a doomed plan as soon as possible, rather than continuing until the failure actually occurs.[5] Plan monitoring also allows for **serendipity**—accidental success. If someone comes along and paints the table red at the same time that the agent is painting the chair red, then the final plan preconditions are satisfied (the goal has been achieved), and the agent can go home early.

It is straightforward to modify a planning algorithm so that each action in the plan is annotated with the action's preconditions, thus enabling action monitoring. It is slightly more complex to enable plan monitoring. Partial-order planners have the advantage that they have already built up structures that contain the relations necessary for plan monitoring. Augmenting state-space planners with the necessary annotations can be done by careful bookkeeping as the goal fluents are regressed through the plan.

Now that we have described a method for monitoring and replanning, we need to ask, "Does it work?" This is a surprisingly tricky question. If we mean, "Can we guarantee that the agent will always achieve the goal?" then the answer is no, because the agent could inadvertently arrive at a dead end from which there is no repair. For example, the vacuum agent might have a faulty model of itself and not know that its batteries can run out. Once they do, it cannot repair any plans. If we rule out dead ends—assume that there exists a plan to reach the goal from *any* state in the environment—and assume that the environment is really nondeterministic, in the sense that such a plan always has *some* chance of success on any given execution attempt, then the agent will eventually reach the goal.

Trouble occurs when a seemingly-nondeterministic action is not actually random, but rather depends on some precondition that the agent does not know about. For example, sometimes a paint can may be empty, so painting from that can has no effect. No amount of retrying is going to change this.[6] One solution is to choose randomly from among the set of possible repair plans, rather than to try the same one each time. In this case, the repair plan of opening another can might work. A better approach is to **learn** a better model. Every prediction failure is an opportunity for learning; an agent should be able to modify its model of the world to accord with its percepts. From then on, the replanner will be able to come up with a repair that gets at the root problem, rather than relying on luck to choose a good repair.

11.6 Time, Schedules, and Resources

Classical planning talks about *what to do,* in *what order,* but does not talk about time: *how long* an action takes and *when* it occurs. For example, in the airport domain we could produce a plan saying what planes go where, carrying what, but could not specify departure and arrival times. This is the subject matter of **scheduling**.

The real world also imposes **resource constraints**: an airline has a limited number of staff, and staff who are on one flight cannot be on another at the same time. This section introduces techniques for planning and scheduling problems with resource constraints.

The approach we take is “plan first, schedule later”: divide the overall problem into a *planning* phase in which actions are selected, with some ordering constraints, to meet the goals of the problem, and a later *scheduling* phase, in which temporal information is added to the plan to ensure that it meets resource and deadline constraints. This approach is common in real-world manufacturing and logistical settings, where the planning phase is sometimes automated, and sometimes performed by human experts.

11.6.1 Representing temporal and resource constraints

A typical **job-shop scheduling problem** (see Section 5.1.2), consists of a set of **jobs**, each of which has a collection of **actions** with ordering constraints among them. Each action has a **duration** and a set of resource constraints required by the action. A constraint specifies a *type* of resource (e.g., bolts, wrenches, or pilots), the number of that resource required, and whether that resource is **consumable** (e.g., the bolts are no longer available for use) or **reusable** (e.g., a pilot is occupied during a flight but is available again when the flight is over). Actions can also produce resources (e.g., manufacturing and resupply actions).

A solution to a job-shop scheduling problem specifies the start times for each action and must satisfy all the temporal ordering constraints and resource constraints. As with search and planning problems, solutions can be evaluated according to a cost function; this can be quite complicated, with nonlinear resource costs, time-dependent delay costs, and so on. For simplicity, we assume that the cost function is just the total duration of the plan, which is called the **makespan**.

Figure 11.13 shows a simple example: a problem involving the assembly of two cars. The problem consists of two jobs, each of the form [*AddEngine*,*AddWheels*,*Inspect*]. Then the *Resources* statement declares that there are four types of resources, and gives the number of each type available at the start: 1 engine hoist, 1 wheel station, 2 inspectors, and 500 lug nuts. The action schemas give the duration and resource needs of each action. The lug nuts are *consumed* as wheels are added to the car, whereas the other resources are “borrowed” at the start of an action and released at the action’s end.

$$
\begin{aligned}
&Jobs(\{AddEngine1 \prec AddWheels1 \prec Inspect1\},\\
&\quad \{AddEngine2 \prec AddWheels2 \prec Inspect2\})\\[1ex]
&Resources(EngineHoists(1),\ WheelStations(1),\ Inspectors(2),\ LugNuts(500))\\[1ex]
&Action(AddEngine1,\ \text{DURATION:}30,\\
&\quad \text{USE:}EngineHoists(1))\\
&Action(AddEngine2,\ \text{DURATION:}60,\\
&\quad \text{USE:}EngineHoists(1))\\
&Action(AddWheels1,\ \text{DURATION:}30,\\
&\quad \text{CONSUME:}LugNuts(20),\ \text{USE:}WheelStations(1))\\
&Action(AddWheels2,\ \text{DURATION:}15,\\
&\quad \text{CONSUME:}LugNuts(20),\ \text{USE:}WheelStations(1))\\
&Action(Inspect_i,\ \text{DURATION:}10,\\
&\quad \text{USE:}Inspectors(1))
\end{aligned}
$$

Figure 11.13 A job-shop scheduling problem for assembling two cars, with resource constraints. The notation $A \prec B$ means that action A must precede action B.

The representation of resources as numerical quantities, such as *Inspectors*(2), rather than as named entities, such as *Inspector*(I_1) and *Inspector*(I_2), is an example of a technique called **aggregation**: grouping individual objects into quantities when the objects are all indistinguishable. In our assembly problem, it does not matter *which* inspector inspects the car, so there is no need to make the distinction. Aggregation is essential for reducing complexity. Consider what happens when a proposed schedule has 10 concurrent *Inspect* actions but only 9 inspectors are available. With inspectors represented as quantities, a failure is detected immediately and the algorithm backtracks to try another schedule. With inspectors represented as individuals, the algorithm would try all 9! ways of assigning inspectors to actions before noticing that none of them work.

11.6.2 Solving scheduling problems

We begin by considering just the temporal scheduling problem, ignoring resource constraints. To minimize makespan (plan duration), we must find the earliest start times for all the actions consistent with the ordering constraints supplied with the problem. It is helpful to view these ordering constraints as a directed graph relating the actions, as shown in Figure 11.14. We can apply the **critical path method** (CPM) to this graph to determine the possible start and end times of each action. A **path** through a graph representing a partial-order plan is a linearly ordered sequence of actions beginning with *Start* and ending with *Finish*. (For example, there are two paths in the partial-order plan in Figure 11.14.)

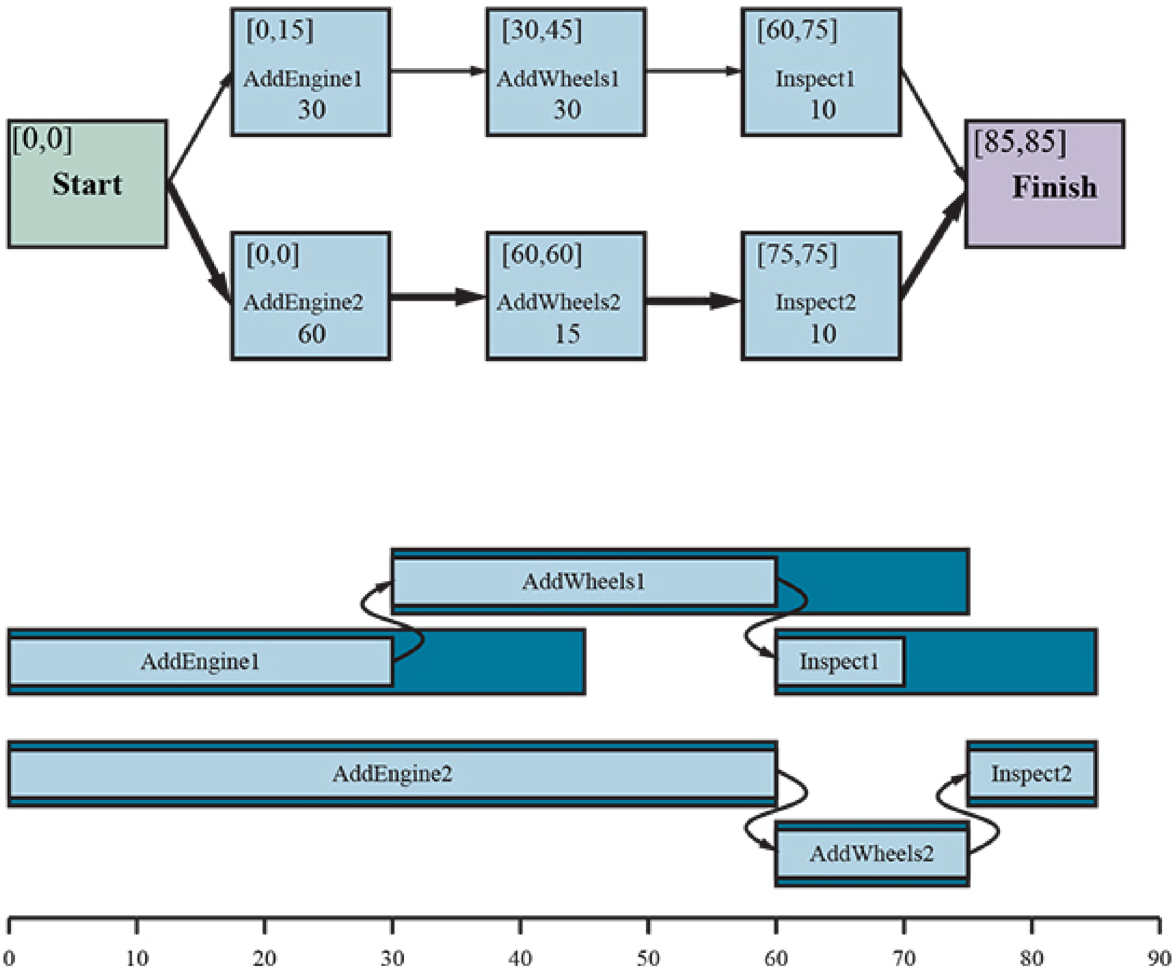


Figure 11.14 Top: a representation of the temporal constraints for the job-shop scheduling problem of Figure 11.13. The duration of each action is given at the bottom of each rectangle. In solving the problem, we compute the earliest and latest start times as the pair [*ES*, *LS*], displayed in the upper left. The difference between these two numbers is the *slack* of an action; actions with zero slack are on the critical path, shown with bold arrows. Bottom: the same solution shown as a timeline. Blue rectangles represent time intervals during which an action may be executed,

provided that the ordering constraints are respected. The unoccupied portion of a blue rectangle indicates the slack.

The **critical path** is that path whose total duration is longest; the path is “critical” because it determines the duration of the entire plan—shortening other paths doesn’t shorten the plan as a whole, but delaying the start of any action on the critical path slows down the whole plan. Actions that are off the critical path have a window of time in which they can be executed. The window is specified in terms of an earliest possible start time, *ES*, and a latest possible start time, *LS*. The quantity *LS* – *ES* is known as the **slack** of an action. We can see in Figure 11.14 that the whole plan will take 85 minutes, that each action in the top job has 15 minutes of slack, and that each action on the critical path has no slack (by definition). Together the *ES* and *LS* times for all the actions constitute a **schedule** for the problem.

The following formulas define *ES* and *LS* and constitute a dynamic-programming algorithm to compute them. *A* and *B* are actions, and $A \prec B$ means that *A* precedes *B*:

$$
\begin{aligned}
ES(Start) &= 0 \\
ES(B) &= \max_{A \prec B} ES(A) + Duration(A) \\
LS(Finish) &= ES(Finish) \\
LS(A) &= \min_{B \succ A} LS(B) - Duration(A)\,.
\end{aligned}
$$

The idea is that we start by assigning *ES(Start)* to be 0. Then, as soon as we get an action *B* such that all the actions that come immediately before *B* have *ES* values assigned, we set *ES*(*B*) to be the maximum of the earliest finish times of those immediately preceding actions, where the earliest finish time of an action is defined as the earliest start time plus the duration. This process repeats until every action has been assigned an *ES* value. The

LS values are computed in a similar manner, working backward from the *Finish* action.

The complexity of the critical path algorithm is just $O(Nb)$, where N is the number of actions and b is the maximum branching factor into or out of an action. (To see this, note that the *LS* and *ES* computations are done once for each action, and each computation iterates over at most b other actions.) Therefore, finding a minimum-duration schedule, given a partial ordering on the actions and no resource constraints, is quite easy.

Mathematically speaking, critical-path problems are easy to solve because they are defined as a *conjunction* of *linear* inequalities on the start and end times. When we introduce resource constraints, the resulting constraints on start and end times become more complicated. For example, the *AddEngine* actions, which begin at the same time in Figure 11.14, require the same *EngineHoist* and so cannot overlap. The “cannot overlap” constraint is a *disjunction* of two linear inequalities, one for each possible ordering. The introduction of disjunctions turns out to make scheduling with resource constraints NP-hard.

Figure 11.15 shows the solution with the fastest completion time, 115 minutes. This is 30 minutes longer than the 85 minutes required for a schedule without resource constraints. Notice that there is no time at which both inspectors are required, so we can immediately move one of our two inspectors to a more productive position.

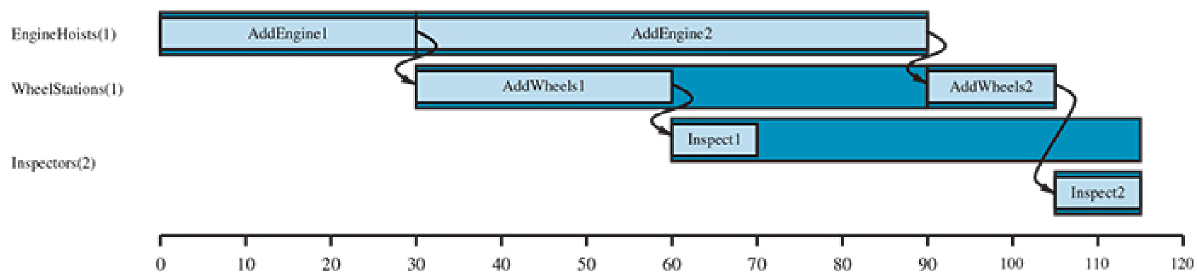


Figure 11.15 A solution to the job-shop scheduling problem from Figure 11.13, taking into account resource constraints. The left-hand margin lists the three reusable resources, and actions are shown aligned horizontally with the resources they use. There are two possible schedules, depending on which assembly uses the engine hoist first; we've shown the shortest-duration solution, which takes 115 minutes.

There is a long history of work on optimal scheduling. A challenge problem posed in 1963—to find the optimal schedule for a problem involving just 10 machines and 10 jobs of 100 actions each—went unsolved for 23 years (Lawler *et al.,* 1993). Many approaches have been tried, including branch-and-bound, simulated annealing, tabu search, and constraint satisfaction. One popular approach is the **minimum slack** heuristic: on each iteration, schedule for the earliest possible start whichever unscheduled action has all its predecessors scheduled and has the least slack; then update the *ES* and *LS* times for each affected action and repeat. This greedy heuristic resembles the minimum-remaining-values (MRV) heuristic in constraint satisfaction. It often works well in practice, but for our assembly problem it yields a 130-minute solution, not the 115-inute solution of Figure 11.15.

Up to this point, we have assumed that the set of actions and ordering constraints is fixed. Under these assumptions, every scheduling problem can be solved by a nonoverlapping sequence that avoids all resource conflicts, provided that each action is feasible by itself. However if a scheduling problem is proving very difficult, it may not be a good idea to solve it this way—it may be better to reconsider the actions and constraints, in case that leads to a much easier scheduling problem. Thus, it makes sense to *integrate* planning and scheduling by taking into account durations and overlaps during the construction of a plan. Several of the planning algorithms in Section 11.2 can be augmented to handle this information.

11.7 Analysis of Planning Approaches

Planning combines the two major areas of AI we have covered so far: *search* and *logic*. A planner can be seen either as a program that searches for a solution or as one that (constructively) proves the existence of a solution. The cross-fertilization of ideas from the two areas has allowed planners to scale up from toy problems where the number of actions and states was limited to around a dozen, to real-world industrial applications with millions of states and thousands of actions.

Planning is foremost an exercise in controlling combinatorial explosion. If there are n propositions in a domain, then there are 2^n states. Against such pessimism, the identification of independent subproblems can be a powerful weapon. In the best case—full decomposability of the problem—we get an exponential speedup. Decomposability is destroyed, however, by negative interactions between actions. SATPLAN can encode logical relations between subproblems. Forward search addresses the problem heuristically by trying to find patterns (subsets of propositions) that cover the independent subproblems. Since this approach is heuristic, it can work even when the subproblems are not completely independent.

Unfortunately, we do not yet have a clear understanding of which techniques work best on which kinds of problems. Quite possibly, new techniques will emerge, perhaps providing a synthesis of highly expressive first-order and hierarchical representations with the highly efficient factored and propositional representations that dominate today. We are seeing examples of **portfolio** planning systems, where a collection of algorithms are available to apply to any given problem. This can be done selectively (the system classifies each new problem to choose the best algorithm for it), or in parallel (all the algorithms run concurrently, each on a different CPU), or by interleaving the algorithms according to a schedule.

Summary

In this chapter, we described the PDDL representation for both classical and extended planning problems, and presented several algorithmic approaches for finding solutions. The points to remember:

- Planning systems are problem-solving algorithms that operate on explicit factored representations of states and actions. These representations make possible the derivation of effective domain-independent heuristics and the development of powerful and flexible algorithms for solving problems.
- PDDL, the Planning Domain Definition Language, describes the initial and goal states as conjunctions of literals, and actions in terms of their preconditions and effects. Extensions represent time, resources, percepts, contingent plans, and hierarchical plans.
- State-space search can operate in the forward direction (**progression**) or the backward direction (**regression**). Effective heuristics can be derived by subgoal independence assumptions and by various relaxations of the planning problem.
- Other approaches include encoding a planning problem as a Boolean satisfiability problem or as a constraint satisfaction problem; and explicitly searching through the space of partially ordered plans.
- **Hierarchical task network** (HTN) planning allows the agent to take advice from the domain designer in the form of **high-level actions** (HLAs) that can be implemented in various ways by lower-level action sequences. The effects of HLAs can be defined with **angelic semantics**, allowing provably correct high-level plans to be derived without consideration of lower-level implementations. HTN methods

can create the very large plans required by many real-world applications.

- **Contingent plans** allow the agent to sense the world during execution to decide what branch of the plan to follow. In some cases, **sensorless** or **conformant planning** can be used to construct a plan that works without the need for perception. Both conformant and contingent plans can be constructed by search in the space of **belief states**. Efficient representation or computation of belief states is a key problem.
- An **online planning agent** uses execution monitoring and splices in repairs as needed to recover from unexpected situations, which can be due to nondeterministic actions, exogenous events, or incorrect models of the environment.
- Many actions consume **resources**, such as money, gas, or raw materials. It is convenient to treat these resources as numeric measures in a pool rather than try to reason about, say, each individual coin and bill in the world. Time is one of the most important resources. It can be handled by specialized scheduling algorithms, or scheduling can be integrated with planning.
- This chapter extends classical planning to cover nondeterministic environments (where outcomes of actions are uncertain), but it is not the last word on planning. Chapter 16 describes techniques for stochastic environments (in which outcomes of actions have probabilities associated with them): Markov decision processes, partially observable Markov decision processes, and game theory. In Chapter 23 we show that reinforcement learning allows an agent to learn how to behave from past successes and failures.

Bibliographical and Historical Notes

AI planning arose from investigations into state-space search, theorem proving, and control theory. STRIPS (Fikes and Nilsson, 1971, 1993), the first major planning system, was designed as the planner for the Shakey robot at SRI. The first version of the program ran on a computer with only 192 KB of memory. Its overall control structure was modeled on GPS, the General Problem Solver (Newell and Simon, 1961), a state-space search system that used means–ends analysis.

The STRIPS representation language evolved into the Action Description Language, or ADL (Pednault, 1986), and then the Problem Domain Description Language, or PDDL (Ghallab *et al.*, 1998), which has been used for the International Planning Competition since 1998. The most recent version is PDDL 3.1 (Kovacs, 2011).

Planners in the early 1970s decomposed problems by computing a subplan for each subgoal and then stringing the subplans together in some order. This approach, called **linear planning** by Sacerdoti (1975), was soon discovered to be incomplete. It cannot solve some very simple problems, such as the Sussman anomaly (see Exercise 11.SUSS), found by Allen Brown during experimentation with the HACKER system (Sussman, 1975). A complete planner must allow for **interleaving** of actions from different subplans within a single sequence. Warren's (1974) WARPLAN system achieved that, and demonstrated how the logic programming language Prolog can produce concise programs; WARPLAN is only 100 lines of code.

Partial-order planning dominated the next 20 years of research, with theoretical work describing the detection of conflicts (Tate, 1975a) and the protection of achieved conditions (Sussman, 1975), and implementations including NOAH (Sacerdoti, 1977) and NONLIN (Tate, 1977). That led to formal models (Chapman, 1987; McAllester and Rosenblitt, 1991) that allowed for theoretical analysis of various algorithms and planning problems, and to a widely distributed system, UCPOP (Penberthy and Weld, 1992).

Drew McDermott suspected that the emphasis on partial-order planning was crowding out other techniques that should perhaps be reconsidered now that computers had 100 times the memory of Shakey's day. His UN POP (McDermott, 1996) was a state-space planning program employing the ignore-delete-list heuristic. HSP, the Heuristic Search Planner (Bonet and Geffner, 1999; Haslum, 2006) made state-space search practical for large planning problems. The FF or Fast Forward planner (Hoffmann, 2001; Hoffmann and Nebel, 2001; Hoffmann, 2005) and the FAST DOWNWARD variant (Helmert, 2006) won international planning competitions in the 2000s.

Bidirectional search (see Section 3.4.5) has also been known to suffer from a lack of heuristics, but some success has been obtained by using backward search to create a **perimeter** around the goal, and then refining a heuristic to search forward towards that perimeter (Torralba *et al.*, 2016). The SYMBA* bidirectional search planner (Torralba *et al.*, 2016) won the 2016 competition.

Researchers turned to PDDL and the planning paradigm so that they could use domain independent heuristics. Hoffmann (2005) analyzes the search space of the ignore-delete- list heuristic. Edelkamp (2009) and Haslum et *al.* (2007) describe how to construct pattern databases for planning heuristics. Felner et *al.* (2004) show encouraging results using pattern databases for sliding-tile puzzles, which can be thought of as a planning domain, but Hoffmann et *al.* (2006) show some limitations of abstraction for classical planning problems. (Rintanen, 2012) discusses planning-specific variable-selection heuristics for SAT solving.

Helmert *et al.* (2011) describe the Fast Downward Stone Soup (FDSS) system, a portfolio planner that, as in the fable of stone soup, invites us to throw in as many planning algorithms as possible. The system maintains a set of training problems, and for each problem and each algorithm records the run time and resulting plan cost of the problem's solution. Then when faced with a new problem, it uses the past experience to decide which algorithm(s) to try, with what time limits, and takes the solution with minimal cost. FDSS was a winner in the 2018 International Planning Competition (Seipp and Röger, 2018). Seipp *et al.* (2015) describe a machine learning approach to automatically learn a good portfolio, given a new problem. Vallati *et al.* (2015) give an overview of portfolio planning. The idea of algorithm portfolios for combinatorial search problems goes back to Gomes and Selman (2001).

Sistla and Godefroid (2004) cover symmetry reduction, and Godefroid (1990) covers heuristics for partial ordering. Richter and Helmert (2009) demonstrate the efficiency gains of forward pruning using preferred actions.

Blum and Furst (1997) revitalized the field of planning with their Graphplan system, which was orders of magnitude faster than the partial-order planners of the time. Bryce and Kambhampati (2007) give an overview of planning graphs. The use of situation calculus for planning was introduced by John McCarthy (1963) and refined by Ray Reiter (2001).

Kautz *et al.* (1996) investigated various ways to propositionalize action schemas, finding that the most compact forms did not necessarily lead to the fastest solution times. A systematic analysis was carried out by Ernst *et al.* (1997), who also developed an automatic "compiler" for generating propositional representations from PDDL problems. The BLACKBOX planner, which combines ideas from Graphplan and SATPLAN, was developed by Kautz and Selman (1998). Planners based on constraint satisfaction include CPLAN van Beek and Chen (1999) and GP-CSP (Do and Kambhampati, 2003).

There has also been interest in the representation of a plan as a **binary decision diagram (BDD)**, a compact data structure for Boolean expressions widely studied in the hardware verification community (Clarke and Grumberg, 1987; McMillan, 1993). There are techniques for proving properties of binary decision diagrams, including the property of being a solution to a planning problem. Cimatti *et al.* (1998) present a planner based on this approach. Other representations have also been used, such as integer programming (Vossen *et al.*, 2001).

There are some interesting comparisons of the various approaches to planning. Helmert (2001) analyzes several classes of planning problems, and shows that constraint-based approaches such as Graphplan and SATPLAN are best for NP-hard domains, while search-based approaches do better in domains where feasible solutions can be found without backtracking. Graphplan and SATPLAN have trouble in domains with many objects because that means they must create many actions. In some cases the problem can be delayed or avoided by generating the propositionalized actions dynamically, only as needed, rather than instantiating them all before the search begins.

The first mechanism for hierarchical planning was a facility in the STRIPS program for learning **macrops**—“macro-operators” consisting of a sequence of primitive steps (Fikes *et al.*, 1972). The ABSTRIPS system (Sacerdoti, 1974) introduced the idea of an **abstraction hierarchy**, whereby planning at higher levels was permitted to ignore lower-level preconditions of actions in order to derive the general structure of a working plan. Austin Tate's Ph.D. thesis (1975b) and work by Earl Sacerdoti (1977) developed the basic ideas of HTN planning. Erol, Hendler, and Nau (1994, 1996) present a complete hierarchical decomposition planner as well as a range of complexity results for pure HTN planners. Our presentation of HLAs and angelic semantics is due to Marthi et *al.* (2007, 2008).

One of the goals of hierarchical planning has been the reuse of previous planning experience in the form of generalized plans. The technique of **explanation-based learning** has been used as a means of generalizing previously computed plans in systems such as SOAR (Laird et *al.,* 1986) and PRODIGY (Carbonell et *al.,* 1989). An alternative approach is to store previously computed plans in their original form and then reuse them to solve new, similar problems by analogy to the original problem. This is the approach taken by the field called **case-based planning** (Carbonell, 1983; Alterman, 1988). Kambhampati (1994) argues that case-based planning should be analyzed as a form of refinement planning and provides a formal foundation for case-based partial-order planning.

Early planners lacked conditionals and loops, but some could use coercion to form conformant plans. Sacerdoti's NOAH solved the "keys and boxes" problem (in which the planner knows little about the initial state) using coercion. Mason (1993) argued that sensing often can and should be dispensed with in robotic planning, and described a sensorless plan that can move a tool into a specific position on a table by a sequence of tilting actions, *regardless* of the initial position.

Goldman and Boddy (1996) introduced the term **conformant planning**, noting that sensorless plans are often effective even if the agent has sensors. The first moderately efficient conformant planner was Smith and Weld's (1998) Conformant Graphplan (CGP). Ferraris and Giunchiglia

(2000) and Rintanen (1999) independently developed SATPLAN-based conformant planners. Bonet and Geffner (2000) describe a conformant planner based on heuristic search in the space of belief states, drawing on ideas first developed in the 1960s for partially observable Markov decision processes, or POMDPs (see Chapter 16).

Currently, there are three main approaches to conformant planning. The first two use heuristic search in belief-state space: HSCP (Bertoli *et al.*, 2001a) uses binary decision diagrams (BDDs) to represent belief states, whereas Hoffmann and Brafman (2006) adopt the lazy approach of computing precondition and goal tests on demand using a SAT solver.

The third approach, championed primarily by Jussi Rintanen (2007), formulates the entire sensorless planning problem as a quantified Boolean formula (QBF) and solves it using a general-purpose QBF solver. Current conformant planners are five orders of magnitude faster than CGP The winner of the 2006 conformant-planning track at the International Planning Competition was T_0 (Palacios and Geffner, 2007), which uses heuristic search in belief-state space while keeping the belief-state representation simple by defining derived literals that cover conditional effects. Bryce and Kambhampati (2007) discuss how a planning graph can be generalized to generate good heuristics for conformant and contingent planning.

The contingent-planning approach described in the chapter is based on Hoffmann and Brafman (2005), and was influenced by the efficient search algorithms for cyclic AND-OR graphs developed by Jimenez and Torras (2000) and Hansen and Zilberstein (2001). The problem of contingent planning received more attention after the publication of Drew McDermott’s (1978a) influential article, *Planning and Acting*. Bertoli *et al.* (2001b) describe MBP (Model-Based Planner), which uses binary decision diagrams to do conformant and contingent planning. Some authors use “conditional planning” and “contingent planning” as synonyms; others make the distinction that “conditional” refers to actions with nondeterministic effects, and “contingent” means using sensing to overcome partial observability.

In retrospect, it is now possible to see how the major classical planning algorithms led to extended versions for uncertain domains. Fast-forward heuristic search through state space led to forward search in belief space (Bonet and Geffner, 2000; Hoffmann and Brafman, 2005); SATPLAN led to stochastic SATPLAN (Majercik and Littman, 2003) and to planning with quantified Boolean logic (Rintanen, 2007); partial order planning led to UWL (Etzioni *et al.*, 1992) and CNLP (Peot and Smith, 1992); Graphplan led to Sensory Graphplan or SGP (Weld *etal.*, 1998).

The first online planner with execution monitoring was PLANEX (Fikes *et al.*, 1972), which worked with the STRIPS planner to control the robot Shakey. SIPE (System for Interactive Planning and Execution monitoring) (Wilkins, 1988) was the first planner to deal systematically with the problem of replanning. It has been used in demonstration projects in several domains, including planning operations on the flight deck of an aircraft carrier, jobshop scheduling for an Australian beer factory, and planning the construction of multistory buildings (Kartam and Levitt, 1990).

In the mid-1980s, pessimism about the slow run times of planning systems led to the proposal of reflex agents called **reactive planning** systems (Brooks, 1986; Agre and Chapman, 1987). “Universal plans” (Schoppers, 1989) were developed as a lookup-table method for reactive planning, but turned out to be a rediscovery of the idea of **policies** that had long been used in Markov decision processes (see Chapter 16). Koenig (2001) surveys online planning techniques, under the name *Agent-Centered Search.*

Planning with time constraints was first dealt with by DEVISER (Vere, 1983). The representation of time in plans was addressed by Allen (1984) and by Dean *et al.* (1990) in the FORBIN system. NONLIN+ (Tate and Whiter, 1984) and SIPE (Wilkins, 1990) could reason about the allocation of limited resources to various plan steps. O-PLAN (Bell and Tate, 1985) has been applied to resource problems such as software procurement planning at Price Waterhouse and back-axle assembly planning at Jaguar Cars.

The two planners SAPA (Do and Kambhampati, 2001) and T4 (Haslum and Geffner, 2001) both used forward state-space search with sophisticated heuristics to handle actions with durations and resources. An alternative is to use very expressive action languages, but guide them by human-written, domain-specific heuristics, as is done by ASPEN (Fukunaga *et al.*, 1997), HSTS (Jonsson *et al.*, 2000), and IxTeT (Ghallab and Laruelle, 1994).

A number of hybrid planning-and-scheduling systems have been deployed: ISIS (Fox *et al.*, 1982; Fox, 1990) has been used for job-shop scheduling at Westinghouse, GARI (De-scotte and Latombe, 1985) planned the machining and construction of mechanical parts, FORBIN was used for factory control, and NONLIN+ was used for naval logistics planning. We chose to present planning and scheduling as two separate problems; Cushing *et al.* (2007) show that this can lead to incompleteness on certain problems.

There is a long history of scheduling in aerospace. T-SCHED (Drabble, 1990) was used to schedule mission-command sequences for the UOSAT-II satellite. OPTIMUM-AIV (Aarup *et al.*, 1994) and PLAN-ERSI (Fuchs *et al.*, 1990), both based on O-PLAN, were used for spacecraft assembly and observation planning, respectively, at the European Space Agency. SPIKE (Johnston and Adorf, 1992) was used for observation planning at NASA for the Hubble Space Telescope, while the Space Shuttle Ground Processing Scheduling System (Deale *et al.*, 1994) does job-shop scheduling of up to 16,000 worker-shifts. Remote Agent (Muscet- tola *et al.* ,1998) became the first autonomous planner-scheduler to control a spacecraft, when it flew onboard the Deep Space One probe in 1999. Space applications have driven the development of algorithms for resource allocation; see Laborie (2003) and Muscettola (2002). The literature on scheduling is presented in a classic survey article (Lawler *et al.*, 1993), a book (Pinedo, 2008), and an edited handbook (Blazewicz *et al.*, 2007).

The computational complexity of planning has been analyzed by several authors (Bylander, 1994; Ghallab *et al.*, 2004; Rintanen, 2016). There are two main tasks: **PlanSAT** is the question of whether there exists any plan that solves a planning problem. **Bounded PlanSAT** asks whether there is a solution of length *k* or less; this can be used to find an optimal plan. Both are decidable for classical planning (because the number of states is finite). But if we add function symbols to the language, then the number of states becomes infinite, and PlanSAT becomes only semidecidable. For propositionalized problems both are in the complexity class PSPACE, a class that is larger (and hence more difficult) than NP and refers to problems that can be solved by a deterministic Turing machine with a polynomial amount of space. These theoretical results are discouraging, but in practice, the problems we want to solve tend to be not so bad. The true advantage of the classical planning formalism is that it has facilitated the development of very accurate domain-independent heuristics; other approaches have not been as fruitful.

Readings in Planning (Allen *et al.*, 1990) is a comprehensive anthology of early work in the field. Weld (1994, 1999) provides two excellent surveys of planning algorithms of the 1990s. It is interesting to see the change in the five years between the two surveys: the first concentrates on partial-order planning, and the second introduces Graphplan and SATPLAN. *Automated Planning and Acting* (Ghallab *et al.*, 2016) is an excellent textbook on all aspects of the field. LaValle's text *Planning Algorithms* (2006) covers both classical and stochastic planning, with extensive coverage of robot motion planning.

Planning research has been central to AI since its inception, and papers on planning are a staple of mainstream AI journals and conferences. There are also specialized conferences such as the International Conference on Automated Planning and Scheduling and the International Workshop on Planning and Scheduling for Space.

---

[1] The blocks world commonly used in planning research is much simpler than SHRDLU'S version (page 38).

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

[3] If cyclic solutions are required for a nondeterministic problem, AND-OR search must be generalized to a loopy version such as LAO* (Hansen and Zilberstein, 2001).

[4] In 1954, a Mrs. Hodges of Alabama was hit by meteorite that crashed through her roof. In 1992, a piece of the Mbale meteorite hit a small boy on the head; fortunately, its descent was slowed by banana leaves (Jenniskens *et al*., 1994). And in 2009, a German boy claimed to have been hit in the hand by a pea-sized meteorite. No serious injuries resulted from any of these incidents, suggesting that the need for preplanning against such contingencies is sometimes overstated.

[5] Plan monitoring means that finally, after 374 pages, we have an agent that is smarter than a dung beetle (see page 59). A plan-monitoring agent would notice that the dung ball was missing from its grasp and would replan to get another ball and plug its hole.

[6] Futile repetition of a plan repair is exactly the behavior exhibited by the sphex wasp (page 59).

CHAPTER 12

QUANTIFYING UNCERTAINTY

In which we see how to tame uncertainty with numeric degrees of belief.

12.1 Acting under Uncertainty

Agents in the real world need to handle **uncertainty**, whether due to partial observability, nondeterminism, or adversaries. An agent may never know for sure what state it is in now or where it will end up after a sequence of actions.

We have seen problem-solving and logical agents handle uncertainty by keeping track of a **belief state**—a representation of the set of all possible world states that it might be in—and generating a contingency plan that handles every possible eventuality that its sensors may report during execution. This approach works on simple problems, but it has drawbacks:

- The agent must consider *every possible* explanation for its sensor observations, no matter how unlikely. This leads to a large belief-state full of unlikely possibilities.
- A correct contingent plan that handles every eventuality can grow arbitrarily large and must consider arbitrarily unlikely contingencies.
- Sometimes there is no plan that is guaranteed to achieve the goal—yet the agent must act. It must have some way to compare the merits of plans that are not guaranteed.

Suppose, for example, that an automated taxi has the goal of delivering a passenger to the airport on time. The taxi forms a plan, A_{90}, that involves leaving home 90 minutes before the flight departs and driving at a reasonable speed. Even though the airport is only 5 miles away, a logical agent will not be able to conclude with absolute certainty that "Plan A_{90} will get us to the airport in time." Instead, it reaches the weaker conclusion "PlanA_{90} will get us to the airport in time, as long as the car doesn't break down, and I don’t get into an accident, and the road isn’t closed, and no meteorite hits the car, and ... .” None of these conditions can be deduced for sure, so we can’t infer that the plan succeeds. This is the logical **qualification problem** (page 259), for which we so far have seen no real solution.

Nonetheless, in some sense A_{90} *is* in fact the right thing to do. What do we mean by this? As we discussed in Chapter 2, we mean that out of all the plans that could be executed, A_{90} is expected to maximize the agent’s performance measure (where the expectation is relative to the agent’s knowledge about the environment). The performance measure includes getting to the airport in time for the flight, avoiding a long, unproductive wait at the airport, and avoiding speeding tickets along the way. The agent’s knowledge cannot guarantee any of these outcomes for A_{90}, but it can provide some degree of belief that they will be achieved. Other plans, such as A_{180},, might increase the agent’s belief that it will get to the airport on time, but also increase the likelihood of a long, boring wait. *The right thing to do—the**rational decision**—therefore depends on both the relative importance of various goals and the likelihood that, and degree to which, they will be achieved.* The remainder of this section hones these ideas, in preparation for the development of the general theories of uncertain reasoning and rational decisions that we present in this and subsequent chapters.

12.1.1 Summarizing uncertainty

Let’s consider an example of uncertain reasoning: diagnosing a dental patient’s toothache. Diagnosis—whether for medicine, automobile repair, or whatever—almost always involves uncertainty. Let us try to write rules for dental diagnosis using propositional logic, so that we can see how the logical approach breaks down. Consider the following simple rule:

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

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

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

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

$$Cavity \Rightarrow Toothache.$$

But this rule is not right either; not all cavities cause pain. The only way to fix the rule is to make it logically exhaustive: to augment the left-hand side with all the qualifications required for a cavity to cause a toothache. Trying to use logic to cope with a domain like medical diagnosis thus fails for three main reasons:

- **Laziness:** It is too much work to list the complete set of antecedents or consequents needed to ensure an exceptionless rule and too hard to use such rules.
- **Theoretical ignorance:** Medical science has no complete theory for the domain.
- **Practical ignorance:** Even if we know all the rules, we might be uncertain about a particular patient because not all the necessary tests have been or can be run.

The connection between toothaches and cavities is not a strict logical consequence in either direction. This is typical of the medical domain, as well as most other judgmental domains: law, business, design, automobile repair, gardening, dating, and so on. The agent's knowledge can at best provide only a **degree of belief** in the relevant sentences. Our main tool for dealing with degrees of belief is **probability theory.** In the terminology of Section 8.1, the **ontological commitments** of logic and probability theory are the same—that the world is composed of facts that do or do not hold in any particular case—but the **epistemological commitments** are different: a logical agent believes each sentence to be true or false or has no opinion, whereas a probabilistic agent may have a numerical degree of belief between 0 (for sentences that are certainly false) and 1 (certainly true).

The theory of *probability provides a way of summarizing the uncertainty that comes from our laziness and ignorance,* thereby solving the qualification problem. We might not know for sure what afflicts a particular patient, but we believe that there is, say, an 80% chance— that is, a probability of 0.8—that the patient who has a toothache has a cavity. That is, we expect that out of all the situations that are indistinguishable from the current situation as far as our knowledge goes, the patient will have a cavity in 80% of them. This belief could be derived from statistical data—80% of the toothache patients seen so far have had cavities—or from some general dental knowledge, or from a combination of evidence sources.

One confusing point is that at the time of our diagnosis, there is no uncertainty in the actual world: the patient either has a cavity or doesn't. So what does it mean to say the probability of a cavity is 0.8? Shouldn't it be either 0 or 1? The answer is that probability statements are made with respect to a knowledge state, not with respect to the real world. We say "The probability that the patient has a cavity, *given that she has a toothache*, is 0.8." If we later learn that the patient has a history of gum disease, we can make a different statement: "The probability that the patient has a cavity, given that she has a toothache and a history of gum disease, is 0.4.” If we gather further conclusive evidence against a cavity, we can say “The probability that the patient has a cavity, given all we now know, is almost 0.” Note that these statements do not contradict each other; each is a separate assertion about a different knowledge state.

12.1.2 Uncertainty and rational decisions

Consider again the A_{90} plan for getting to the airport. Suppose it gives us a 97% chance of catching our flight. Does this mean it is a rational choice? Not necessarily: there might be other plans, such as A_{180}, with higher probabilities. If it is *vital* not to miss the flight, then it is worth risking the longer wait at the airport. What about A_{1440}, a plan that involves leaving home 24 hours in advance? In most circumstances, this is not a good choice, because although it almost guarantees getting there on time, it involves an intolerable wait—not to mention a possibly unpleasant diet of airport food.

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

The utility of a state is relative to an agent. For example, the utility of a state in which White has checkmated Black in a game of chess is obviously high for the agent playing White, but low for the agent playing Black. But we can’t go strictly by the scores of 1, 1/2, and 0 that are dictated by the rules of tournament chess—some players (including the authors) might be thrilled with a draw against the world champion, whereas other players (including the former world champion) might not. There is no accounting for taste or preferences: you might think that an agent who prefers jalapeño bubble-gum ice cream to chocolate chip is odd, but you could not say the agent is irrational. A utility function can account for any set of preferences—quirky or typical, noble or perverse. Note that utilities can account for altruism, simply by including the welfare of others as one of the factors.

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

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

The fundamental idea of decision theory is that *an agent is rational if and only if it chooses the action that yields the highest expected utility, averaged over all the possible outcomes of the action.* This is called the principle of **maximum expected utility (MEU).** Here, “expected” means the “average,” or “statistical mean” of the outcome utilities, weighted by the probability of the outcome. We saw this principle in action in Chapter 6 when we touched briefly on optimal decisions in backgammon; it is in fact a completely general principle for single-agent decision making.

Figure 12.1 sketches the structure of an agent that uses decision theory to select actions. The agent is identical, at an abstract level, to the agents described in Chapters 4 and 7 that maintain a belief state reflecting the history of percepts to date. The primary difference is that the decision-theoretic agent’s belief state represents not just the *possibilities* for world states but also their *probabilities.* Given the belief state and some knowledge of the effects of actions, the agent can make probabilistic predictions of action outcomes and hence select the action with the highest expected utility.

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

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

Figure 12.1 A decision-theoretic agent that selects rational actions.

This chapter and the next concentrate on the task of representing and computing with probabilistic information in general. Chapter 14 deals with methods for the specific tasks of representing and updating the belief state over time and predicting outcomes. Chapter 18 looks at ways of combining probability theory with expressive formal languages such as firstorder logic and general-purpose programming languages. Chapter 15 covers utility theory in more depth, and Chapter 16 develops algorithms for planning sequences of actions in stochastic environments. Chapter 17 covers the extension of these ideas to multiagent environments.

12.2 Basic Probability Notation

For our agent to represent and use probabilistic information, we need a formal language. The language of probability theory has traditionally been informal, written by human mathematicians for other human mathematicians. Appendix A includes a standard introduction to elementary probability theory; here, we take an approach more suited to the needs of AI and connect it with the concepts of formal logic.

12.2.1 What probabilities are about

Like logical assertions, probabilistic assertions are about possible worlds. Whereas logical assertions say which possible worlds are strictly ruled out (all those in which the assertion is false), probabilistic assertions talk about how probable the various worlds are. In probability theory, the set of all possible worlds is called the **sample space**. The possible worlds are *mutually exclusive* and *exhaustive*—two possible worlds cannot both be the case, and one possible world must be the case. For example, if we are about to roll two (distinguishable) dice, there are 36 possible worlds to consider: (1,1), (1,2), ..., (6,6). The Greek letter Ω (uppercase omega) is used to refer to the sample space, and ω (lowercase omega) refers to elements of the space, that is, particular possible worlds.

A fully specified **probability model** associates a numerical probability $P(\omega)$ with each possible world.[1] The basic axioms of probability theory say that every possible world has a probability between 0 and 1 and that the total probability of the set of possible worlds is 1:

$$0 \leq P(\omega) \leq 1 \text{ for every } \omega \text{ and} \sum_{\omega \in \Omega} P(\omega) = 1. \qquad (12.1)$$

For example, if we assume that each die is fair and the rolls don't interfere with each other, then each of the possible worlds (1,1), (1,2), ..., (6,6) has probability 1/36. If the dice are loaded then some worlds will have higher probabilities and some lower, but they will all still sum to 1.

Probabilistic assertions and queries are not usually about particular possible worlds, but about sets of them. For example, we might ask for the probability that the two dice add up to 11, the probability that doubles are rolled, and so on. In probability theory, these sets are called **events**—a term already used extensively in Chapter 10 for a different concept. In logic, a set of worlds corresponds to a **proposition** in a formal language; specifically, for each proposition, the corresponding set contains just those possible worlds in which the proposition holds. (Hence, "event" and "proposition" mean roughly the same thing in this context, except that a proposition is expressed in a formal language.) The probability associated with a proposition is defined to be the sum of the probabilities of the worlds in which it holds:

$$\text{For any proposition } \phi,\ P(\phi) = \sum_{\omega \in \phi} P(\omega). \qquad (12.2)$$

For example, when rolling fair dice, we have $P(Total = 11) = P((5, 6)) + P((6, 5)) = 1/36 + 1/36 = 1/18$. Note that probability theory does not require complete knowledge of the probabilities of each possible world. For example, if we believe the dice conspire to produce the same number, we might *assert* that $P(doubles) = 1/4$ without knowing whether the dice prefer double 6 to double 2. Just as with logical assertions, this assertion *constrains* the underlying probability model without fully determining it.

Probabilities such as $P(Total = 11)$ and $P(doubles)$ are called **unconditional** or **prior probabilities** (and sometimes just "priors" for short); they refer to degrees of belief in propositions *in the absence of any other information.* Most of the time, however, we have *some* information, usually called **evidence**, that has already been revealed. For example, the first die may already be showing a 5 and we are waiting with bated breath for the other one to stop spinning. In that case, we are interested not in the unconditional probability of rolling doubles, but the **conditional** or **posterior** probability (or just "posterior" for short) of rolling doubles *given that the first die is a 5.* This probability is written $P(doubles \mid Die_1 = 5)$, where the "|" is pronounced "given."[2]

Similarly, if I am going to the dentist for a regularly scheduled checkup, then the prior probability $P(cavity) = 0.2$ might be of interest; but if I go to the dentist because I have a toothache, it's the conditional probability $P(cavity \mid toothache) = 0.6$ that matters.

It is important to understand that $P(cavity) = 0.2$ is still *valid* after *toothache* is observed; it just isn't especially useful. When making decisions, an agent needs to condition on *all* the evidence it has observed. It is also important to understand the difference between conditioning and logical implication. The assertion that $P(cavity \mid toothache) = 0.6$ does not mean "Whenever *toothache* is true, conclude that *cavity* is true with probability 0.6" rather it means "Whenever *toothache* is true *and we have no further information*, conclude that *cavity* is true with probability 0.6." The extra condition is important; for example, if we had the further information that the dentist found no cavities, we definitely would not want to conclude that *cavity* is true with probability 0.6; instead we need to use $P(cavity \mid toothache \wedge \neg cavity) = 0$.

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

$$P(a \middle| b) = \frac{P(a \wedge b)}{P(b)}, \qquad (12.3)$$

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

$$P\left(doubles \,\middle|\, Die_1 = 5\right) = \frac{P\left(doubles \wedge Die_1 = 5\right)}{P(Die_1 = 5)}.$$

The definition makes sense if you remember that observing b rules out all those possible worlds where b is false, leaving a set whose total probability is just $P(b)$. Within that set, the worlds where a is true must satisfy $a \wedge b$ and constitute a fraction $P(a \wedge b)/P(b)$.

The definition of conditional probability, Equation (12.3), can be written in a different form called the **product rule**:

$$P(a \wedge b) = P(a|b)P(b). \qquad (12.4)$$

The product rule is perhaps easier to remember: it comes from the fact that for a and b to be true, we need b to be true, and we also need a to be true given b.

12.2.2 The language of propositions in probability assertions

In this chapter and the next, propositions describing sets of possible worlds are usually written in a notation that combines elements of propositional logic and constraint satisfaction notation. In the terminology of Section 2.4.7, it is a **factored representation**, in which a possible world is represented by a set of variable/value pairs. A more expressive **structured representation** is also possible, as shown in Chapter 18.

Variables in probability theory are called **random variables**, and their names begin with an uppercase letter. Thus, in the dice example, *Total* and Die_1 are random variables. Every random variable is a function that maps from the domain of possible worlds Ω to some **range**—the set of possible values it can take on. The range of *Total* for two dice is the set {2,..., 12} and the range of Die_1 is {1,..., 6}. Names for values are always lowercase, so we might write $\sum_x P(X = x)$ to sum over the values of X. A Boolean random variable has the range {*true*,*false*}. For example, the proposition that doubles are rolled can be written as *Doubles* = *true.* (An alternative range for

Boolean variables is the set {0,1}, in which case the variable is said to have a **Bernoulli** distribution.) By convention, propositions of the form *A* = *true* are abbreviated simply as *a*, while *A* = *false* is abbreviated as ¬*a*. (The uses of *doubles*, *cavity*, and *toothache* in the preceding section are abbreviations of this kind.)

Ranges can be sets of arbitrary tokens. We might choose the range of *Age* to be the set *{juvenile, teen, adult}* and the range of *Weather* might be *{sun, rain, cloud, snow}*. When no ambiguity is possible, it is common to use a value by itself to stand for the proposition that a particular variable has that value; thus, *sun* can stand for *Weather = sun*[3].

The preceding examples all have finite ranges. Variables can have infinite ranges, too— either discrete (like the integers) or continuous (like the reals). For any variable with an ordered range, inequalities are also allowed, such as *NumberOfAtomsInUniverse* $\geq 10^{70}$.

Finally, we can combine these sorts of elementary propositions (including the abbreviated forms for Boolean variables) by using the connectives of propositional logic. For example, we can express "The probability that the patient has a cavity, given that she is a teenager with no toothache, is 0.1" as follows:

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

In probability notation, it is also common to use a comma for conjunction, so we could write *P*(*cavity* | ¬*toothache, teen).*

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

$$P(Weather = sun) \ = \ 0.6$$
$$P(Weather = rain) \ = \ 0.1$$
$$P(Weather = cloud) \ = \ 0.29$$
$$P(Weather = snow) \ = \ 0.01,$$

but as an abbreviation we will allow

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

where the bold **P** indicates that the result is a vector of numbers, and where we assume a predefined ordering *(sun, rain, cloud, snow)* on the range of *Weather*. We say that the **P** statement defines a **probability distribution** for the random variable *Weather*—that is, an assignment of a probability for each possible value of the random variable. (In this case, with a finite, discrete range, the distribution is called a **categorical distribution**.) The **P** notation is also used for conditional distributions: **P**(*X* | *Y)* gives the values of $P(X = x_i \mid Y = y_j)$ for each possible *i, j* pair.

For continuous variables, it is not possible to write out the entire distribution as a vector, because there are infinitely many values. Instead, we can define the probability that a random variable takes on some value *x* as a parameterized function of *x*, usually called a **probability density function**. For example, the sentence

$$P(NoonTemp = x) \ = \ Uniform(x; 18C, 26C)$$

expresses the belief that the temperature at noon is distributed uniformly between 18 and 26 degrees Celsius.

Probability density functions (sometimes called **pdfs**) differ in meaning from discrete distributions. Saying that the probability density is uniform from 18*C* to 26*C* means that there is a 100% chance that the temperature will fall somewhere in that 8*C*-wide region and a 50% chance that it will fall in any 4*C*-wide sub-region, and so on. We write the probability density for a continuous random variable *X* at value x as *P*(*X* = *x*) or just *P*(*x*); the intuitive definition of *P*(*x*) is the probability that *X* falls within an arbitrarily small region beginning at *x*, divided by the width of the region:

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

For *NoonTemp* we have

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

where C stands for centigrade (not for a constant). In $P(NoonTemp = 20.18C) = \frac{1}{8C}$, note that $\frac{1}{8C}$ is not a probability, it is a probability density. The probability that *NoonTemp* is *exactly* 20.18C is zero, because 20.18C is a region of width 0. Some authors use different symbols for discrete probabilities and probability densities; we use P for specific probability values and **P** for vectors of values in both cases, since confusion seldom arises and the equations are usually identical. Note that probabilities are unitless numbers, whereas density functions are measured with a unit, in this case reciprocal degrees centigrade. If the same temperature interval were to be expressed in degrees Fahrenheit, it would have a width of 14.4 degrees, and the density would be 1/14.4F.

In addition to distributions on single variables, we need notation for distributions on multiple variables. Commas are used for this. For example, **P**(*Weather, Cavity*) denotes the probabilities of all combinations of the values of *Weather* and *Cavity.* This is a 4 × 2 table of probabilities called the **joint probability distribution** of *Weather* and *Cavity.* We can also mix variables and specific values; **P**(*sun*, *Cavity*) would be a two-element vector giving the probabilities of a cavity with a sunny day and no cavity with a sunny day.

The **P** notation makes certain expressions much more concise than they might otherwise be. For example, the product rules (see Equation (12.4)) for all possible values of *Weather* and *Cavity* can be written as a single equation:

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

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

$$\begin{aligned}
&P(W = sun \wedge C = true) = P(W = sun|C = true)P(C = true)\\
&P(W = rain \wedge C = true) = P(W = rain|C = true)P(C = true)\\
&P(W = cloud \wedge C = true) = P(W = cloud|C = true)P(C = true)\\
&P(W = snow \wedge C = true) = P(W = snow|C = true)P(C = true)\\
&P(W = sun \wedge C = false) = P(W = sun|C = false)P(C = false)\\
&P(W = rain \wedge C = false) = P(W = rain|C = false)P(C = false)\\
&P(W = cloud \wedge C = false) = P(W = cloud|C = false)P(C = false)\\
&P(W = snow \wedge C = false) = P(W = snow|C = false)P(C = false).
\end{aligned}$$

As a degenerate case, **P**(*sun*, *cavity)* has no variables and thus is a zero-dimensional vector, which we can think of as a scalar value.

Now we have defined a syntax for propositions and probability assertions and we have given part of the semantics: Equation (12.2) defines the probability of a proposition as the sum of the probabilities of worlds in which it holds. To complete the semantics, we need to say what the worlds are and how to determine whether a proposition holds in a world. We borrow this part directly from the semantics of propositional logic, as follows. *A possible world is defined to be an assignment of values to all of the random variables under consideration.*

It is easy to see that this definition satisfies the basic requirement that possible worlds be mutually exclusive and exhaustive (Exercise 12.EXEX). For example, if the random variables are *Cavity, Toothache,* and *Weather,* then there are 2 × 2 × 4 = 16 possible worlds. Furthermore, the truth of any given proposition can be determined easily in such worlds by the same recursive truth calculation we used for propositional logic (see page 236).

Note that some random variables may be redundant, in that their values can be obtained in all cases from the values of other variables. For example, the *Doubles* variable in the two-dice world is true exactly when $Die_1 = Die_2$. Including *Doubles* as one of the random variables, in addition to Die_1 and Die_2, seems to increase the number of possible worlds from 36 to 72, but of course exactly half of the 72 will be logically impossible and will have probability 0.

From the preceding definition of possible worlds, it follows that a probability model is completely determined by the joint distribution for all of the random variables—the so-called **full joint probability distribution.** For example, given *Cavity*, *Toothache*, and *Weather*, the full joint distribution is $\mathbf{P}(Cavity, Toothache, Weather)$. This joint distribution can be represented as a 2 × 2 × 4 table with 16 entries. Because every proposition's probability is a sum over possible worlds, a full joint distribution suffices, in principle, for calculating the probability of any proposition. We will see examples of how to do this in Section 12.3.

12.2.3 Probability axioms and their reasonableness

The basic axioms of probability (Equations (12.1) and (12.2)) imply certain relationships among the degrees of belief that can be accorded to logically related propositions. For example, we can derive the familiar relationship between the probability of a proposition and the probability of its negation:

$$\begin{aligned}
P(\neg a) &= \textstyle\sum_{\omega \in \neg a} P(\omega) && \text{by Equation (12.2)}\\
&= \textstyle\sum_{\omega \in \neg a} P(\omega) + \sum_{\omega \in a} P(\omega) - \sum_{\omega \in a} P(\omega) &&\\
&= \textstyle\sum_{\omega \in \Omega} P(\omega) - \sum_{\omega \in a} P(\omega) && \text{grouping the first two terms}\\
&= 1 - P(a) && \text{by (12.1) and (12.2).}
\end{aligned}$$

We can also derive the well-known formula for the probability of a disjunction, sometimes called the **inclusion-exclusion principle**:

$$P(a \vee b) = P(a) + P(b) - P(a \wedge b). \qquad (12.5)$$

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

Equations (12.1) and (12.5) are often called **Kolmogorov's axioms** in honor of the mathematician Andrei Kolmogorov, who showed how to build up the rest of probability theory from this simple foundation and how to handle the difficulties caused by continuous variables.[4] While Equation (12.2) has a definitional flavor, Equation (12.5) reveals that the axioms really do constrain the degrees of belief an agent can have concerning logically related propositions. This is analogous to the fact that a logical agent cannot simultaneously believe A, B, and $\neg(A \wedge B)$, because there is no possible world in which all three are true. With probabilities, however, statements refer not to the world directly, but to the agent's own state of knowledge. Why, then, can an agent not hold the following set of beliefs (even though they violate Kolmogorov's axioms)?

$$P(a) = 0.4 \qquad\qquad P(b) = 0.3 \qquad\qquad P(a \wedge b) = 0.0$$

This kind of question has been the subject of decades of intense debate between those who advocate the use of probabilities as the only legitimate form for degrees of belief and those who advocate alternative approaches.

One argument for the axioms of probability, first stated in 1931 by Bruno de Finetti (see de Finetti, 1993, for an English translation), is as follows: If an agent has some degree of belief in a proposition a, then the agent should be able to state odds at which it is indifferent to a bet for or against a.[5] Think of it as a game between two agents: Agent 1 states, "my degree of belief in event a is 0.4." Agent 2 is then free to choose whether to wager for or against *a* at stakes that are consistent with the stated degree of belief. That is, Agent 2 could choose to accept Agent 1's bet that *a* will occur, offering \$6 against Agent 1's \$4. Or Agent 2 could accept Agent 1's bet that ¬*a* will occur, offering \$4 against Agent 1's \$6. Then we observe the outcome of *a*, and whoever is right collects the money. If one's degrees of belief do not accurately reflect the world, then one would expect to lose money over the long run to an opposing agent whose beliefs more accurately reflect the state of the world.

De Finetti's theorem is not concerned with choosing the right values for individual probabilities, but with choosing values for the probabilities of logically related propositions: *If Agent 1 expresses a set of degrees of belief that violate the axioms of probability theory then there is a combination of bets by Agent 2 that guarantees that Agent 1 will lose money every time.* For example, suppose that Agent 1 has the set of degrees of belief from Equation (12.6). Figure 12.2 shows that if Agent 2 chooses to bet \$4 on *a*, \$3 on *b*, and \$2 on $\neg(a \vee b)$, then Agent 1 always loses money, regardless of the outcomes for *a* and *b*. De Finetti's theorem implies that no rational agent can have beliefs that violate the axioms of probability.

Proposition	Agent 1's belief	Agent 2 bets	Agent 1 bets	Agent 1 payoffs for each outcome a, b	$a, \neg b$	$\neg a, b$	$\neg a, \neg b$
a	0.4	\$4 on a	\$6 on $\neg a$	–\$6	–\$6	\$4	\$4
b	0.3	\$3 on b	\$7 on $\neg b$	–\$7	\$3	–\$7	\$3
$a \vee b$	0.8	\$2 on $\neg(a \vee b)$	\$8 on $a \vee b$	\$2	\$2	\$2	–\$8
				–\$11	–\$1	–\$1	–\$1

Figure 12.2 Because Agent 1 has inconsistent beliefs, Agent 2 is able to devise a set of three bets that guarantees a loss for Agent 1, no matter what the outcome of *a* and *b*.

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

Other strong philosophical arguments have been put forward for the use of probabilities, most notably those of Cox (1946), Carnap (1950), and Jaynes (2003). They each construct a set of axioms for reasoning with degrees of beliefs: no contradictions, correspondence with ordinary logic (for example, if belief in *A* goes up, then belief in ¬*A* must go down), and so on. The only controversial axiom is that degrees of belief must be numbers, or at least act like numbers in that they must be transitive (if belief in *A* is greater than belief in *B*, which is greater than belief in *C*, then belief in *A* must be greater than *C*) and comparable (the belief in *A* must be one of equal to, greater than, or less than belief in *B*). It can then be proved that probability is the only approach that satisfies these axioms.

The world being the way it is, however, practical demonstrations sometimes speak louder than proofs. The success of reasoning systems based on probability theory has been much more effective than philosophical arguments in making converts. We now look at how the axioms can be deployed to make inferences.

12.3 Inference Using Full Joint Distributions

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

We begin with a simple example: a domain consisting of just the three Boolean variables *Toothache*, *Cavity*, and *Catch* (the dentist’s nasty steel probe catches in my tooth). The full joint distribution is a 2 × 2 × 2 table as shown in Figure 12.3.

	toothache		*¬toothache*	
	catch	*¬catch*	*catch*	*¬catch*
cavity	0.108	0.012	0.072	0.008
¬cavity	0.016	0.064	0.144	0.576

Figure 12.3 A full joint distribution for the *Toothache, Cavity, Catch* world.

Notice that the probabilities in the joint distribution sum to 1, as required by the axioms of probability. Notice also that Equation (12.2) gives us a direct way to calculate the probability of any proposition, simple or complex: simply identify those possible worlds in which the proposition is true and add up their probabilities. For example, there are six possible worlds in which *cavity* ∨ *toothache* holds:

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

One particularly common task is to extract the distribution over some subset of variables or a single variable. For example, adding the entries in the first row gives the unconditional or **marginal probability**[6] of *cavity:*

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

This process is called **marginalization**, or **summing out**—because we sum up the probabilities for each possible value of the other variables, thereby taking them out of the equation. We can write the following general marginalization rule for any sets of variables **Y** and **Z**:

$$\mathbf{P}(\mathbf{Y}) = \sum_{\mathbf{z}} \mathbf{P}(\mathbf{Y}, \mathbf{Z} = \mathbf{z}), \qquad (12.7)$$

where $\sum_{\mathbf{z}}$ sums over all the possible combinations of values of the set of variables **Z**. As usual we can abbreviate **P**(**Y**, **Z** = **z**) in this equation by **P**(**Y**, **z**). For the *Cavity* example, Equation (12.7) corresponds to the following equation:

$$\begin{aligned}\mathbf{P}(Cavity) &= \mathbf{P}(Cavity, toothache, catch) + \mathbf{P}(Cavity,\ toothache, \neg catch)\\ &+ \mathbf{P}(Cavity, \neg toothache,\ catch) + \mathbf{P}(Cavity, \neg toothache, \neg catch)\\ &= \langle 0.108, 0.016\rangle + \langle 0.012, 0.064\rangle + \langle 0.072, 0.144\rangle + \langle 0.008, 0.576\rangle\\ &= \langle 0.2,\ 0.8\rangle.\end{aligned}$$

Using the product rule (Equation (12.4)), we can replace **P**(**Y**, **z**) in Equation (12.7) by **P**(**Y** | **z**)*P*(**z**), obtaining a rule called **conditioning**:

$$\mathbf{P}(\mathbf{Y}) = \sum_{\mathbf{z}} \mathbf{P}(\mathbf{Y}|\mathbf{z})P(\mathbf{z}). \qquad (12.8)$$

Marginalization and conditioning turn out to be useful rules for all kinds of derivations involving probability expressions.

In most cases, we are interested in computing *conditional* probabilities of some variables, given evidence about others. Conditional probabilities can be found by first using Equation (12.3) to obtain an expression in terms of unconditional probabilities and then evaluating the expression from the full joint distribution. For example, we can compute the probability of a cavity, given evidence of a toothache, as follows:

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

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

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

The two values sum to 1.0, as they should. Notice that the term *P*(*toothache*) is in the denominator for both of these calculations. If the variable *Cavity* had more than two values, it would be in the denominator for all of them. In fact, it can be viewed as a **normalization** constant for the distribution **P**(*Cavity* | *toothache*), ensuring that it adds up to 1. Throughout the chapters dealing with probability, we use *a* to denote such constants. With this notation, we can write the two preceding equations in one:

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

In other words, we can calculate **P**(*Cavity* | *toothache*) even if we don't know the value of *P*(*toothache*)! We temporarily forget about the factor *1/P*(*toothache*) and add up the values for *cavity* and ¬*cavity*, getting 0.12 and 0.08. Those are the correct relative proportions, but they don't sum to 1, so we normalize them by dividing each one by 0.12 + 0.08, getting the true probabilities of 0.6 and 0.4. Normalization turns out to be a useful shortcut in many probability calculations, both to make the computation easier and to allow us to proceed when some probability assessment (such as *P*(*toothache*)) is not available.

From the example, we can extract a general inference procedure. We begin with the case in which the query involves a single variable, *X (Cavity* in the example). Let **E** be the list of evidence variables (just *Toothache* in the example), let **e** be the list of observed values for them, and let **Y** be the remaining unobserved variables (just *Catch* in the example). The query is **P**(*X* | **e**) and can be evaluated as

$$\mathbf{P}(X \mid \mathbf{e}) = \alpha\mathbf{P}(X, \mathbf{e}) = \alpha\sum_{\mathbf{y}}\mathbf{P}(X, \mathbf{e}, \mathbf{y}), \tag{12.9}$$

where the summation is over all possible **y** s (i.e., all possible combinations of values of the unobserved variables **Y**). Notice that together the variables *X*, **E**, and **Y** constitute the complete set of variables for the domain, so **P**(*X*, **e**, **y**) is simply a subset of probabilities from the full joint distribution.

Given the full joint distribution to work with, Equation (12.9) can answer probabilistic queries for discrete variables. It does not scale well, however: for a domain described by *n* Boolean variables, it requires an input table of size $O(2^n)$ and takes $O(2^n)$ time to process the table. In a realistic problem we could easily have *n* = 100, making $O(2^n)$ impractical—a table with $2^{100} \approx 10^{30}$ entries! The problem is not just memory and computation: the real issue is that if each of the 10^{30} probabilities has to be estimated separately from examples, the number of examples required will be astronomical.

For these reasons, the full joint distribution in tabular form is seldom a practical tool for building reasoning systems. Instead, it should be viewed as the theoretical foundation on which more effective approaches may be built, just as truth tables formed a theoretical foundation for more practical algorithms like DPLL in Chapter 7. The remainder of this chapter introduces some of the basic ideas required in preparation for the development of realistic systems in Chapter 13.

12.4 Independence

Let us expand the full joint distribution in Figure 12.3 by adding a fourth variable, *Weather*. The full joint distribution then becomes **P**(*Toothache, Catch, Cavity, Weather*), which has 2 × 2 × 2 × 4 = 32 entries. It contains four "editions" of the table shown in Figure 12.3, one for each kind of weather. What relationship do these editions have to each other and to the original three-variable table? How is the value of *P*(*toothache, catch, cavity, cloud*) related to the value of *P*(*toothache, catch, cavity*)? We can use the product rule (Equation (12.4)):

$$\begin{aligned}&P(toothache,\ catch,\ cavity,\ cloud)\\&\quad = P(cloud \mid toothache,\ catch,\ cavity)P(toothache,\ catch,\ cavity).\end{aligned}$$

Now, unless one is in the deity business, one should not imagine that one's dental problems influence the weather. And for indoor dentistry, at least, it seems safe to say that the weather does not influence the dental variables. Therefore, the following assertion seems reasonable:

$$P(cloud \mid toothache,\ catch,\ cavity) = P(cloud). \tag{12.10}$$

From this, we can deduce

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

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

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

Thus, the 32-element table for four variables can be constructed from one 8-element table and one 4-element table. This decomposition is illustrated schematically in Figure 12.4(a).

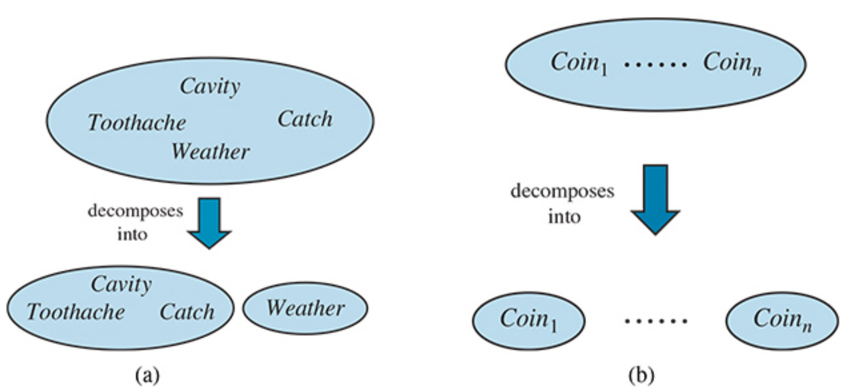


Figure 12.4 Two examples of factoring a large joint distribution into smaller distributions, using absolute independence. (a) Weather and dental problems are independent. (b) Coin flips are independent.

The property we used in Equation (12.10) is called **independence** (also **marginal independence** and **absolute independence**). In particular, the weather is independent of one's dental problems. Independence between propositions a and b can be written as

$$P(a|b) = P(a) \quad \text{or} \quad P(b|a) = P(b) \quad \text{or} \quad P(a \wedge b) = P(a)P(b). \tag{12.11}$$

All these forms are equivalent (Exercise 12.INDI). Independence between variables X and Y can be written as follows (again, these are all equivalent):

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

Independence assertions are usually based on knowledge of the domain. As the toothache–weather example illustrates, they can dramatically reduce the amount of information necessary to specify the full joint distribution. If the complete set of variables can be divided into independent subsets, then the full joint distribution can be *factored* into separate joint distributions on those subsets. For example, the full joint distribution on the outcome of n independent coin flips, $\mathbf{P}(C_1,..., C_n)$, has 2^n entries, but it can be represented as the product of n single-variable distributions $\mathbf{P}(C_i)$. In a more practical vein, the independence of dentistry and meteorology is a good thing, because otherwise the practice of dentistry might require intimate knowledge of meteorology, and vice versa.

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

12.5 Bayes' Rule and Its Use

On page 408, we defined the **product rule** (Equation (12.4)). It can actually be written in two forms:

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

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

$$P(b\Big|a) = \frac{P(a|b)P(b)}{P(a)}. \qquad (12.12)$$

This equation is known as **Bayes' rule** (also Bayes' law or Bayes' theorem). This simple equation underlies most modern AI systems for probabilistic inference.

The more general case of Bayes' rule for multivalued variables can be written in the **P** notation as follows:

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

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

$$\mathbf{P}(Y\Big|X,\mathbf{e}) = \frac{\mathbf{P}(X|Y,\mathbf{e})\,\mathbf{P}(Y|\mathbf{e})}{\mathbf{P}(X|\mathbf{e})}. \qquad (12.13)$$

12.5.1 Applying Bayes' rule: The simple case

On the surface, Bayes' rule does not seem very useful. It allows us to compute the single term *P*(*b* | *a*) in terms of three terms: *P*(*a* | *b*), *P*(*b*), and *P*(*a*). That seems like two steps backwards; but Bayes' rule is useful in practice because there are many cases where we do have good probability estimates for these three numbers and need to compute the fourth. Often, we perceive as evidence the *effect* of some unknown *cause* and we would like to determine that cause. In that case, Bayes' rule becomes

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

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

For example, a doctor knows that the disease meningitis causes a patient to have a stiff neck, say, 70% of the time. The doctor also knows some unconditional facts: the prior probability that any patient has meningitis is 1/50,000, and the prior probability that any patient has a stiff neck is 1%. Letting *s* be the proposition that the patient has a stiff neck and *m* be the proposition that the patient has meningitis, we have

$$P(s|m) = 0.7$$

$$P(m) = 1/50000$$

$$P(s) = 0.01$$

$$P(m\Big|s) = \frac{P(s|m)P(m)}{P(s)} = \frac{0.7 \times 1/50000}{0.01} = 0.0014. \qquad (12.14)$$

That is, we expect only 0.14% of patients with a stiff neck to have meningitis. Notice that even though a stiff neck is quite strongly indicated by meningitis (with probability 0.7), the probability of meningitis in patients with stiff necks remains small. This is because the prior probability of stiff necks (from any cause) is much higher than the prior for meningitis.

Section 12.3 illustrated a process by which one can avoid assessing the prior probability of the evidence (here, $P(s)$) by instead computing a posterior probability for each value of the query variable (here, m and $\neg m$) and then normalizing the results. The same process can be applied when using Bayes' rule. We have

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

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

$$\mathbf{P}(Y|X) \ = \ \alpha\mathbf{P}(X|Y)\mathbf{P}(Y), \qquad (12.15)$$

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

One obvious question to ask about Bayes' rule is why one might have available the conditional probability in one direction, but not the other. In the meningitis domain, perhaps the doctor knows that a stiff neck implies meningitis in 1 out of 5000 cases; that is, the doctor has quantitative information in the **diagnostic** direction from symptoms to causes. Such a doctor has no need to use Bayes' rule.

Unfortunately, *diagnostic knowledge is often more fragile than causal knowledge.* If there is a sudden epidemic of meningitis, the unconditional probability of meningitis, $P(m)$, will go up. The doctor who derived the diagnostic probability $P(m \mid s)$ directly from statistical observation of patients before the epidemic will have no idea how to update the value, but the doctor who computes $P(m \mid s)$ from the other three values will see that $P(m \mid s)$ should go up proportionately with $P(m)$. Most important, the causal information $P(s \mid m)$ is *unaffected* by the epidemic, because it simply reflects the way meningitis works. The use of this kind of direct causal or model-based knowledge provides the crucial robustness needed to make probabilistic systems feasible in the real world.

12.5.2 Using Bayes' rule: Combining evidence

We have seen that Bayes' rule can be useful for answering probabilistic queries conditioned on one piece of evidence—for example, the stiff neck. In particular, we have argued that probabilistic information is often available in the form $P(\mathit{effect} \mid \mathit{cause})$. What happens when we have two or more pieces of evidence? For example, what can a dentist conclude if her nasty steel probe catches in the aching tooth of a patient? If we know the full joint distribution (Figure 12.3), we can read off the answer:

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

We know, however, that such an approach does not scale up to larger numbers of variables. We can try using Bayes' rule to reformulate the problem:

$$\begin{aligned}&\mathbf{P}(\mathit{Cavity}|\mathit{toothache} \wedge \mathit{catch})\\ &\quad = \alpha\,\mathbf{P}(\mathit{toothache} \wedge \mathit{catch}|\mathit{Cavity})\mathbf{P}(\mathit{Cavity}).\end{aligned} \qquad (12.16)$$

For this reformulation to work, we need to know the conditional probabilities of the conjunction $\mathit{toothache} \wedge \mathit{catch}$ for each value of *Cavity*. That might be feasible for just two evidence variables, but again it does not scale up. If there are n possible evidence variables (X rays, diet, oral hygiene, etc.), then there are $O(2^n)$ possible combinations of observed values for which we would need to know conditional probabilities. This is no better than using the full joint distribution.

To make progress, we need to find some additional assertions about the domain that will enable us to simplify the expressions. The notion of **independence** in Section 12.4 provides a clue, but needs refining. It would be nice if *Toothache* and *Catch* were independent, but they are not: if the probe catches in the tooth, then it is likely that the tooth has a cavity and that the cavity causes a toothache. These variables *are* independent, however, *given the presence or the absence of a cavity.* Each is directly caused by the cavity, but neither has a direct effect on the other: toothache depends on the state of the nerves in the tooth, whereas the probe's accuracy depends primarily on the dentist's skill, to which the toothache is irrelevant.[7] Mathematically, this property is written as

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

This equation expresses the **conditional independence** of *toothache* and *catch* given *Cavity.* We can plug it into Equation (12.16) to obtain the probability of a cavity:

$$\mathbf{P}(Cavity|toothache \wedge catch)$$
$$= \alpha\, \mathbf{P}(toothache|Cavity)\mathbf{P}(catch|\ Cavity)\mathbf{P}(Cavity). \qquad (12.18)$$

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

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

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

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

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

Notice that this assertion is somewhat stronger than Equation (12.17), which asserts independence only for specific values of *Toothache* and *Catch.* As with absolute independence in Equation (12.11), the equivalent forms

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

can also be used (see Exercise 12.PXYZ). Section 12.4 showed that absolute independence assertions allow a decomposition of the full joint distribution into much smaller pieces. It turns out that the same is true for conditional independence assertions. For example, given the assertion in Equation (12.19), we can derive a decomposition as follows:

$$\mathbf{P}(Toothache, Catch,\ Cavity)$$
$$=\ \mathbf{P}(Toothache, Catch|Cavity)\mathbf{P}(Cavity) \quad (product\ rule)$$
$$=\ \mathbf{P}\ (Toothache|Cavity)\mathbf{P}(Catch|Cavity)\mathbf{P}(Cavity) \qquad (\text{using } 12.19).$$

(The reader can easily check that this equation does in fact hold in Figure 12.3.) In this way, the original large table is decomposed into three smaller tables. The original table has 7 independent numbers. (The table has $2^3 = 8$ entries, but they must sum to 1, so 7 are independent). The smaller tables contain a total of 2 + 2 + 1 = 5 independent numbers. (For a conditional probability distribution such as **P**(*Toothache* | *Cavity*) there are two rows of two numbers, and each row sums to 1, so that's two independent numbers; for a prior distribution such as **P**(*Cavity*) there is only one independent number.) Going from 7 to 5 might not seem like a major triumph, but the gains can be much greater with larger number of symptoms.

In general, for *n* symptoms that are all conditionally independent given *Cavity,* the size of the representation grows as $O(n)$ instead of $O(2^n)$. That means that *conditional independence assertions can allow probabilistic systems to scale up; moreover, they are much more commonly available than absolute independence assertions.* Conceptually, *Cavity* **separates** *Toothache* and *Catch* because it is a direct cause of both of them. The decomposition of large probabilistic domains into weakly connected subsets through conditional independence is one of the most important developments in the recent history of AI.

12.6 Naive Bayes Models

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

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

Such a probability distribution is called a **naive Bayes** model—"naive" because it is often used (as a simplifying assumption) in cases where the "effect" variables are *not* strictly independent given the cause variable. (The naive Bayes model is sometimes called a **Bayesian classifier**, a somewhat careless usage that has prompted true Bayesians to call it the **idiot Bayes** model.) In practice, naive Bayes systems often work very well, even when the conditional independence assumption is not strictly true.

To use a naive Bayes model, we can apply Equation (12.20) to obtain the probability of the cause given some observed effects. Call the observed effects **E** = **e**, while the remaining effect variables **Y** are unobserved. Then the standard method for inference from the joint distribution (Equation (12.9)) can be applied:

$$\mathbf{P}(Cause\,\Big|\,\mathbf{e}) = \alpha \sum_{\mathbf{y}} \mathbf{P}\,(Cause,\, \mathbf{e}, \mathbf{y}).$$

From Equation (12.20), we then obtain

$$\begin{aligned}
\mathbf{P}(Cause|\mathbf{e}) &= \alpha \sum_{\mathbf{y}} \mathbf{P} \\
&= \alpha\, \mathbf{P}(Cause)(\prod_j \mathbf{P}(e_j|Cause)) \sum_{\mathbf{y}} \mathbf{P}(\mathbf{y} \\
&= \alpha\, \mathbf{P}(Cause) \prod_j \mathbf{P}(e_j|Ca
\end{aligned}$$

where the last line follows because the summation over **y** is 1. Reinterpreting this equation in words: for each possible cause, multiply the prior probability of the cause by the product of the conditional probabilities of the observed effects given the cause; then normalize the result. The run time of this calculation is linear in the number of observed effects and does not depend on the number of unobserved effects (which may be very large in domains such as medicine). We will see in the next chapter that this is a common phenomenon in probabilistic inference: evidence variables whose values are unobserved usually "disappear" from the computation altogether.

12.6.1 Text classification with naive Bayes

Let's see how a naive Bayes model can be used for the task of **text classification:** given a text, decide which of a predefined set of classes or categories it belongs to. Here the "cause" is the *Category* variable, and the "effect" variables are the presence or absence of certain key words, $HasWord_i$. Consider these two example sentences, taken from newspaper articles:

1. Stocks rallied on Monday, with major indexes gaining 1% as optimism persisted over the first quarter earnings season.

2. Heavy rain continued to pound much of the east coast on Monday, with flood warnings issued in New York City and other locations.

The task is to classify each sentence into a *Category*—the major sections of the newspaper: *news, sports, business, weather,* or *entertainment.* The naive Bayes model consists of the prior probabilities $\mathbf{P}(Category)$ and the conditional probabilities $\mathbf{P}(HasWord_i| \, Category)$. For each category c, $P(Category = c)$ is estimated as the fraction of all previously seen documents that are of category c. For example, if 9% of articles are about weather, we set $P(Category = weather) = 0.09$. Similarly, $\mathbf{P}(HasWord_i \mid Category)$ is estimated as the fraction of documents of each category that contain word i; perhaps 37% of articles about business contain word 6, "stocks," so $P(HasWord_6 = true \mid Category = business)$ is set to 0.37.[8]

To categorize a new document, we check which key words appear in the document and then apply Equation (12.21) to obtain the posterior probability distribution over categories. If we have to predict just one category, we take the one with the highest posterior probability. Notice that, for this task, every effect variable is observed, since we can always tell whether a given word appears in the document.

The naive Bayes model assumes that words occur independently in documents, with frequencies determined by the document category. This independence assumption is clearly violated in practice. For example, the phrase "first quarter" occurs more frequently in business (or sports) articles than would be suggested by multiplying the probabilities of "first" and "quarter." The violation of independence usually means that the final posterior probabilities will be much closer to 1 or 0 than they should be; in other words, the model is overconfident in its predictions. On the other hand, even with these errors, the *ranking* of the possible categories is often quite accurate.

Naive Bayes models are widely used for language determination, document retrieval, spam filtering, and other classification tasks. For tasks such as medical diagnosis, where the actual values of the posterior probabilities really matter—for example, in deciding whether to perform an appendectomy—one would usually prefer to use the more sophisticated models described in the next chapter.

12.7 The Wumpus World Revisited

We can combine the ideas in this chapter to solve probabilistic reasoning problems in the wumpus world. (See Chapter 7 for a complete description of the wumpus world.) Uncertainty arises in the wumpus world because the agent's sensors give only partial information about the world. For example, Figure 12.5 shows a situation in which each of the three unvisited but reachable squares—[1,3], [2,2], and [3,1]—might contain a pit. Pure logical inference can conclude nothing about which square is most likely to be safe, so a logical agent might have to choose randomly. We will see that a probabilistic agent can do much better than the logical agent.

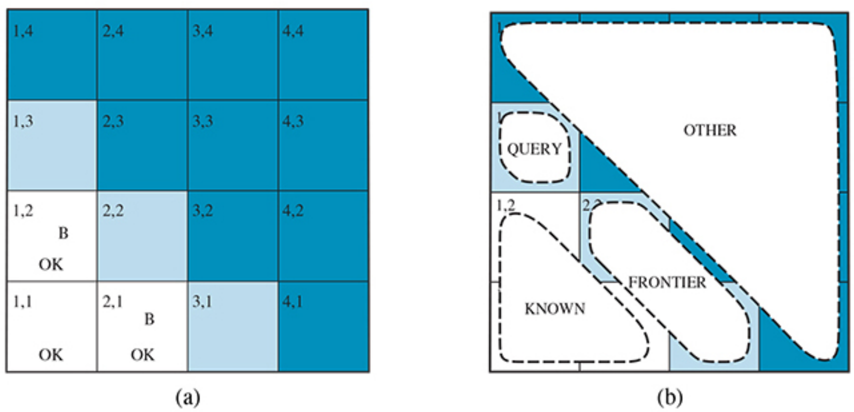


Figure 12.5 (a) After finding a breeze in both [1,2] and [2,1], the agent is stuck—there is no safe place to explore. (b) Division of the squares into *Known*, *Frontier*, and *Other*, for a query about [1,3].

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

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

The next step is to specify the full joint distribution, $\mathbf{P}(P_{1,1},...,P_{4,4},B_{1,1},B_{12},B_{21})$. Applying the product rule, we have

$$\begin{aligned}\mathbf{P}(P_{1,1}, \ldots, P_{4,4}, B_{1,1}, B_{1,2}, B_{2,1}) &= \\ &\mathbf{P}(B_{1,1}, B_{1,2}, B_{2,1} \mid P_{1,1}, \ldots, P_{4,4})\mathbf{P}(\mathrm{P}_{1,1}, \ldots, P_{4,4})\,.\end{aligned}$$

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

$$\mathbf{P}(P_{1,1},\ldots,P_{4,4}) = \prod_{\mathrm{i},\, j=1,1}^{4,4} \mathbf{P}(P_{i,j}). \qquad (12.22)$$

For a particular configuration with exactly n pits, the probability is $0.2^{n} \times 0.8^{16-n}$.

In the situation in Figure 12.5(a), the evidence consists of the observed breeze (or its absence) in each square that is visited, combined with the fact that each such square contains no pit. We abbreviate these facts as $b = \neg b_{1,1} \wedge b_{1,2} \wedge b_{2,1}$ and $known = \neg p_{1,1} \wedge \neg p_{1,2} \wedge \neg p_{2,1}$. We are interested in answering queries such as $\mathbf{P}(P_{1,3} \mid known, b)$: how likely is it that [1,3] contains a pit, given the observations so far?

To answer this query, we can follow the standard approach of Equation (12.9), namely, summing over entries from the full joint distribution. Let *Unknown* be the set of $P_{i,j}$ variables for squares other than the known squares and the query square [1,3]. Then, by Equation (12.9), we have

$$\mathbf{P}(P_{1,3} \mid known, b) = \alpha \sum_{unknown} \mathbf{P}(P_{1,3}, known, b, unknown). \qquad (12.23)$$

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

Surely, one might ask, aren't the other squares irrelevant? How could [4,4] affect whether [1,3] has a pit? Indeed, this intuition is roughly correct, but it needs to be made more precise. What we really mean is that if we knew the values of all the pit variables adjacent to the squares we care about, then pits (or their absence) in other, more distant squares could have no further effect on our belief.

Let *Frontier* be the pit variables (other than the query variable) that are adjacent to visited squares, in this case just [2,2] and [3,1]. Also, let *Other* be the pit variables for the other unknown squares; in this case, there are 10 other squares, as shown in Figure 12.5(b). With these definitions, $Unknown = Frontier \cup Other$. The key insight given above can now be stated as follows: the observed breezes are *conditionally independent* of the other variables, given the known, frontier, and query variables. To use this insight, we manipulate the query formula into a form in which the breezes are conditioned on all the other variables, and then we apply conditional independence:

$$\begin{aligned}
&\mathbf{P}(P_{1,3} \mid known, b) \\
&= \alpha \sum_{unknown} \mathbf{P}(P_{1,3}, known, b, unknown) \quad \text{(from Equation (12.23))} \\
&= \alpha \sum_{unknown} \mathbf{P}(b \mid P_{1,3}, known, unknown)\mathbf{P}(P_{1,3}, known, unknown) \ (\textit{product rule}) \\
&= \alpha \sum_{frontier} \sum_{other} \mathbf{P}(b \mid known, P_{1,3}, frontier, other)\mathbf{P}(P_{1,3}, known, frontier, other) \\
&= \alpha \sum_{frontier} \sum_{other} \mathbf{P}(b \mid known, P_{1,3}, frontier)\mathbf{P}(P_{1,3}, known, frontier, other)\ ,
\end{aligned}$$

where the final step uses conditional independence: b is independent of *other* given *known*, $P_{1,3}$, and *frontier*. Now, the first term in this expression does not depend on the *Other* variables, so we can move the summation inward:

$$\mathbf{P}(P_{1,3}|\ known, b)$$

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

By independence, as in Equation (12.22), the term on the right can be factored, and then the terms can be reordered:

$$\mathbf{P}(P_{1,3}|\ known, b)$$

$$= \alpha \sum_{frontier} \mathbf{P}(b \middle| known, p_{1,3}, frontier) \sum_{other} \mathbf{P}(P_{1,3}) P(known) P(\ frontier) P(other)$$

$$= \alpha P(know) \mathbf{P}(P_{1,3}) \sum_{frontier} \mathbf{P}(b \middle| known, p_{1,3}, frontier)\ P(frontier) \sum_{other} P(other)$$

$$= \alpha' \mathbf{P}(P_{1,3}) \sum_{frontier} \mathbf{P}(b | known, p_{1,3}, frontier)\ P(frontier).$$

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

Now, there are just four terms in the summation over the frontier variables, $P_{2,2}$ and $P_{3,1}$. The use of independence and conditional independence has completely eliminated the other squares from consideration.

Notice that the probabilities in $\mathbf{P}(b \mid known,\ P_{1,3},\ frontier)$ are 1 when the breeze observations are consistent with the other variables and 0 otherwise. Thus, for each value of $P_{1,3}$, we sum over the *logical models* for the frontier variables that are consistent with the known facts. (Compare with the enumeration over models in Figure 7.5 on page 233.) The models and their associated prior probabilities—*P*(*frontier*)—are shown in Figure 12.6. We have

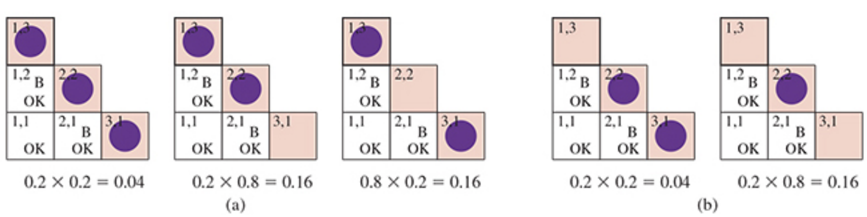


Figure 12.6 Consistent models for the frontier variables, $P_{2,2}$ and $P_{3,1}$, showing *P(frontier)* for each model: (a) three models with $P_{1,3}$= *true* showing two or three pits, and (b) two models with $P_{1,3}$ = *false*

showing one or two pits.

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

That is, [1,3] (and [3,1] by symmetry) contains a pit with roughly 31% probability. A similar calculation, which the reader might wish to perform, shows that [2,2] contains a pit with roughly 86% probability. The wumpus agent should definitely avoid [2,2]! Note that our logical agent from Chapter 7 did not know that [2,2] was worse than the other squares. Logic can tell us that it is unknown whether there is a pit in [2, 2], but we need probability to tell us how likely it is.

What this section has shown is that even seemingly complicated problems can be formulated precisely in probability theory and solved with simple algorithms. To get *efficient* solutions, independence and conditional independence relationships can be used to simplify the summations required. These relationships often correspond to our natural understanding of how the problem should be decomposed. In the next chapter, we develop formal representations for such relationships as well as algorithms that operate on those representations to perform probabilistic inference efficiently.

Summary

This chapter has suggested probability theory as a suitable foundation for uncertain reasoning and provided a gentle introduction to its use.

- Uncertainty arises because of both laziness and ignorance. It is inescapable in complex, nondeterministic, or partially observable environments.
- **Probabilities** express the agent’s inability to reach a definite decision regarding the truth of a sentence. Probabilities summarize the agent’s beliefs relative to the evidence.
- **Decision theory** combines the agent’s beliefs and desires, defining the best action as the one that maximizes expected **utility**.
- Basic probability statements include **prior** or **unconditional probabilities** and **posterior** or **conditional probabilities** over simple and complex propositions.
- The axioms of probability constrain the probabilities of logically related propositions. An agent that violates the axioms must behave irrationally in some cases.
- The **full joint probability distribution** specifies the probability of each complete assignment of values to random variables. It is usually too large to create or use in its explicit form, but when it is available it can be used to answer queries simply by adding up entries for the possible worlds corresponding to the query propositions.
- **Absolute independence** between subsets of random variables allows the full joint distribution to be factored into smaller joint distributions, greatly reducing its complexity.

- **Bayes' rule** allows unknown probabilities to be computed from known conditional probabilities, usually in the causal direction. Applying Bayes' rule with many pieces of evidence runs into the same scaling problems as does the full joint distribution.
- **Conditional independence** brought about by direct causal relationships in the domain allows the full joint distribution to be factored into smaller, conditional distributions. The **naive Bayes** model assumes the conditional independence of all effect variables, given a single cause variable; its size grows linearly with the number of effects.
- A wumpus-world agent can calculate probabilities for unobserved aspects of the world, thereby improving on the decisions of a purely logical agent. Conditional independence makes these calculations tractable.

Bibliographical and Historical Notes

Probability theory was invented as a way of analyzing games of chance. In about 850 CE the Indian mathematician Mahaviracarya described how to arrange a set of bets that can't lose (what we now call a Dutch book). In Europe, the first significant systematic analyses were produced by Girolamo Cardano around 1565, although publication was posthumous (1663). By that time, probability had been established as a mathematical discipline due to a series of results from a famous correspondence between Blaise Pascal and Pierre de Fermat in 1654. The first published textbook on probability was *De Ratiociniis in Ludo Aleae* (On Reasoning in a Game of Chance) by Huygens (1657). The "laziness and ignorance" view of uncertainty was described by John Arbuthnot in the preface of his translation of Huygens (Arbuthnot, 1692): "It is impossible for a Die, with such determin'd force and direction, not to fall on such determin'd side, only I don't know the force and direction which makes it fall on such determin'd side, and therefore I call it Chance, which is nothing but the want of art."

The connection between probability and reasoning dates back at least to the nineteenth century: in 1819, Pierre Laplace said, "Probability theory is nothing but common sense reduced to calculation." In 1850, James Maxwell said, "The true logic for this world is the calculus of Probabilities, which takes account of the magnitude of the probability which is, or ought to be, in a reasonable man's mind."

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

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

The **subjectivist** view describes probabilities as a way of characterizing an agent’s beliefs, rather than as having any external physical significance. The subjective **Bayesian** view allows any self-consistent ascription of prior probabilities to propositions, but then insists on proper Bayesian updating as evidence arrives.

Even a strict frequentist position involves subjectivity because of the **reference class** problem: in trying to determine the outcome probability of a *particular* experiment, the frequentist has to place it in a reference class of “similar” experiments with known outcome frequencies. But what’s the right class? I. J. Good wrote, “every event in life is unique, and every real-life probability that we estimate in practice is that of an event that has never occurred before” (Good, 1983, p. 27).

For example, given a particular patient, a frequentist who wants to estimate the probability of a cavity will consider a reference class of other patients who are similar in important ways—age, symptoms, diet—and see what proportion of them had a cavity. If the dentist considers everything that is known about the patient—hair color, weight to the nearest gram, mother's maiden name—then the reference class becomes empty. This has been a vexing problem in the philosophy of science.

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

Leibniz introduced the "classical" notion of probability as a proportion of enumerated, equally probable cases, which was also used by Bernoulli, although it was brought to prominence by Laplace (1816). This notion is ambiguous between the frequency interpretation and the subjective interpretation. The cases can be thought to be equally probable either because of a natural, physical symmetry between them, or simply because we do not have any knowledge that would lead us to consider one more probable than another. The use of this latter, subjective consideration to justify assigning equal probabilities is known as the **principle of indifference**. The principle is often attributed to Laplace (1816), but he never used the name explicitly; Keynes (1921) did. George Boole and John Venn both referred to it as the **principle of insufficient reason**.

The debate between objectivists and subjectivists became sharper in the 20th century. Kolmogorov (1963), R. A. Fisher (1922), and Richard von Mises (1928) were advocates of the relative frequency interpretation. Karl

Popper’s “propensity” interpretation (1959, first published in German in 1934) traces relative frequencies to an underlying physical symmetry. Frank Ramsey (1931), Bruno de Finetti (1937), R. T. Cox (1946), Leonard Savage (1954), Richard Jeffrey (1983), and E. T. Jaynes (2003) interpreted probabilities as the degrees of belief of specific individuals. Their analyses of degree of belief were closely tied to utilities and to behavior—specifically, to the willingness to place bets.

Rudolf Carnap offered a different interpretation of probability—not as the degree of belief that an individual actually has, but as the degree of belief that an idealized reasoner *should* have in a particular proposition *a*, given a particular body of evidence e. Carnap attempted to make this notion of degree of **confirmation** mathematically precise, as a logical relation between *a* and e. Currently it is believed that there is no unique logic of this kind; rather, any such logic rests on a subjective prior probability distribution whose effect is diminished as more observations are collected.

The study of this relation was intended to constitute a mathematical discipline called **inductive logic**, analogous to ordinary deductive logic (Carnap, 1948, 1950). Carnap was not able to extend his inductive logic much beyond the propositional case, and Putnam (1963) showed by adversarial arguments that some difficulties were inherent. More recent work by Bacchus, Grove, Halpern, and Koller (1992) extends Carnap’s methods to first-order theories.

The first rigorously axiomatic framework for probability theory was proposed by Kolmogorov (1950, first published in German in 1933). Renyi (1970) later gave an axiomatic presentation that took conditional probability, rather than absolute probability, as primitive.

In addition to de Finetti’s arguments for the validity of the axioms, Cox (1946) showed that any system for uncertain reasoning that meets his set of assumptions is equivalent to probability theory. This gave renewed confidence to probability fans, but others were not convinced, objecting to the assumption that belief must be represented by a single number. Halpern (1999) describes the assumptions and shows some gaps in Cox's original formulation. Horn (2003) shows how to patch up the difficulties. Jaynes (2003) has a similar argument that is easier to read.

The Rev. Thomas Bayes (1702-1761) introduced the rule for reasoning about conditional probabilities that was posthumously named after him (Bayes, 1763). Bayes only considered the case of uniform priors; it was Laplace who independently developed the general case. Bayesian probabilistic reasoning has been used in AI since the 1960s, especially in medical diagnosis. It was used not only to make a diagnosis from available evidence, but also to select further questions and tests by using the theory of information value (Section 15.6) when available evidence was inconclusive (Gorry, 1968; Gorry et *al.*, 1973). One system outperformed human experts in the diagnosis of acute abdominal illnesses (de Dombal et *al*., 1974). Lucas et *al.* (2004) provide an overview.

These early Bayesian systems suffered from a number of problems. Because they lacked any theoretical model of the conditions they were diagnosing, they were vulnerable to unrepresentative data occurring in situations for which only a small sample was available (de Dom- bal et *al.*, 1981). Even more fundamentally, because they lacked a concise formalism (such as the one to be described in Chapter 13) for representing and using conditional independence information, they depended on the acquisition, storage, and processing of enormous tables of probabilistic data. Because of these difficulties, probabilistic methods for coping with uncertainty fell out of favor in AI from the 1970s to the mid-1980s. Developments since the late 1980s are described in the next chapter.

The naive Bayes model for joint distributions has been studied extensively in the pattern recognition literature since the 1950s (Duda and Hart, 1973). It has also been used, often unwittingly, in information retrieval, beginning with the work of Maron (1961). The probabilistic foundations of this technique, described further in Exercise 12.BAYS, were elucidated by Robertson and Sparck Jones (1976). Domingos and Pazzani (1997) provide an explanation for the surprising success of naive Bayesian reasoning even in domains where the independence assumptions are clearly violated.

There are many good introductory textbooks on probability theory, including those by Bertsekas and Tsitsiklis (2008), Ross (2015), and Grinstead and Snell (1997). DeGroot and Schervish (2001) offer a combined introduction to probability and statistics from a Bayesian standpoint, and Walpole *et al.* (2016) offer an introduction for scientists and engineers. Jaynes (2003) gives a very persuasive exposition of the Bayesian approach. Billingsley (2012) and Venkatesh (2012) provide more mathematical treatments, including all the complications with continuous variables that we have left out. Hacking (1975) and Hald (1990) cover the early history of the concept of probability, and Bernstein (1996) gives a popular account.

---

[1] For now, we assume a discrete, countable set of worlds. The proper treatment of the continuous case brings in certain complications that are less relevant for most purposes in AI.

[2] Note that the precedence of "|" is such that any expression of the form *P(...|...)* always means *P((...)|(...))*.

[3] These conventions taken together lead to a potential ambiguity in notation when summing over values of a Boolean variable: *P(a)* is the probability that *A* is *true*, whereas in the expression $\sum_a P(a)$ it just refers to the probability of one of the values *a* of *A*.

[4] The difficulties include the **Vitali set**, a well-defined subset of the interval [0,1] with no well-defined size.

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

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

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

[8] One needs to be careful not to assign probability zero to words that have not been seen previously in a given category of documents, since the zero would wipe out all the other evidence in Equation (12.21). Just because you haven't seen a word yet doesn't mean you will *never* see it. Instead, reserve a small portion of the probability distribution to represent "previously unseen" words. See Chapter 21 for more on this issue in general, and Section 24.1.4 for the particular case of word models.

CHAPTER 13

PROBABILISTIC REASONING

In which we explain how to build efficient network models to reason under uncertainty according to the laws of probability theory, and how to distinguish between correlation and causality.

Chapter 12 introduced the basic elements of probability theory and noted the importance of independence and conditional independence relationships in simplifying probabilistic representations of the world. This chapter introduces a systematic way to represent such relationships explicitly in the form of **Bayesian networks**. We define the syntax and semantics of these networks and show how they can be used to capture uncertain knowledge in a natural and efficient way. We then show how probabilistic inference, although computationally intractable in the worst case, can be done efficiently in many practical situations. We also describe a variety of approximate inference algorithms that are often applicable when exact inference is infeasible. Chapter 18 extends the basic ideas of Bayesian networks to more expressive formal languages for defining probability models.

13.1 Representing Knowledge in an Uncertain Domain

In Chapter 12, we saw that the full joint probability distribution can answer any question about the domain, but can become intractably large as the number of variables grows. Furthermore, specifying probabilities for possible worlds one by one is unnatural and tedious.

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

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

1. Each node corresponds to a random variable, which may be discrete or continuous.
2. Directed links or arrows connect pairs of nodes. If there is an arrow from node X to node Y, X is said to be a *parent* of Y. The graph has no directed cycles and hence is a directed acyclic graph, or DAG.
3. Each node X_i has associated probability information $\theta(X_i \mid Parents(X_i))$ that quantifies the effect of the parents on the node using a finite number of **parameters**.

The topology of the network—the set of nodes and links—specifies the conditional independence relationships that hold in the domain, in a way that will be made precise shortly. The *intuitive* meaning of an arrow is typically that *X* has a *direct influence* on *Y*, which suggests that causes should be parents of effects. It is usually easy for a domain expert to decide what direct influences exist in the domain—much easier, in fact, than actually specifying the probabilities themselves. Once the topology of the Bayes net is laid out, we need only specify the local probability information for each variable, in the form of a conditional distribution given its parents. The full joint distribution for all the variables is defined by the topology and the local probability information.

Recall the simple world described in Chapter 12, consisting of the variables *Toothache*, *Cavity*, *Catch*, and *Weather*. We argued that *Weather* is independent of the other variables; furthermore, we argued that *Toothache* and *Catch* are conditionally independent, given *Cavity*. These relationships are represented by the Bayes net structure shown in Figure 13.1. Formally, the conditional independence of *Toothache* and *Catch*, given *Cavity*, is indicated by the *absence* of a link between *Toothache* and *Catch*. Intuitively, the network represents the fact that *Cavity* is a direct cause of *Toothache* and *Catch*, whereas no direct causal relationship exists between *Toothache* and *Catch*.

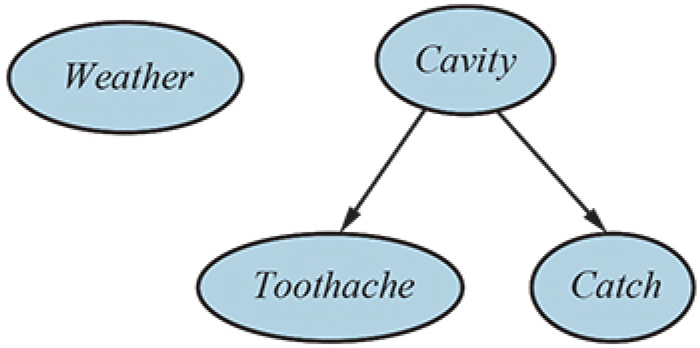


Figure 13.1 A simple Bayesian network in which *Weather* is independent of the other three variables and *Toothache* and *Catch* are conditionally independent, given *Cavity*.

Now consider the following example, which is just a little more complex. You have a new burglar alarm installed at home. It is fairly reliable at detecting a burglary, but is occasionally set off by minor earthquakes. (This example is due to Judea Pearl, a resident of earthquake-prone Los Angeles.) You also have two neighbors, John and Mary, who have promised to call you at work when they hear the alarm. John nearly always calls when he hears the alarm, but sometimes confuses the telephone ringing with the alarm and calls then, too. Mary, on the other hand, likes rather loud music and often misses the alarm altogether. Given the evidence of who has or has not called, we would like to estimate the probability of a burglary.

A Bayes net for this domain appears in Figure 13.2. The network structure shows that burglary and earthquakes directly affect the probability of the alarm's going off, but whether John and Mary call depends only on the alarm. The network thus represents our assumptions that they do not perceive burglaries directly, they do not notice minor earthquakes, and they do not confer before calling.

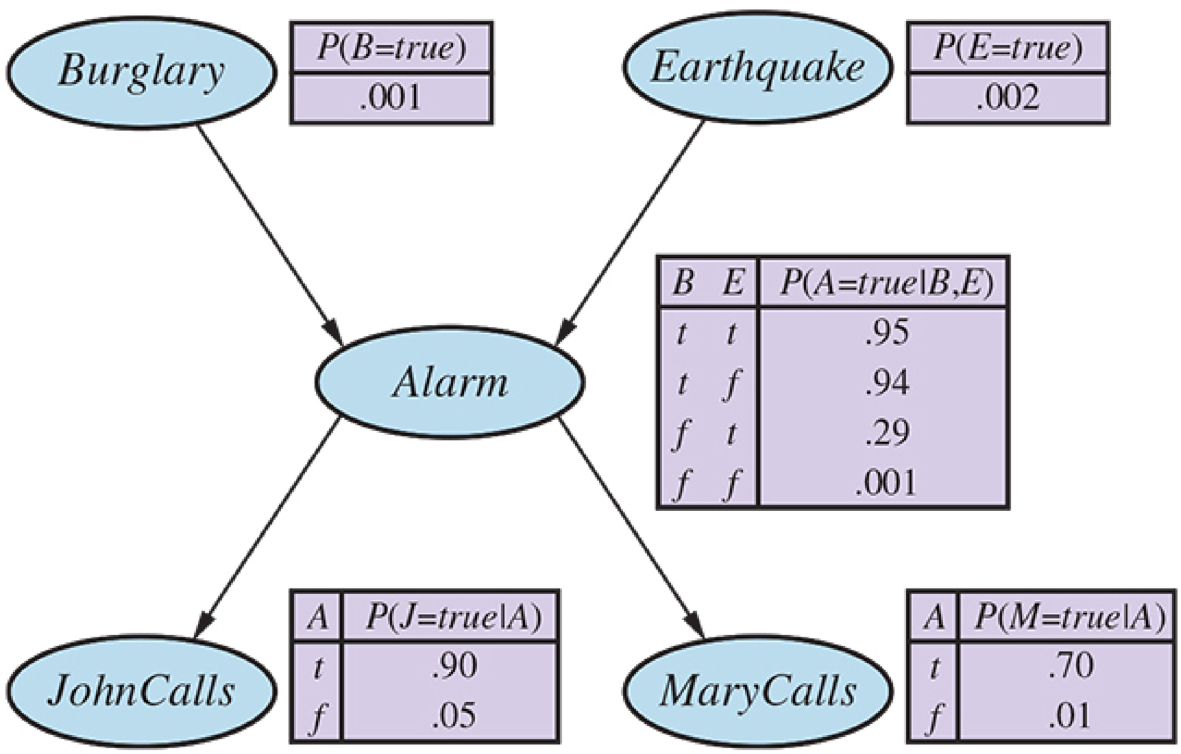


Figure 13.2 A typical Bayesian network, showing both the topology and the conditional probability tables (CPTs). In the CPTs, the letters *B*, *E*, *A*, *J*, and *M* stand for *Burglary*, *Earthquake*, *Alarm*, *JohnCalls*, and *MaryCalls*, respectively.

The local probability information attached to each node in Figure 13.2 takes the form of a **conditional probability table (CPT)**. (CPTs can be used only for discrete variables; other representations, including those suitable for continuous variables, are described in Section 13.2.) Each row in a CPT contains the conditional probability of each node value for a

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

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

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

13.2 The Semantics of Bayesian Networks

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

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

$$P(x_1, \ldots, x_n) = \prod_{i=1}^{n} \theta(x_i|\text{parents}(X_i)), \qquad (13.1)$$

where parents (X_i) denotes the values of Parents (X_i) that appear in $x_1,...,x_n$. Thus, each entry in the joint distribution is represented by the product of the appropriate elements of the local conditional distributions in the Bayes net.

To illustrate this, we can calculate the probability that the alarm has sounded, but neither a burglary nor an earthquake has occurred, and both John and Mary call. We simply multiply the relevant entries from the local conditional distributions (abbreviating the variable names):

$$\begin{aligned} P(j, m, a, \neg b, \neg e) &= P(j|a)P(m|a)P(a|\neg b \wedge \neg e)P(\neg b)P(\neg e) \\ &= 0.90 \times 0.70 \times 0.001 \times 0.999 \times 0.998 = 0.000628. \end{aligned}$$

Section 12.3 explained that the full joint distribution can be used to answer any query about the domain. If a Bayes net is a representation of the joint distribution, then it too can be used to answer any query, by summing all the relevant joint probability values, each calculated by multiplying probabilities from the local conditional distributions. Section 13.3 explains this in more detail, but also describes methods that are much more efficient.

So far, we have glossed over one important point: what is the meaning of the numbers that go into the local conditional distributions $\theta\ (x_i \mid parents(X_i))$? It turns out that from Equation (13.1) we can prove that the parameters $\theta\ (x_i \mid parents(X_i))$ are exactly the conditional probabilities $P(x_i \mid parents(X_i))$ implied by the joint distribution. Remember that the conditional probabilities can be computed from the joint distribution as follows:

$$\begin{aligned} P(x_i \mid parents(X_i)) &\equiv \frac{P(x_i, parents(X_i))}{P(parents(X_i))} \\ &= \frac{\sum_y P(x_i, parents(X_i), \mathbf{y})}{\sum x'_{i,y} P(x'_i, parents(X_i), \mathbf{y})} \end{aligned}$$

where **y** represents the values of all variables other than X_i and its parents. From this last line one can prove that $P(x_i \mid parents(X_i)) = \theta\ (x_i \mid parents(X_i))$ (Exercise 13.CPTE). Hence, we can rewrite Equation (13.1) as

$$P(x_1,\ldots,x_n) = \prod_{i=1}^{n} P(x_i|parents(X_i))\ . \qquad (13.2)$$

This means that when one estimates values for the local conditional distributions, they need to be the actual conditional probabilities for the variable given its parents. So, for example, when we specify $\theta(JohnCalls = true \mid Alarm = true) = 0.90$, it should be the case that about 90% of the time when the alarm sounds, John calls. The fact that each parameter of the network has a precise meaning in terms of only a small set of variables is crucially important for robustness and ease of specification of the models.

A method for constructing Bayesian networks

Equation (13.2) defines what a given Bayes net means. The next step is to explain how to *construct* a Bayesian network in such a way that the resulting joint distribution is a good representation of a given domain. We will now show that Equation (13.2) implies certain conditional independence relationships that can be used to guide the knowledge engineer in constructing the topology of the network. First, we rewrite the entries in the joint distribution in terms of conditional probability, using the product rule (see page 408):

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

Then we repeat the process, reducing each joint probability to a conditional probability and a joint probability on a smaller set of variables. We end up with one big product:

$$P(x_1,\ldots,x_n) = P(x_n|x_{n-1},\ldots,x_1)P(x_{n-1}|x_{n-2},\ldots,x_1)\cdots P(x_2|x_1)P(x_1)$$

$$= \prod_{i=1}^{n} P\left(x_i|x_{i-1},\ldots,x_1\right).$$

This identity is called the **chain rule**. It holds for any set of random variables. Comparing it with Equation (13.2), we see that the specification of the joint distribution is equivalent to the general assertion that, for every variable X_i in the network,

$$\mathbf{P}(X_i|X_{i-1},\ldots,X_1) \ = \ \mathbf{P}(X_i|Parents(X_i)), \qquad (13.3)$$

provided that $Parents(X_i) \subseteq \{X_{i-1},\ldots,X_1\}$. This last condition is satisfied by numbering the nodes in **topological order**—that is, in any order consistent with the directed graph structure. For example, the nodes in Figure 13.2 could be ordered *B,E,A,J,M;E,B, A,M, J*; and so on.

What Equation (13.3) says is that the Bayesian network is a correct representation of the domain only if each node is conditionally independent of its other predecessors in the node ordering, given its parents. We can satisfy this condition with this methodology:

1. *Nodes:* First determine the set of variables that are required to model the domain. Now order them, $\{X_1,\ldots,X_n\}$. Any order will work, but the resulting network will be more compact if the

variables are ordered such that causes precede effects.

2. *Links:* For $i = 1$ to n do:
   - Choose a minimal set of parents for X_i from $X_1, \ldots, X_{i-1}$, such that Equation (13.3) is satisfied.
   - For each parent insert a link from the parent to X_i.
   - CPTs: Write down the conditional probability table, $\mathbf{P}(X_i \mid Parents(X_i))$.

Intuitively, the parents of node X_i should contain all those nodes in $X_1, \ldots, X_{i-1}$ that directly influence X_i. For example, suppose we have completed the network in Figure 13.2 except for the choice of parents for *MaryCalls*. *MaryCalls* is certainly influenced by whether there is a *Burglary* or an *Earthquake*, but not *directly* influenced. Intuitively, our knowledge of the domain tells us that these events influence Mary's calling behavior only through their effect on the alarm. Also, given the state of the alarm, whether John calls has no influence on Mary's calling. Formally speaking, we believe that the following conditional independence statement holds:

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

Thus, *Alarm* will be the only parent node for *MaryCalls.*

Because each node is connected only to earlier nodes, this construction method guarantees that the network is acyclic. Another important property of Bayes nets is that they contain no redundant probability values. If there is no redundancy, then there is no chance for inconsistency: *it is impossible for the knowledge engineer or domain expert to create a Bayesian network that violates the axioms of probability.*

Compactness and node ordering

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

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

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

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

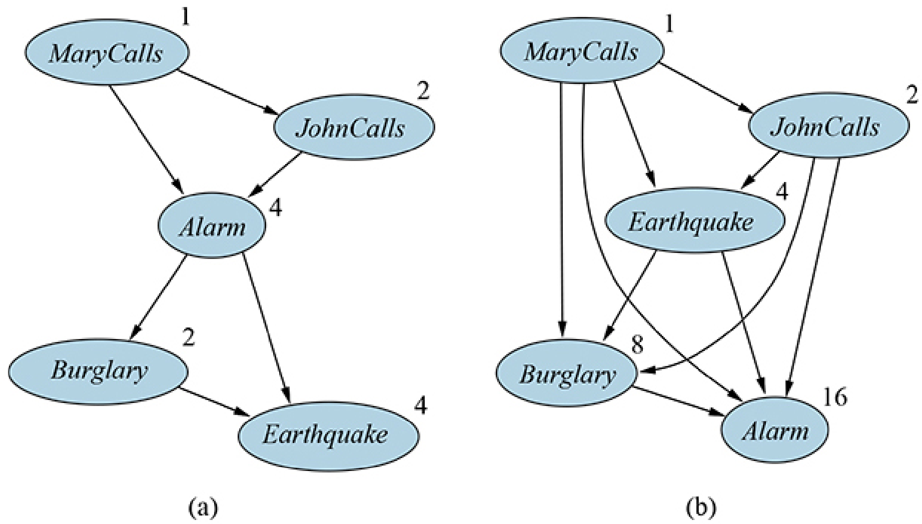


Figure 13.3 Network structure and number of parameters depends on order of introduction. (a) The structure obtained with ordering *M,J,A,B,E*. (b) The structure obtained with *M,J,E,B,A*. Each node is annotated with the number of parameters required; 13 in all for (a) and 31 for (b). In Figure 13.2,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

- Adding *MaryCalls:* No parents.
- Adding *JohnCalls:* If Mary calls, that probably means the alarm has gone off, which makes it more likely that John calls. Therefore, *JohnCalls* needs *MaryCalls* as a parent.

- Adding *Alarm:* Clearly, if both call, it is more likely that the alarm has gone off than if just one or neither calls, so we need both *MaryCalls* and *JohnCalls* as parents.
- Adding *Burglary:* If we know the alarm state, then the call from John or Mary might give us information about our phone ringing or Mary's music, but not about burglary:

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

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

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

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

13.2.1 Conditional independence relations in Bayesian networks

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

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

For example, in Figure 13.2, the variable *JohnCalls* is independent of *Burglary*, *Earthquake*, and *MaryCalls* given the value of *Alarm.* The definition is illustrated in Figure 13.4(a).

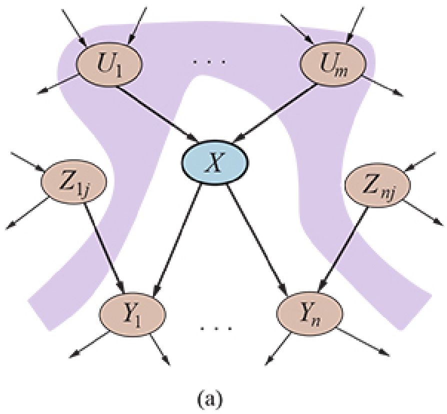


Figure 13.4 (a) A node X is conditionally independent of its non-descendants (e.g., the Z_{ij}s) given its parents (the U_is shown in the lavender area). (b) A node X is conditionally independent of all other nodes in the network given its Markov blanket (the lavender area).

It turns out that the non-descendants property combined with interpretation of the network parameters $\theta(X_i \mid Parents(X_i))$ as conditional probabilities $\mathbf{P}(X_i \mid Parents(X_i))$ suffices to reconstruct the full joint distribution given in Equation (13.2). In other words, one can view the semantics of Bayes nets in a different way: instead of defining the full joint distribution as the product of conditional distributions, the network defines a set of conditional independence properties. The full joint distribution can be derived from those properties.

Another important independence property is implied by the non-descendants property:

> a variable is conditionally independent of all other nodes in the network, given its parents, children, and children's parents—that is, given its **Markov blanket**.

(Exercise 13.MARB asks you to prove this.) For example, the variable *Burglary* is independent of *JohnCalls* and *MaryCalls*, given *Alarm* and *Earthquake*. This property is illustrated in Figure 13.4(b). The Markov blanket property makes possible inference algorithms that use completely local and distributed stochastic sampling processes, as explained in Section 13.4.2.

The most general conditional independence question one might ask in a Bayes net is whether a set of nodes **X** is conditionally independent of another set **Y**, given a third set **Z**. This can be determined efficiently by examining the Bayes net to see whether **Z d-separates X** and **Y**. The process works as follows:

1. Consider just the **ancestral subgraph** consisting of **X**, **Y**, **Z**, and their ancestors.
2. Add links between any unlinked pair of nodes that share a common child; now we have the so-called **moral graph.**
3. Replace all directed links by undirected links.

4. If **Z** blocks all paths between **X** and **Y** in the resulting graph, then **Z** d-separates **X** and **Y**. In that case, **X** is conditionally independent of **Y**, given **Z**. Otherwise, the original Bayes net does not require conditional independence.

In brief, then, d-separation means separation in the undirected, moralized, ancestral subgraph. Applying the definition to the burglary network in Figure 13.2, we can deduce that *Burglary* and *Earthquake* are independent given the empty set (i.e., they are absolutely independent); that they are *not* necessarily conditionally independent given *Alarm;* and that *JohnCalls* and *MaryCalls* are conditionally independent given *Alarm.* Notice also that the Markov blanket property follows directly from the d-separation property, since a variable's Markov blanket d-separates it from all other variables.

13.2.2 Efficient Representation of Conditional Distributions

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

The simplest example is provided by **deterministic nodes**. A deterministic node has its value specified exactly by the values of its parents, with no uncertainty. The relationship can be a logical one: for example, the relationship between the parent nodes *Canadian*, *US*, *Mexican* and the child node *NorthAmerican* is simply that the child is the disjunction of the parents. The relationship can also be numerical: for example, the *BestPrice* for a car is the minimum of the prices at each dealer in the area; and the *WaterStored* in a reservoir at year's end is the sum of the original amount, plus the inflows (rivers, runoff, precipitation) and minus the outflows (releases, evaporation, seepage).

Many Bayes net systems allow the user to specify deterministic functions using a generalpurpose programming language; this makes it possible to include complex elements such as global climate models or power-grid simulators within a probabilistic model.

Another important pattern that occurs often in practice is **context-specific independence** or CSI. A conditional distribution exhibits CSI if a variable is conditionally independent of some of its parents given *certain values* of others. For example, let's suppose that the *Damage* to your car occurring during a given period of time depends on the *Ruggedness* of your car and whether or not an *Accident* occurred in that period. Clearly, if *Accident* is false, then the *Damage*, if any, doesn't depend on the *Ruggedness* of your car. (There might be vandalism damage to the car's paintwork or windows, but we'll assume all cars are equally subject to such damage.) We say that *Damage* is context-specifically independent of *Ruggedness* given *Accident* = *false.* Bayes net systems often implement CSI using an if-then-else syntax for specifying conditional distributions; for example, one might write

$$\mathbf{P}(Damage|Ruggedness,\ Accident)\ =$$
$$\mathbf{if}\ (Accident = false)\ \mathbf{then}\ d_1\ \mathbf{else}\ d_2(Ruggedness)$$

where d_1 and d_2 represent arbitrary distributions. As with determinism, the presence of CSI in a network may facilitate efficient inference. All of the exact inference algorithms mentioned in Section 13.3 can be modified to take advantage of CSI to speed up computation.

Uncertain relationships can often be characterized by so-called **noisy** logical relationships. The standard example is the **noisy-OR** relation, which is a generalization of the logical OR. In propositional logic, we might say that *Fever* is true if and only if *Cold*, *Flu*, or *Malaria* are true. The noisy-OR model allows for uncertainty about the ability of each parent to cause the child to be true—the causal relationship between parent and child may be *inhibited*, and so a patient could have a cold, but not exhibit a fever.

The model makes two assumptions. First, it assumes that all the possible causes are listed. (If some are missing, we can always add a so-called **leak node** that covers “miscellaneous causes.”) Second, it assumes that inhibition of each parent is independent of inhibition of any other parents: for example, whatever inhibits *Malaria* from causing a fever is independent of whatever inhibits *Flu* from causing a fever. Given these assumptions, *Fever* is *false* if and only if all its *true* parents are inhibited, and the probability of this is the product of the inhibition probabilities *q j* for each parent. Let us suppose these individual inhibition probabilities are as follows:

$$\begin{aligned} q_{cold} &= P(\neg fever \mid cold, \neg flu, \neg malaria) = 0.6, \\ q_{flu} &= P(\neg fever | \neg cold, flu, \neg malaria) = 0.2, \\ q_{malaria} &= P(\neg fever | \neg cold, \neg flu, malaria) = 0.1. \end{aligned}$$

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

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

where the product is taken over the parents that are set to true for that row of the CPT. Figure 13.5 illustrates this calculation.

Cold	*Flu*	*Malaria*	$P(fever \mid \cdot)$	$P(\neg fever \mid \cdot)$
f	*f*	*f*	0.0	1.0
f	*f*	*t*	0.9	**0.1**
f	*t*	*f*	0.8	**0.2**
f	*t*	*t*	0.98	$0.02 = 0.2 \times 0.1$
t	*f*	*f*	0.4	**0.6**
t	*f*	*t*	0.94	$0.06 = 0.6 \times 0.1$
t	*t*	*f*	0.88	$0.12 = 0.6 \times 0.2$
t	*t*	*t*	0.988	$0.012 = 0.6 \times 0.2 \times 0.1$

Figure 13.5 A complete conditional probability table for **P**(*Fever* | *Cold*,*Flu*,*Malaria*), assuming a noisy-OR model with the the three *q*-values shown in bold.

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

13.2.3 Bayesian nets with continuous variables

Many real-world problems involve continuous quantities, such as height, mass, temperature, and money. By definition, continuous variables have an infinite number of possible values, so it is impossible to specify conditional probabilities explicitly for each value. One way to handle continuous variables is with **discretization**—that is, dividing up the possible values into a fixed set of intervals. For example, temperatures could be divided into three categories: ($<0^{o}$C), (0^{o}C-100^{o}C), and ($>100^{o}$C). In choosing the number of categories, there is a tradeoff between loss of accuracy and large CPTs which can lead to slow run times.

Another approach is to define a continuous variable using one of the standard families of probability density functions (see Appendix A). For example, a Gaussian (or normal) distribution $N(x; \mu, \sigma^2)$ is specified by just two parameters, the mean μ and the variance σ^2. Yet another solution—sometimes called a **nonparametric** representation—is to define the conditional distribution implicitly with a collection of instances, each containing specific values of the parent and child variables. We explore this approach further in Chapter 19.

A network with both discrete and continuous variables is called a **hybrid Bayesian network**. To specify a hybrid network, we have to specify two new kinds of distributions: the conditional distribution for a continuous variable given discrete or continuous parents; and the conditional distribution for a discrete variable given continuous parents. Consider the simple example in Figure

13.6, in which a customer buys some fruit depending on its cost, which depends in turn on the size of the harvest and whether the government's subsidy scheme is operating. The variable *Cost* is continuous and has continuous and discrete parents; the variable *Buys* is discrete and has a continuous parent.

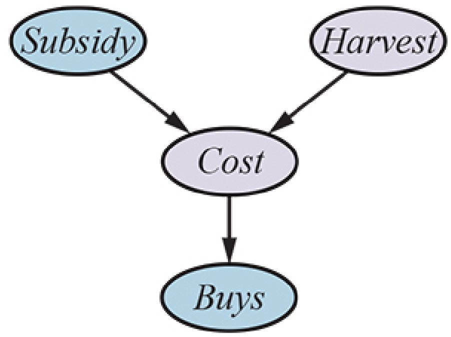


Figure 13.6 A simple network with discrete variables *(Subsidy* and *Buys)* and continuous variables *(Harvest* and *Cost).*

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

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

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

For this example, then, the conditional distribution for *Cost* is specified by naming the linear- Gaussian distribution and providing the parameters a_t, b_t, σ_t, a_f, b_f, and σ_f. Figures 13.7(a) and (b) show these two relationships. Notice that in each case the slope of *c* versus *h* is negative, because cost decreases as the harvest size increases. (Of course, the assumption of linearity implies that the cost becomes negative at some point; the linear model is reasonable only if the harvest size is limited to a narrow range.) Figure 13.7(c) shows the distribution $P(c \mid h)$, averaging over the two possible values of *Subsidy* and assuming that each has prior probability 0.5. This shows that even with very simple models, quite interesting distributions can be represented.

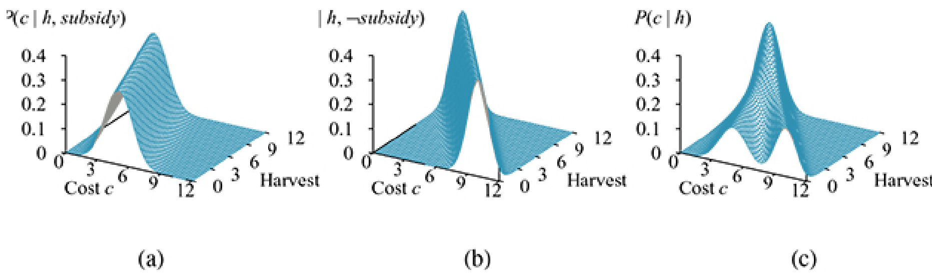


Figure 13.7 The graphs in (a) and (b) show the probability distribution over *Cost* as a function of *Harvest* size, with *Subsidy* true and false, respectively. Graph (c) shows the distribution *P(Cost|Harvest),* obtained by summing over the two subsidy cases.

The linear–Gaussian conditional distribution has some special properties. A network containing only continuous variables with linear-Gaussian distributions has a joint distribution that is a multivariate Gaussian distribution (see Appendix A) over all the variables (Exercise 13.LGEX). Furthermore, the posterior distribution given any evidence also has this property.[2] When discrete variables are added as parents (not as children) of continuous variables, the network defines a **conditional Gaussian**, or CG, distribution: given any assignment to the discrete variables, the distribution over the continuous variables is a multivariate Gaussian.

Now we turn to the distributions for discrete variables with continuous parents. Consider, for example, the *Buys* node in Figure 13.6. It seems reasonable to assume that the customer will buy if the cost is low and will not buy if it is high and that the probability of buying varies smoothly in some intermediate region. In other words, the conditional distribution is like a “soft” threshold function. One way to make soft thresholds is to use the *integral* of the standard normal distribution:

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

$\Phi(x)$ is an increasing function of x, whereas the probability of buying decreases with cost, so here we flip the function around:

$$P(buys|Cost = c) = 1 - \Phi\left((c-\mu)/\sigma\right),$$

which means that the cost threshold occurs around μ, the width of the threshold region is proportional to σ, and the probability of buying decreases as cost increases. This **probit** model (pronounced “pro-bit” and short for “probability unit”) is illustrated in Figure 13.8(a). The form can be justified by proposing that the underlying decision process has a hard threshold, but that the precise location of the threshold is subject to random Gaussian noise.

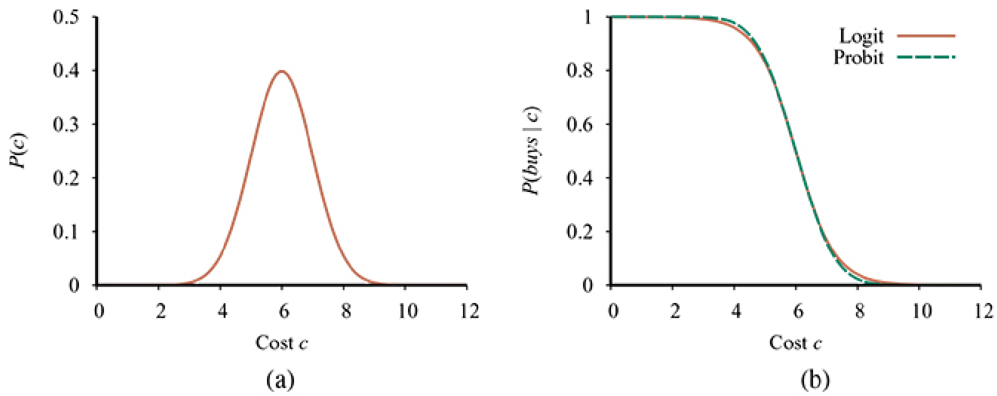


Figure 13.8 (a) A normal (Gaussian) distribution for the cost threshold, centered on $\mu = 6.0$ with standard deviation $\sigma = 1.0$. (b) Expit and probit models for the probability of *buys* given *cost*, for the parameters $\mu = 6.0$ and $\sigma = 1.0$.

An alternative to the probit model is the **expit** or **inverse logit** model. It uses the **logistic function** $1/(1 + e^{-x})$ to produce a soft threshold—it maps any x to a value between 0 and 1. Again, for our example, we flip it around to make a decreasing function; we also scale the exponent by $4/\sqrt{2\pi}$ to match the probit's slope at the mean:

$$P(buys \Big| Cost = c) = 1 - \frac{1}{1+exp(-\frac{4}{\sqrt{2\pi}}, \frac{c-\mu}{\sigma})}.$$

This is illustrated in Figure 13.8(b). The two distributions look similar, but the logit actually has much longer "tails." The probit is often a better fit to real situations, but the logistic function is sometimes easier to deal with mathematically. It is used widely in machine learning. Both models can be generalized to handle multiple continuous parents by taking a linear combination of the parent values. This also works for discrete parents if their values are integers; for example, with k Boolean parents, each viewed as having values 0 or 1, the input to the expit or probit distribution would be a weighted linear combination with k parameters, yielding a model quite similar to the noisy-OR model discussed earlier.

13.2.4 Case study: Car insurance

A car insurance company receives an application from an individual to insure a specific vehicle and must decide on the appropriate annual premium to charge, based on the anticipated claims it will pay out for this applicant. The task is to build a Bayes net that captures the causal structure of the domain and gives an accurate, well-calibrated distribution over the output variables given the evidence available from the application form.[3] The Bayes net will include **hidden variables** that are neither input nor output variables, but are essential for structuring the network so that it is reasonably sparse with a manageable number of parameters. The hidden variables are shaded brown in Figure 13.9.

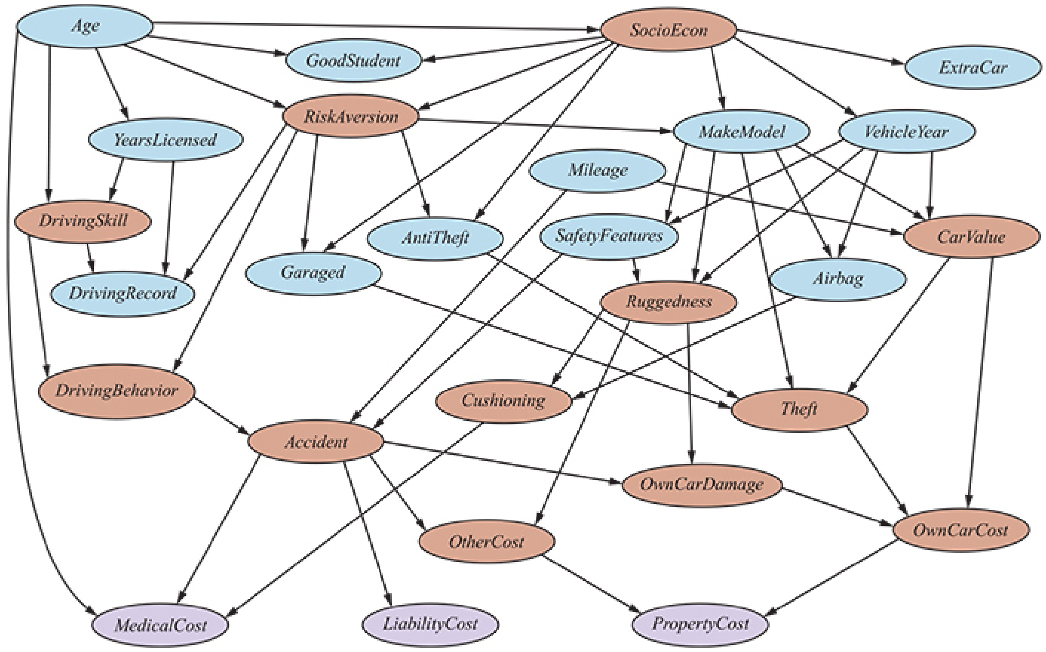


Figure 13.9 A Bayesian network for evaluating car insurance applications.

The claims to be paid out—shaded lavender in Figure 13.9—are of three kinds: the *MedicalCost* for any injuries sustained by the applicant; the *LiabilityCost* for lawsuits filed by other parties against the applicant and the company; and the *PropertyCost* for vehicle damage to either party and vehicle loss by theft. The application form asks for the following input information (the light blue nodes in Figure 13.9):

- About the applicant: *Age*; *YearsLicensed*—how long since a driving license was first obtained; *DrivingRecord*—some summary, perhaps based on "points," of recent accidents and traffic violations; and (for students) a *GoodStudent* indicator for a grade-point average of 3.0 (B) on a 4-point scale.
- About the vehicle: the *MakeModel* and *VehicleYear;* whether it has an *Airbag*; and some summary of *SafetyFeatures* such as anti-lock braking and collision warning.
- About the driving situation: the annual *Mileage* driven and how securely the vehicle is *Garaged*, if at all.

Now we need to think about how to arrange these into a causal structure. The key hidden variables are whether or not a *Theft* or *Accident* will occur in the next time period. Obviously, one cannot ask the applicant to predict these; they have to be inferred from the available information and the insurer's previous experience.

What are the causal factors leading to *Theft*? The *MakeModel* is certainly important— some models are stolen much more often than others because there is an efficient resale market for vehicles and parts; the *CarValue* also matters, because an old, beat-up, or high- mileage vehicle has lower resale value. Moreover, a vehicle that is *Garaged* and has an *AntiTheft* device is harder to steal. The hidden variable *CarValue* depends in turn on the *MakeModel, VehicleYear,* and *Mileage. CarValue* also dictates the loss amount when a *Theft* occurs, so that is one of the contributors to *OwnCarCost* (the other being accidents, which we will get to shortly).

It is common in models of this type to introduce another hidden variable, *SocioEcon,* the socioeconomic category of the applicant. This is thought to influence a wide range of behaviors and characteristics. In our model, there is no *direct* evidence in the form of observed income and occupation variables;[4] but *SocioEcon* influences *MakeModel* and *VehicleYear;* it also affects *ExtraCar* and *GoodStudent*, and depends somewhat on *Age*.

For any insurance company, perhaps the most important hidden variable is *RiskAversion:* people who are risk-averse are good insurance risks! *Age* and *SocioEcon* affect *RiskAversion,* and its “symptoms” include the applicant’s choice of whether the vehicle is *Garaged* and has *AntiTheft* devices and *SafetyFeatures*.

In predicting future accidents, the key is the applicant’s future *DrivingBehavior,* which is influenced by both *RiskAversion* and *DrivingSkill*; the latter in turn depends on *Age* and *YearsLicensed.* The applicant’s past driving behavior is reflected in the *DrivingRecord,* which also depends on *RiskAversion* and *DrivingSkill* as well as on *YearsLicensed* (because someone who started driving only recently may not have had time to accumulate a litany of accidents and violations). In this way, *DrivingRecord* provides evidence about *RiskAversion* and *DrivingSkill*, which in turn help to predict future *DrivingBehavior*.

We can think of *DrivingBehavior* as a per-mile tendency to drive in an accident-prone way; whether an *Accident* actually occurs in a fixed time period depends also on the annual *Mileage* and on the *SafetyFeatures* of the vehicle. If an *Accident* occurs, there are three kinds of costs: the *MedicalCost* for the applicant depends on *Age* and *Cushioning*, which depends in turn on the *Ruggedness* of the car and whether it has an *Airbag*; the *LiabilityCost*(medical, pain and suffering, loss of income, etc.) for the other driver; and the *PropertyCost* for the applicant and the other driver, both of which depend (in different ways) on the car’s *Ruggedness* and on the applicant’s *CarValue.*

We have illustrated the kind of reasoning that goes into developing the topology and hidden variables in a Bayes net. We also need to specify the ranges and the conditional distributions for each variable. For the ranges, the primary decision is often whether to make the variable discrete or continuous. For example, the *Ruggedness* of the vehicle could be a continuous variable between 0 and 1, or a discrete variable with range {*TinCan,Normal, Tank*}.

Continuous variables provide more precision, but they make exact inference impossible except in a few special cases. A discrete variable with many possible values can make it tedious to fill in the correspondingly large conditional probability tables and makes exact inference more expensive unless the variable's value is always observed. For example, *MakeModel* in a real system would have thousands of possible values, and this causes its child *CarValue* to have an enormous CPT that would have to be filled in from industry databases; but, because the *MakeModel* is always observed, this does not contribute to inference complexity: in fact, the observed values for the three parents pick out exactly one relevant row of the CPT for *CarValue*.

The conditional distributions in the model are given in the code repository for the book; we provide a version with only discrete variables, for which exact inference can be performed. In practice, many of the variables would be continuous and the conditional distributions would be learned from historical data on applicants and their insurance claims. We will see how to learn Bayes net models from data in Chapter 21.

The final question is, of course, how to do inference in the network to make predictions. We turn now to this question. For each inference method that we describe, we will evaluate the method on the insurance net to measure the time and space requirements of the method.

13.3 Exact Inference in Bayesian Networks

The basic task for any probabilistic inference system is to compute the posterior probability distribution for a set of **query variables**, given some observed **event**—usually, some assignment of values to a set of **evidence variables**.[5] To simplify the presentation, we will consider only one query variable at a time; the algorithms can easily be extended to queries with multiple variables. (For example, we can solve the query $\mathbf{P}(U, V \mid \mathbf{e})$ by multiplying $\mathbf{P}(V \mid e)$ and $\mathbf{P}(U \mid V, \mathbf{e})$.) We will use the notation from Chapter 12: X denotes the query variable; $\mathbf{E}$ denotes the set of evidence variables $E_1,\ldots, E_m$, and $\mathbf{e}$ is a particular observed event; $\mathbf{Y}$ denotes the hidden (nonevidence, nonquery) variables $Y_1,\ldots,Y_\ell$. Thus, the complete set of variables is $\{X\}\cup\mathbf{E}\cup\mathbf{Y}$. A typical query asks for the posterior probability distribution $\mathbf{P}(X \mid \mathbf{e})$.

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

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

In this section we discuss exact algorithms for computing posterior probabilities as well as the complexity of this task. It turns out that the general case is intractable, so Section 13.4 covers methods for approximate inference.

13.3.1 Inference by enumeration

Chapter 12 explained that any conditional probability can be computed by summing terms from the full joint distribution. More specifically, a query $\mathbf{P}(X \mid \mathbf{e})$ can be answered using Equation (12.9), which we repeat here for convenience:

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

Now, a Bayes net gives a complete representation of the full joint distribution. More specifically, Equation (13.2) on page 433 shows that the terms $P(x, \mathbf{e}, \mathbf{y})$ in the joint distribution can be written as products of conditional probabilities from the network. Therefore, *a query can be answered using a Bayes net by computing sums of products of conditional probabilities from the network.*

Consider the query $\mathbf{P}(Burglary|JohnCalls = true, MaryCalls = true)$. The hidden variables for this query are *Earthquake* and *Alarm*. From Equation (12.9), using initial letters for the variables to shorten the expressions, we have

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

The semantics of Bayes nets (Equation (13.2)) then gives us an expression in terms of CPT entries. For simplicity, we do this just for *Burglary* = *true*:

$$P(b|j,m) = \alpha\sum_{e}\sum_{a}P(b)P(e)P(a|b,e)P(j|a)P(m|a). \qquad (13.4)$$

To compute this expression, we have to add four terms, each computed by multiplying five numbers. In the worst case, where we have to sum out almost all the variables, there will be $O(2^n)$ terms in the sum, each a product of $O(n)$ probability values. A naive implementation would therefore have complexity $O(n2^n)$.

This can be reduced to $O(2^n)$ by taking advantage of the nested structure of the computation. In symbolic terms, this means moving the summations inwards as far as possible in expressions such as Equation (13.4). We can do this because not all the factors in the product of probabilities depend on all the variables. Thus we have

$$P(b|j,m) = \alpha P(b) \sum_e P(e) \sum_a P(a|b,e) P(j|a) P(m|a). \tag{13.5}$$

This expression can be evaluated by looping through the variables in order, multiplying CPT entries as we go. For each summation, we also need to loop over the variable's possible values. The structure of this computation is shown as a tree in Figure 13.10. Using the numbers from Figure 13.2, we obtain $P(b \mid j, m) = \alpha \times 0.00059224$. The corresponding computation for $\neg b$ yields $\alpha \times 0.0014919$; hence,

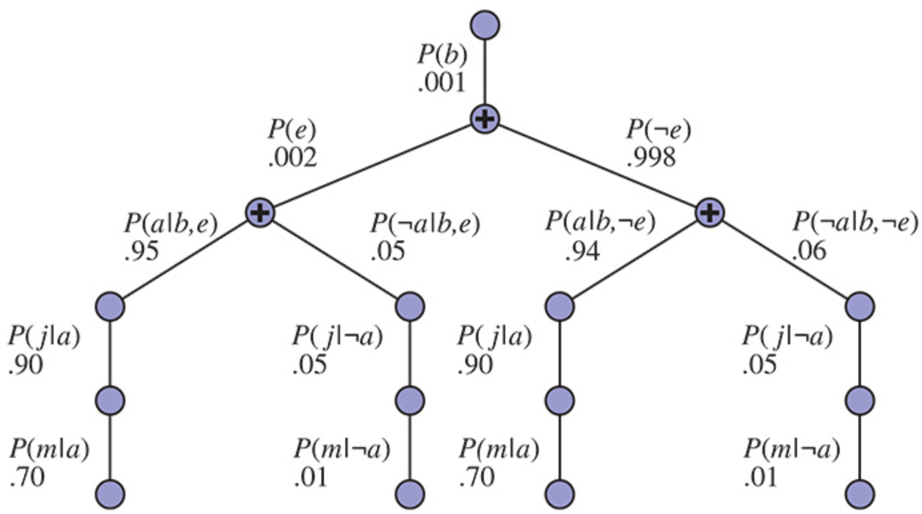


Figure 13.10 The structure of the expression shown in Equation (13.5). The evaluation proceeds top down, multiplying values along each path and summing at the "+" nodes. Notice the repetition of the paths for j and m.

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

That is, the chance of a burglary, given calls from both neighbors, is about 28%.

The ENUMERATION-ASK algorithm in Figure 13.11 evaluates these expression trees using depth-first, left-to-right recursion. The algorithm is very similar in structure to the backtracking algorithm for solving CSPs (Figure 5.5) and the DPLL algorithm for satisfiability (Figure 7.17). Its space complexity is only linear in the number of variables: the algorithm sums over the full joint distribution without ever constructing it explicitly. Unfortunately, its time complexity for a network with n Boolean variables (not counting the evidence variables) is always $O(2^n)$—better than the $O(n\,2^n)$ for the simple approach described earlier, but still rather grim. For the insurance network in Figure 13.9, which is relatively small, exact inference using enumeration requires around 227 million arithmetic operations for a typical query on the cost variables.

```
function ENUMERATION-ASK(X, e, bn) returns a distribution over X
   inputs: X, the query variable
           e, observed values for variables E
           bn, a Bayes net with variables vars

   Q(X) ← a distribution over X, initially empty
   for each value x_i of X do
      Q(x_i) ← ENUMERATE-ALL(vars, e_{x_i})
         where e_{x_i} is e extended with X = x_i
   return NORMALIZE(Q(X))

function ENUMERATE-ALL(vars, e) returns a real number
   if EMPTY?(vars) then return 1.0
   V ← FIRST(vars)
   if V is an evidence variable with value v in e
      then return P(v | parents(V)) × ENUMERATE-ALL(REST(vars), e)
      else return ∑_v P(v | parents(V)) × ENUMERATE-ALL(REST(vars), e_v)
         where e_v is e extended with V = v
```

Figure 13.11 The enumeration algorithm for exact inference in Bayes nets.

If you look carefully at the tree in Figure 13.10, however, you will see that it contains *repeated subexpressions.* The products $P(j \mid a)P(m \mid a)$ and $P(j \mid \neg a)P(m \mid \neg a)$ are computed twice, once for each value of E. The key to efficient inference in Bayes nets is avoiding such wasted computations. The next section describes a general method for doing this.

13.3.2 The variable elimination algorithm

The enumeration algorithm can be improved substantially by eliminating repeated calculations of the kind illustrated in Figure 13.10. The idea is simple: do the calculation once and save the results for later use. This is a form of dynamic programming. There are several versions of this approach; we present the **variable elimination** algorithm, which is the simplest. Variable elimination works by evaluating expressions such as Equation (13.5) in *right-to-left* order (that is, *bottom up* in Figure 13.10). Intermediate results are stored, and summations over each variable are done only for those portions of the expression that depend on the variable.

Let us illustrate this process for the burglary network. We evaluate the expression

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

Notice that we have annotated each part of the expression with the name of the corresponding **factor**; each factor is a matrix indexed by the values of its argument variables. For example, the factors $\mathbf{f}_4(A)$ and $\mathbf{f}_5(A)$ corresponding to $P(j|a)$ and $P(m|a)$ depend just on A because J and M are fixed by the query. They are therefore two-element vectors:

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

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

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

Here the " × " operator is not ordinary matrix multiplication but instead the **pointwise product** operation, to be described shortly.

The evaluation process sums out variables (right to left) from pointwise products of factors to produce new factors, eventually yielding a factor that constitutes the solution—that is, the posterior distribution over the query variable. The steps are as follows:

- First, we sum out A from the product of $\mathbf{f}_3$, $\mathbf{f}_4$, and $\mathbf{f}_5$. This gives us a new 2×2 factor $\mathbf{f}_6(B,E)$ whose indices range over just B and E:

  $$\begin{aligned}\mathbf{f}_6(B, E) &= \sum_a \mathbf{f}_3(A, B, E) \times \mathbf{f}_4(A) \times \mathbf{f}_5(A) \\ &= (\mathbf{f}_3(a, B, E) \times \mathbf{f}_4(a) \times \mathbf{f}_5(a)) + (\mathbf{f}_3(\neg a, B, E) \times \mathbf{f}_4(\neg a) \times \mathbf{f}_5(\neg a)).\end{aligned}$$

  Now we are left with the expression

  $$\mathbf{P}(B \middle| j, m) = a\, \mathbf{f}_1(B) \times \sum_e \mathbf{f}_2(E) \times \mathbf{f}_6(B, E).$$

- Next, we sum out E from the product of $\mathbf{f}_2$ and $\mathbf{f}_6$:

  $$\begin{aligned}\mathbf{f}_7(B) &= \sum_e \mathbf{f}_2(E) \times \mathbf{f}_6(B, E) \\ &= \mathbf{f}_2(e) \times \mathbf{f}_6(B, e) + \mathbf{f}_2(\neg e) \times \mathbf{f}_6(B, \neg e).\end{aligned}$$

  This leaves the expression

  $$\mathbf{P}(B|j, m) \ = \alpha\ \mathbf{f}_1(B) \times \mathbf{f}_7(B)$$

  which can be evaluated by taking the pointwise product and normalizing the result.

Examining this sequence, we see that two basic computational operations are required: point- wise product of a pair of factors, and summing out a variable from a product of factors. The next section describes each of these operations.

Operations on factors

The pointwise product of two factors **f** and **g** yields a new factor **h** whose variables are the *union* of the variables in **f** and **g** and whose elements are given by the product of the corresponding elements in the two factors. Suppose the two factors have variables $Y_1,\ldots,Y_k$ in common. Then we have

$$\mathbf{f}(X_1\ \ldots X_j, Y_1\ \ldots Y_k) \times g(Y_1\ \ldots Y_k, Z_1,\ \ldots Z_l)\ =\ \mathbf{h}(X_1\ \ldots X_j, Y_1\ \ldots Y_k, Z_1\ \ldots Z_\ell)$$

If all the variables are binary, then **f** and **g** have 2^{j+k} and $2^{k+\ell}$ entries, respectively, and the pointwise product has $2^{j+k+\ell}$ entries. For example, given two factors $\mathbf{f}(X, Y)$ and $\mathbf{g}(Y,Z)$, the pointwise product $\mathbf{f} \times \mathbf{g} = \mathbf{h}(X, Y, Z)$ has $2^{1+1+1} = 8$ entries, as illustrated in Figure 13.12. Notice that the factor resulting from a pointwise product can contain more variables than any of the factors being multiplied and that the size of a factor is exponential in the number of variables. This is where both space and time complexity arise in the variable elimination algorithm.

X	Y	$\mathbf{f}(X,Y)$	Y	Z	$\mathbf{g}(Y,Z)$	X	Y	Z	$\mathbf{h}(X,Y,Z)$
t	t	.3	t	t	.2	t	t	t	$.3 \times .2 = .06$
t	f	.7	t	f	.8	t	t	f	$.3 \times .8 = .24$
f	t	.9	f	t	.6	t	f	t	$.7 \times .6 = .42$
f	f	.1	f	f	.4	t	f	f	$.7 \times .4 = .28$
						f	t	t	$.9 \times .2 = .18$
						f	t	f	$.9 \times .8 = .72$
						f	f	t	$.1 \times .6 = .06$
						f	f	f	$.1 \times .4 = .04$

Figure 13.12 Illustrating pointwise multiplication: $\mathbf{f}(X, Y) \times \mathbf{g}(Y,Z) = \mathbf{h}(X,Y,Z)$.

Summing out a variable from a product of factors is done by adding up the submatrices formed by fixing the variable to each of its values in turn. For example, to sum out X from $\mathbf{h}(X, Y, Z)$, we write

$$\mathbf{h}_2(Y,Z) = \sum_x \mathbf{h}(X,Y,Z) = \mathbf{h}(x,Y,Z) + \mathbf{h}(\neg x,Y,Z)$$

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

The only trick is to notice that any factor that does *not* depend on the variable to be summed out can be moved outside the summation. For example, to sum out X from the product of **f** and **g**, we can move g outside the summation:

$$\sum_x \mathbf{f}(X,Y) \times \mathbf{g}(Y,Z) = \mathbf{g}(Y,Z) \times \sum_x \mathbf{f}(X,Y).$$

This is potentially much more efficient than computing the larger pointwise product **h** first and then summing X out from that.

Notice that matrices are *not* multiplied until we need to sum out a variable from the accumulated product. At that point, we multiply just those matrices that include the variable to be summed out. Given functions for pointwise product and summing out, the variable elimination algorithm itself can be written quite simply, as shown in Figure 13.13.

```
function ELIMINATION-ASK(X, e, bn) returns a distribution over X
  inputs: X, the query variable
          e, observed values for variables E
          bn, a Bayesian network with variables vars

  factors ← []
  for each V in ORDER(vars) do
     factors ← [MAKE-FACTOR(V, e)] + factors
     if V is a hidden variable then factors ← SUM-OUT(V, factors)
  return NORMALIZE(POINTWISE-PRODUCT(factors))
```

Figure 13.13 The variable elimination algorithm for exact inference in Bayes nets.

Variable ordering and variable relevance

The algorithm in Figure 13.13 includes an unspecified ORDER function to choose an ordering for the variables. Every choice of ordering yields a valid algorithm, but different orderings cause different intermediate factors to be generated during the calculation. For example, in the calculation shown previously, we eliminated *A* before *E*; if we do it the other way, the calculation becomes

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

during which a new factor $\mathbf{f}_6(A, B)$ will be generated.

In general, the time and space requirements of variable elimination are dominated by the size of the largest factor constructed during the operation of the algorithm. This in turn is determined by the order of elimination of variables and by the structure of the network. It turns out to be intractable to determine the optimal ordering, but several good heuristics are available. One fairly effective method is a greedy one: eliminate whichever variable minimizes the size of the next factor to be constructed.

Let us consider one more query: $\mathbf{P}(JohnCalls \mid Burglary = true)$. As usual (see Equation (13.5)), the first step is to write out the nested summation:

$$\mathbf{P}(J \middle| b) = \alpha\, P(b) \sum_e P(e) \sum_a P(a|b, e) \mathbf{P}(J \middle| a) \sum_m P(m|a).$$

Evaluating this expression from right to left, we notice something interesting: $\sum_m P(m \mid a)$ is equal to 1 by definition! Hence, there was no need to include it in the first place; the variable *M* is *irrelevant* to this query. Another way of saying this is that the result of the query $P(JohnCalls \mid Burglary = true)$ is unchanged if we remove *MaryCalls* from the network altogether. In general, we can remove any leaf node that is not a query variable or an evidence variable. After its removal, there may be some more leaf nodes, and these too may be irrelevant. Continuing this process, we eventually find that *every variable that is not an ancestor of a query variable or evidence variable is irrelevant to the query.* A variable elimination algorithm can therefore remove all these variables before evaluating the query.

When applied to the insurance network shown in Figure 13.9, variable elimination shows considerable improvement over the naive enumeration algorithm. Using reverse topological order for the variables, exact inference using elimination is about 1,000 times faster than the enumeration algorithm.

13.3.3 The complexity of exact inference

The complexity of exact inference in Bayes nets depends strongly on the structure of the network. The burglary network of Figure 13.2 belongs to the family of networks in which there is at most one undirected path (i.e., ignoring the direction of the arrows) between any two nodes in the network. These are called **singly connected** networks or **polytrees**, and they have a particularly nice property: *The time and space complexity of exact inference in poly trees is linear in the size of the network.* Here, the size is defined as the number of CPT entries; if the number of parents of each node is bounded by a constant, then the complexity will also be linear in the number of nodes. These results hold for any ordering consistent with the topological ordering of the network (Exercise 13.VEEX).

For **multiply connected** networks, such as the insurance network in Figure 13.9, variable elimination can have exponential time and space complexity in the worst case, even when the number of parents per node is bounded. This is not surprising when one considers that *because it includes inference in propositional logic as a special case, inference in Bayes nets is NP-hard.* To prove this, we need to work out how to encode a propositional satisfiability problem as a Bayes net, such that running inference on this net tells us whether or not the original propositional sentences are satisfiable. (In the language of complexity theory, we **reduce** satisfiability problems to Bayes net inference problems.) This turns out to be quite straightforward. Figure 13.14 shows how to encode a particular 3-SAT problem. The propositional variables become the root variables of the network, each with prior probability 0.5. The next layer of nodes corresponds to the clauses, with each clause variable C_j connected to the appropriate variables as parents. The conditional distribution for a clause variable is a deterministic disjunction, with negation as needed, so that each clause variable is true if and only if the assignment to its parents satisfies that clause. Finally, S is the conjunction of the clause variables.

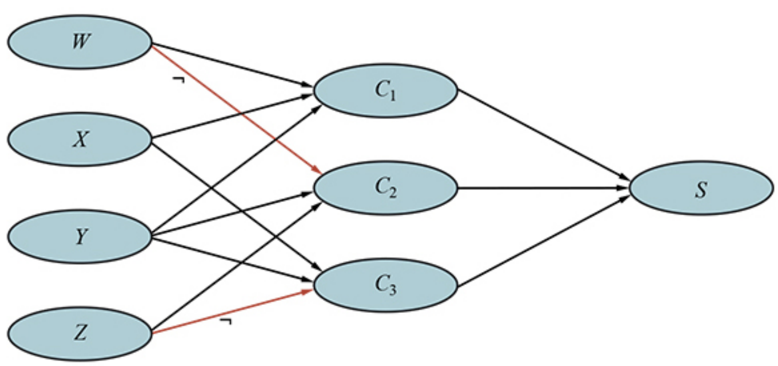


Figure 13.14 Bayes net encoding of the 3-CNF sentence

To determine it the original sentence is satisfiable, we simply evaluate $P(S = true)$. If the sentence is *satisfiable*, then there is some possible assignment to the logical variables that makes S true; in the Bayes net, this means that there is a possible world with nonzero probability in which the root variables have that assignment, the clause variables have value *true,* and S has value *true*. Therefore, $P(S = true) > 0$ for a satisfiable sentence. Conversely, $P(S = true) = 0$ for an unsatisfiable sentence: all worlds with $S = true$ have probability 0. Hence, we can use Bayes net inference to solve 3-SAT problems; from this, we conclude that Bayes net inference is NP-hard.

We can, in fact, do more than this. Notice that the probability of each satisfying assignment is 2^{-n} for a problem with n variables. Hence, the *number* of satisfying assignments is $P(S = true)/(2^{-n})$. Because computing the *number* of satisfying assignments for a 3-SAT problem is #P-complete ("number-P complete"), this means that Bayes net inference is #P- hard—that is, strictly harder than NP-complete problems.

There is a close connection between the complexity of Bayes net inference and the complexity of constraint satisfaction problems (CSPs). As we discussed in Chapter 5, the difficulty of solving a discrete CSP is related to how "treelike" its constraint graph is. Measures such as **tree width**, which bound the complexity of solving a CSP, can also be applied directly to Bayes nets. Moreover, the variable elimination algorithm can be generalized to solve CSPs as well as Bayes nets.

As well as reducing satisfiability problems to Bayes net inference, we can reduce Bayes net inference to satisfiability, which allows us to take advantage of the powerful machinery developed for SAT-solving (see Chapter 7). In this case, the reduction is to a particular form of SAT solving called **weighted model counting** (WMC). Regular model counting counts the number of satisfying assignments for a SAT expression; WMC sums the total weight of those satisfying assignments—where, in this application, the weight is essentially the product of the conditional probabilities for each variable assignment given its parents. (See Exercise 13.WMCX for details.) Partly because SAT-solving technology has been so well optimized for large-scale applications, Bayes net inference via WMC is competitive with and sometimes superior to other exact algorithms on networks with large tree width.

13.3.4 Clustering algorithms

The variable elimination algorithm is simple and efficient for answering individual queries. If we want to compute posterior probabilities for all the variables in a network, however, it can be less efficient. For example, in a polytree network, one would need to issue $O(n)$ queries costing $O(n)$ each, for a total of $O(n^2)$ time. Using **clustering** algorithms (also known as **join tree** algorithms), the time can be reduced to $O(n)$. For this reason, these algorithms are widely used in commercial Bayes net tools.

The basic idea of clustering is to join individual nodes of the network to form cluster nodes in such a way that the resulting network is a polytree. For example, the multiply connected network shown in Figure 13.15(a) can be converted into a polytree by combining the *Sprinkler* and *Rain* node into a cluster node called *Sprinkler+Rain*, as shown in Figure 13.15(b). The two Boolean nodes are replaced by a **meganode** that takes on four possible values: *tt*, *tf*, *ft*, and *ff*. The meganode has only one parent, the Boolean variable *Cloudy*, so there are two conditioning cases. Although this example doesn't show it, the process of clustering often produces meganodes that share some variables.

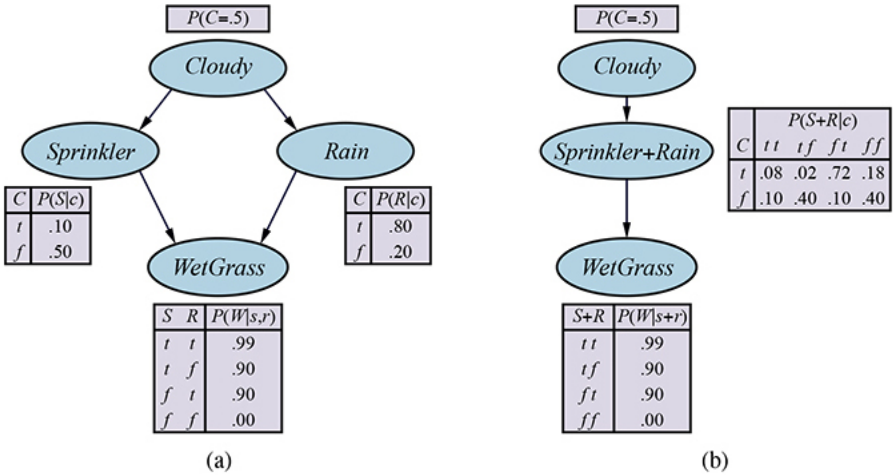


Figure 13.15 (a) A multiply connected network describing Mary's daily lawn routine: each morning, she checks the weather; if it's cloudy, she usually doesn't turn on the sprinkler; if the sprinkler is on, or if it rains during the day, the grass will be wet. Thus, *Cloudy* affects *WetGrass* via two different causal pathways. (b) A clustered equivalent of the multiply connected network.

Once the network is in polytree form, a special-purpose inference algorithm is required, because ordinary inference methods cannot handle meganodes that share variables with each other. Essentially, the algorithm is a form of constraint propagation (see Chapter 5) where the constraints ensure that neighboring meganodes agree on the posterior probability of any variables that they have in common. With careful bookkeeping, this algorithm is able to compute posterior probabilities for all the nonevidence nodes in the network in time *linear* in the size of the clustered network. However, the NP-hardness of the problem has not disappeared: if a network requires exponential time and space with variable elimination, then the CPTs in the clustered network will necessarily be exponentially large.

13.4 Approximate Inference for Bayesian Networks

Given the intractability of exact inference in large networks, we will now consider approximate inference methods. This section describes randomized sampling algorithms, also called **Monte Carlo** algorithms, that provide approximate answers whose accuracy depends on the number of samples generated. They work by generating random events based on the probabilities in the Bayes net and counting up the different answers found in those random events. With enough samples, we can get arbitrarily close to recovering the true probability distribution—provided the Bayes net has no deterministic conditional distributions.

Monte Carlo algorithms, of which simulated annealing (page 133) is an example, are used in many branches of science to estimate quantities that are difficult to calculate exactly. In this section, we are interested in sampling applied to the computation of posterior probabilities in Bayes nets. We describe two families of algorithms: direct sampling and Markov chain sampling. Several other approaches for approximate inference are mentioned in the notes at the end of the chapter.

13.4.1 Direct sampling methods

The primitive element in any sampling algorithm is the generation of samples from a known probability distribution. For example, an unbiased coin can be thought of as a random variable *Coin* with values {*heads*, *tails*) and a prior distribution $\mathbf{P}(Coin) = \langle 0.5, 0.5 \rangle$. Sampling from this distribution is exactly like flipping the coin: with probability 0.5 it will return *heads*, and with probability 0.5 it will return *tails*. Given a source of random numbers r uniformly distributed in the range [0,1], it is a simple matter to sample any distribution on a single variable, whether discrete or continuous. This is done by constructing the cumulative distribution for the variable and returning the first value whose cumulative probability exceeds r(see Exercise 13.PRSA).

We begin with a random sampling process for a Bayes net that has no evidence associated with it. The idea is to sample each variable in turn, in topological order. The probability distribution from which the value is sampled is conditioned on the values already assigned to the variable's parents. (Because we sample in topological order, the parents are guaranteed to have values already.) This algorithm is shown in Figure 13.16. Applying it to the network in Figure 13.15(a) with the ordering *Cloudy, Sprinkler, Rain, WetGrass*, we might produce a random event as follows:

---

function PRIOR-SAMPLE(bn) **returns** an event sampled from the prior specified by bn
  inputs: bn, a Bayesian network specifying joint distribution $\mathbf{P}(X_1, \ldots, X_n)$

  $\mathbf{x} \leftarrow$ an event with n elements
  for each variable X_i **in** $X_1, \ldots, X_n$ **do**
    $\mathbf{x}[i] \leftarrow$ a random sample from $\mathbf{P}(X_i \mid parents(X_i))$
  return x

Figure 13.16 A sampling algorithm that generates events from a Bayesian network. Each variable is sampled according to the conditional distribution given the values already sampled for the variable's parents.

1. Sample from $\mathbf{P}(Cloudy) = \langle 0.5, 0.5 \rangle$, value is *true.*
2. Sample from $\mathbf{P}(Sprinkler \mid Cloudy = true) = \langle 0.1, 0.9 \rangle$, value is *false.*
3. Sample from $\mathbf{P}(Rain \mid Cloudy = true) = \langle 0.8, 0.2 \rangle$, value is *true.*
4. Sample from $\mathbf{P}(WetGrass \mid Sprinkler = false, Rain = true) = \langle 0.9, 0.1 \rangle$, value is *true.*

In this case, PRIOR-SAMPLE returns the event *[true,false, true, true].*

It is easy to see that PRIOR-SAMPLE generates samples from the prior joint distribution specified by the network. First, let $S_{PS}(x_1,\ldots,x_n)$ be the probability that a specific event is generated by the PRIOR-SAMPLE algorithm. Just looking at the sampling process, we have

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

because each sampling step depends only on the parent values. This expression should look familiar, because it is also the probability of the event according to the Bayesian net's representation of the joint distribution, as stated in Equation (13.2). That is, we have

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

This simple fact makes it easy to answer questions by using samples.

In any sampling algorithm, the answers are computed by counting the actual samples generated. Suppose there are N total samples produced by the PRIOR-SAMPLE algorithm, and let $N_{PS}(x_1,\ldots, x_n)$ be the number of times the specific event $x_1,\ldots, x_n$ occurs in the set of samples. We expect this number, as a fraction of the total, to converge in the limit to its expected value according to the sampling probability:

$$\lim_{N\to\infty} \frac{N_{PS}(x_1,\ldots,x_n)}{N} = S_{PS}(x_1,\ldots,x_n) = P(x_1,\ldots,x_n) . \tag{13.6}$$

For example, consider the event produced earlier: [*true, false, true, true*]. The sampling probability for this event is

$$S_{PS}(true,\ false,\ true,\ true) = 0.5 \times 0.9 \times 0.8 \times 0.9 = 0.324.$$

Hence, in the limit of large N, we expect 32.4% of the samples to be of this event.

Whenever we use an approximate equality ("≈") in what follows, we mean it in exactly this sense—that the estimated probability becomes exact in the large-sample limit. Such an estimate is called consistent. For example, one can produce a consistent estimate of the probability of any partially specified event $x_1,\ldots,x_m$, where $m \le n$, as follows:

$$P(x_1,\ \ldots, x_m) \approx N_{PS}(x_1,\ldots,x_m)/N. \tag{13.7}$$

That is, the probability of the event can be estimated as the fraction of all complete events generated by the sampling process that match the partially specified event. We will use P (pronounced "P-hat") to mean an estimated probability. So, if we generate 1,000 samples from the sprinkler network, and 511 of them have *Rain = true,* then the estimated probability of rain is $\widehat{P}(Rain = true) = 0.511$.

Rejection sampling in Bayesian networks

Rejection sampling is a general method for producing samples from a hard-to-sample distribution given an easy-to-sample distribution. In its simplest form, it can be used to compute conditional probabilities—that is, to determine $P(X \mid e)$. The REJECTION-SAMPLING algorithm is shown in Figure 13.17. First, it generates samples from the prior distribution specified by the network. Then, it rejects all those that do not match the evidence. Finally, the estimate $\widehat{P}(X = x \mid \mathbf{e})$ is obtained by counting how often $X = x$ occurs in the remaining samples.

```
function REJECTION-SAMPLING(X, e, bn, N) returns an estimate of P(X | e)
   inputs: X, the query variable
           e, observed values for variables E
           bn, a Bayesian network
           N, the total number of samples to be generated
   local variables: C, a vector of counts for each value of X, initially zero

   for j = 1 to N do
      x ← PRIOR-SAMPLE(bn)
      if x is consistent with e then
         C[j] ← C[j]+1 where x_j is the value of X in x
   return NORMALIZE(C)
```

Figure 13.17 The rejection-sampling algorithm for answering queries given evidence in a Bayesian network.

Let $\widehat{\mathbf{P}}(X \mid \mathbf{e})$ be the estimated distribution that the algorithm returns; this distribution is computed by normalizing $\mathbf{N}_{PS}(X, \mathbf{e})$, the vector of sample counts for each value of X where the sample agrees with the evidence $\mathbf{e}$:

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

From Equation (13.7), this becomes

$$\overset{\wedge}{\mathbf{P}}(X\ \mathbf{e}) \approx \frac{\mathbf{P}(X,\mathbf{e})}{P(\mathbf{e})} = \mathbf{P}(X\ \mathbf{e}).$$

That is, rejection sampling produces a consistent estimate of the true probability.

Continuing with our example from Figure 13.15(a), let us assume that we wish to estimate $\mathbf{P}(Rain \mid Sprinkler = true)$, using 100 samples. Of the 100 that we generate, suppose that 73 have *Sprinkler* = *false* and are rejected, while 27 have *Sprinkler* = *true;* of the 27, 8 have *Rain* = *true* and 19 have *Rain* = *false.* Hence,

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

The true answer is ⟨0.3,0.7⟩. As more samples are collected, the estimate will converge to the true answer. The standard deviation of the error in each probability will be proportional to $1/\sqrt{n}$, where n is the number of samples used in the estimate.

Now we know that rejection sampling converges to the correct answer, the next question is, how fast does that happen? More precisely, how many samples are required before we know that the resulting estimates are close to the correct answers with high probability? Whereas the complexity of exact algorithms depends to a large extent on the topology of the network—trees are easy, densely connected networks are hard—the complexity of rejection sampling depends primarily on the fraction of samples that are accepted. This fraction is exactly equal to the prior probability of the evidence, $P(\mathbf{e})$. Unfortunately, for complex problems with many evidence variables, this fraction is vanishingly small. When applied to the discrete version of the car insurance network in Figure 13.9, the fraction of samples consistent with a typical evidence case sampled from the network itself is usually between one in a thousand and one in ten thousand. Convergence is extremely slow (see Figure 13.19 below).

We expect the fraction of samples consistent with the evidence **e** to drop exponentially as the number of evidence variables grows, so the procedure is unusable for complex problems. It also has difficulties with continuous-valued evidence variables, because the probability of producing a sample consistent with such evidence is zero (if it is really continuous-valued) or infinitesimal (if it is merely a finite-precision floating-point number).

Notice that rejection sampling is very similar to the estimation of conditional probabilities in the real world. For example, to estimate the conditional probability that any humans survive after a 1km-diameter asteroid crashes into the Earth, one can simply count how often any humans survive after a 1km-diameter asteroid crashes into the Earth, ignoring all those days when no such event occurs. (Here, the universe itself plays the role of the sample-generation algorithm.) To get a decent estimate, one might need to wait for 100 such events to occur. Obviously, this could take a long time, and that is the weakness of rejection sampling.

Importance sampling

The general statistical technique of **importance sampling** aims to emulate the effect of sampling from a distribution P using samples from another distribution Q. We ensure that the answers are correct in the limit by applying a correction factor $P(\mathbf{x}) / Q(\mathbf{x})$, also known as a **weight**, to each sample **x** when counting up the samples.

The reason for using importance sampling in Bayes nets is simple: we would like to sample from the true posterior distribution conditioned on all the evidence, but usually this is too hard;[6] so instead, we sample from an easy distribution and apply the necessary corrections. The reason why importance sampling works is also simple. Let the nonevidence variables be **Z**. If we could sample directly from $P(\mathbf{z} \mid \mathbf{e})$, we could construct estimates like this:

$$\hat{P}(\mathbf{Z}\ \mathbf{e}) = \frac{N_P(\mathbf{z})}{N} \approx P(\mathbf{z}\ \mathbf{e})$$

where $N_P(\mathbf{z})$ is the number of samples with $\mathbf{Z} = \mathbf{z}$ when sampling from P. Now suppose instead that we sample from $Q(\mathbf{z})$. The estimate in this case includes the correction factors:

$$\hat{P}(\mathbf{z}\ \mathbf{e}) = \frac{N_Q(\mathbf{z})}{N}\frac{P(\mathbf{z}|\mathbf{e})}{Q(\mathbf{z})} \approx Q(\mathbf{z})\frac{P(\mathbf{z}|\mathbf{e})}{Q(\mathbf{z})} = P(\mathbf{z}\ \mathbf{e}).$$

Thus, the estimate converges to the correct value *regardless of which sampling distribution Q is used.* (The only technical requirement is that $Q(\mathbf{z})$ should not be zero for any **z** where $P(\mathbf{z} \mid \mathbf{e})$ is nonzero.) Intuitively, the correction factor compensates for oversampling or undersampling. For example, if $Q(\mathbf{z})$ is much bigger than $P(\mathbf{z} \mid \mathbf{e})$ for some **z**, then there will be many more samples of that z than there should be, but each will have a small weight, so it works out just as if there were the right number.

As for which Q to use, we want one that is easy to sample from and as close as possible to the true posterior $P(\mathbf{z} \mid \mathbf{e})$. The most common approach is called **likelihood weighting** (for reasons we will see shortly). As shown in the WEIGHTED-SAMPLE function in Figure 13.18, the algorithm fixes the values for the evidence variables **E** and samples all the nonevidence variables in topological order, each conditioned on its parents. This guarantees that each event generated is consistent with the evidence.

```
function LIKELIHOOD-WEIGHTING(X, e, bn, N) returns an estimate of P(X | e)
  inputs: X, the query variable
          e, observed values for variables E
          bn, a Bayesian network specifying joint distribution P(X_1, ..., X_n)
          N, the total number of samples to be generated
  local variables: W, a vector of weighted counts for each value of X, initially zero

  for j = 1 to N do
     x, w ← WEIGHTED-SAMPLE(bn, e)
     W[j] ← W[j] + w where x_j is the value of X in x
  return NORMALIZE(W)

function WEIGHTED-SAMPLE(bn, e) returns an event and a weight
  w ← 1; x ← an event with n elements, with values fixed from e
  for i = 1 to n do
     if X_i is an evidence variable with value x_ij in e
        then w ← w × P(X_i = x_ij | parents(X_i))
        else x[i] ← a random sample from P(X_i | parents(X_i))
  return x, w
```

Figure 13.18 The likelihood-weighting algorithm for inference in Bayesian networks. In WEIGHTED-SAMPLE, each nonevidence variable is sampled according to the conditional distribution given the values already sampled for the variable's parents, while a weight is accumulated based on the likelihood for each evidence variable.

Let's call the sampling distribution produced by this algorithm Q_{WS}. If the nonevidence variables are $\mathbf{Z} = \{Z_1, \ldots, Z_l\}$, then we have

$$Q_{WS}(\mathbf{z}) = \prod_{i=1}^{l} P(z_i | parents(Z_i)) \qquad (13.8)$$

because each variable is sampled conditioned on its parents. In order to complete the algorithm, we need to know how to compute the weight for each sample generated from Q_{WS}. According to the general scheme for importance sampling, the weight should be

$$w(\mathbf{z}) = P(\mathbf{z}|\mathbf{e})/Q_{WS}(\mathbf{z}) = \alpha P(\mathbf{z}, \mathbf{e})/Q_{WS}(\mathbf{z})$$

where the normalizing factor $\alpha = 1/P(\mathbf{e})$ is the same for all samples. Now $\mathbf{z}$ and $\mathbf{e}$ together cover all the variables in the Bayes net, so $P(\mathbf{z}, \mathbf{e})$ is just the product of all the conditional probabilities (Equation (13.2) page 433); and we can write this as the product of the conditional probabilities for the nonevidence variables times the product of the conditional probabilities for the evidence variables:

$$w(\mathbf{z}) = \alpha \frac{P(\mathbf{z}|\mathbf{e})}{Q_{WS}(\mathbf{z})} = \alpha \frac{\prod_{i=1}^{l} P(z_i|parents(Z_i)) \ \prod_{i=1}^{m} P(e_i|parents(E_i))}{\prod_{i=1}^{l} P(zi|parents(Z_i))}$$
$$= \alpha \prod_{i=1}^{m} P(e_i|parents(E_i)). \qquad (13.9)$$

Thus the weight is the product of the conditional probabilities for the evidence variables given their parents. (Probabilities of evidence are generally called **likelihoods**, hence the name.) The weight calculation is implemented incrementally in WEIGHTED-SAMPLE, multiplying by the conditional probability each time an evidence variable is encountered. The normalization is done at the end before the query result is returned.

Let us apply the algorithm to the network shown in Figure 13.15(a), with the query **P***(Rain|Cloudy = true, WetGrass = true)* and the ordering *Cloudy, Sprinkler, Rain, WetGrass.* (Any topological ordering will do.) The process goes as follows: First, the weight *w* is set to 1.0. Then an event is generated:

1. *Cloudy* is an evidence variable with value *true.* Therefore, we set

   $w \leftarrow w \times P(Cloudy = true) \ = \ 0.5.$

2. *Sprinkler* is not an evidence variable, so sample from **P**(*Sprinkler* | *Cloudy* = *true*) = (0.1,0.9); suppose this returns *false.*
3. *Rain* is not an evidence variable, so sample from **P**(*Rain* | *Cloudy* = *true*) = (0.8,0.2); suppose this returns *true*.
4. *WetGrass* is an evidence variable with value *true.* Therefore, we set

   $$w \leftarrow w \times P(WetGrass = true|Sprinkler = false, Rain = true)$$
   $$= \ 0.5 \times 0.9 = 0.45.$$

Here WEIGHTED-SAMPLE returns the event [*true,false, true, true*] with weight 0.45, and this is tallied under *Rain* = *true.*

Notice that *Parents*(Z_i) can include both nonevidence variables and evidence variables. Unlike the prior distribution *P*(**z**), the distribution Q_{WS} pays some attention to the evidence: the sampled values for each Z_i will be influenced by evidence among Z_i's ancestors. For example, when sampling *Sprinkler* the algorithm pays attention to the evidence *Cloudy* = *true* in its parent variable. On the other hand, Q_{WS} pays less attention to the evidence than does the true posterior distribution P(**z** | **e**), because the sampled values for each Z_i *ignore* evidence among Z_i's non-ancestors. For example, when sampling *Sprinkler* and *Rain* the algorithm ignores the evidence in the child variable *WetGrass* = *true;* this means it will generate many samples with *Sprinkler* = *false* and *Rain* = *false* despite the fact that the evidence actually rules out this case. Those samples will have zero weight.

Because likelihood weighting uses all the samples generated, it can be much more efficient than rejection sampling. It will, however, suffer a degradation in performance as the number of evidence variables increases. This is because most samples will have very low weights and hence the weighted estimate will be dominated by the tiny fraction of samples that accord more than an infinitesimal likelihood to the evidence. The problem is exacerbated if the evidence variables occur "downstream"—that is, late in the variable ordering—because then the nonevidence variables will have no evidence in their parents and ancestors to guide the generation of samples. This means the samples will be mere hallucinations—simulations that bear little resemblance to the reality suggested by the evidence.

When applied to the discrete version of the car insurance network in Figure 13.9, likelihood weighting is considerably more efficient than rejection sampling (see Figure 13.19). The insurance network is a relatively benign case for likelihood weighting because much of the evidence is “upstream” and the query variables are leaf nodes of the network.

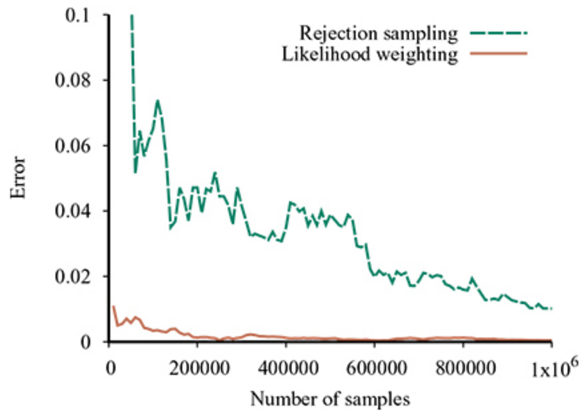


Figure 13.19 Performance of rejection sampling and likelihood weighting on the insurance network. The x-axis shows the number of samples generated and the y-axis shows the maximum absolute error in any of the probability values for a query on *PropertyCost*.

13.4.2 Inference by Markov chain simulation

Markov chain Monte Carlo (MCMC) algorithms work differently from rejection sampling and likelihood weighting. Instead of generating each sample from scratch, MCMC algorithms generate a sample by making a random change to the preceding sample. Think of an MCMC algorithm as being in a particular *current state* that specifies a value for every variable and generating a *next state* by making random changes to the current state.

The term **Markov chain** refers to a random process that generates a sequence of states. (Markov chains also figure prominently in Chapters 14 and 16; the simulated annealing algorithm in Chapter 4 and the WALK SAT algorithm in Chapter 7 are also members of the MCMC family.) We begin by describing a particular form of MCMC called **Gibbs sampling**, which is especially well suited for Bayes nets. We then describe the more general **Metropolis–Hastings** algorithm, which allows much greater flexibility in generating samples.

Gibbs sampling in Bayesian networks

The Gibbs sampling algorithm for Bayesian networks starts with an arbitrary state (with the evidence variables fixed at their observed values) and generates a next state by randomly sampling a value for one of the nonevidence variables X_i. Recall from page 437 that X_i is independent of all other variables given its Markov blanket (its parents, children, and children’s other parents); therefore, Gibbs sampling for X_i means sampling *conditioned on the current values of the variables in its Markov blanket.* The algorithm wanders randomly around the state space —the space of possible complete assignments—flipping one variable at a time, but keeping the evidence variables fixed. The complete algorithm is shown in Figure 13.20.

function GIBBS-ASK(X, $\mathbf{e}$, bn, N) **returns** an estimate of $\mathbf{P}(X \mid \mathbf{e})$
local variables: $\mathbf{C}$, a vector of counts for each value of X, initially zero
$\mathbf{Z}$, the nonevidence variables in bn
$\mathbf{x}$, the current state of the network, initialized from $\mathbf{e}$

```
initialize x with random values for the variables in Z
for k = 1 to N do
    choose any variable Z_i from Z according to any distribution ρ(i)
    set the value of Z_i in x by sampling from P(Z_i | mb(Z_i))
    C[j] ← C[j] + 1 where x_j is the value of X in x
return NORMALIZE(C)
```

Figure 13.20 The Gibbs sampling algorithm for approximate inference in Bayes nets; this version chooses variables at random, but cycling through the variables but also works.

Consider the query **P**(*Rain* | *Sprinkler* = *true*, *WetGrass* = *true)* for the network in Figure 13.15(a). The evidence variables *Sprinkler* and *WetGrass* are fixed to their observed values (both *true*), and the nonevidence variables *Cloudy* and *Rain* are initialized randomly to, say, *true* and *false* respectively. Thus, the initial state is [*true*, **true**, *false*, **true**], where we have marked the fixed evidence values in bold. Now the nonevidence variables Z_i are sampled repeatedly in some random order according to a probability distribution $\rho(i)$ for choosing variables. For example:

1. *Cloudy* is chosen and then sampled, given the current values of its Markov blanket: in this case, we sample from **P**(*Cloudy* | *Sprinkler* = *true*, *Rain* = *false*). Suppose the result is *Cloudy* = *false*. Then the new current state is [*false*, **true**, *false*, **true**].
2. *Rain* is chosen and then sampled, given the current values of its Markov blanket: in this case, we sample from **P**(*Rain*| *Cloudy* = *false*,*Sprinkler* = *true*, *WetGrass* = *true)*. Suppose this yields *Rain* = *true*. The new current state is [*false*, **true**, *true*, **true**].

The one remaining detail concerns the method of calculating the Markov blanket distribution $\mathbf{P}(X_i \mid mb(X_i))$, where $mb(X_i)$ denotes the values of the variables in X_i's Markov blanket, $MB(X_i)$. Fortunately, this does not involve any complex inference. As shown in Exercise 13.MARB, the distribution is given by

$$P(x_i\ mb(X_i)) = \alpha\ P(x_i\ parents(X_i)) \prod_{Y_j \in Children(X_i)} P(y_j | parents(Y_j))\ .$$

In other words, for each value x_i, the probability is given by multiplying probabilities from the CPTs of X_i and its children. For example, in the first sampling step shown above, we sampled from **P**(*Cloudy* | *Sprinkler* = *true*, *Rain* = *false*). By Equation (13.10), and abbreviating the variable names, we have

$$\begin{aligned} P(c|s,\neg r) &= \alpha\ P(c)P(s|c)P(\neg r|c) = \alpha\ 0.5 \cdot 0.1 \cdot 0.2 \\ P(\neg c|s,\neg r) &= P(\neg c)P(s|\neg c)P(\neg r|\neg c) = \alpha\ 0.5 \cdot 0.5 \cdot 0.8, \end{aligned}$$

so the sampling distribution is $\alpha\langle 0.001, 0.020\rangle \approx \langle 0.048, 0.952\rangle$.

Figure 13.21(a) shows the complete Markov chain for the case where variables are chosen uniformly, i.e., *ρ(Cloudy)* = *ρ(Rain)* = 0.5. The algorithm is simply wandering around in this graph, following links with the stated probabilities. Each state visited during this process is a sample that contributes to the estimate for the query variable *Rain*. If the process visits 20 states where *Rain* is true and 60 states where *Rain* is false, then the answer to the query is NORMALIZE(⟨20, 60⟩) = ⟨0.25,0.75⟩.

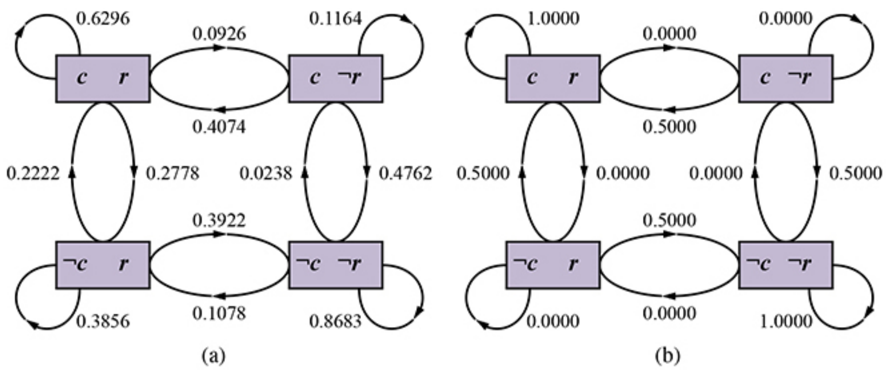


Figure 13.21 (a) The states and transition probabilities of the Markov chain for the query **P**(*Rain* | *Sprinkler* = *true*, *WetGrass* = *true*). Note the self-loops: the state stays the same when *either* variable is chosen and then resamples the same value it already has. (b) The transition probabilities when the CPT for *Rain* constrains it to have the same value as *Cloudy*.

Analysis of Markov chains

We have said that Gibbs sampling works by wandering randomly around the state space to generate samples. To explain why Gibbs sampling works *correctly*—that is, why its estimates converge to correct values in the limit—we will need some careful analysis. (This section is somewhat mathematical and can be skipped on first reading.)

We begin with some of the basic concepts for analyzing Markov chains in general. Any such chain is defined by its initial state and its **transition kernel** $k(\mathbf{x} \to \mathbf{x}')$—the probability of a transition to state $\mathbf{x}'$ starting from state $\mathbf{x}$. Now suppose that we run the Markov chain for t steps, and let $\pi_t(\mathbf{x})$ be the probability that the system is in state $\mathbf{x}$ at time t. Similarly, let $\pi_{t+1}(\mathbf{x}')$ be the probability of being in state $\mathbf{x}'$ at time $t+1$. Given $\pi_t(\mathbf{x})$, we can calculate $\pi_{t+1}(\mathbf{x}')$ by summing, for all states $\mathbf{x}$ the system could be in at time t, the probability of being in $\mathbf{x}$ times the probability of making the transition to $\mathbf{x}'$:

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

We say that the chain has reached its **stationary distribution** if $\pi_t = \pi_{t+1}$. Let us call this stationary distribution π; its defining equation is therefore

$$\pi(\mathbf{x}') = \sum_{\mathbf{x}} \pi(\mathbf{x}) k(\mathbf{x} \to \mathbf{x}') \qquad \text{for all } \mathbf{x}' . \qquad (13.11)$$

Provided the transition kernel k is **ergodic**—that is, every state is reachable from every other and there are no strictly periodic cycles—there is exactly one distribution π satisfying this equation for any given k.

Equation (13.11) can be read as saying that the expected "outflow" from each state (i.e., its current "population") is equal to the expected "inflow" from all the states. One obvious way to satisfy this relationship is if the expected flow between any pair of states is the same in both directions; that is,

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

When these equations hold, we say that $k(\mathbf{x} \to \mathbf{x}')$ is in detailed balance with $\pi(\mathbf{x})$. One special case is the self-loop $\mathbf{x} = \mathbf{x}'$, i.e., a transition from a state to itself. In that case, the detailed balance condition becomes $\pi(\mathbf{x})k(\mathbf{x} \to \mathbf{x}) = \pi(\mathbf{x})k(\mathbf{x} \to \mathbf{x})$ which is of course trivially true for any stationary distribution π and any transition kernel k.

We can show that detailed balance implies stationarity simply by summing over $\mathbf{x}$ in Equation (13.12). We have

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

where the last step follows because a transition from $\mathbf{x}'$ is guaranteed to occur.

Why Gibbs sampling works

We will now show that Gibbs sampling returns consistent estimates for posterior probabilities. The basic claim is straightforward: *the stationary distribution of the Gibbs sampling process is exactly the posterior distribution for the nonevidence variables conditioned on the evidence.* This remarkable property follows from the specific way in which the Gibbs sampling process moves from state to state.

The general definition of Gibbs sampling is that a variable X_i is chosen and then sampled conditionally on the current values of *all* the other variables. (When applied specifically to Bayes nets, we simply use the additional fact that sampling conditionally on all variables is equivalent to sampling conditionally on the variable's Markov blanket, as shown on page 437.) We will use the notation $\overline{\mathbf{X}}_i$ to refer to these other variables (except the evidence variables); their values in the current state are $\overline{\mathbf{x}}_i$.

To write down the transition kernel $k(\mathbf{x} \to \mathbf{x}')$ for Gibbs sampling, there are three cases to consider:

1. The states $\mathbf{x}$ and $\mathbf{x}'$ differ in two or more variables. In that case, $k(\mathbf{x} \to \mathbf{x}') = 0$ because Gibbs sampling changes only a single variable.
2. The states differ in exactly one variable X_i that changes its value from x_i to x'_i. The probability of such an occurrence is

$$k(\mathbf{x} \to \mathbf{x}') = k((x_i, \overline{\mathbf{x}}_i) \to (x'_i, \overline{\mathbf{x}}_i)) = \rho(i)P(x'_i|\overline{\mathbf{x}}_i). \qquad (13.13)$$

3. The states are the same: $\mathbf{x} = \mathbf{x}'$. In that case, *any* variable could be chosen but then the sampling process produces the same value the variable already has. The probability of such an occurrence is

$$k(\mathbf{x} \to \mathbf{x}) = \sum_i \rho(i)k((x_i, \overline{x}_i) \to (x_i, \overline{x}_i)) = \sum_i \rho(i)P(x_i|\overline{\mathbf{x}}_i).$$

Now we show that this general definition of Gibbs sampling satisfies the detailed balance equation with a stationary distribution equal to $P(\mathbf{x} \mid \mathbf{e})$, the true posterior distribution on the nonevidence variables. That is, we show that $\pi(\mathbf{x})k(\mathbf{x} \to \mathbf{x}') = \pi(\mathbf{x}')k(\mathbf{x}' \to \mathbf{x})$ where $\pi(\mathbf{x}) = P(\mathbf{x} \mid \mathbf{e})$, for all states $\mathbf{x}$ and $\mathbf{x}'$.

For the first and third cases given above, detailed balance is ***always*** satisfied: if two states differ in two or more variables, the transition probability in both directions is zero. If $\mathbf{x} \neq \mathbf{x}'$ then from Equation (13.13), we have

$$\pi(\mathbf{x})k(\mathbf{x} \to \mathbf{x}\prime) = P(\mathbf{x}|\mathbf{e})\rho(i)P(x_i'|\mathbf{x}_i, \mathbf{e}) = \rho(i)P(x_i, \mathbf{x}_i|\mathbf{e})P(x_i'|\mathbf{x}_i, \mathbf{e})$$
$$= \rho(i)P(x_i|\mathbf{x}_i, \mathbf{e})P(\mathbf{x}_i|\mathbf{e})P(x_i'|\mathbf{x}_i, \mathbf{e}) \qquad \text{(using the chain rule on the first term)}$$
$$= \rho(i)P(x_i|\mathbf{x}_i, \mathbf{e})P(x_i'|\mathbf{x}_i, \mathbf{e}) \qquad \text{(reverse chain rule on last two terms)}$$
$$= \pi(\mathbf{x}\prime)k(\mathbf{x}\prime \to \mathbf{x}).$$

The final piece of the puzzle is the ergodicity of the chain—that is, every state must be reachable from every other and there are no periodic cycles. Both conditions are satisfied provided the CPTs do not contain probabilities of 0 or 1. Reachability comes from the fact that we can convert one state into another by changing one variable at a time, and the absence of periodic cycles comes from the fact that every state has a self-loop with nonzero probability. Hence, under the stated conditions, k is ergodic, which means that the samples generated by Gibbs sampling will eventually be drawn from the true posterior distribution.

Complexity of Gibbs sampling

First, the good news: each Gibbs sampling step involves calculating the Markov blanket distribution for the chosen variable X_i, which requires a number of multiplications proportional to the number of X_i's children and the size of X_i's range. This is important because it means that *the work required to generate each sample is independent of the size of the network.*

Now, the not necessarily bad news: the complexity of Gibbs sampling is much harder to analyze than that of rejection sampling and likelihood weighting. The first thing to notice is that Gibbs sampling, unlike likelihood weighting, *does* pay attention to downstream evidence. Information propagates from evidence nodes in all directions: first, any neighbors of the evidence nodes sample values that reflect the evidence in those nodes; then *their* neighbors, and so on. Thus, we expect Gibbs sampling to outperform likelihood weighting when evidence is mostly downstream; and indeed, this is borne out in Figure 13.22.

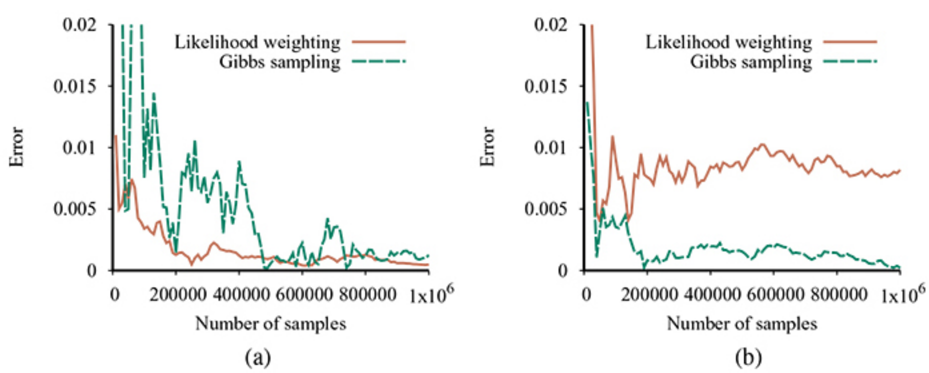


Figure 13.22 Performance of Gibbs sampling compared to likelihood weighting on the car insurance network: (a) for the standard query on *PropertyCost*, and (b) for the case where the output variables are observed and *Age* is the query variable.

The rate of convergence for Gibbs sampling—the **mixing rate** of the Markov chain defined by the algorithm —depends strongly on the quantitative properties of the conditional distributions in the network. To see this, consider what happens in Figure 13.15(a) as the CPT for *Rain* becomes deterministic: it rains *if and only if* it is cloudy. In that case, the true posterior distribution for the query **P**(*Rain* | *sprinkler, wetGrass*) is roughly ⟨0.18,0.82⟩ but Gibbs sampling will never reach this value. The problem is that the only two joint states for *Cloudy* and *Rain* that have non-zero probability are [*true*, *true*] and [*false*, *false*]. Starting in *[true, true]*, the chain can never reach [*false, false*] because transitions to the intermediate states have probability zero (see Figure 13.21(b)). So, if the process starts in [*true*, *true*] it always reports a posterior probability for the query of ⟨1.0,0.0⟩; if it starts in [*false, false*] it always reports a posterior probability for the query of ⟨0.0,1.0⟩.

Gibbs sampling fails in this case because the deterministic relationship between *Cloudy* and *Rain* breaks the property of ergodicity that is required for convergence. If, however, we make the relationship *nearly* deterministic, then convergence is restored, but happens arbitrarily slowly. There are several fixes that help MCMC algorithms mix more quickly. One is **block sampling**: sampling multiple variables simultaneously. In this case, we could sample *Cloudy* and *Rain* jointly, conditioned on their combined Markov blanket. Another is to generate next states more intelligently, as we will see in the next section.

Metropolis–Hastings sampling

The Metropolis–Hastings or MH sampling method is perhaps the most broadly applicable MCMC algorithm. Like Gibbs sampling, MH is designed to generate samples $\mathbf{x}$ (eventually) according to target probabilities $\pi(\mathbf{x})$; in the case of inference in Bayesian networks, we want $\pi(\mathbf{x}) = P(\mathbf{x} \mid \mathbf{e})$. Like simulated annealing (page 133), MH has two stages in each iteration of the sampling process:

1. Sample a new state $\mathbf{x}'$ from a **proposal distribution** $q(\mathbf{x}' \mid \mathbf{x})$, given the current state $\mathbf{x}$.
2. Probabilistically accept or reject $\mathbf{x}'$ according to the **acceptance probability**

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

   If the proposal is rejected, the state remains at $\mathbf{x}$.

The transition kernel for MH consists of this two-step process. Note that if the proposal is rejected, the chain stays in the same state.

The proposal distribution is responsible, as its name suggests, for proposing a next state $\mathbf{x}'$. For example, $q(\mathbf{x}' \mid \mathbf{x})$ could be defined as follows:

- With probability 0.95, perform a Gibbs sampling step to generate $\mathbf{x}'$.
- Otherwise, generate $\mathbf{x}'$ by running the WEIGHTED-SAMPLE algorithm from page 458.

This proposal distribution causes MH to do about 20 steps of Gibbs sampling then "restarts" the process from a new state (assuming it is accepted) that is generated from scratch. By this stratagem, it gets around the problem of Gibbs sampling getting stuck in one part of the state space and being unable to reach the other parts.

You might ask how on Earth we know that MH with such a weird proposal actually converges to the right answer. The remarkable thing about MH is that *convergence to the correct stationary distribution is guaranteed for* any *proposal distribution*, provided the resulting transition kernel is ergodic.

This property follows from the way the acceptance probability is defined. As with Gibbs sampling, the self-loop with $\mathbf{x} = \mathbf{x}'$ automatically satisfies detailed balance, so we focus on the case where $\mathbf{x} \neq \mathbf{x}'$. This can occur only if the proposal is accepted. The probability of such a transition occurring is

$$k(\mathbf{x} \to \mathbf{x}') = q(\mathbf{x}'|\mathbf{x})a(\mathbf{x}'|\mathbf{x}).$$

As with Gibbs sampling, proving detailed balance means showing that the flow from $\mathbf{x}$ to $\mathbf{x}'$, $\pi(\mathbf{x})k(\mathbf{x} \to \mathbf{x}')$, matches the flow from $\mathbf{x}'$ to $\mathbf{x}$, $\pi(\mathbf{x}')k(\mathbf{x}' \to \mathbf{x})$. After plugging in the expression above for $k(\mathbf{x} \to \mathbf{x}')$, the proof is quite straightforward:

$$\begin{aligned}
\pi(\mathbf{x})q(\mathbf{x}\prime\ \mathbf{x})a(\mathbf{x}\prime\ \mathbf{x}) = \pi(\mathbf{x})q(\mathbf{x}\prime\ \mathbf{x})\min\ (1, \tfrac{\pi(\mathbf{x}\prime)q(\mathbf{x}|\mathbf{x}\prime)}{\pi(\mathbf{x})q(\mathbf{x}\prime|\mathbf{x})}) &\quad (\text{definition of } a\,(\ \cdot \mid \cdot\ )) \\
= \ \min(\pi(\mathbf{x})q, (\mathbf{x}\prime|\mathbf{x}), \pi(\mathbf{x}\prime)q(\mathbf{x}|\mathbf{x}\prime)) &\quad (\text{multiplying in}) \\
= \ \pi(\mathbf{x}\prime)q(\mathbf{x}\ \mathbf{x}\prime)\ \min\ (\tfrac{\pi(\mathbf{x})q(\mathbf{x}\prime|\mathbf{x})}{\pi(\mathbf{x}\prime)q(\mathbf{x}|\mathbf{x}\prime)}, 1) &\quad (\text{dividing out}) \\
= \pi(\mathbf{x}\prime)q(\mathbf{x}|\mathbf{x}\prime)a(\mathbf{x}|\mathbf{x}\prime). &
\end{aligned}$$

Mathematical properties aside, the important part of MH to focus on is the ratio $\pi(\mathbf{x}')/\pi(\mathbf{x})$ in the acceptance probability. This says that if a next state is proposed that is *more* likely than the current state, it will definitely be accepted. (We are overlooking, for now, the term $q(\mathbf{x} \mid \mathbf{x}')/q(\mathbf{x}' \mid \mathbf{x})$, which is there to ensure detailed balance and is, in many state spaces, equal to 1 because of symmetry.) If the proposed state is *less* likely than the current state, its probability of being accepted drops proportionally.

Thus, one guideline for designing proposal distributions is to make sure the new states being proposed are reasonably likely. Gibbs sampling does this automatically: it proposes from the Gibbs distribution $P(X_i \mid \mathbf{X}_i)$, which means that the probability of generating any particular new value for X_i is directly proportional to its probability. (Exercise 13.GIBM asks you to show that Gibbs is a special case of MH with an acceptance probability of 1.)

Another guideline is to make sure that the chain mixes well, which means sometimes proposing large moves to distant parts of the state space. In the example given above, the occasional use of WEIGHTED-SAMPLE to restart the chain in a new state serves this purpose.

Besides near-complete freedom in designing proposal distributions, MH has two additional properties that make it practical. First, the posterior probability $\pi(\mathbf{x}) = P(\mathbf{x}|\mathbf{e})$ appears in the acceptance calculation only in the form of a ratio $\pi(\mathbf{x}')/\pi(\mathbf{x})$, which is very fortunate. Computing $P(\mathbf{x} \mid \mathbf{e})$ directly is the very computation we're trying to approximate using MH, so it wouldn't make sense to do it for each sample! Instead, we use the following trick:

$$\frac{\pi(\mathbf{x}\prime)}{\pi(\mathbf{x})} = \frac{P(\mathbf{x}\prime|\mathbf{e})}{P(\mathbf{x}|\mathbf{e})} = \frac{P(\mathbf{x}\prime,\mathbf{e})}{P(\mathbf{e})}\,\frac{P(\mathbf{e})}{P(\mathbf{x},\mathbf{e})} = \frac{P(\mathbf{x}\prime,\mathbf{e})}{P(\mathbf{x},\mathbf{e})}.$$

The terms in this ratio are full joint probabilities, i.e., products of conditional probabilities in the Bayes net. The second useful property of this ratio is that as long as the proposal distribution makes only local changes in $\mathbf{x}$ to produce $\mathbf{x}'$, only a small number of terms in the product of conditional probabilities will be different. All of the conditional probabilities involving variables whose values are unchanged will cancel out in the ratio. So, as with Gibbs sampling, the work required to generate each sample is independent of the size of the network as long as the state changes are local.

13.4.3 Compiling approximate inference

The sampling algorithms in Figures 13.17, 13.18, and 13.20 share a common property: they operate on a Bayes net represented as a data structure. This seems quite natural: after all, a Bayes net is a directed acyclic graph, so how else could it be represented? The problem with this approach is that the operations required to access the data structure—for example to find a node's parents—are repeated thousands or millions of times as the sampling algorithm runs, and *all of these computations are completely unnecessary.*

The network's structure and conditional probabilities remain fixed throughout the computation, so there is an opportunity to *compile* the network into model-specific inference code that carries out just the inference computations needed for that specific network. (In case this sounds familiar, it is the same idea used in the compilation of logic programs in Chapter 9.) For example, suppose we want to Gibbs-sample the *Earthquake* variable in the burglary network of Figure 13.2. According to the GIBBS-ASK algorithm in Figure 13.20, we need to perform the following computation:

$$\text{set the value of } Earthquake \text{ in } \mathbf{x} \text{ by sampling from } \mathbf{P}(Earthquake|mb(Earthquake))$$

where the latter distribution is computed according to Equation (13.10), repeated here:

$$P(x_i\ mb(X_i)) = \alpha\, P(x_i\ parents(X_i)) \prod_{Y_j \in Children(X_i)} P(y_j|\ parents(Y_j)).$$

This computation, in turn, requires looking up the parents and children of *Earthquake* in the Bayes net structure; looking up their current values; using those values to index into the corresponding CPTs (which also have to be found from the Bayes net); and multiplying together all the appropriate rows from those CPTs to form a new distribution from which to sample. Finally, as noted on page 454, the sampling step itself has to construct the cumulative version of the discrete distribution and then find the value therein that corresponds to a random number sampled from [0,1].

If, instead, we compile the network, we obtain model-specific sampling code for the *Earthquake* variable that looks like this:

$r \leftarrow$ a uniform random sample from $[0,1]$
if $Alarm = true$
  then if $Burglary = true$
    then return $[r < 0.0020212]$
    else return $[r < 0.36755]$
  else if $Burglary = true$
    then return $[r < 0.0016672]$
    else return $[r < 0.0014222]$

Here, Bayes net variables *Alarm, Burglary,* and so on become ordinary program variables with values that comprise the current state of the Markov chain. The numerical threshold expressions evaluate to *true* or *false* and represent the precomputed Gibbs distributions for each combination of values in the Markov blanket of *Earthquake.* The code is not especially pretty—typically, it will be roughly as large as the Bayes net itself—but it is incredibly efficient. Compared to GIBBS-ASK, the compiled code will typically be 2-3 orders of magnitude faster. It can perform tens of millions of sampling steps per second on an ordinary laptop, and its speed is limited largely by the cost of generating random numbers.

13.5 Causal Networks

We have discussed several advantages of keeping node ordering in Bayes nets compatible with the direction of causation. In particular, we noted the ease with which conditional probabilities can be assessed if such ordering is maintained, as well as the compactness of the resultant network structure. We noted however that, in principle, any node ordering permits a consistent construction of the network to represent the joint distribution function. This was demonstrated in Figure 13.3, where changing the node ordering produced networks that were bushier and a lot less natural than the original network in Figure 13.2 but enabled us, nevertheless, to represent the same distribution on all variables.

This section describes **causal networks**, a restricted class of Bayesian networks that forbids all but causally compatible orderings. We will explore how to construct such networks, what is gained by such construction, and how to leverage this gain in decision-making tasks.

Consider the simplest Bayesian network imaginable, a single arrow, $Fire \rightarrow Smoke$. It tells us that variables *Fire* and *Smoke* may be dependent, so one needs to specify the prior $P(Fire)$ and the conditional probability $P(Smoke \mid Fire)$ in order to specify the joint distribution $P(Fire, Smoke)$. However, this distribution can be represented equally well by the reverse arrow $Fire \leftarrow Smoke$, using the appropriate $P(Smoke)$ and $P(Fire \mid Smoke)$ computed from Bayes' rule. The idea that these two networks are equivalent, hence convey the same information, evokes discomfort and even resistance in most people. How could they convey the same information when we know that *Fire* causes *Smoke* and not the other way around?

In other words, we know from our experience and scientific understanding that clearing the smoke would not stop the fire and extinguishing the fire will stop the smoke. We expect therefore to represent this asymmetry through the directionality of the arrow between them. But if arrow reversal only makes things equivalent, how can we hope to represent this important information formally?

Causal Bayesian networks, sometimes called Causal Diagrams, were devised to permit us to represent causal asymmetries and to leverage the asymmetries towards reasoning with causal information. The idea is to decide on arrow directionality by considerations that go beyond probabilistic dependence and invoke a totally different type of judgment. Instead of asking an expert whether *Smoke* and *Fire* are probabilistically dependent, as we do in ordinary Bayesian networks, we now ask which responds to which, *Smoke* to *Fire* or *Fire* to *Smoke*?

This may sound a bit mystical, but it can be made precise through the notion of "assignment," similar to the assignment operator in programming languages. If nature assigns a value to *Smoke* on the basis of what nature learns about *Fire*, we draw an arrow from *Fire* to *Smoke*. More importantly, if we judge that nature assigns *Fire* a truth value that depends on other variables, not *Smoke*, we refrain from drawing the arrow $Fire \leftarrow Smoke$. In other words, the value x_i of each variable X_i is determined by an equation $x_i = f_i(OtherVariables)$, and an arrow $X_j \rightarrow X_i$ is drawn if and only if X_j is one of the arguments of f_i.

The equation $x_i = f_i(\cdot)$ is called a **structural equation**, because it describes a stable mechanism in nature which, unlike the probabilities that quantify a Bayesian network, remains invariant to measurements and local changes in the environment.

To appreciate this stability to local changes, consider Figure 13.23(a), which depicts a slightly modified version of the lawn sprinkler story of Figure 13.15. To represent a disabled sprinkler, for example, we simply delete from the network all links incident to the *Sprinkler* node. To represent a lawn covered by a tent, we simply delete the arrow $Rain \rightarrow WetGrass$. Any local reconfiguration of the mechanisms in the environment can thus be translated, with only minor modification, into an isomorphic reconfiguration of the network topology. A much more elaborate transformation would be required had the network been constructed contrary to causal ordering. This local stability is particularly important for representing actions or interventions, our next topic of discussion.

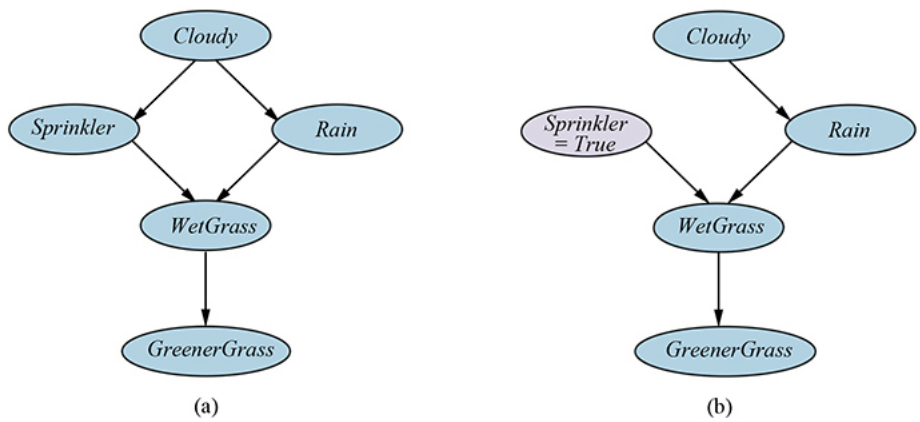


Figure 13.23 (a) A causal Bayesian network representing cause-effect relations among five variables. (b) The network after performing the action "turn *Sprinkler* on."

13.5.1 Representing actions: The *do*-operator

Consider again the *Sprinkler* story of Figure 13.23(a). According to the standard semantics of Bayes nets, the joint distribution of the five variables is given by a product of five conditional distributions:

$$P(c,\ r,\ s, w, g)\ =\ P(c)\ P(r|c)\ P(s|c)\ P(w|r, s)\ P(g|w) \qquad (13.14)$$

where we have abbreviated each variable name by its first letter. As a system of structural equations, the model looks like this:

$$\begin{aligned} C &= f_C(U_C) \\ R &= f_R(C, U_R) \\ S &= f_S(C, U_S) \\ W &= f_W(R, S, U_W) \\ G &= f_G(W, U_G) \end{aligned} \qquad (13.15)$$

where, without loss of generality, f_C can be the identity function. The *U*-variables in these equations represent **unmodeled variables**, also called **error terms** or **disturbances**, that perturb the functional relationship between each variable and its parents. For example, U_W may represent another potential source of wetness, in addition to *Sprinkler* and *Rain*—perhaps *MorningDew* or *FirefightingHelicopter.*

If all the *U*-variables are mutually independent random variables with suitably chosen priors, the joint distribution in Equation (13.14) can be represented exactly by the structural equations in Equation (13.15). Thus, a system of stochastic relationships can be captured by a system of deterministic relationships, each of which is affected by an exogenous disturbance. However, the system of structural equations gives us more than that: it allows us to predict how *interventions* will affect the operation of the system and hence the observable consequences of those interventions. This is not possible given just the joint distribution.

For example, suppose we *turn the sprinkler on*—that is, if we (who are, by definition, not part of the causal processes described by the model) *intervene* to impose the condition *Sprinkler*=*true*. In the notation of the **do-calculus**, which is a key part of the theory of causal networks, this is written as *do*(*Sprinkler* = *true*). Once done, this means that the sprinkler variable is no longer dependent on whether it's a cloudy day. We therefore delete the equation $S = f_S(C, U_S)$ from the system of structural equations and replace it with $S = true$, giving us

$$\begin{aligned} C &= f_C(U_C) \\ R &= f_R(C, U_R) \\ S &= true \\ W &= f_W(R, S, U_W) \\ G &= f_G(W, U_G). \end{aligned} \qquad (13.16)$$

From these equations, we obtain the new joint distribution for the remaining variables conditioned on *do*(*Sprinkler* = *true*):

$$P(c, r, w, g | do(S = true)) = P(c)\, P(r|c)\, P(w|r,\ s = true)\, P(g|w) \qquad (13.1$$

This corresponds to the "mutilated" network in Figure 13.23(b). From Equation (13.17), we see that the only variables whose probabilities change are *WetGrass* and *GreenerGrass*, that is, the descendants of the manipulated variable *Sprinkler*.

Note the difference between conditioning on the *action* *do*(*Sprinkler* = *true*) in the original network and conditioning on the *observation* *Sprinkler* = *true*. The original network tells us that the sprinkler is less likely to be on when the weather is cloudy, so if we *observe* the sprinkler to be on, that reduces the probability that the weather is cloudy. But common sense tells us that if we (operating from outside the world, so to speak) reach in and turn on the sprinkler, that doesn't affect the weather or provide new information about what the weather is like that day. As shown in Figure 13.23(b), intervening breaks the normal causal link between the weather and the sprinkler. This prevents any influence flowing backward from *Sprinkler* to *Cloudy*. Thus, conditioning on *do*(*Sprinkler* = *true*) in the original graph is equivalent to conditioning on *Sprinkler* = *true* in the mutilated graph.

A similar approach can be taken to analyze the effect of $do(X_j = x_{jk})$ in a general causal network with variables $X_1,...,X_n$. The network corresponds to a joint distribution defined in the usual way (see Equation (13.2)):

$$P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i | parents(X_i)). \qquad (13.18)$$

After applying $do(X_j = X_{jk})$, the new joint distribution P_{xjk} simply omits the factor for X_j:

$$P_{xjk}(x_1, \ldots, x_n) = \begin{cases} \prod_{i \neq j} P(x_i\ parents(X_i)) = \dfrac{\mathrm{P}(x_1, \ldots, x_n)}{P(x_j | parents(X_j))} & \text{if } \ x_j = x_{jk} \\ 0 & \text{if } \ x_j \neq x_{jk} \end{cases}$$

This follows from the fact that setting X_j to a particular value X_{jk} corresponds to deleting the equation $X_j = f_j(Parents(X_j), U_j)$ from the system of structural equations and replacing it with $X_j = X_{jk}$. With a bit more algebraic manipulation, one can derive a formula for the effect of setting variable X_j on any other variable X_i:

$$P(X_i \ = \ x_i | do(X_j = x_{jk})) = P_{x\ jk}\ (X_i = x_i)$$
$$= \sum_{parents(X_j)} P(x_i | x_{jk},\ parents(X_j)) P(parents(X_j))\,. \qquad (13.20)$$

The probability terms in the sum are obtained by computation on the original network, by any of the standard inference algorithms. This equation is known as an **adjustment formula**; it is a probability-weighted average of the influence of X_j and its parents on X_i, where the weights are the priors on the parent values. The effects of intervening on multiple variables can be computed by imagining that the individual interventions happen in sequence, each one in turn deleting the causal influences on a variable and yielding a new, mutilated model.

13.5.2 The back-door criterion

The ability to predict the effect of any intervention is a remarkable result, but it does require accurate knowledge of the necessary conditional distributions in the model, particularly $P(x_j \mid parents(X_j))$. In many real-world settings, however, this is too much to ask. For example, we know that “genetic factors” play a role in obesity, but we do not know which genes play a role or the precise nature of their effects. Even in the simple story of Mary’s sprinkler decisions (Figure 13.15, which also applies in Figure 13.23(a)), we might know that she checks the weather before deciding whether to turn on the sprinkler, but we might not know *how* she makes her decision.

The specific reason this is problematic in this instance is that we would like to predict the effect of turning on the sprinkler on a downstream variable such as *GreenerGrass*, but the adjustment formula (Equation (13.20)) must take into account not only the direct route from *Sprinkler*, but also the “back door” route via *Cloudy* and *Rain*. If we knew the value of *Rain*, this back-door path would be blocked—which suggests that there might be a way to write an adjustment formula that conditions on *Rain* instead of *Cloudy*. And indeed this is possible:

$$P(g\ do(S = true)) = \sum_r P(g | S = true,\ r) P(r) \qquad (13.21)$$

In general, if we wish to find the effect of $do(X_j = x_{jk})$ on a variable X_i, the **back-door criterion** allows us to write an adjustment formula that conditions on any set of variables **Z** that closes the back door, so to speak. In more technical language, we want a set **Z** such that X_i is conditionally independent of $Parents(X_j)$ given X_j and **Z**. This is a straightforward application of d-separation (see page 437).

The back-door criterion is a basic building block for a theory of causal reasoning that has emerged in the past two decades. It provides a way to argue against a century of statistical dogma asserting that only a **randomized controlled trial** can provide causal information. The theory has provided conceptual tools and algorithms for causal analysis in a wide range of non-experimental and quasi-experimental settings; for computing probabilities on counterfactual statements (“if this had happened instead, what would the probability have been?”); for determining when findings in one population can be transferred to another; and for handling all forms of missing data when learning probability models.

Summary

This chapter has described **Bayesian networks**, a well-developed representation for uncertain knowledge. Bayesian networks play a role roughly analogous to that of propositional logic for definite knowledge.

- A Bayesian network is a directed acyclic graph whose nodes correspond to random variables; each node has a conditional distribution for the node, given its parents.
- Bayesian networks provide a concise way to represent **conditional independence** relationships in the domain.
- A Bayesian network specifies a joint probability distribution over its variables. The probability of any given assignment to all the variables is defined as the product of the corresponding entries in the local conditional distributions. A Bayesian network is often exponentially smaller than an explicitly enumerated joint distribution.
- Many conditional distributions can be represented compactly by canonical families of distributions. **Hybrid Bayesian networks**, which include both discrete and continuous variables, use a variety of canonical distributions.
- Inference in Bayesian networks means computing the probability distribution of a set of query variables, given a set of evidence variables. Exact inference algorithms, such as **variable elimination**, evaluate sums of products of conditional probabilities as efficiently as possible.
- In **polytrees** (singly connected networks), exact inference takes time linear in the size of the network. In the general case, the problem is intractable.

- Random sampling techniques such as **likelihood weighting** and **Markov chain Monte Carlo** can give reasonable estimates of the true posterior probabilities in a network and can cope with much larger networks than can exact algorithms.
- Whereas Bayes nets capture probabilistic influences, **causal networks** capture causal relationships and allow prediction of the effects of interventions as well as observations.

Bibliographical and Historical Notes

The use of networks to represent probabilistic information began early in the 20th century, with the work of Sewall Wright on the probabilistic analysis of genetic inheritance and animal growth factors (Wright, 1921, 1934). I. J. Good (1961), in collaboration with Alan Turing, developed probabilistic representations and Bayesian inference methods that could be regarded as a forerunner of modern Bayesian networks—although the paper is not often cited in this context.[7] The same paper is the original source for the noisy-OR model.

The **influence diagram** representation for decision problems, which incorporated a DAG representation for random variables, was used in decision analysis in the late 1970s (see Chapter 15), but only enumeration was used for evaluation. Judea Pearl developed the messagepassing method for inference in tree networks (Pearl, 1982a) and polytree networks (Kim and Pearl, 1983) and explained the importance of causal rather than diagnostic probability models. The first expert system using Bayesian networks was CONVINCE (Kim, 1983).

As chronicled in Chapter 1, the mid-1980s saw a boom in rule-based expert systems, which incorporated ad hoc methods for handling uncertainty. Probability was considered both impractical and “cognitively implausible” as a basis for reasoning. Peter Cheeseman’s (1985) pugnacious “In Defense of Probability” and his later article “An Inquiry into Computer Understanding” (Cheeseman, 1988, with commentaries) helped to turn the tables.

The resurgence of probability depended mainly, however, on Pearl’s development of Bayesian networks and the broad development of a probabilistic approach to AI as outlined in his book, *Probabilistic Reasoning in Intelligent Systems* (Pearl, 1988). The book covered both representational issues, including conditional independence relationships and the d-separation criterion, and algorithmic approaches. Geiger *et al.* (1990a) and Tian *et al.*(1998) presented key computational results on efficient detection of d-separation.

Eugene Charniak helped present Pearl's ideas to AI researchers with a popular article, "Bayesian networks without tears"[8] (1991), and book (1993). The book by Dean and Wellman (1991) also helped introduce Bayesian networks to AI researchers. Shachter (1998) presented a simplified way to determine d-separation called the "Bayes-ball" algorithm.

As applications of Bayes nets were developed, researchers found it necessary to go beyond the basic model of discrete variables with CPTs. For example, the CPCS system (Prad- han *et al.*, 1994), a Bayesian network for internal medicine with 448 nodes and 906 links, made extensive use of the noisy logical operators proposed by Good (1961). Boutilier *et al.*(1996) analyzed the algorithmic benefits of context-specific independence. The inclusion of continuous random variables in Bayesian networks was considered by Pearl (1988) and Shachter and Kenley (1989); these papers discussed networks containing only continuous variables with linear Gaussian distributions.

Hybrid networks with both discrete and continuous variables were investigated by Lauritzen and Wermuth (1989) and implemented in the cHUGIN system (Olesen, 1993). Further analysis of linear–Gaussian models, with connections to many other models used in statistics, appears in Roweis and Ghahramani (1999); Lerner (2002) provides a very thorough discussion of their use in hybrid Bayes nets. The probit distribution is usually attributed to Gaddum (1933) and Bliss (1934), although it had been discovered several times in the 19th century. Bliss's work was expanded considerably by Finney (1947). The probit has been used widely for modeling discrete choice phenomena and can be extended to handle more than two choices (Daganzo, 1979). The expit (inverse logit) model was introduced by Berkson (1944); initially much derided, it eventually became more popular than the probit model. Bishop (1995) gives a simple justification for its use.

Early applications of Bayes nets in medicine included the MUNIN system for diagnosing neuromuscular disorders (Andersen *et al.*, 1989) and the PATHFINDER system for pathology (Heckerman, 1991). Applications in engineering include the Electric Power Research Institute's work on monitoring power generators (Morjaria *et al.*, 1995), NASA's work on displaying time-critical information at Mission Control in Houston (Horvitz and Barry, 1995), and the general field of **network tomography**, which aims to infer unobserved local properties of nodes and links in the Internet from observations of end-to-end message performance (Castro *et al.*, 2004). Perhaps the most widely used Bayesian network systems have been the diagnosis-and-repair modules (e.g., the Printer Wizard) in Microsoft Windows (Breese and Heckerman, 1996) and the Office Assistant in Microsoft Office (Horvitz *et al.*, 1998).

Another important application area is biology: the mathematical models used to analyze genetic inheritance in family trees (so-called **pedigree analysis**) are in fact a special form of Bayesian networks. Exact inference algorithms for pedigree analysis, resembling variable elimination, were developed in the 1970s (Cannings *et al.*, 1978). Bayesian networks have been used for identifying human genes by reference to mouse genes (Zhang et *al.*, 2003), inferring cellular networks (Friedman, 2004), genetic linkage analysis to locate disease-related genes (Silberstein et *al.*, 2013), and many other tasks in bioinformatics. We could go on, but instead we'll refer you to Pourret et *al.* (2008), a 400-page guide to applications of Bayesian networks. Published applications over the last decade run into the tens of thousands, ranging from dentistry to global climate models.

Judea Pearl (1985), in the first paper to use the term "Bayesian networks," briefly described an inference algorithm for general networks based on the cutset conditioning idea introduced in Chapter 5. Independently, Ross Shachter (1986), working in the influence diagram community, developed a complete algorithm based on goal-directed reduction of the network using posterior-preserving transformations.

Pearl (1986) developed a clustering algorithm for exact inference in general Bayesian networks, utilizing a conversion to a directed polytree of clusters in which message passing was used to achieve consistency over variables shared between clusters. A similar approach, developed by the statisticians David Spiegelhalter and Steffen Lauritzen (Lauritzen and Spiegelhalter, 1988), is based on conversion to an undirected form of graphical model called a **Markov network**. This approach is implemented in the HUGIN system, an efficient and widely used tool for uncertain reasoning (Andersen et *al.*, 1989).

The basic idea of variable elimination—that repeated computations within the overall sum-of-products expression can be avoided by caching—appeared in the symbolic probabilistic inference (SPI) algorithm (Shachter et *al.*, 1990). The elimination algorithm we describe is closest to that developed by Zhang and Poole (1994). Criteria for pruning irrelevant variables were developed by Geiger et *al.* (1990b) and by Lauritzen et *al.* (1990); the criterion we give is a simple special case of these. Dechter (1999) shows how the variable elimination idea is essentially identical to **nonserial dynamic programming** (Bertele and Brioschi, 1972).

This connects Bayesian network algorithms to related methods for solving CSPs and gives a direct measure of the complexity of exact inference in terms of the tree width of the network. Preventing the exponential growth in the size of factors in variable elimination can be done by dropping variables from large factors (Dechter and Rish, 2003); it is also possible to bound the error introduced thereby (Wexler and Meek, 2009). Alternatively, factors can be compressed by representing them using algebraic decision diagrams instead of tables (Gogate and Domingos, 2011).

Exact methods based on recursive enumeration (see Figure 13.11) combined with caching include the recursive conditioning algorithm (Darwiche, 2001), the value elimination algorithm (Bacchus et *al.*, 2003), and AND-OR search (Dechter and Mateescu, 2007). The method of weighted model counting (Sang *et al.*, 2005; Chavira and Darwiche, 2008) is usually based on a DPLL-style SAT solver (see Figure 7.17 on page 252). As such, it is also performing a recursive enumeration of variable assignments with caching, so the approach is in fact quite similar. All three of these algorithms can implement a complete range of space/time tradeoffs. Because they consider variable assignments, the algorithms can easily take advantage of determinism and context-specific independence in the model. They can also be modified to use an efficient linear-time algorithm whenever the partial assignment makes the remaining network a polytree. (This is a version of the method of **cutset conditioning**, which was described for CSPs in Chapter 5.) For exact inference in large models, where the space requirements of clustering and variable elimination become enormous, these recursive algorithms are often the most practical approach.

There are other important inference tasks in Bayes nets besides computing marginal probabilities. The **most probable explanation** or MPE is the most likely assignment to the nonevidence variables given the evidence. (MPE is a special case of MAP—maximum a posteriori—inference, which asks for the most likely assignment to a *subset* of nonevidence variables given the evidence.) For such problems, many different algorithms have been developed, some related to shortest-path or AND-OR search algorithms; for a summary, see Marinescu and Dechter (2009).

The first result on the complexity of inference in Bayes nets is due to Cooper (1990), who showed that the general problem of computing marginals in Bayesian networks is NP- hard; as noted in the chapter, this can be strengthened to #P-hardness through a reduction from counting satisfying assignments (Roth, 1996). This also implies the NP-hardness of approximate inference (Dagum and Luby, 1993); however, for the case where probabilities can be bounded away from 0 and 1, a form of likelihood weighting converges in (randomized) polynomial time (Dagum and Luby, 1997). Shimony (1994) showed that finding the most probable explanation is NP-complete—intractable, but somewhat easier than computing marginals—while Park and Darwiche (2004) provide a thorough complexity analysis of MAP computation, showing that it falls into the class of NP^{PP} -complete problems—that is, somewhat harder than computing marginals.

The development of fast approximation algorithms for Bayesian network inference is a very active area, with contributions from statistics, computer science, and physics. The rejection sampling method is a general technique dating back at least to Buffon's needle (1777); it was first applied to Bayesian networks by Max Henrion (1988), who called it **logic sampling**. Importance sampling was invented originally for applications in physics (Kahn, 1950a, 1950b) and applied to Bayes net inference by Fung and Chang (1989) (who called the algorithm “evidence weighting”) and by Shachter and Peot (1989).

In statistics, **adaptive sampling** has been applied to all sorts of Monte Carlo algorithms to speed up convergence. The basic idea is to adapt the distribution from which samples are generated, based on the outcome from previous samples. Gilks and Wild (1992) developed adaptive rejection sampling, while adaptive importance sampling appears to have originated independently in physics (Lepage, 1978), civil engineering (Karamchandani et *al.*, 1989), statistics (Oh and Berger, 1992), and computer graphics (Veach and Guibas, 1995). Cheng and Druzdzel (2000) describe an adaptive version of importance sampling applied to Bayes net inference. More recently, Le et *al.* (2017) have demonstrated the use of deep learning systems to produce proposal distributions that speed up importance sampling by many orders of magnitude.

Markov chain Monte Carlo (MCMC) algorithms began with the Metropolis algorithm, due to Metropolis et *al.* (1953), which was also the source of the simulated annealing algorithm described in Chapter 4. Hastings (1970) introduced the accept/reject step that is an integral part of what we now call the Metropolis-Hastings algorithm. The Gibbs sampler was devised by Geman and Geman (1984) for inference in undirected Markov networks. The application of Gibbs sampling to Bayesian networks is due to Pearl (1987). The papers collected by Gilks et *al.* (1996) cover both theory and applications of MCMC.

Since the mid-1990s, MCMC has become the workhorse of Bayesian statistics and statistical computation in many other disciplines including physics and biology. The *Handbook of Markov Chain Monte Carlo* (Brooks *et al.*, 2011) covers many aspects of this literature. The BUGS package (Gilks *et al.*, 1994) was an early and influential system for Bayes net modeling and inference using Gibbs sampling. STAN (named after Stanislaw Ulam, an originator of Monte Carlo methods in physics) is a more recent system that uses Hamiltonian Monte Carlo inference (Carpenter *et al.*, 2017).

There are two very important families of approximation methods that we did not cover in the chapter. The first is the family of **variational approximation** methods, which can be used to simplify complex calculations of all kinds. The basic idea is to propose a reduced version of the original problem that is simple to work with, but that resembles the original problem as closely as possible. The reduced problem is described by some **variational parameters** λ that are adjusted to minimize a distance function D between the original and the reduced problem, often by solving the system of equations $\partial D/\partial\lambda = 0$. In many cases, strict upper and lower bounds can be obtained. Variational methods have long been used in statistics (Rustagi, 1976). In statistical physics, the **mean-field** method is a particular variational approximation in which the individual variables making up the model are assumed to be completely independent.

This idea was applied to solve large undirected Markov networks (Peterson and Anderson, 1987; Parisi, 1988). Saul *et al.* (1996) developed the mathematical foundations for applying variational methods to Bayesian networks and obtained accurate lower-bound approximations for sigmoid networks with the use of mean-field methods. Jaakkola and Jordan (1996) extended the methodology to obtain both lower and upper bounds. Since these early papers, variational methods have been applied to many specific families of models. The remarkable paper by Wainwright and Jordan (2008) provides a unifying theoretical analysis of the literature on variational methods.

A second important family of approximation algorithms is based on Pearl's polytree message-passing algorithm (1982a). This algorithm can be applied to general "loopy" networks, as suggested by Pearl (1988). The results might be incorrect, or the algorithm might fail to terminate, but in many cases, the values obtained are close to the true values. Little attention was paid to this so-called **loopy belief propagation** approach until McEliece *et al.*(1998) observed that it is exactly the computation performed by the **turbo decoding** algorithm (Berrou *et al.*, 1993), which provided a major breakthrough in the design of efficient error-correcting codes.

The implication of these observations is if loopy BP is both fast and accurate on the very large and very highly connected networks used for decoding, it might therefore be useful more generally. Theoretical support for these findings, including convergence proofs for some special cases, was provided by Weiss (2000b), Weiss and Freeman (2001), and Yedidia *et al.* (2005), drawing on connections to ideas from statistical physics.

Theories of causal inference going beyond randomized controlled trials were proposed by Rubin (1974) and Robins (1986), but these ideas remained both obscure and controversial until Judea Pearl developed and presented a fully articulated theory of causality based on causal networks (Pearl, 2000). Peters *et al.* (2017) further develop the theory, with an emphasis on learning. A more recent work, *The Book of Why* (Pearl and McKenzie, 2018), provides a less mathematical but more readable and wide-ranging introduction.

Uncertain reasoning in AI has not always been based on probability theory. As noted in Chapter 12, early probabilistic systems fell out of favor in the early 1970s, leaving a partial vacuum to be filled by alternative methods. These included rule-based expert systems, Dempster-Shafer theory, and (to some extent) fuzzy logic.[9]

Rule-based approaches to uncertainty hoped to build on the success of logical rule- based systems, but add a sort of “fudge factor”—more politely called a **certainty factor**—to each rule to accommodate uncertainty. The first such system was MYCIN (Shortliffe, 1976), a medical expert system for bacterial infections. The collection *Rule-Based Expert Systems* (Buchanan and Shortliffe, 1984) provides a complete overview of MYCIN and its descendants (see also Stefik, 1995).

David Heckerman (1986) showed that a slightly modified version of certainty factor calculations gives correct probabilistic results in some cases, but results in serious overcounting of evidence in other cases. As rule sets became larger, undesirable interactions between rules became more common, and practitioners found that the certainty factors of many other rules had to be “tweaked” when new rules were added. The basic mathematical properties that allow *chains* of reasoning in logic simply do not hold for probability.

Dempster-Shafer theory originates with a paper by Arthur Dempster (1968) proposing a generalization of probability to interval values and a combination rule for using them. Such an approach might alleviate the difficulty of specifying probabilities exactly. Later work by Glenn Shafer (1976) led to the Dempster-Shafer theory’s being viewed as a competing approach to probability. Pearl (1988) and Ruspini *et al.* (1992) analyze the relationship between the Dempster-Shafer theory and standard probability theory. In many cases, probability theory does not require probabilities to be specified exactly: we can express uncertainty about probability values as (second-order) probability distributions, as explained in Chapter 21.

Fuzzy sets were developed by Lotfi Zadeh (1965) in response to the perceived difficulty of providing exact inputs to intelligent systems. A fuzzy set is one in which membership is a matter of degree. **Fuzzy logic** is a method for reasoning with logical expressions describing membership in fuzzy sets. **Fuzzy control** is a methodology for constructing control systems in which the mapping between real-valued input and output parameters is represented by fuzzy rules. Fuzzy control has been very successful in commercial products such as automatic transmissions, video cameras, and electric shavers. The text by Zimmermann (2001) provides a thorough introduction to fuzzy set theory; papers on fuzzy applications are collected in Zimmermann (1999).

Fuzzy logic has often been perceived incorrectly as a direct competitor to probability theory, whereas in fact it addresses a different set of issues: rather than considering uncertainty about the truth of well-defined propositions, fuzzy logic handles **vagueness** in the mapping from terms in a symbolic theory to an actual world. Vagueness is a real issue in any application of logic, probability, or indeed standard mathematical models to reality. Even a variable as impeccable as the mass of the Earth turns out, on inspection, to vary with time as meteorites and molecules come and go. It is also imprecise—does it include the atmosphere? If so, to what height? In some cases, further elaboration of the model can reduce vagueness, but fuzzy logic takes vagueness as a given and develops a theory around it.

Possibility theory (Zadeh, 1978) was introduced to handle uncertainty in fuzzy systems and has much in common with probability (Dubois and Prade, 1994).

Many AI researchers in the 1970s rejected probability because the numerical calculations that probability theory was thought to require were not apparent to introspection and presumed an unrealistic level of precision in our uncertain knowledge. The development of **qualitative probabilistic networks** (Wellman, 1990a) provided a purely qualitative abstraction of Bayesian networks, using the notion of positive and negative influences between variables. Wellman shows that in many cases such information is sufficient for optimal decision making without the need for the precise specification of probability values. Goldszmidt and Pearl (1996) take a similar approach. Work by Darwiche and Ginsberg (1992) extracts the basic properties of conditioning and evidence combination from probability theory and shows that they can also be applied in logical and default reasoning.

Several excellent texts (Jensen, 2007; Darwiche, 2009; Koller and Friedman, 2009; Korb and Nicholson, 2010; Dechter, 2019) provide thorough treatments of the topics we have covered in this chapter. New research on probabilistic reasoning appears both in mainstream AI journals, such as *Artificial Intelligence* and the *Journal of AI Research*, and in more specialized journals, such as the *International Journal of Approximate Reasoning*. Many papers on graphical models, which include Bayesian networks, appear in statistical journals. The proceedings of the conferences on Uncertainty in Artificial Intelligence (UAI), Neural Information Processing Systems (NeurIPS), and Artificial Intelligence and Statistics (AISTATS) are good sources for current research.

---

[1] Bayesian networks, often abbreviated to "Bayes net," were called belief networks in the 1980s and 1990s. A causal network is a Bayes net with additional constraints on the meaning of the arrows (see Section 13.5). The term graphical model refers to a broader class that includes Bayesian networks.

[2] It follows that inference in linear-Gaussian networks takes only $O(n^3)$ time in the worst case, regardless of the network topology. In Section 13.3, we see that inference for networks of discrete variables is NP-hard.

[3] The network shown in Figure 13.9 is not in actual use, but its structure has been vetted with insurance experts. In practice, the information requested on application forms varies by company and

jurisdiction—for example, some ask for *Gender*—and the model could certainly be made more detailed and sophisticated.

[4] Some insurance companies also acquire the applicant's credit history to help in assessing risk; this provides considerably more information about socioeconomic category. Whenever using hidden variables of this kind, one must be careful that they do not inadvertently become proxies for variables such as race that may not be used in insurance decisions. Techniques for avoiding biases of this kind are described in Chapter 19.

[5] Another widely studied task is finding the most probable explanation for some observed evidence. This and other tasks are discussed in the notes at the end of the chapter.

[6] If it was easy, then we could approximate the desired probability to arbitrary accuracy with a polynomial number of samples. It can be shown that no such polynomial-time approximation scheme can exist.

[7] I. J. Good was chief statistician for Turing's code-breaking team in World War II. In *2001: A Space Odyssey*(Clarke, 1968), Good and Minsky are credited with making the breakthrough that led to the development of the HAL 9000 computer.

[8] The title of the original version of the article was "Pearl for swine."

[9] A fourth approach, **default reasoning**, treats conclusions not as "believed to a certain degree," but as "believed until a better reason is found to believe something else." It is covered in Chapter 10.

CHAPTER 14

PROBABILISTIC REASONING OVER TIME

In which we try to interpret the present, understand the past, and perhaps predict the future, even when very little is crystal clear

Agents in partially observable environments must be able to keep track of the current state, to the extent that their sensors allow. In Section 4.4 we showed a methodology for doing that: an agent maintains a **belief state** that represents which states of the world are currently possible. From the belief state and a **transition model**, the agent can predict how the world might evolve in the next time step. From the percepts observed and a **sensor model**, the agent can update the belief state. This is a pervasive idea: in Chapter 4 belief states were represented by explicitly enumerated sets of states, whereas in Chapters 7 and 11 they were represented by logical formulas. Those approaches defined belief states in terms of which world states were *possible*, but could say nothing about which states were *likely* or *unlikely*. In this chapter, we use probability theory to quantify the degree of belief in elements of the belief state.

As we show in Section 14.1, time itself is handled in the same way as in Chapter 7: a changing world is modeled using a variable for each aspect of the world state *at each point in time*. The transition and sensor models may be uncertain: the transition model describes the probability distribution of the variables at time t, given the state of the world at past times, while the sensor model describes the probability of each percept at time t, given the current state of the world. Section 14.2 defines the basic inference tasks and describes the general structure of inference algorithms for temporal models. Then we describe three specific kinds of models: **hidden Markov models, Kalman filters**, and **dynamic Bayesian networks** (which include hidden Markov models and Kalman filters as special cases).

14.1 Time and Uncertainty

We have developed our techniques for probabilistic reasoning in the context of *static* worlds, in which each random variable has a single fixed value. For example, when repairing a car, we assume that whatever is broken remains broken during the process of diagnosis; our job is to infer the state of the car from observed evidence, which also remains fixed.

Now consider a slightly different problem: treating a diabetic patient. As in the case of car repair, we have evidence such as recent insulin doses, food intake, blood sugar measurements, and other physical signs. The task is to assess the current state of the patient, including the actual blood sugar level and insulin level. Given this information, we can make a decision about the patient's food intake and insulin dose. Unlike the case of car repair, here the *dynamic* aspects of the problem are essential. Blood sugar levels and measurements thereof can change rapidly over time, depending on recent food intake and insulin doses, metabolic activity, the time of day, and so on. To assess the current state from the history of evidence and to predict the outcomes of treatment actions, we must model these changes.

The same considerations arise in many other contexts, such as tracking the location of a robot, tracking the economic activity of a nation, and making sense of a spoken or written sequence of words. How can dynamic situations like these be modeled?

14.1.1 States and observations

This chapter discusses **discrete-time** models, in which the world is viewed as a series of snapshots or **time slices**.[1] We'll just number the time slices 0, 1, 2, and so on, rather than assigning specific times to them. Typically, the time interval Δ between slices is assumed to be the same for every interval. For any particular application, a specific value of Δ has to be chosen. Sometimes this is dictated by the sensor; for example, a video camera might supply images at intervals of 1/30 of a second. In other cases, the interval is dictated by the typical rates of change of the relevant variables; for example, in the case of blood glucose monitoring, things can change significantly in the course of ten minutes, so a one-minute interval might be appropriate. On the other hand, in modeling continental drift over geological time, an interval of a million years might be fine.

Each time slice in a discrete-time probability model contains a set of random variables, some observable and some not. For simplicity, we will assume that the same subset of variables is observable in each time slice (although this is not strictly necessary in anything that follows). We will use $\mathbf{X}_t$ to denote the set of state variables at time t, which are assumed to be unobservable, and $\mathbf{E}_t$ to denote the set of observable evidence variables. The observation at time t is $\boldsymbol{E}_t = \mathbf{e}_t$ for some set of values $\mathbf{e}_t$.

Consider the following example: You are the security guard stationed at a secret underground installation. You want to know whether it's raining today, but your only access to the outside world occurs each morning when you see the director coming in with, or without, an umbrella. For each day t, the set $\mathbf{E}_t$ thus contains a single evidence variable $Umbrella_t$ or U_t for short (whether the umbrella appears), and the set $\mathbf{X}_t$ contains a single state variable Rain_t or R_t for short (whether it is raining). Other problems can involve larger sets of variables. In the diabetes example, the evidence variables might be $MeasuredBloodSugar_t$ and $PulseRate_t$ while the state variables might include $BloodSugar_t$ and $StomachContents_t$. (Notice that $BloodSugar_t$ and $MeasuredBloodSugar_t$ are not the same variable; this is how we deal with noisy measurements of actual quantities.)

We will assume that the state sequence starts at $t = 0$ and evidence starts arriving at $t = 1$. Hence, our umbrella world is represented by state variables R_0, R_1, R_2,... and evidence variables U_1, U_2,.... We will use the notation $a{:}b$ to denote the sequence of integers from a to b inclusive and the notation $\mathbf{X}_{a:b}$ to denote the set of variables from $\mathbf{X}_a$ to $\mathbf{X}_b$ inclusive. For example, $U_{1:3}$ corresponds to U_1, U_2, U_3. (Note that this is different from the notation used in programming languages such as Python and Go, where U[1:3] would *not* include U[3].)

14.1.2 Transition and sensor models

With the set of state and evidence variables for a given problem decided on, the next step is to specify how the world evolves (the transition model) and how the evidence variables get their values (the sensor model).

The transition model specifies the probability distribution over the latest state variables, given the previous values, that is, $\mathbf{P}(\mathbf{X}_t \mid \mathbf{X}_{0:t-1})$. Now we face a problem: the set $\mathbf{X}_{0:t-1}$ is unbounded in size as t increases. We solve the problem by making a **Markov assumption**— that the current state depends on only a *finite fixed number* of previous states. Processes satisfying this assumption were first studied in depth by the statistician Andrei Markov (1856— 1922) and are called **Markov processes** or **Markov chains**. They come in various flavors; the simplest is the **first-order Markov process**, in which the current state depends only on the previous state and not on any earlier states. In other words, a state provides enough information to make the future conditionally independent of the past, and we have

$$\mathbf{P}(\mathbf{X}_t|\mathbf{X}_{0:t-1}) \;=\; \mathbf{P}(\mathbf{X}_t|\mathbf{X}_{t-1}). \qquad (14.1)$$

Hence, in a first-order Markov process, the transition model is the conditional distribution $\mathbf{P}(\mathbf{X}_t \mid \mathbf{X}_{t-1})$. The transition model for a second-order Markov process is the conditional distribution $\mathbf{P}(\mathbf{X}_t \mid \mathbf{X}_{t-2}, \mathbf{X}_{t-1})$. Figure 14.1 shows the Bayesian network structures corresponding to first-order and second-order Markov processes.

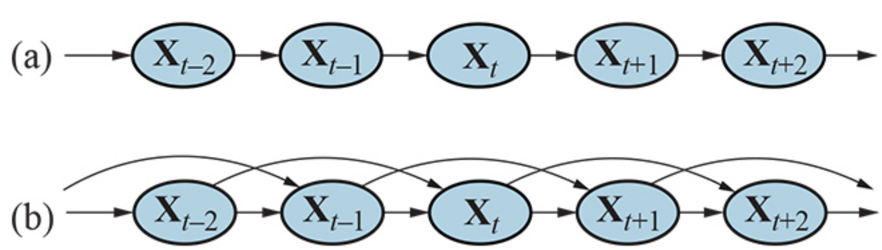


Figure 14.1 (a) Bayesian network structure corresponding to a first-order Markov process with state defined by the variables $\mathbf{X}_t$. (b) A second-order Markov process.

Even with the Markov assumption there is still a problem: there are infinitely many possible values of t. Do we need to specify a different distribution for each time step? We avoid this problem by assuming that changes in the world state are caused by a **time-homogeneous** process—that is, a process of change that is governed by laws that do not themselves change over time. In the umbrella world, then, the conditional probability of rain, $\mathbf{P}(R_t \mid R_{t-1})$, is the same for all t, and we need specify only one conditional probability table.

Now for the sensor model. The evidence variables $\mathbf{E}_t$ *could* depend on previous variables as well as the current state variables, but any state that's worth its salt should suffice to generate the current sensor values. Thus, we make a **sensor Markov assumption** as follows:

$$\mathbf{P}(\mathbf{E}_t|\mathbf{X}_{0:t}, \mathbf{E}_{1:t-1}) = \mathbf{P}(\mathbf{E}_t|\mathbf{X}_t). \quad (14.2)$$

Thus, $\mathbf{P}(\mathbf{E}_t \mid \mathbf{X}_t)$ is our sensor model (sometimes called the **observation model**). Figure 14.2 shows both the transition model and the sensor model for the umbrella example. Notice the direction of the dependence between state and sensors: the arrows go from the actual state of the world to sensor values because the state of the world *causes* the sensors to take on particular values: the rain *causes* the umbrella to appear. (The inference process, of course, goes in the other direction; the distinction between the direction of modeled dependencies and the direction of inference is one of the principal advantages of Bayesian networks.)

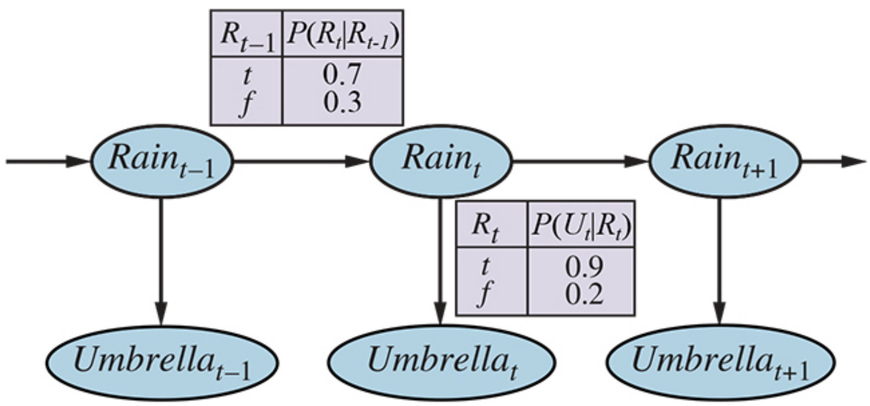


Figure 14.2 Bayesian network structure and conditional distributions describing the umbrella world. The transition model is $\mathbf{P}(Rain_t \mid Rain_{t-1})$ and the sensor model is $\mathbf{P}(Umbrella \mid Rain_t)$.

In addition to specifying the transition and sensor models, we need to say how everything gets started—the prior probability distribution at time 0, $\mathbf{P}(\mathbf{X}_0)$. With that, we have a specification of the complete joint distribution over all the variables, using Equation (13.2). For any time step t,

$$\mathbf{P}(\mathbf{X}_{0:t}, \mathbf{E}_{1:t}) = \mathbf{P}(\mathbf{X}_0) \prod_{i=1}^{t} \mathbf{P}(\mathbf{X}_i|\mathbf{X}_{i-1})\mathbf{P}(\mathbf{E}_i|\mathbf{X}_i). \quad (14.3)$$

The three terms on the right-hand side are the initial state model $\mathbf{P}(\mathbf{X}_0)$, the transition model $\mathbf{P}(\mathbf{X}_i \mid \mathbf{X}_{i-1})$, and the sensor model $\mathbf{P}(\mathbf{E}_i \mid \mathbf{X}_i)$. This equation defines the semantics of the family of temporal models represented by the three terms. Notice that standard Bayesian networks cannot represent such models because they require a finite set of variables. The ability to handle an infinite set of variables comes from two things: first, defining the infinite set using integer indices; and second, the use of implicit universal quantification (see Section 8.2) to define the sensor and transition models for every time step.

The structure in Figure 14.2 is a first-order Markov process—the probability of rain is assumed to depend only on whether it rained the previous day. Whether such an assumption is reasonable depends on the domain itself. The first-order Markov assumption says that the state variables contain *all* the information needed to characterize the probability distribution for the next time slice. Sometimes the assumption is exactly true—for example, if a particle is executing a random walk along the x-axis, changing its position by ±1 at each time step, then using the x-coordinate as the state gives a first-order Markov process. Sometimes the assumption is only approximate, as in the case of predicting rain only on the basis of whether it rained the previous day. There are two ways to improve the accuracy of the approximation:

1. Increasing the order of the Markov process model. For example, we could make a second-order model by adding $Rain_{t-2}$ as a parent of $Rain_t$, which might give slightly more accurate predictions. For example, in Palo Alto, California, it very rarely rains more than two days in a row.
2. Increasing the set of state variables. For example, we could add $Season_t$ to allow us to incorporate historical records of rainy seasons, or we could add $Temperature_t$, $Humidity_t$, and $Pressure_t$ (perhaps at a range of locations) to allow us to use a physical model of rainy conditions.

Exercise 14.AUGM asks you to show that the first solution—increasing the order—can always be reformulated as an increase in the set of state variables, keeping the order fixed. Notice that adding state variables might improve the system's predictive power but also increases the prediction *requirements:* we now have to predict the new variables as well. Thus, we are looking for a "self-sufficient" set of variables, which really means that we have to understand the "physics" of the process being modeled. The requirement for accurate modeling of the process is obviously lessened if we can add new sensors (e.g., measurements of temperature and pressure) that provide information directly about the new state variables.

Consider, for example, the problem of tracking a robot wandering randomly on the X–Y plane. One might propose that the position and velocity are a sufficient set of state variables: one can simply use Newton's laws to calculate the new position, and the velocity may change unpredictably. If the robot is battery-powered, however, then battery exhaustion would tend to have a systematic effect on the change in velocity. Because this in turn depends on how much power was used by all previous maneuvers, the Markov property is violated.

We can restore the Markov property by including the charge level $Battery_t$ as one of the state variables that make up $\mathbf{X}_t$. This helps in predicting the motion of the robot, but in turn requires a model for predicting $Battery_t$ from $Battery_{t-1}$ and the velocity. In some cases, that can be done reliably, but more often we find that error accumulates over time. In that case, accuracy can be improved by *adding a new sensor* for the battery level. We will return to the battery example in Section 14.5.

14.2 Inference in Temporal Models

Having set up the structure of a generic temporal model, we can formulate the basic inference tasks that must be solved:

- **Filtering**[2] or **state estimation** is the task of computing the **belief state** $\mathbf{P}(\mathbf{X}_t \mid \mathbf{e}_{1:t})$—the posterior distribution over the most recent state given all evidence to date. In the umbrella example, this would mean computing the probability of rain today, given all the umbrella observations made so far. Filtering is what a rational agent does to keep track of the current state so that rational decisions can be made. It turns out that an almost identical calculation provides the likelihood of the evidence sequence, $P(\mathbf{e}_{1:t})$.
- **Prediction:** This is the task of computing the posterior distribution over the *future* state, given all evidence to date. That is, we wish to compute $\mathbf{P}(\mathbf{X}_{t+k} \mid \mathbf{e}_{1:t})$ for some $k > 0$. In the umbrella example, this might mean computing the probability of rain three days from now, given all the observations to date. Prediction is useful for evaluating possible courses of action based on their expected outcomes.
- **Smoothing**: This is the task of computing the posterior distribution over a *past* state, given all evidence up to the present. That is, we wish to compute $\mathbf{P}(\mathbf{X}_k \mid \mathbf{e}_{1:t})$ for some k such that $0 \leq k < t$. In the umbrella example, it might mean computing the probability that it rained last Wednesday, given all the observations of the umbrella carrier made up to today. Smoothing provides a better estimate of the state at time k than was available at that time, because it incorporates more evidence.[3]
- **Most likely explanation:** Given a sequence of observations, we might wish to find the sequence of states that is most likely to have generated those observations. That is, we wish to compute $\operatorname{argmax}_{\mathbf{x}_{1:t}} P(\mathbf{x}_{1:t} \mid \mathbf{e}_{1:t})$. For example, if the umbrella appears on each of the first three days and is absent on the fourth, then the most likely explanation is that it rained on the first three days and did not rain on the fourth. Algorithms for this task are useful in many applications, including speech recognition—where the aim is to find the most likely sequence of words, given a series of sounds—and the reconstruction of bit strings transmitted over a noisy channel.

In addition to these inference tasks, we also have

- **Learning:** The transition and sensor models, if not yet known, can be learned from observations. Just as with static Bayesian networks, dynamic Bayes net learning can be done as a by-product of inference. Inference provides an estimate of what transitions actually occurred and of what states generated the sensor readings, and these estimates can be used to learn the models. The learning process can operate via an iterative update algorithm called expectation-maximization or EM, or it can result from Bayesian updating of the model parameters given the evidence. See Chapter 21 for more details.

The remainder of this section describes generic algorithms for the four inference tasks, independent of the particular kind of model employed. Improvements specific to each model are described in subsequent sections.

14.2.1 Filtering and prediction

As we pointed out in Section 7.7.3, a useful filtering algorithm needs to maintain a current state estimate and update it, rather than going back over the entire history of percepts for each update. (Otherwise, the cost of each update increases as time goes by.) In other words, given the result of filtering up to time t, the agent needs to compute the result for $t + 1$ from the new evidence $\mathbf{e}_{t+1}$. So we have

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

for some function *f*. This process is called **recursive estimation**. (See also Sections 4.4 and 7.7.3.) We can view the calculation as being composed of two parts: first, the current state distribution is projected forward from t to *t* +1; then it is updated using the new evidence $\mathbf{e}_{t+1}$. This two-part process emerges quite simply when the formula is rearranged:

$$\begin{aligned}
\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t+1}) &= \mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t},\ \mathbf{e}_{t+1}) \quad \text{(dividing up the evidence)} \\
&= \alpha\, \mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1}, \mathbf{e}_{1:t})\, \mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t}) \quad \text{(using Bayes's rule, given } \mathbf{e}_{1:t}\text{)} \\
&= \alpha\ \underbrace{\mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1})}_{\text{update}}\ \underbrace{\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t})}_{\text{prediction}} \quad \text{(by the sensor Markov assumption).} \quad (14.4)
\end{aligned}$$

Here and throughout this chapter, α is a normalizing constant used to make probabilities sum up to 1. Now we plug in an expression for the one-step prediction $\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{e}_{1:t})$, obtained by conditioning on the current state $\mathbf{X}_t$. The resulting equation for the new state estimate is the central result in this chapter:

$$\begin{aligned}
\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t+1}) &= \alpha\, \mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1}) \sum_{\mathbf{x}_t} \mathbf{P}(\mathbf{X}_{t+1}|x_t, \mathbf{e}_{1:t}) P(\mathbf{x}_t|\mathbf{e}_{1:t}) \\
&= \alpha\, \underbrace{\mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1})}_{\text{sensor model}} \sum_{\mathbf{x}_t} \underbrace{\mathbf{P}(\mathbf{X}_{t+1}|\, x_t)}_{\text{transition model}} \underbrace{P(x_t|\mathrm{e}_{1:t})}_{\text{recursion}} \quad \text{(Markov assumption).} \quad (14.5)
\end{aligned}$$

In this expression, all the terms come either from the model or from the previous state estimate. Hence, we have the desired recursive formulation. We can think of the filtered estimate $\mathbf{P}(\mathbf{X}_t \mid \mathbf{e}_{1:t})$ as a “message” $\mathbf{f}_{1:t}$ that is propagated forward along the sequence, modified by each transition and updated by each new observation. The process is given by

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

where FORWARD implements the update described in Equation (14.5) and the process begins with $\mathbf{f}_{1:0} = \mathbf{P}(\mathbf{X}_0)$. When all the state variables are discrete, the time for each update is constant (i.e., independent of *t*), and the space required is also constant. (The constants depend, of course, on the size of the state space and the specific type of the temporal model in question.) *The time and space requirements for updating must be constant if a finite agent is to keep track of the current state distribution indefinitely.*

Let us illustrate the filtering process for two steps in the basic umbrella example (Figure 14.2). That is, we will compute $\mathbf{P}(R_2 \mid u_{1:2})$ as follows:

- On day 0, we have no observations, only the security guard’s prior beliefs; let’s assume that consists of $\mathbf{P}(R_0)$ = (0.5,0.5).
- On day 1, the umbrella appears, so $U_1 = true$. The prediction from $t = 0$ to $t = 1$ is

  $$\begin{aligned}
  \mathbf{P}(R_1) &= \sum_{r_0} \mathbf{P}(R_1|\, r_0) P(r_0) \\
  &= \langle 0.7{,}0.3 \times 0.5 + \langle 0.3{,}0.7\rangle \times 0.5 = \langle 0.5{,}0.5\rangle.
  \end{aligned}$$

  Then the update step simply multiplies by the probability of the evidence for $t = 1$ and normalizes, as shown in Equation (14.4):

$$\begin{aligned}\mathbf{P}(R_1|\ u_1\ ) &= \alpha\,\mathbf{P}(u_1|R_1)\mathbf{P}(R_1) = \alpha\,\langle 0.9{,}0.2\rangle\,\langle 0.5{,}0.5\rangle.\\ &= \alpha\,\langle 0.45{,}0.1\rangle\ \approx\ \langle 0.818{,}0.182\rangle.\end{aligned}$$

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

$$\begin{aligned}\mathbf{P}(R_2|\ u_1\ ) &= \sum_{r_1}\mathbf{P}(R_2|r_1)P(r_1|u_1)\\ &= \langle 0.7{,}0.3\rangle\ \times 0.818 + \langle 0.3{,}0.7\rangle\ \times 0.182\ \approx\ \langle 0.627{,}0.373\rangle,\end{aligned}$$

and updating it with the evidence for t = 2 gives

$$\begin{aligned}\mathbf{P}(R_2|\ u_1, u_2\ ) &= \alpha\,\mathbf{P}(u_2|R_2)\mathbf{P}(R_2|u_1) = \alpha\,\langle 0.9{,}0.2\rangle\,\langle 0.627{,}0.373\rangle\\ &= \alpha\,\langle 0.565{,}0.075\rangle\ \approx\ \langle 0.883{,}0.117\rangle.\end{aligned}$$

Intuitively, the probability of rain increases from day 1 to day 2 because rain persists. Exercise 14.CONV(a) asks you to investigate this tendency further.

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

$$\mathbf{P}(\mathbf{X}_{t+k+1}\,|\,\mathbf{e}_{1:t}) = \sum_{\mathbf{x}_{t+k}}\underbrace{\mathbf{P}(\mathbf{X}_{t+k+1}|\mathbf{x}_{t+k})}_{\text{transition model}}\underbrace{P(\mathbf{x}_{t+k}|\mathbf{e}_{1:t}}_{\text{recursion}}. \qquad (14.6)$$

Naturally, this computation involves only the transition model and not the sensor model.

It is interesting to consider what happens as we try to predict further and further into the future. As Exercise 14.CONV(b) shows, the predicted distribution for rain converges to a fixed point ⟨0.5,0.5⟩, after which it remains constant for all time.[4] This is the **stationary distribution** of the Markov process defined by the transition model. (See also page 462.) A great deal is known about the properties of such distributions and about the **mixing time**—roughly, the time taken to reach the fixed point. In practical terms, this dooms to failure any attempt to predict the *actual* state for a number of steps that is more than a small fraction of the mixing time, unless the stationary distribution itself is strongly peaked in a small area of the state space. The more uncertainty there is in the transition model, the shorter will be the mixing time and the more the future is obscured.

In addition to filtering and prediction, we can use a forward recursion to compute the **likelihood** of the evidence sequence, $P(\mathbf{e}_{1:t})$. This is a useful quantity if we want to compare different temporal models that might have produced the same evidence sequence (e.g., two different models for the persistence of rain). For this recursion, we use a likelihood message $\ell_{1:t}(\mathbf{X}_t) = \mathbf{P}(\mathbf{X}_t, \mathbf{e}_{1:t})$. It is easy to show (Exercise 14.LIKL) that the message calculation is identical to that for filtering:

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

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

$$L_{1:t} = P(\mathbf{e}_{1:t}) = \sum_{\mathbf{x}_t}\ell_{1:t}(\mathbf{X}_t). \qquad (14.7)$$

Notice that the likelihood message represents the probabilities of longer and longer evidence sequences as time goes by and so becomes numerically smaller and smaller, leading to underflow problems with floating-point arithmetic. This is an important problem in practice, but we shall not go into solutions here.

14.2.2 Smoothing

As we said earlier, smoothing is the process of computing the distribution over past states given evidence up to the present—that is, $\mathbf{P}(\mathbf{X}_k|\mathbf{e}_{1:t})$ for $0 \le k < t$. (See Figure 14.3.) In anticipation of another recursive message-passing approach, we can split the computation into two parts—the evidence up to k and the evidence from $k + 1$ to t,

$$\begin{aligned}
\mathrm{P}(\mathbf{X}_k \,|\, \mathbf{e}_{1:t}) &= \mathbf{P}(\mathbf{X}_k|\mathbf{e}_{1:k},\mathbf{e}_{k+1:t}) \\
&= \alpha\, \mathbf{P}(\mathbf{X}_k|\, \mathbf{e}_{1:k})\mathbf{P}(\mathbf{e}_{k+1:t}|\mathbf{X}_k, \mathbf{e}_{1:k}) \quad \text{(using Bayes's rule, given } \mathbf{e}_{1:k}) \\
&= \alpha\, \mathbf{P}(\mathbf{X}_k|\, \mathbf{e}_{1:k})\mathbf{P}(\mathbf{e}_{k+1:t}|\mathbf{X}_k) \quad \text{(using conditional independence)} \\
&= \alpha\, \mathbf{f}_{1:k} \times \mathbf{b}_{k+1:t}\,. \qquad (14.8)
\end{aligned}$$

where "×" represents pointwise multiplication of vectors. Here we have defined a "back-ward" message $\mathbf{b}_{k+1:t} = \mathbf{P}(\mathbf{e}_{k+1:t} \mid \mathbf{X}_k)$, analogous to the forward message $\mathbf{f}_{1:k}$. The forward message $\mathbf{f}_{1:k}$ can be computed by filtering forward from 1 to k, as given by Equation (14.5). It turns out that the backward message $\mathbf{b}_{k+1:t}$ can be computed by a recursive process that runs *backward* from t:

$$\begin{aligned}
\mathrm{P}(\mathbf{e}_{k+1:t} \,|\, \mathbf{X}_k) &= \sum_{\mathbf{x}_{k+1}} \mathbf{P}(\mathbf{e}_{k+1:t} \,|\, \mathbf{X}_k,\mathbf{x}_{k+1})\mathbf{P}(\mathbf{x}_{k+1} \,|\mathbf{X}_k) \quad \text{(conditioning on } \mathbf{X}_{k+1}) \\
&= \sum_{\mathbf{x}_{k+1}} P(\mathbf{e}_{k+1:t} \,|\, \mathbf{X}_{k+1})\mathbf{P}(\mathbf{x}_{k+1} \,|\mathbf{X}_k) \quad \text{(by conditional independence)} \\
&= \sum_{\mathbf{x}_{k+1}} P(\mathbf{e}_{k+1},\, \mathbf{e}_{k+2:t}|\, \mathbf{x}_{k+1})\mathbf{P}(\mathbf{x}_{k+1} \,|\mathbf{X}_k) \\
&= \sum_{\mathbf{x}_{k+1}} \underbrace{P(\mathbf{e}_{k+1}|\mathbf{x}_{k+1})}_{\text{sensor model}}\underbrace{P(\mathbf{e}_{k+2:t}|\mathbf{x}_{k+1})}_{\text{recursion}}\underbrace{\mathbf{P}(\mathbf{x}_{k+1}|\mathbf{x}_k)}_{\text{transition model}}, \qquad (14.9)
\end{aligned}$$

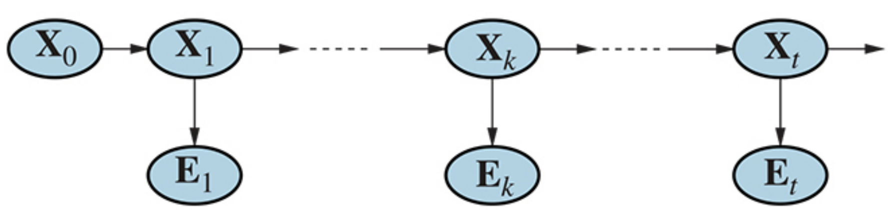


Figure 14.3 Smoothing computes $\mathbf{P}(\mathbf{X}_k \mid \mathbf{e}_{1:t})$, the posterior distribution of the state at some past time k given a complete sequence of observations from 1 to t.

where the last step follows by the conditional independence of $\mathbf{e}_{k+1}$ and $\mathbf{e}_{k+2:t}$, given $\mathbf{x}_{k+1}$. In this expression, all the terms come either from the model or from the previous backward message. Hence, we have the desired recursive formulation. In message form, we have

$$\mathbf{b}_{k+1:t} = \text{BACKWARD}\,(\mathbf{b}_{k+2:t}, \mathbf{e}_{k+1}),$$

where BACKWARD implements the update described in Equation (14.9). As with the forward recursion, the time and space needed for each update are constant and thus independent of t.

We can now see that the two terms in Equation (14.8) can both be computed by recursions through time, one running forward from 1 to k and using the filtering equation (14.5) and the other running backward from t to $k + 1$ and using Equation (14.9).

For the initialization of the backward phase, we have $\mathbf{b}_{t+1:t} = \mathbf{P}(\mathbf{e}_{t+1:t} \mid \mathbf{X}_t) = \mathbf{P}(\ \mid \mathbf{X}_t\ ) = \mathbf{1}$, where $\mathbf{1}$ is a vector of 1s. The reason for this is that $\mathbf{e}_{t+1:t}$ is an empty sequence, so the probability of observing it is 1.

Let us now apply this algorithm to the umbrella example, computing the smoothed estimate for the probability of rain at time $k = 1$, given the umbrella observations on days 1 and 2. From Equation (14.8), this is given by

$$\mathbf{P}(R_1|u_1,u_2) \ = \ \alpha\mathbf{P}(R_1|u_1)\mathbf{P}(u_2|R_1). \qquad (14.10)$$

The first term we already know to be $\langle .818,.182\rangle$, from the forward filtering process described earlier. The second term can be computed by applying the backward recursion in Equation (14.9):

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

Plugging this into Equation (14.10), we find that the smoothed estimate for rain on day 1 is

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

Thus, the smoothed estimate for rain on day 1 is *higher* than the filtered estimate (0.818) in this case. This is because the umbrella on day 2 makes it more likely to have rained on day 2; in turn, because rain tends to persist, that makes it more likely to have rained on day 1.

Both the forward and backward recursions take a constant amount of time per step; hence, the time complexity of smoothing with respect to evidence $\mathbf{e}_{1:t}$ is $O(t)$. This is the complexity for smoothing at a particular time step k. If we want to smooth the whole sequence, one obvious method is simply to run the whole smoothing process once for each time step to be smoothed. This results in a time complexity of $O(t^2)$.

A better approach uses a simple application of dynamic programming to reduce the complexity to $O(t)$. A clue appears in the preceding analysis of the umbrella example, where we were able to reuse the results of the forward-filtering phase. The key to the linear-time algorithm is to *record the results* of forward filtering over the whole sequence. Then we run the backward recursion from t down to 1, computing the smoothed estimate at each step k from the computed backward message $\mathbf{b}_{k+1:t}$ and the stored forward message $\mathbf{f}_{1:k}$. The algorithm, aptly called the **forward-backward algorithm,** is shown in Figure 14.4.

```
function FORWARD-BACKWARD(ev, prior) returns a vector of probability distributions
  inputs: ev, a vector of evidence values for steps 1,...,t
          prior, the prior distribution on the initial state, P(X_0)
  local variables: fv, a vector of forward messages for steps 0,...,t
                   b, a representation of the backward message, initially all 1s
                   sv, a vector of smoothed estimates for steps 1,...,t

  fv[0] ← prior
  for i = 1 to t do
      fv[i] ← FORWARD(fv[i − 1], ev[i])
  for i = t down to 1 do
      sv[i] ← NORMALIZE(fv[i] × b)
      b ← BACKWARD(b, ev[i])
  return sv
```

Figure 14.4 The forward–backward algorithm for smoothing: computing posterior probabilities of a sequence of states given a sequence of observations. The FORWARD and BACKWARD operators are defined by Equations (14.5) and (14.9), respectively.

The alert reader will have spotted that the Bayesian network structure shown in Figure 14.3 is a *polytree* as defined on page 451. This means that a straightforward application of the clustering algorithm also yields a linear-time algorithm that computes smoothed estimates for the entire sequence. It is now understood that the forward-backward algorithm is in fact a special case of the polytree propagation algorithm used with clustering methods (although the two were developed independently).

The forward-backward algorithm forms the computational backbone for many applications that deal with sequences of noisy observations. As described so far, it has two practical drawbacks. The first is that its space complexity can be too high when the state space is large and the sequences are long. It uses $O(|\mathbf{f} \mid t)$ space where $|\mathbf{f}|$ is the size of the representation of the forward message. The space requirement can be reduced to $O(\mid \mathbf{f} \mid \log t)$ with a concomitant increase in the time complexity by a factor of $\log_t$, as shown in Exercise 14.ISLE. In some cases (see Section 14.3), a constant-space algorithm can be used.

The second drawback of the basic algorithm is that it needs to be modified to work in an *online* setting where smoothed estimates must be computed for earlier time slices as new observations are continuously added to the end of the sequence. The most common requirement is for **fixed-lag smoothing**, which requires computing the smoothed estimate $\mathbf{P}(\mathbf{X}_{t-d} \mid \mathbf{e}_{1:t})$ for fixed d. That is, smoothing is done for the time slice d steps behind the current time t; as t increases, the smoothing has to keep up. Obviously, we can run the forward–backward algorithm over the d-step "window" as each new observation is added, but this seems inefficient. In Section 14.3, we will see that fixed-lag smoothing can, in some cases, be done in constant time per update, independent of the lag d.

14.2.3 Finding the most likely sequence

Suppose that [*true, true, false, true, true*] is the observed umbrella sequence for the security guard's first five days on the job. What weather sequence is most likely to explain this? Does the absence of the umbrella on day 3 mean that it wasn't raining, or did the director forget to bring it? If it didn't rain on day 3, perhaps (because weather tends to persist) it didn't rain on day 4 either, but the director brought the umbrella just in case. In all, there are 2^5

possible weather sequences we could pick. Is there a way to find the most likely one, short of enumerating all of them and calculating their likelihoods?

We could try this linear-time procedure: use smoothing to find the posterior distribution for the weather at each time step; then construct the sequence, using at each step the weather that is most likely according to the posterior. Such an approach should set off alarm bells in the reader's head, because the posterior distributions computed by smoothing are distributions over *single* time steps, whereas to find the most likely *sequence* we must consider *joint* probabilities over all the time steps. The results can in fact be quite different. (See Exercise 14.VITE.)

There ***is*** a linear-time algorithm for finding the most likely sequence, but it requires more thought. It relies on the same Markov property that yielded efficient algorithms for filtering and smoothing. The idea is to view each sequence as a *path* through a graph whose nodes are the possible *states* at each time step. Such a graph is shown for the umbrella world in Figure 14.5(a). Now consider the task of finding the most likely path through this graph, where the likelihood of any path is the product of the transition probabilities along the path and the probabilities of the given observations at each state.

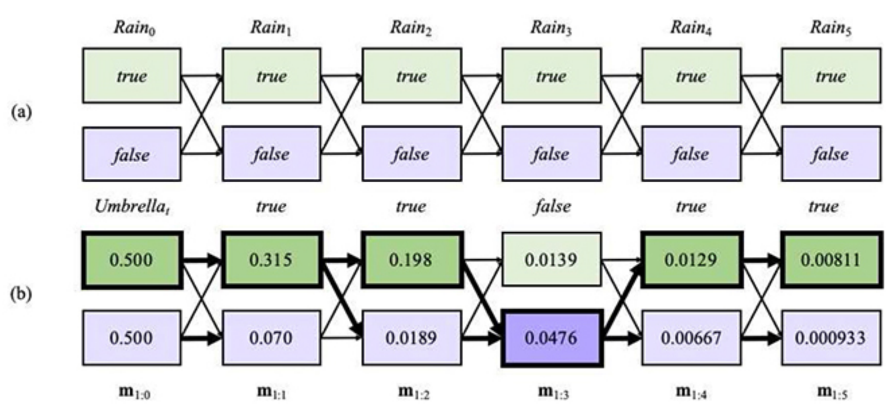


Figure 14.5 (a) Possible state sequences for Rain_t can be viewed as paths through a graph of the possible states at each time step. (States are shown as rectangles to avoid confusion with nodes in a Bayes net.) (b) Operation of the Viterbi algorithm for the umbrella observation sequence [*true*, *true*, *false*, *true*, *true*], where the evidence starts at time 1. For each *t*, we have shown the values of the message $\mathbf{m}_{1:t}$, which gives the probability of the best sequence reaching each state at time *t*. Also, for each state, the bold arrow leading into it indicates its best predecessor as measured by the product of the preceding sequence probability and the transition probability. Following the bold arrows back from the most likely state in $\mathbf{m}_{1:5}$ gives the most likely sequence, shown by the bold outlines and darker shading.

Let's focus in particular on paths that reach the state $Rain_5 = true$. Because of the Markov property, it follows that the most likely path to the state $Rain_5 = true$ consists of the most likely path to *some* state at time 4 followed by a transition to $Rain_5 = true$; and the state at time 4 that will become part of the path to $Rain_5 = true$ is whichever maximizes the likelihood of that path. In other words, *there is a recursive relationship between most likely paths to each state* $\mathbf{x}_{t+1}$ *and most likely paths to each state* $\mathbf{x}_t$.

We can use this property directly to construct a recursive algorithm for computing the most likely path given the evidence. We will use a recursively computed message $m_{1:t}$, like the forward message $\mathbf{f}_{1:t}$ in the filtering algorithm. The message is defined as follows:[5]

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

To obtain the recursive relationship between $\mathbf{m}_{1:t+1}$ and $\mathbf{m}_{1:t}$, we can repeat more or less the same steps that we used for Equation (14.5):

$$\begin{aligned}\mathbf{m}_{1:t+1} &= \max_{\mathbf{x}_{1:t}} \mathbf{P}(\mathbf{x}_{1:t},\ \mathbf{X}_{t+1}, \mathbf{e}_{1:t+1}) = \max_{\mathbf{x}_{1:t}} \mathbf{P}(\mathbf{x}_{1:t},\ \mathbf{X}_{t+1}, \mathbf{e}_{1:t}, e_{t+1}) \\ &= \max_{\mathbf{x}_{1:t}} \mathbf{P}(\mathbf{e}_{t+1}\ \ \mathbf{x}_{1:t}, \mathbf{X}_{t+1}, \mathbf{e}_{1:t})\mathbf{P}(\mathbf{x}_{1:t},\ \mathbf{X}_{t+1}, \mathbf{e}_{1:t}) \\ &= \mathbf{P}(e_{t+1}\ \ \mathbf{X}_{t+1}) \max_{\mathbf{x}_{1:t}}\ \mathbf{P}(\mathbf{X}_{t+1},\ \ \mathbf{x}_t) P(\mathbf{x}_{1:t}, \mathbf{e}_{1:t}) \\ &= \mathbf{P}(e_{t+1}\ \ \mathbf{X}_{t+1}) \max_{\mathbf{x}_t}\ \mathbf{P}(\mathbf{X}_{t+1},\ \ \mathbf{x}_t) \max_{\mathbf{x}_{1:t..1}} P(\mathbf{x}_{1:t-1}, \mathbf{x}_t, \mathbf{e}_{1:t}) \qquad (14.11)\end{aligned}$$

where the final term $\max_{\mathbf{x}_{1:t-1}} P(\mathbf{x}_{1:t-1}, \mathbf{x}_t, \mathbf{e}_{1:t})$ is exactly the entry for the particular state $\mathbf{x}_t$ in the message vector $\mathbf{m}_{1:t}$. Equation (14.11) is essentially identical to the filtering equation (14.5) except that the summation over $\mathbf{x}_t$ in Equation (14.5) is replaced by the maximization over $\mathbf{x}_t$ in Equation (14.11), and there is no normalization constant a in Equation (14.11). Thus, the algorithm for computing the most likely sequence is similar to filtering: it starts at time 0 with the prior $\mathbf{m}_{1:0} = \mathbf{P}(\mathbf{X}_0)$ and then runs forward along the sequence, computing the $\mathbf{m}$ message at each time step using Equation (14.11). The progress of this computation is shown in Figure 14.5(b).

At the end of the observation sequence, $\mathbf{m}_{1:t}$ will contain the probability for the most likely sequence reaching *each* of the final states. One can thus easily select the final state of the most likely sequence overall (the state outlined in bold at step 5). In order to identify the actual sequence, as opposed to just computing its probability, the algorithm will also need to record, for each state, the best state that leads to it; these are indicated by the bold arrows in Figure 14.5(b). The optimal sequence is identified by following these bold arrows backwards from the best final state.

The algorithm we have just described is called the **Viterbi algorithm**, after its inventor, Andrew Viterbi. Like the filtering algorithm, its time complexity is linear in t , the length of the sequence. Unlike filtering, which uses constant space, its space requirement is also linear in t. This is because the Viterbi algorithm needs to keep the pointers that identify the best sequence leading to each state.

One final practical point: numerical underflow is a significant issue for the Viterbi algorithm. In Figure 14.5(b), the probabilities are getting smaller and smaller—and this is just a toy example. Real applications in DNA analysis or message decoding may have thousands or millions of steps. One possible solution is simply to normalize $\mathbf{m}$ at each step; this rescaling does not affect correctness because $\max(cx, cy) = c \cdot \max(x,y)$. A second solution is to use log probabilities everywhere and replace multiplication by addition. Again, correctness is unaffected because the log function is monotonic, so $\max(\log x, \log y) = \log \max(x, y)$.

14.3 Hidden Markov Models

The preceding section developed algorithms for temporal probabilistic reasoning using a general framework that was independent of the specific form of the transition and sensor models and independent of the nature of the state and evidence variables. In this and the next two sections, we discuss more concrete models and applications that illustrate the power of the basic algorithms and in some cases allow further improvements.

We begin with the **hidden Markov model**, or **HMM.** An HMM is a temporal probabilistic model in which the state of the process is described by a *single, discrete* random variable. The possible values of the variable are the possible states of the world. The umbrella example described in the preceding section is therefore an HMM, since it has just one state variable: $Rain_t$. What happens if you have a model with two or more state variables? You can still fit it into the HMM framework by combining the variables into a single "megavariable" whose values are all possible tuples of values of the individual state variables. We will see that the restricted structure of HMMs allows for a simple and elegant matrix implementation of all the basic algorithms.[6]

Although HMMs require the *state* to be a single, discrete variable, there is no corresponding restriction on the *evidence* variables. This is because the evidence variables are always observed, which means that there is no need to keep track of any distribution over their values. (If a variable is not observed, it can simply be dropped from the model for that time step.) There can be many evidence variables, both discrete and continuous.

14.3.1 Simplified matrix algorithms

With a single, discrete state variable X_t, we can give concrete form to the representations of the transition model, the sensor model, and the forward and backward messages. Let the state variable X_t have values denoted by integers $1,\ldots,S$, where S is the number of possible states. The transition model $\mathbf{P}(X_t \mid X_{t-1})$ becomes an $S \times S$ matrix $\mathbf{T}$, where

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

That is, $\mathbf{T}_{ij}$ is the probability of a transition from state i to state j. For example, if we number the states $Rain = true$ and $Rain = false$ as 1 and 2, respectively, then the transition matrix for the umbrella world defined in Figure 14.2 is

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

We also put the sensor model in matrix form. In this case, because the value of the evidence variable E_t is known at time t (call it e_t), we need only specify, for each state, how likely it is that the state causes e_t to appear: we need $P(e_t \mid X_t = i)$ for each state i. For mathematical convenience we place these values into an $S \times S$ diagonal **observation matrix**, $\mathbf{O}_t$, one for each time step. The i th diagonal entry of $\mathbf{O}_t$ is $P(e_t \mid X_t = i)$ and the other entries are 0. For example, on day 1 in the umbrella world of Figure 14.5, $U_1 = true$, and on day 3, $U_3 = false$, so we have

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

Now, if we use column vectors to represent the forward and backward messages, all the computations become simple matrix-vector operations. The forward equation (14.5) becomes

$$\mathbf{f}_{1:t+1} = \alpha\, \mathbf{O}_{t+1} \mathbf{T}^{\top} \mathbf{f}_{1:t} \qquad (14.12)$$

and the backward equation (14.9) becomes

$$\mathbf{b}_{k+1:t} = \mathbf{T}\mathbf{O}_{k+1}\,\mathbf{b}_{k+2:t}. \tag{14.13}$$

From these equations, we can see that the time complexity of the forward–backward algorithm (Figure 14.4) applied to a sequence of length t is $O(S^2t)$, because each step requires multiplying an S-element vector by an $S \times S$ matrix. The space requirement is $O(St)$, because the forward pass stores t vectors of size S.

Besides providing an elegant description of the filtering and smoothing algorithms for HMMs, the matrix formulation reveals opportunities for improved algorithms. The first is a simple variation on the forward-backward algorithm that allows smoothing to be carried out in *constant* space, independently of the length of the sequence. The idea is that smoothing for any particular time slice k requires the simultaneous presence of both the forward and backward messages, $\mathbf{f}_{1:k}$ and $\mathbf{b}_{k+1:t}$, according to Equation (14.8). The forward-backward algorithm achieves this by storing the **f**s computed on the forward pass so that they are available during the backward pass. Another way to achieve this is with a single pass that propagates both **f** and **b** in the same direction. For example, the "forward" message **f** can be propagated backward if we manipulate Equation (14.12) to work in the other direction:

$$\mathbf{f}_{1:t} = \alpha'(\mathbf{T}^{\top})^{-1}\mathbf{O}_{t+1}^{-1}\mathbf{f}_{1:t+1}.$$

The modified smoothing algorithm works by first running the standard forward pass to compute $\mathbf{f}_{t:t}$ (forgetting all the intermediate results) and then running the backward pass for both **b** and **f** together, using them to compute the smoothed estimate at each step. Since only one copy of each message is needed, the storage requirements are constant (i.e., independent of t, the length of the sequence). There are two significant restrictions on this algorithm: it requires that the transition matrix be invertible and that the sensor model have no zeroes—that is, that every observation be possible in every state.

A second area in which the matrix formulation reveals an improvement is in *online* smoothing with a fixed lag. The fact that smoothing can be done in constant space suggests that there should exist an efficient recursive algorithm for online smoothing—that is, an algorithm whose time complexity is independent of the length of the lag. Let us suppose that the lag is d; that is, we are smoothing at time slice $t - d$, where the current time is t. By Equation (14.8), we need to compute

$$\alpha\,\mathbf{f}_{1:t-d} \times \mathbf{b}_{t-d+1:t}$$

for slice $t - d$. Then, when a new observation arrives, we need to compute

$$\alpha\,\mathbf{f}_{1:t-d+1} \times \mathbf{b}_{t-d+2:t+1}$$

for slice $t - d + 1$. How can this be done incrementally? First, we can compute $\mathbf{f}_{1:t-d+1}$ from $\mathbf{f}_{1:t-d}$, using the standard filtering process, Equation (14.5).

Computing the backward message incrementally is trickier, because there is no simple relationship between the old backward message $\mathbf{b}_{t-d+1:t}$ and the new backward message $\mathbf{b}_{t-d+2:t+1}$. Instead, we will examine the relationship between the old backward message $\mathbf{b}_{t-d+1:t}$ and the backward message at the front of the sequence, $\mathbf{b}_{t+1:t}$. To do this, we apply Equation (14.13) d times to get

$$\mathbf{b}_{t-d+1:t} = \left(\prod_{i=t-d+1}^{t} \mathbf{T}\mathbf{O}_i\right)\mathbf{b}_{t+1:t} = \mathbf{B}_{t-d+1:t}\mathbf{1}, \tag{14.14}$$

where the matrix $\mathbf{B}_{t-d+1:t}$ is the product of the sequence of **T** and **O** matrices and **1** is a vector of 1s. **B** can be thought of as a "transformation operator" that transforms a later backward message into an earlier one. A similar equation holds for the new backward messages *after* the next observation arrives:

$$\mathbf{b}_{t-d+2:t+1} = \left(\prod_{i=t-d+2}^{t+1} \mathbf{TO}_i\right)\mathbf{b}_{t+2:t+1} = \mathbf{B}_{t-d+2:t+1}\mathbf{1}. \tag{14.15}$$

Examining the product expressions in Equations (14.14) and (14.15), we see that they have a simple relationship: to get the second product, "divide" the first product by the first element $\mathbf{TO}_{t-d+1}$, and multiply by the new last element $\mathbf{TO}_{t+1}$. In matrix language, then, there is a simple relationship between the old and new **B** matrices:

$$\mathbf{B}_{t-d+2:t+1} = \mathbf{O}_{t-d+1}^{-1}\,\mathbf{T}^{-1}\,\mathbf{B}_{t-d+1:t}\,\mathbf{TO}_{t+1}. \tag{14.16}$$

This equation provides an incremental update for the **B** matrix, which in turn (through Equation (14.15)) allows us to compute the new backward message $\mathbf{b}_{t-d+2:t+1}$. The complete algorithm, which requires storing and updating **f** and **B**, is shown in Figure 14.6.

---

function FIXED-LAG-SMOOTHING(e_t, *hmm*, d) **returns** a distribution over $\mathbf{X}_{t-d}$
  inputs: e_t, the current evidence for time step t
    hmm, a hidden Markov model with $S \times S$ transition matrix $\mathbf{T}$
    d, the length of the lag for smoothing
  persistent: t, the current time, initially 1
    f, the forward message $\mathbf{P}(X_t \mid e_{1:t})$, initially *hmm*.PRIOR
    B, the d-step backward transformation matrix, initially the identity matrix
    $e_{t-d:t}$, double-ended list of evidence from $t-d$ to t, initially empty
  local variables: $\mathbf{O}_{t-d}, \mathbf{O}_t$, diagonal matrices containing the sensor model information

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

Figure 14.6 An algorithm for smoothing with a fixed time lag of d steps, implemented as an online algorithm that outputs the new smoothed estimate given the observation for a new time step. Notice that the final output NORMALIZE($\mathbf{f} \times \mathbf{B1}$) is just $\alpha\, \mathbf{f} \times \mathbf{b}$, by Equation (14.14).

14.3.2 Hidden Markov model example: Localization

On page 151, we introduced a simple form of the **localization** problem for the vacuum world. In that version, the robot had a single nondeterministic *Move* action and its sensors reported perfectly whether or not obstacles lay immediately to the north, south, east, and west; the robot's belief state was the set of possible locations it could be in.

Here we make the problem slightly more realistic by allowing for noise in the sensors, and formalizing the idea that the robot moves randomly—it is equally likely to move to any adjacent empty square. The state variable

X_t represents the location of the robot on the discrete grid; the domain of this variable is the set of empty squares, which we will label by the integers$\{1,...,S\}$. Let NEIGHBORS(i) be the set of empty squares that are adjacent to i and let $N(i)$ be the size of that set. Then the transition model for the *Move* action says that the robot is equally likely to end up at any neighboring square:

$$P(X_{t+1} = j \,\big|\, X_t = i) = \mathbf{T}_{ij} = \begin{cases} 1/N(i) \text{ if } j \in \text{NEIGHBORS}(i) \\ 0 \text{ otherwise.} \end{cases}$$

We don't know where the robot starts, so we will assume a uniform distribution over all the squares; that is, $P(X_0 = i) = 1/S$. For the particular environment we consider (Figure 14.7), $S = 42$ and the transition matrix $\mathbf{T}$ has $42 \times 42 = 1764$ entries.

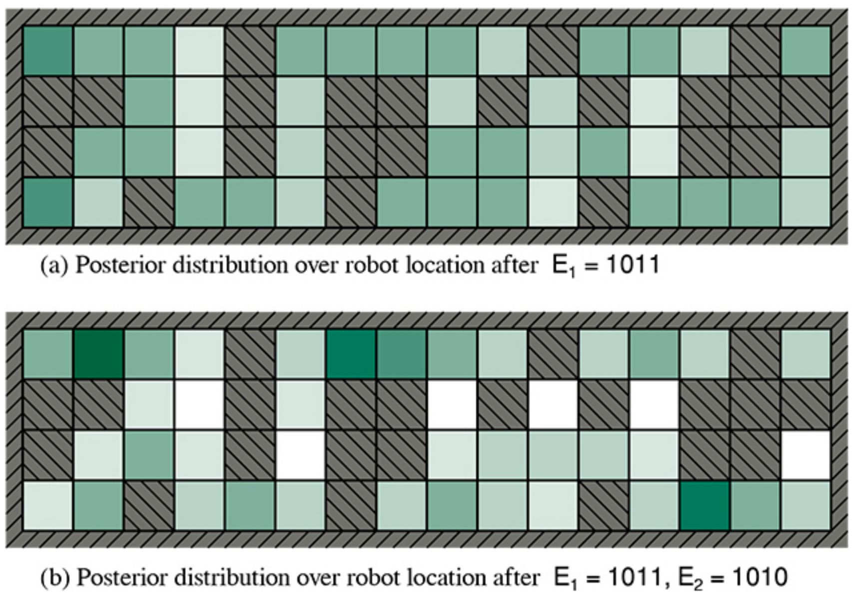


Figure 14.7 Posterior distribution over robot location: (a) after one observation $E_1 = 1011$ (i.e., obstacles to the north, south, and west); (b) after a random move to an adjacent location and a second observation $E_2 = 1010$ (i.e., obstacles to the north and south). The darkness of each square corresponds to the probability that the robot is at that location. The sensor error rate for each bit is $\epsilon = 0.2$.

The sensor variable E_t has 16 possible values, each a four-bit sequence giving the presence or absence of an obstacle in each of the compass directions NESW. For example, 1010 means that the north and south sensors report an obstacle and the east and west do not. Suppose that each sensor's error rate is ϵ and that errors occur independently for the four sensor directions. In that case, the probability of getting all four bits right is $(1-\epsilon)^4$ and the probability of getting them all wrong is ϵ^4. Furthermore, if d_{it} is the discrepancy—the number of bits that are different—between the true values for square i and the actual reading e_t, then the probability that a robot in square i would receive a sensor reading e_t is

$$P(E_t = e_t \mid X_t = i) = (\mathbf{O}_t)_{ii} = (1-\epsilon)^{4-d_{it}}\, \epsilon^{d_{it}}.$$

For example, the probability that a square with obstacles to the north and south would produce a sensor reading 1110 is $(1-\epsilon)^3\epsilon^1$.

Given the matrices $\mathbf{T}$ and $\mathbf{O}_t$, the robot can use Equation (14.12) to compute the posterior distribution over locations—that is, to work out where it is. Figure 14.7 shows the distributions $\mathbf{P}(X_1 \mid E_1 = 1011)$ and $\mathbf{P}(X_2 \mid E_1 = 1011, E_2 = 1010)$. This is the same maze we saw before in Figure 4.18 (page 152), but there we used logical filtering to find the locations that were *possible*, assuming perfect sensing. Those same locations are still the most *likely* with noisy sensing, but now *every* location has some nonzero probability because any location could produce any sensor values.

In addition to filtering to estimate its current location, the robot can use smoothing (Equation (14.13)) to work out where it was at any given past time—for example, where it began at time 0—and it can use the Viterbi algorithm to work out the most likely path it has taken to get where it is now. Figure 14.8 shows the localization error and Viterbi path error for various values of the per-bit sensor error rate ϵ. Even when ϵ is 0.20—which means that the overall sensor reading is wrong 59% of the time—the robot is usually able to work out its location to within two squares after 20 observations. This is because of the algorithm's ability to integrate evidence over time and to take into account the probabilistic constraints imposed on the location sequence by the transition model. When ϵ is 0.10 or less, the robot needs only a few observations to work out where it is and to track its position accurately. When ϵ is 0.40, both the localization error and the Viterbi path error remain large; in other words, the robot is lost. This is because a sensor with an error probability of 0.40 provides too little information to counteract the loss of information about the robot's position that comes from the unpredictable random motion.

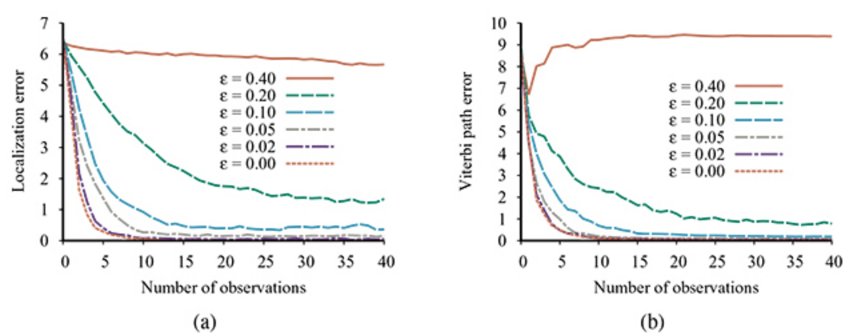


Figure 14.8 Performance of HMM localization as a function of the length of the observation sequence for various different values of the sensor error probability ϵ; data averaged over 400 runs. (a) The localization error, defined as the Manhattan distance from the true location. (b) The Viterbi path error, defined as the average Manhattan distance of states on the Viterbi path from corresponding states on the true path.

The state variable for the example we have considered in this section is a physical location in the world. Other problems can, of course, include other aspects of the world. Exercise 14.ROOM asks you to consider a version of the vacuum robot that has the policy of going straight for as long as it can; only when it encounters an obstacle does it change to a new heading. To model this robot, each state in the model consists of a (*location*, *heading*) pair. For the environment in Figure 14.7, which has 42 empty squares, this leads to 168 states and a transition matrix with $168^2 = 28{,}224$ entries—still a manageable number.

If we add the possibility of dirt in each of the 42 squares, the number of states is multiplied by 2^{42} and the transition matrix has more than 10^{29} entries—no longer a manageable number. In general, if the state is composed of n discrete variables with at most d values each, the corresponding HMM transition matrix will have size $O(d^{2n})$ and the per-update computation time will also be $O(d^{2n})$.

For these reasons, although HMMs have many uses in areas ranging from speech recognition to molecular biology, they are fundamentally limited in their ability to represent complex processes. In the terminology introduced in Chapter 2, HMMs are an atomic representation: states of the world have no internal structure and are simply labeled by integers. Section 14.5 shows how to use dynamic Bayesian networks—a factored representation—to model domains with many state variables. The next section shows how to handle domains with continuous state variables, which of course lead to an infinite state space.

14.4 Kalman Filters

Imagine watching a small bird flying through dense jungle foliage at dusk: you glimpse brief, intermittent flashes of motion; you try hard to guess where the bird is and where it will appear next so that you don't lose it. Or imagine that you are a World War II radar operator peering at a faint, wandering blip that appears once every 10 seconds on the screen. Or, going back further still, imagine you are Kepler trying to reconstruct the motions of the planets from a collection of highly inaccurate angular observations taken at irregular and imprecisely measured intervals.

In all these cases, you are doing filtering: estimating state variables (here, the position and velocity of a moving object) from noisy observations over time. If the variables were discrete, we could model the system with a hidden Markov model. This section examines methods for handling continuous variables, using an algorithm called **Kalman filtering**, after one of its inventors, Rudolf Kalman.

The bird's flight might be specified by six continuous variables at each time point; three for position (X_t, Y_t, Z_t) and three for velocity $(\dot{X}_t, \dot{Y}_t, \dot{Z}_t)$. We will need suitable conditional densities to represent the transition and sensor models; as in Chapter 13, we will use **linear–Gaussian distributions**. This means that the next state $\mathbf{X}_{t+1}$ must be a linear function of the current state $\mathbf{X}_t$, plus some Gaussian noise, a condition that turns out to be quite reasonable in practice. Consider, for example, the X-coordinate of the bird, ignoring the other coordinates for now. Let the time interval between observations be Δ, and assume constant velocity during the interval; then the position update is given by $X_{t+\Delta} = X_t + \dot{X}\Delta$. Adding Gaussian noise (to account for wind variation, etc.), we obtain a linear–Gaussian transition model:

$$P(X_{t+\Delta} = x_{t+\Delta} \big| X_t = x_t,\, \dot{X}_t = \dot{x}_t) = N(x_{t+\Delta}; x_t + \dot{x}_t\Delta, \sigma^2).$$

The Bayesian network structure for a system with position vector $\mathbf{X}_t$ and velocity $\dot{\mathbf{X}}_t$ is shown in Figure 14.9. Note that this is a very specific form of linear–Gaussian model; the general form will be described later in this section and covers a vast array of applications beyond the simple motion examples of the first paragraph. The reader might wish to consult Appendix A for some of the mathematical properties of Gaussian distributions; for our immediate purposes, the most important is that a multivariate Gaussian distribution for d variables is specified by a d-element mean μ and a d x d covariance matrix ∑.

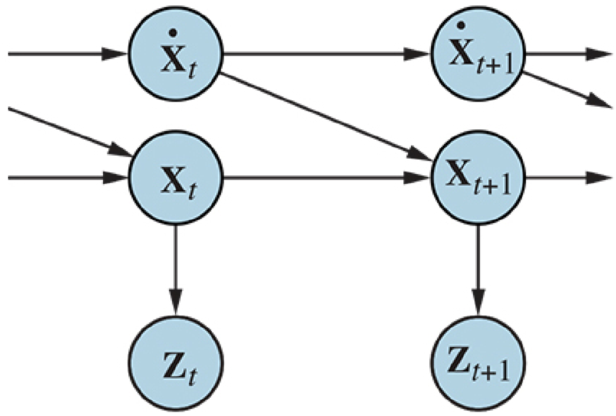


Figure 14.9 Bayesian network structure for a linear dynamical system with position $\mathbf{X}_t$, velocity $\dot{X}t$, and position measurement $\mathbf{Z}_t$.

14.4.1 Updating Gaussian distributions

In Chapter 13 on page 441, we alluded to a key property of the linear–Gaussian family of distributions: it remains closed under Bayesian updating. (That is, given any evidence, the posterior is still in the linear–Gaussian family.) Here we make this claim precise in the context of filtering in a temporal probability model. The required properties correspond to the two- step filtering calculation in Equation (14.5):

1. If the current distribution $\mathbf{P}(\mathbf{X}_t|\mathbf{e}_{1:t})$ is Gaussian and the transition model $\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{x}_t)$ is linear–Gaussian, then the one-step predicted distribution given by

$$\mathbf{P}(X_{t+1}\Big|\mathbf{e}_{1:t}) = \int_{\mathbf{x}_t} \mathbf{P}(\mathbf{X}_{t+1}|\mathbf{x}_t)\; P(\mathbf{x}_t|\mathbf{e}_{1:t})d\mathbf{x}_t \qquad (14.17)$$

   is also a Gaussian distribution.

2. If the prediction $\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t})$ is Gaussian and the sensor model $\mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1})$ is linear– Gaussian, then, after conditioning on the new evidence, the updated distribution

$$\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t+1}) = \alpha\mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1})\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t}) \qquad (14.18)$$

   is also a Gaussian distribution.

Thus, the FORWARD operator for Kalman filtering takes a Gaussian forward message $\mathbf{f}_{1:t}$, specified by a mean μ_t and covariance Σ_t, and produces a new multivariate Gaussian forward message $\mathbf{f}_{1:t+1}$, specified by a mean μ_{t+1} and covariance Σ_{t+1}. So if we start with a Gaussian prior $\mathbf{f}_{1:0} = \mathbf{P}(\mathbf{X}_0) = N(\mu_0,, \Sigma_0)$, filtering with a linear–Gaussian model produces a Gaussian state distribution for all time.

This seems to be a nice, elegant result, but why is it so important? The reason is that except for a few special cases such as this, *filtering with continuous or hybrid (discrete and continuous) networks* *generates state distributions whose representation grows without bound over time.* This statement is not easy to prove in general, but Exercise 14.KFSW shows what happens for a simple example.

14.4.2 A simple one-dimensional example

We have said that the FORWARD operator for the Kalman filter maps a Gaussian into a new Gaussian. This translates into computing a new mean and covariance from the previous mean and covariance. Deriving the update rule in the general (multivariate) case requires rather a lot of linear algebra, so we will stick to a very simple univariate case for now, and later give the results for the general case. Even for the univariate case, the calculations are somewhat tedious, but we feel that they are worth seeing because the usefulness of the Kalman filter is tied so intimately to the mathematical properties of Gaussian distributions.

The temporal model we consider describes a **random walk** of a single continuous state variable X_t with a noisy observation Z_t. An example might be the "consumer confidence" index, which can be modeled as undergoing a random Gaussian-distributed change each month and is measured by a random consumer survey that also introduces Gaussian sampling noise. The prior distribution is assumed to be Gaussian with variance σ_0^2

$$P(x_0) = \alpha\, e^{-\frac{1}{2}\left(\frac{(x_0-\mu_0)^2}{\sigma_0^2}\right)}.$$

(For simplicity, we use the same symbol α for all normalizing constants in this section.) The transition model adds a Gaussian perturbation of constant variance σ_x^2 to the current state:

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

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

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

Now, given the prior $P(X_0)$, the one-step predicted distribution comes from Equation (14.17):

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

This integral looks rather complicated. The key to progress is to notice that the exponent is the sum of two expressions that are *quadratic* in x_0 and hence is itself a quadratic in x_0. A simple trick known as **completing the square** allows the rewriting of any quadratic $ax_0^2 + bx_0 + c$ as the sum of a squared term $a(x_0 - \frac{-b}{2a})^2$ and a residual term $c - \frac{b^2}{4a}$ that is independent of x_0. In this case, we have $a = (\sigma_0^2 + \sigma_x^2)/(\sigma_0^2\sigma_x^2), b = -2(\sigma_0^2 x_1 + \sigma_x^2\mu_0)/(\sigma_0^2\sigma_x^2)$, and $c = (\sigma_0^2 x_1^2 + \sigma_x^2\mu_0^2)/(\sigma_0^2\sigma_x^2)$. The residual term can be taken outside the integral, giving us

$$P(x_1) \;=\; \alpha\; e^{-\frac{1}{2}\left(c-\frac{b^2}{4a}\right)} \int_{-\infty}^{\infty} e^{-\frac{1}{2}\left(a(x_0-\frac{-b}{2a})^2\right)}\; dx_0$$

Now the integral is just the integral of a Gaussian over its full range, which is simply 1. Thus, we are left with only the residual term from the quadratic. Plugging back in the expressions for a, b, and c and simplifying, we obtain

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

That is, the one-step predicted distribution is a Gaussian with the same mean μ_0 and a variance equal to the sum of the original variance σ_0^2 and the transition variance σ_0^x.

To complete the update step, we need to condition on the observation at the first time step, namely, z_1. From Equation (14.18), this is given by

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

Once again, we combine the exponents and complete the square (Exercise 14.KALM), obtaining the following expression for the posterior:

$$P(x_1 \middle| z_1) = \alpha\, e^{-\frac{1}{2}\frac{\left(x_1-\frac{(\sigma_0^2+\sigma_x^2)z_1+\sigma_z^2\mu_0}{\sigma_0^2+\sigma_x^2+\sigma_z^2}\right)^2}{(\sigma_0^2+\sigma_x^2)\sigma_z^2/(\sigma_0^2+\sigma_x^2+\sigma_z^2)}}. \qquad (14.19)$$

Thus, after one update cycle, we have a new Gaussian distribution for the state variable.

From the Gaussian formula in Equation (14.19), we see that the new mean and standard deviation can be calculated from the old mean and standard deviation as follows:

$$\mu_{t+1} = \frac{(\sigma_t^2+\sigma_x^2)z_{t+1}+\sigma_z^2\mu_t}{\sigma_t^2+\sigma_x^2+\sigma_z^2} \qquad \text{and} \qquad \sigma_{t+1}^2 = \frac{(\sigma_t^2+\sigma_x^2)\sigma_z^2}{\sigma_t^2+\sigma_x^2+\sigma_z^2} \qquad (14.20) \quad .$$

Figure 14.10 shows one update cycle of the Kalman filter in the one-dimensional case for particular values of the transition and sensor models.

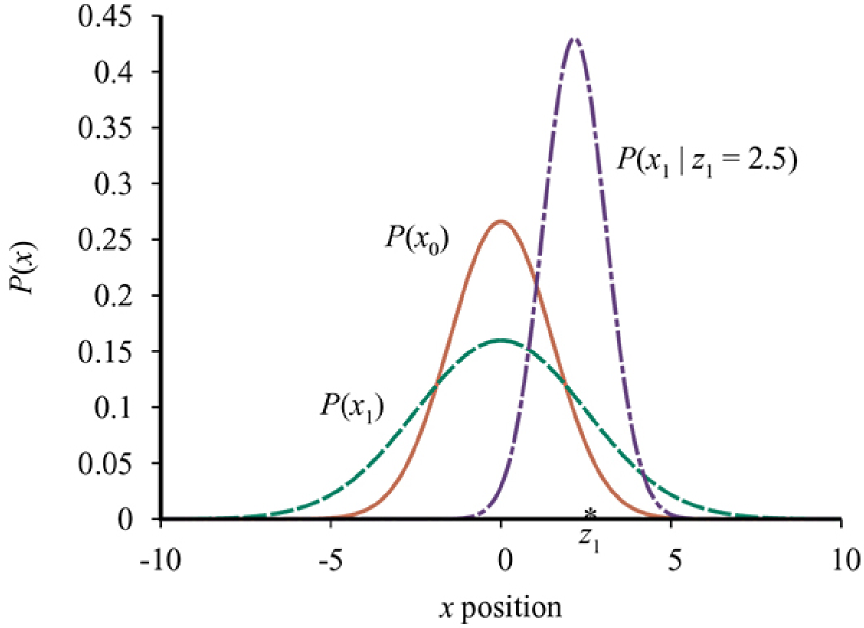


Figure 14.10 Stages in the Kalman filter update cycle for a random walk with a prior given by $\mu_0 = 0.0$ and $\sigma_0 = 1.5$, transition noise given by $\sigma_x = 2.0$, sensor noise given by $\sigma_z = 1.0$, and a first observation $z_1 = 2.5$ (marked on the x-axis). Notice how the prediction $P(x_1)$ is flattened out, relative to $P(x_0)$, by the transition noise. Notice also that the mean of the posterior distribution $P(x_1 \mid z_1)$ is slightly to the left of the observation z_1 because the mean is a weighted average of the prediction and the observation.

Equation (14.20) plays exactly the same role as the general filtering equation (14.5) or the HMM filtering equation (14.12). Because of the special nature of Gaussian distributions, however, the equations have some interesting additional properties.

First, we can interpret the calculation for the new mean μ_{t+1} as a *weighted mean* of the new observation z_{t+1} and the old mean μ_t. If the observation is unreliable, then σ_z^2 is large and we pay more attention to the old mean; if the old mean is unreliable (σ_t^2 is large) or the process is highly unpredictable (σ_x^2 is large), then we pay more attention to the observation.

Second, notice that the update for the variance σ_{t+1}^2 is *independent of the observation.* We can therefore compute in advance what the sequence of variance values will be. Third, the sequence of variance values converges quickly to a fixed value that depends only on σ_x^2 and σ_z^2, thereby substantially simplifying the subsequent calculations. (See Exercise 14.VARI.)

14.4.3 The general case

The preceding derivation illustrates the key property of Gaussian distributions that allows Kalman filtering to work: the fact that the exponent is a quadratic form. This is true not just for the univariate case; the full multivariate Gaussian distribution has the form

$$N(\mathbf{x};\mu,\textstyle\sum)=\alpha e^{-\frac{1}{2}\left((\mathbf{x}-\mu)^{\mathrm{T}}\sum^{-1}(\mathbf{x}-\mu)\right)}.$$

Multiplying out the terms in the exponent, we see that the exponent is also a quadratic function of the values x_i in **x**. Thus, filtering preserves the Gaussian nature of the state distribution.

Let us first define the general temporal model used with Kalman filtering. Both the transition model and the sensor model are required to be a *linear* transformation with additive Gaussian noise. Thus, we have

$$\begin{aligned}P(\mathbf{x}_{t+1}|\mathbf{x}_t) &= N(\mathbf{x}_{t+1};\mathbf{F}\mathbf{x}_t,\textstyle\sum_x)\\ P(\mathbf{z}_t|\mathbf{x}_t) &= N(\mathbf{z}_t;\mathbf{H}\mathbf{x}_t,\textstyle\sum_z)\,.\end{aligned} \qquad (14.21)$$

where **F** and $\sum_x$ are matrices describing the linear transition model and transition noise covariance, and **H** and $\sum_z$ are the corresponding matrices for the sensor model. Now the update equations for the mean and covariance, in their full, hairy horribleness, are

$$\begin{aligned}\mu_{t+1} &= \mathbf{F}\mu_t+\mathbf{K}_{t+1}(\mathbf{z}_{t+1}-\mathbf{H}\mathbf{F}\mu_t)\\ \textstyle\sum_{t+1} &= (\mathbf{I}-\mathbf{K}_{t+1}\mathbf{H})\left(\mathbf{F}\textstyle\sum_t\mathbf{F}^\top+\sum_x\right),\end{aligned} \qquad (14.22)$$

where $\mathbf{K}_{t+1}=(\mathbf{F}\sum_t\mathbf{F}^\top+\sum_x)\mathbf{H}^\top(\mathbf{H}(\mathbf{F}\sum_t\mathbf{F}^\top+\sum_x)\mathbf{H}^\top+\sum_z)^{-1}$ is the **Kalman gain matrix**. Believe it or not, these equations make some intuitive sense. For example, consider the update for the mean state estimate μ. The term $\mathbf{F}\mu_t$ is the *predicted* state at $t+1$, so $\mathbf{HF}\mu_t$ is the *predicted* observation. Therefore, the term $\mathbf{z}_{t+1}-\mathbf{HF}\mu_t$ represents the error in the predicted observation. This is multiplied by $\mathbf{K}_{t+1}$ to correct the predicted state; hence, $\mathbf{K}_{t+1}$ is a measure of *how seriously to take the new observation* relative to the prediction. As in Equation (14.20), we also have the property that the variance update is independent of the observations. The sequence of values for $\sum_t$ and $\mathbf{K}_t$ can therefore be computed offline, and the actual calculations required during online tracking are quite modest.

To illustrate these equations at work, we have applied them to the problem of tracking an object moving on the X–Y plane. The state variables are $\mathbf{X}=(X,Y,\dot{X},\dot{Y})^\top$, so **F**, $\sum_x$, **H**, and $\sum_z$ are 4×4 matrices. Figure 14.11(a) shows the true trajectory, a series of noisy observations, and the trajectory estimated by Kalman filtering, along with the covariances indicated by the one-standard-deviation contours. The filtering process does a good job of tracking the actual motion, and, as expected, the variance quickly reaches a fixed point.

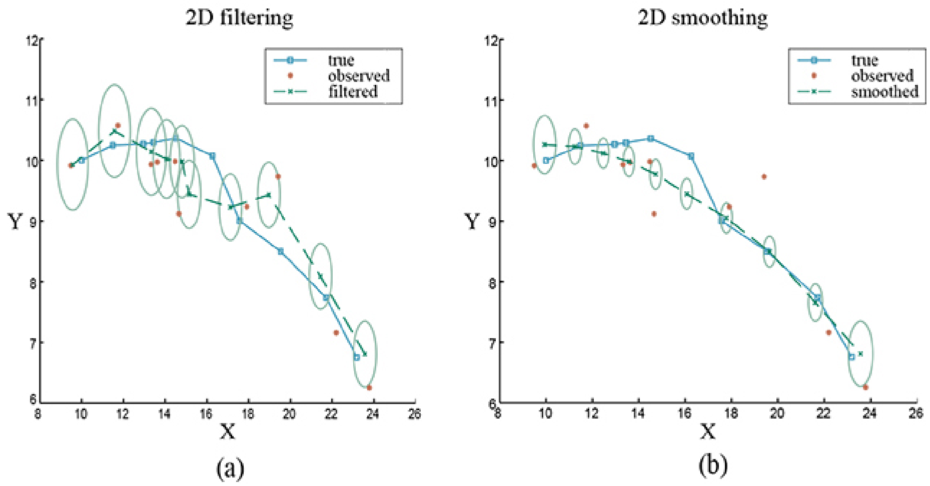


Figure 14.11 (a) Results of Kalman filtering for an object moving on the *X*–*Y* plane, showing the true trajectory (left to right), a series of noisy observations, and the trajectory estimated by Kalman filtering. Variance in the position estimate is indicated by the ovals. (b) The results of Kalman smoothing for the same observation sequence.

We can also derive equations for *smoothing* as well as filtering with linear–Gaussian models. The smoothing results are shown in Figure 14.11(b). Notice how the variance in the position estimate is sharply reduced, except at the ends of the trajectory (why?), and that the estimated trajectory is much smoother.

14.4.4 Applicability of Kalman filtering

The Kalman filter and its elaborations are used in a vast array of applications. The "classical" application is in radar tracking of aircraft and missiles. Related applications include acoustic tracking of submarines and ground vehicles and visual tracking of vehicles and people. In a slightly more esoteric vein, Kalman filters are used to reconstruct particle trajectories from bubble-chamber photographs and ocean currents from satellite surface measurements. The range of application is much larger than just the tracking of motion: any system characterized by continuous state variables and noisy measurements will do. Such systems include pulp mills, chemical plants, nuclear reactors, plant ecosystems, and national economies.

The fact that Kalman filtering can be applied to a system does not mean that the results will be valid or useful. The assumptions made—linear–Gaussian transition and sensor models—are very strong. The **extended Kalman filter (EKF)** attempts to overcome nonlinearities in the system being modeled. A system is **nonlinear** if the transition model cannot be described as a matrix multiplication of the state vector, as in Equation (14.21). The EKF works by modeling the system as *locally* linear in $\mathbf{x}_t$ in the region of $\mathbf{x}_t = \mu_t$, the mean of the current state distribution. This works well for smooth, well-behaved systems and allows the tracker to maintain and update a Gaussian state distribution that is a reasonable approximation to the true posterior. A detailed example is given in Chapter 26.

What does it mean for a system to be "unsmooth" or "poorly behaved"? Technically, it means that there is significant nonlinearity in system response within the region that is "close" (according to the covariance Σ_t) to the current mean μ_t. To understand this idea in nontechnical terms, consider the example of trying to track a bird as it flies through the jungle. The bird appears to be heading at high speed straight for a tree trunk. The Kalman filter, whether regular or extended, can make only a Gaussian prediction of the location of the bird, and the mean of this Gaussian will be centered on the trunk, as shown in Figure 14.12(a). A reasonable model of the bird, on the other hand, would predict evasive action to one side or the other, as shown in Figure 14.12(b). Such a model is highly nonlinear, because the bird's decision varies sharply depending on its precise location relative to the trunk.

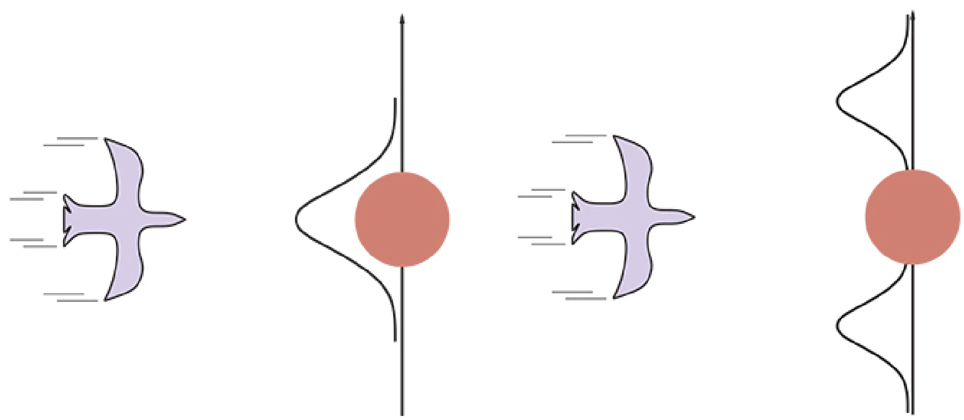


Figure 14.12 A bird flying toward a tree (top views). (a) A Kalman filter will predict the location of the bird using a single Gaussian centered on the obstacle. (b) A more realistic model allows for the bird's evasive action, predicting that it will fly to one side or the other.

To handle examples like these, we clearly need a more expressive language for representing the behavior of the system being modeled. Within the control theory community, for which problems such as evasive maneuvering by aircraft raise the same kinds of difficulties, the standard solution is the **switching Kalman filter**. In this approach, multiple Kalman filters run in parallel, each using a different model of the system—for example, one for straight flight, one for sharp left turns, and one for sharp right turns. A weighted sum of predictions is used, where the weight depends on how well each filter fits the current data. We will see in the next section that this is simply a special case of the general dynamic Bayesian network model, obtained by adding a discrete “maneuver” state variable to the network shown in Figure 14.9. Switching Kalman filters are discussed further in Exercise 14.KFSW.

14.5 Dynamic Bayesian Networks

Dynamic Bayesian networks, or **DBNs**, extend the semantics of standard Bayesian networks to handle temporal probability models of the kind described in Section 14.1. We have already seen examples of DBNs: the umbrella network in Figure 14.2 and the Kalman filter network in Figure 14.9. In general, each slice of a DBN can have any number of state variables $\mathbf{X}_t$ and evidence variables $\mathbf{E}_t$. For simplicity, we assume that the variables, their links, and their conditional distributions are exactly replicated from slice to slice and that the DBN represents a first-order Markov process, so that each variable can have parents only in its own slice or the immediately preceding slice. In this way, the DBN corresponds to a Bayesian network with infinitely many variables.

It should be clear that every hidden Markov model can be represented as a DBN with a single state variable and a single evidence variable. It is also the case that every discrete-variable DBN can be represented as an HMM; as explained in Section 14.3, we can combine all the state variables in the DBN into a single state variable whose values are all possible tuples of values of the individual state variables. Now, if every HMM is a DBN and every DBN can be translated into an HMM, what's the difference? The difference is that, *by decomposing the state of a complex system into its constituent variables, we can take advantage of* sparseness *in the temporal probability model.*

To see what this means in practice, remember that in Section 14.3 we said that an HMM representation for a temporal process with n discrete variables, each with up to d values, needs a transition matrix of size $O(d^{2n})$. The DBN representation, on the other hand, has size $O(nd^k)$ if the number of parents of each variable is bounded by k. In other words, the DBN representation is linear rather than exponential in the number of variables. For the vacuum robot with 42 possibly dirty locations, the number of probabilities required is reduced from 5×10^{29} to a few thousand.

We have already explained that every Kalman filter model can be represented in a DBN with continuous variables and linear–Gaussian conditional distributions (Figure 14.9). It should be clear from the discussion at the end of the preceding section that ***not*** every DBN can be represented by a Kalman filter model. In a Kalman filter, the current state distribution is always a single multivariate Gaussian distribution—that is, a single “bump” in a particular location. DBNs, on the other hand, can model arbitrary distributions.

For many real-world applications, this flexibility is essential. Consider, for example, the current location of my keys. They might be in my pocket, on the bedside table, on the kitchen counter, dangling from the front door, or locked in the car. A single Gaussian bump that included all these places would have to allocate significant probability to the keys being in mid-air above the front garden. Aspects of the real world such as purposive agents, obstacles, and pockets introduce “nonlinearities” that require combinations of discrete and continuous variables in order to get reasonable models.

14.5.1 Constructing DBNs

To construct a DBN, one must specify three kinds of information: the prior distribution over the state variables, $\mathbf{P}(\mathbf{X}_0)$; the transition model $\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{X}_t)$; and the sensor model $\mathbf{P}(\mathbf{E}_t \mid \mathbf{X}_t)$. To specify the transition and sensor models, one must also specify the topology of the connections between successive slices and between the state and evidence variables. Because the transition and sensor models are assumed to be time-homogeneous—the same for all t —it is most convenient simply to specify them for the first slice. For example, the complete DBN specification for the umbrella world is given by the three-node network shown in Figure 14.13(a). From this specification, the complete DBN with an unbounded number of time slices can be constructed as needed by copying the first slice.

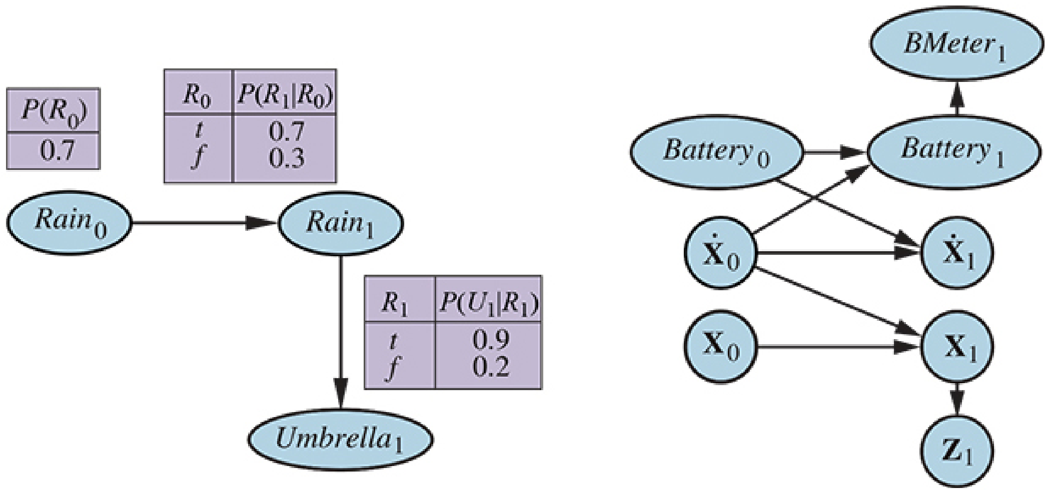


Figure 14.13 Left: Specification of the prior, transition model, and sensor model for the umbrella DBN. Subsequent slices are copies of slice 1. Right: A simple DBN for robot motion in the X–Y plane.

Let us now consider a more interesting example: monitoring a battery-powered robot moving in the X–Y plane, as introduced at the end of Section 14.1. First, we need state variables, which will include both $\mathbf{X}_t = (X_t, Y_t)$ for position and $\dot{\mathbf{X}}_t = (\dot{X}_t, \dot{Y}_t)$ for velocity. We assume some method of measuring position—perhaps a fixed camera or onboard GPS (Global Positioning System)—yielding measurements $\mathbf{Z}_t$. The position at the next time step depends on the current position and velocity, as in the standard Kalman filter model. The velocity at the next step depends on the current velocity and the state of the battery. We add $Battery_t$ to represent the actual battery charge level, which has as parents the previous battery level and the velocity, and we add $BMeter_t$, which measures the battery charge level. This gives us the basic model shown in Figure 14.13(b).

It is worth looking in more depth at the nature of the sensor model for $BMeter_t$. Let us suppose, for simplicity, that both Battery_t and $BMeter_t$ can take on discrete values 0 through 5. (Exercise 14.BATT asks you to relate this discrete model to a corresponding continuous model.) If the meter is always accurate, then the CPT $\mathbf{P}(BMeter_t \mid Battery_t)$

should have probabilities of 1.0 “along the diagonal” and probabilities of 0.0 elsewhere. In reality, noise always creeps into measurements. For continuous measurements, a Gaussian distribution with a small variance might be used.[7] For our discrete variables, we can approximate a Gaussian using a distribution in which the probability of error drops off in the appropriate way, so that the probability of a large error is very small. We use the term **Gaussian error model** to cover both the continuous and discrete versions.

Anyone with hands-on experience of robotics, computerized process control, or other forms of automatic sensing will readily testify to the fact that small amounts of measurement noise are often the least of one’s problems. Real sensors *fail*. When a sensor fails, it does not necessarily send a signal saying, “Oh, by the way, the data I’m about to send you is a load of nonsense.” Instead, it simply sends the nonsense. The simplest kind of failure is called a **transient failure,** where the sensor occasionally decides to send some nonsense. For example, the battery level sensor might have a habit of sending a reading of 0 when someone bumps the robot, even if the battery is fully charged.

Let’s see what happens when a transient failure occurs with a Gaussian error model that doesn’t accommodate such failures. Suppose, for example, that the robot is sitting quietly and observes 20 consecutive battery readings of 5. Then the battery meter has a temporary seizure and the next reading is $BMeter_{21} = 0$. What will the simple Gaussian error model lead us to believe about $Battery_{21}$? According to Bayes’ rule, the answer depends on both the sensor model $\mathbf{P}(BMeter_{21} = 0 | \; Battery_{21})$ and the prediction $\mathbf{P}(Battery_{21} \mid BMeter_{1:20})$. If the probability of a large sensor error is significantly less than the probability of a transition to $Battery_{21} = 0$, even if the latter is very unlikely, then the posterior distribution will assign a high probability to the battery’s being empty.

A second reading of 0 at $t = 22$ will make this conclusion almost certain. If the transient failure then disappears and the reading returns to 5 from $t = 23$ onwards, the estimate for the battery level will quickly return to 5. (This does not mean the algorithm thinks the battery magically recharged itself, which may be physically impossible; instead, the algorithm now believes that the battery was never low and the extremely unlikely hypothesis that the battery meter had two consecutive huge errors must be the right explanation.) This course of events is illustrated in the upper curve of Figure 14.14(a), which shows the expected value (see Appendix A) of $Battery_t$ over time, using a discrete Gaussian error model.

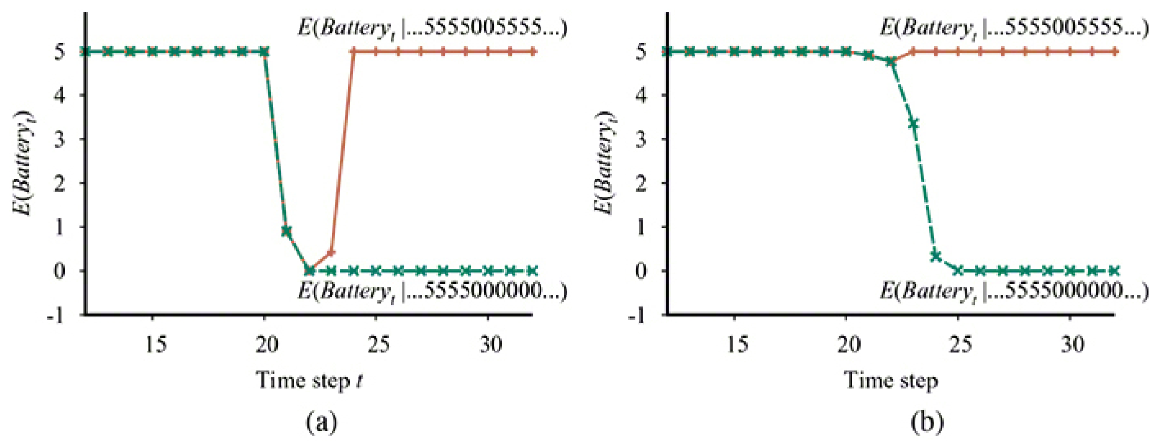


Figure 14.14 (a) Upper curve: trajectory of the expected value of $Battery_t$ for an observation sequence consisting of all 5s except for 0s at $t = 21$ and $t = 22$, using a simple Gaussian error model. Lower curve: trajectory when the observation remains at 0 from $t = 21$ onwards. (b) The same experiment run with the transient failure model. The transient failure is handled well, but the persistent failure results in excessive pessimism about the battery charge.

Despite the recovery, there is a time ($t = 22$) when the robot is convinced that its battery is empty; presumably, then, it should send out a mayday signal and shut down. Alas, its oversimplified sensor model has led it astray. The moral of the story is simple: *for the system to handle sensor failure properly, the sensor model must include the possibility of failure.*

The simplest kind of failure model for a sensor allows a certain probability that the sensor will return some completely incorrect value, regardless of the true state of the world. For example, if the battery meter fails by returning 0, we might say that

$$P(BMeter_t = 0 | Battery_t = 5) = 0.03,$$

which is presumably much larger than the probability assigned by the simple Gaussian error model. Let's call this the **transient failure model**. How does it help when we are faced with a reading of 0? Provided that the *predicted* probability of an empty battery, according to the readings so far, is much less than 0.03, then the best explanation of the observation $BMeter_{21} = 0$ is that the sensor has temporarily failed. Intuitively, we can think of the belief about the battery level as having a certain amount of "inertia" that helps to overcome temporary blips in the meter reading. The upper curve in Figure 14.14(b) shows that the transient failure model can handle transient failures without a catastrophic change in beliefs.

So much for temporary blips. What about a persistent sensor failure? Sadly, failures of this kind are all too common. If the sensor returns 20 readings of 5 followed by 20 readings of 0, then the transient sensor failure model described in the preceding paragraph will result in the robot gradually coming to believe that its battery is empty when in fact it may be that the meter has failed. The lower curve in Figure 14.14(b) shows the belief "trajectory" for this case. By $t = 25$—five readings of 0—the robot is convinced that its battery is empty. Obviously, we would prefer the robot to believe that its battery meter is broken—if indeed this is the more likely event.

Unsurprisingly, to handle persistent failure, we need a **persistent failure model** that describes how the sensor behaves under normal conditions and after failure. To do this, we need to augment the state of the system with an additional variable, say, *BMBroken*, that describes the status of the battery meter. The persistence of failure must be modeled by an arc linking $BMBroken_0$ to $BMBroken_1$. This **persistence arc** has a CPT that gives a small probability of failure in any given time step, say, 0.001, but specifies that the sensor stays broken once it breaks. When the sensor is OK, the sensor model for *BMeter* is identical to the transient failure model; when the sensor is broken, it says *BMeter* is always 0, regardless of the actual battery charge.

The persistent failure model for the battery sensor is shown in Figure 14.15(a). Its performance on the two data sequences (temporary blip and persistent failure) is shown in Figure 14.15(b). There are several things to notice about these curves. First, in the case of the temporary blip, the probability that the sensor is broken rises significantly after the second 0 reading, but immediately drops back to zero once a 5 is observed. Second, in the case of persistent failure, the probability that the sensor is broken rises quickly to almost 1 and stays there. Finally, once the sensor is known to be broken, the robot can only assume that its battery discharges at the "normal" rate. This is shown by the gradually descending level of $E(Battery_t \mid \ldots)$.

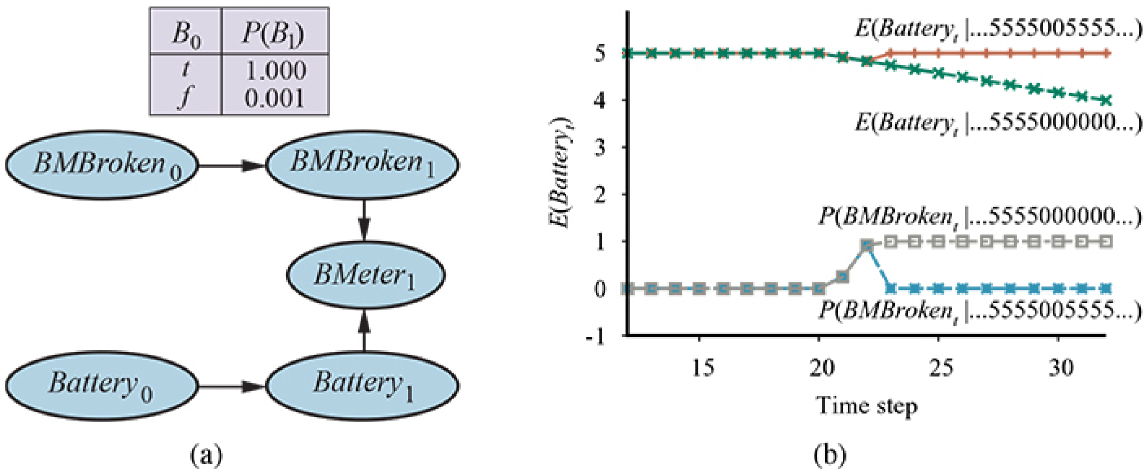


Figure 14.15 (a) A DBN fragment showing the sensor status variable required for modeling persistent failure of the battery sensor. (b) Upper curves: trajectories of the expected value of $Battery_t$ for the "transient failure" and "permanent failure" observations sequences. Lower curves: probability trajectories for *BMBroken* given the two observation sequences.

So far, we have merely scratched the surface of the problem of representing complex processes. The variety of transition models is huge, encompassing topics as disparate as modeling the human endocrine system and modeling multiple vehicles driving on a freeway. Sensor modeling is also a vast subfield in itself. But dynamic Bayesian networks can model even subtle phenomena, such as sensor drift, sudden decalibration, and the effects of exogenous conditions (such as weather) on sensor readings.

14.5.2 Exact inference in DBNs

Having sketched some ideas for representing complex processes as DBNs, we now turn to the question of inference. In a sense, this question has already been answered: dynamic Bayesian networks *are* Bayesian networks, and we already have algorithms for inference in Bayesian networks. Given a sequence of observations, one can construct the full Bayesian network representation of a DBN by replicating slices until the network is large enough to accommodate the observations, as in Figure 14.16. This technique is called

unrolling. (Technically, the DBN is equivalent to the semi-infinite network obtained by unrolling forever. Slices added beyond the last observation have no effect on inferences within the observation period and can be omitted.) Once the DBN is unrolled, one can use any of the inference algorithms— variable elimination, clustering methods, and so on— described in Chapter 13.

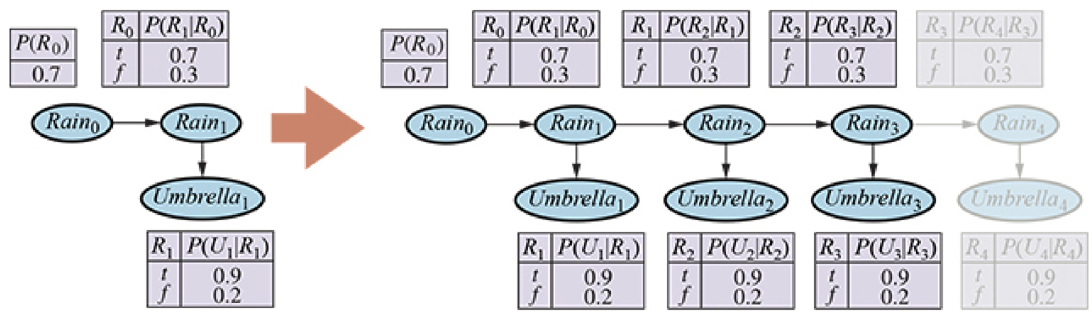


Figure 14.16 Unrolling a dynamic Bayesian network: slices are replicated to accommodate the observation sequence $Umbrella_{1:3}$. Further slices have no effect on inferences within the observation period.

Unfortunately, a naive application of unrolling would not be particularly efficient. If we want to perform filtering or smoothing with a long sequence of observations $\mathbf{e}_{1:t}$, the unrolled network would require $O(t)$ space and would thus grow without bound as more observations were added. Moreover, if we simply run the inference algorithm anew each time an observation is added, the inference time per update will also increase as $O(t)$.

Looking back to Section 14.2.1, we see that constant time and space per filtering update can be achieved if the computation can be done recursively. Essentially, the filtering update in Equation (14.5) works by *summing out* the state variables of the previous time step to get the distribution for the new time step. Summing out variables is exactly what the **variable elimination** (Figure 13.13) algorithm does, and it turns out that running variable elimination with the variables in temporal order exactly mimics the operation of the recursive filtering update in Equation (14.5). The modified algorithm keeps at most two slices in memory at any one time: starting with slice 0, we add slice 1, then sum out slice 0, then add slice 2, then sum out slice 1, and so on. In this way, we can achieve constant space and time per filtering update. (The same performance can be achieved by suitable modifications to the clustering algorithm.) Exercise 14.DBNE asks you to verify this fact for the umbrella network.

So much for the good news; now for the bad news: It turns out that the “constant” for the per-update time and space complexity is, in almost all cases, exponential in the number of state variables. What happens is that, as the variable elimination proceeds, the factors grow to include all the state variables (or, more precisely, all those state variables that have parents in the previous time slice). The maximum factor size is $O(d^{n+k})$ and the total update cost per step is $O(nd^{n+k})$, where d is the domain size of the variables and k is the maximum number of parents of any state variable.

Of course, this is much less than the cost of HMM updating, which is $O(d^{2n})$, but it is still infeasible for large numbers of variables. This grim fact means is that *even though we can use DBNs to* represent *very complex temporal processes with many sparsely connected variables, we cannot* reason *efficiently and exactly about those processes.* The DBN model itself, which represents the prior joint distribution over all the variables, is factorable into its constituent CPTs, but the posterior joint distribution conditioned on an observation sequence—that is, the forward message—is generally *not* factorable. The problem is intractable in general, so we must fall back on approximate methods.

14.5.3 Approximate inference in DBNs

Section 13.4 described two approximation algorithms: likelihood weighting (Figure 13.18) and Markov chain Monte Carlo (MCMC, Figure 13.20). Of the two, the former is most easily adapted to the DBN context. (An MCMC filtering algorithm is described briefly in the notes at the end of this chapter.) We will see, however, that several improvements are required over the standard likelihood weighting algorithm before a practical method emerges.

Recall that likelihood weighting works by sampling the nonevidence nodes of the network in topological order, weighting each sample by the likelihood it accords to the observed evidence variables. As with the exact algorithms, we could apply likelihood weighting directly to an unrolled DBN, but this would suffer from the same problems of increasing time and space requirements per update as the observation sequence grows. The problem is that the standard algorithm runs each sample in turn, all the way through the network.

Instead, we can simply run all N samples together through the DBN, one slice at a time. The modified algorithm fits the general pattern of filtering algorithms, with the set of N samples as the forward message. The first key innovation, then, is to *use the samples themselves as an approximate representation of the current state distribution.* This meets the requirement of a "constant" time per update, although the constant depends on the number of samples required to maintain an accurate approximation. There is also no need to unroll the DBN, because we need to have in memory only the current slice and the next slice. This approach is called **sequential importance sampling** or SIS.

In our discussion of likelihood weighting in Chapter 13, we pointed out that the algorithm's accuracy suffers if the evidence variables are "downstream" from the variables being sampled, because in that case the samples are generated without any influence from the evidence and will nearly all have very low weights.

Now if we look at the typical structure of a DBN—say, the umbrella DBN in Figure 14.16—we see that indeed the early state variables will be sampled without the benefit of the later evidence. In fact, looking more carefully, we see that *none* of the state variables have *any* evidence variables among its ancestors! Hence, although the weight of each sample will depend on the evidence, the actual set of samples generated will be *completely independent* of the evidence. For example, even if the boss brings in the umbrella every day, the sampling process could still hallucinate endless days of sunshine.

What this means in practice is that the fraction of samples that remain reasonably close to the actual series of events (and therefore have non-negligible weights) drops exponentially with t, the length of the sequence. In other words, to maintain a given level of accuracy, we need to increase the number of samples exponentially with t. Given that a real-time filtering algorithm can use only a bounded number of samples, what happens in practice is that the error blows up after a very small number of update steps. Figure 14.19 on page 512 shows this effect for SIS applied to the grid-world localization problem from Section 14.3: even with 100,000 samples, the SIS approximation fails completely after about 20 steps.

Clearly, we need a better solution. The second key innovation is to *focus the set of samples on the high-probability regions of the state space.* This can be done by throwing away samples that have very low weight, according to the observations, while replicating those that have high weight. In that way, the population of samples will stay reasonably close to reality. If we think of samples as a resource for modeling the posterior distribution, then it makes sense to use more samples in regions of the state space where the posterior is higher.

A family of algorithms called **particle filtering** is designed to do just that. (Another early name was **sequential importance sampling with resampling**, but for some reason it failed on catch on.) Particle filtering works as follows: First, we generate a population of N samples from the prior distribution $\mathbf{P}(\mathbf{X}_0)$. Then the update cycle is repeated for each time step:

1. Each sample is propagated forward by sampling the next state value $\mathbf{x}_{t+1}$ given the current value $\mathbf{x}_t$ for the sample, based on the transition model $\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{x}_t)$.
2. Each sample is weighted by the likelihood it assigns to the new evidence, $P(\mathbf{e}_{t+1} \mid \mathbf{x}_{t+1})$.
3. The population is *resampled* to generate a new population of N samples. Each new sample is selected from the current population; the probability that a particular sample is selected is proportional to its weight. The new samples are unweighted.

The algorithm is shown in detail in Figure 14.17, and its operation for the umbrella DBN is illustrated in Figure 14.18.

---

function PARTICLE-FILTERING($\mathbf{e}, N, dbn$) **returns** a set of samples for the next time step
  inputs: $\mathbf{e}$, the new incoming evidence
    N, the number of samples to be maintained
    dbn, a DBN defined by $\mathbf{P}(\mathbf{X}_0)$, $\mathbf{P}(\mathbf{X}_1 \mid \mathbf{X}_0)$, and $\mathbf{P}(\mathbf{E}_1 \mid \mathbf{X}_1)$
  persistent: S, a vector of samples of size N, initially generated from $\mathbf{P}(\mathbf{X}_0)$
  local variables: W, a vector of weights of size N

  for $i = 1$ to N **do**
    $S[i] \leftarrow$ sample from $\mathbf{P}(\mathbf{X}_1 \mid \mathbf{X}_0 = S[i])$ *// step 1*
    $W[i] \leftarrow \mathbf{P}(\mathbf{e} \mid \mathbf{X}_1 = S[i])$ *// step 2*
  $S \leftarrow$ WEIGHTED-SAMPLE-WITH-REPLACEMENT(N, S, W)   *// step 3*
  return S

Figure 14.17 The particle filtering algorithm implemented as a recursive update operation with state (the set of samples). Each of the sampling operations involves sampling the relevant slice variables in topological order, much as in PRIOR-SAMPLE. The WEIGHTED-SAMPLE-WITH-REPLACEMENT operation can be implemented to run in $O(N)$ expected time. The step numbers refer to the description in the text.

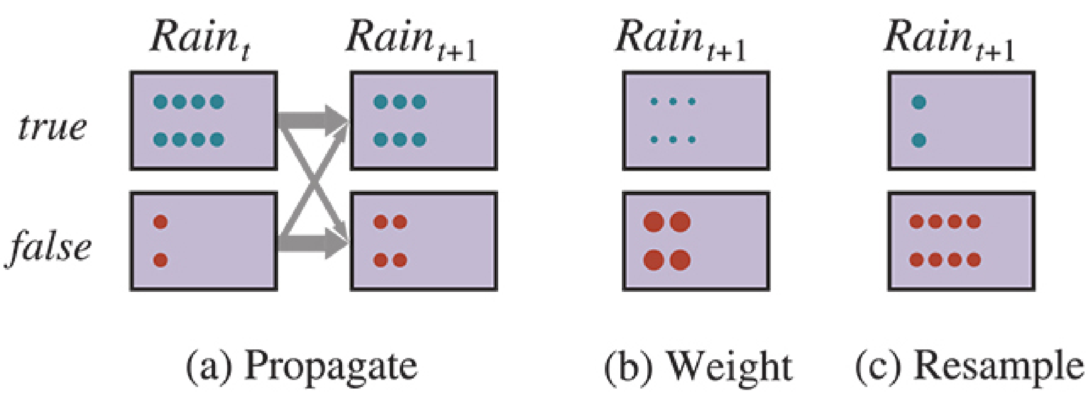


Figure 14.18 The particle filtering update cycle for the umbrella DBN with $N = 10$, showing the sample populations of each state. (a) At time t, 8 samples indicate *rain* and 2 indicate ¬*rain*. Each is propagated forward by sampling the next state through the transition model. At time $t + 1$, 6 samples indicate *rain* and 4 indicate ¬*rain*. (b) ¬*umbrella* is observed at $t + 1$. Each sample is weighted by its likelihood for the observation, as indicated by the size of the circles. (c) A new set of 10 samples is generated by weighted random selection from the current set, resulting in 2 samples that indicate *rain* and 8 that indicate ¬*rain*.

We can show that this algorithm is consistent—gives the correct probabilities as N tends to infinity—by examining the operations in one update cycle. We assume that the sample population starts with a correct representation of the forward message—that is, $\mathbf{f}_{1:t} = \mathbf{P}(\mathbf{X}_t|\mathbf{e}_{1:t})$ at time t. Writing $\mathbf{N}(\mathbf{x}_t \mid \mathbf{e}_{1:t})$ for the number of samples occupying state $\mathbf{x}_t$ after observations $\mathbf{e}_{1:t}$ have been processed, we therefore have

$$N(\mathbf{x}_t|\mathbf{e}_{1:t})/N = P(\mathbf{x}_t|\mathbf{e}_{1:t}) \quad (14.23)$$

for large N. Now we propagate each sample forward by sampling the state variables at $t + 1$, given the values for the sample at t. The number of samples reaching state $\mathbf{x}_{t+1}$ from each $\mathbf{x}_t$ is the transition probability times the population of $\mathbf{x}_t$; hence, the total number of samples reaching $\mathbf{x}_{t+1}$ is

$$N(\mathbf{x}_{t+1} \Big| \mathbf{e}_{1:t}) = \sum_{\mathbf{x}_t} P(\mathbf{x}_{t+1}|\mathbf{x}_t) N(\mathbf{x}_t|\mathbf{e}_{1:t}).$$

Now we weight each sample by its likelihood for the evidence at $t + 1$. A sample in state $\mathbf{x}_{t+i}$ receives weight $P(\mathbf{e}_{t+1} \mid \mathbf{x}_{t+1})$. The total weight of the samples in $\mathbf{x}_{t+1}$ after seeing $\mathbf{e}_{t+1}$ is therefore

$$W(\mathbf{x}_{t+1}|\mathbf{e}_{1:t+1}) = P(\mathbf{e}_{t+1}|\mathbf{x}_{t+1}) N(\mathbf{x}_{t+1}|\mathbf{e}_{1:t}).$$

Now for the resampling step. Since each sample is replicated with probability proportional to its weight, the number of samples in state $\mathbf{x}_{t+1}$ after resampling is proportional to the total weight in $\mathbf{x}_{t+1}$ before resampling:

$$\begin{aligned}
N(\mathbf{x}_{t+1}|\mathbf{e}_{1:t+1})/N &= \alpha\, W(\mathbf{x}_{t+1}|\mathbf{e}_{1:t+1}) \\
&= \alpha\, P(\mathbf{e}_{t+1}|\mathbf{x}_{t+1}) N(\mathbf{x}_{t+1}|\mathbf{e}_{1:t}) \\
&= \alpha\, P(\mathbf{e}_{t+1} \Big| \mathbf{x}_{t+1}) \sum_{\mathbf{x}_t} P(\mathbf{x}_{t+1}|\mathbf{x}_t) N(\mathbf{x}_t|\mathbf{e}_{1:t}) \\
&= \alpha\, N P(\mathbf{e}_{t+1} \Big| \mathbf{x}_{t+1}) \sum_{\mathbf{x}_t} P(\mathbf{x}_{t+1}|\mathbf{x}_t) P(\mathbf{x}_t|\mathbf{e}_{1:t}) \qquad \text{(by 14.23)} \\
&= \alpha' P(\mathbf{e}_{t+1} \Big| \mathbf{x}_{t+1}) \sum_{\mathbf{x}_t} P(\mathbf{x}_{t+1}|\mathbf{x}_t) P(\mathbf{x}_t|\mathbf{e}_{1:t}) \\
&= P(\mathbf{x}_{t+1}|\mathbf{e}_{1:t+1}) \quad \text{(by 14.5).}
\end{aligned}$$

Therefore the sample population after one update cycle correctly represents the forward message at time $t + 1$.

Particle filtering is *consistent*, therefore, but is it *efficient*? For many practical cases, it seems that the answer is yes: particle filtering seems to maintain a good approximation to the true posterior using a constant number of samples. Figure 14.19 shows that particle filtering does a good job on the grid-world localization problem with only a thousand samples. It also works on real-world problems: the algorithm supports thousands of applications in science and engineering. (Some references are given at the end of the chapter.) It handles combinations of discrete and continuous variables as well as nonlinear and non-Gaussian models for continuous variables. Under certain assumptions—in particular, that the probabilities in the transition and sensor models are bounded away from 0 and 1—it is also possible to prove that the approximation maintains bounded error with high probability, as the figure suggests.

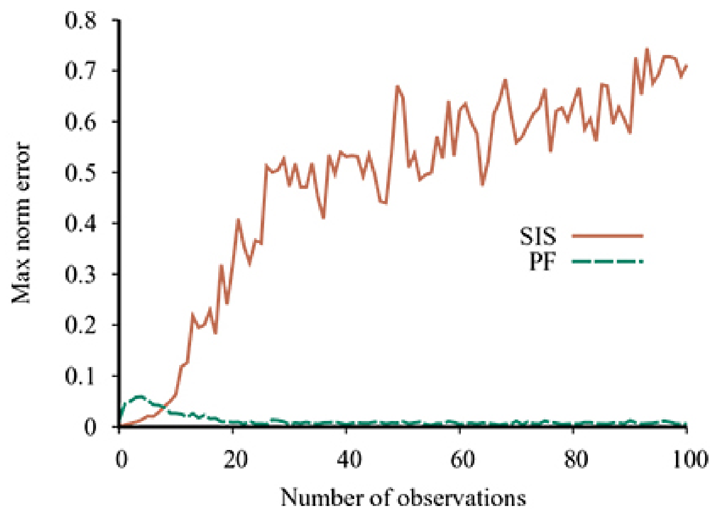


Figure 14.19 Max norm error in the grid-world location estimate (compared to exact inference) for likelihood weighting (sequential importance sampling) with 100,000 samples and particle filtering with 1,000 samples; data averaged over 50 runs.

The particle filtering algorithm does have weaknesses, however. Let's see how it performs for the vacuum world with dirt added. Recall from Section 14.3.2 that this increases the state space size by a factor of 2^{42}, making exact HMM inference infeasible. We want the robot to wander around and build a map of where the dirt is located. (This is a simple example of **simultaneous localization and mapping** or **SLAM,** which we cover in more depth in Chapter 26.) Let $Dirt_{i,t}$ mean that square i is dirty at time t and let $DirtSensor_t$ be true if and only if the robot detects dirt at time t. We'll assume that, in any given square, dirt persists with probability p, whereas a clean square becomes dirty with probability 1 – p (which means that each square is dirty half the time, on average). The robot has a dirt sensor for its current location; the sensor is accurate with probability 0.9. Figure 14.20 shows the DBN.

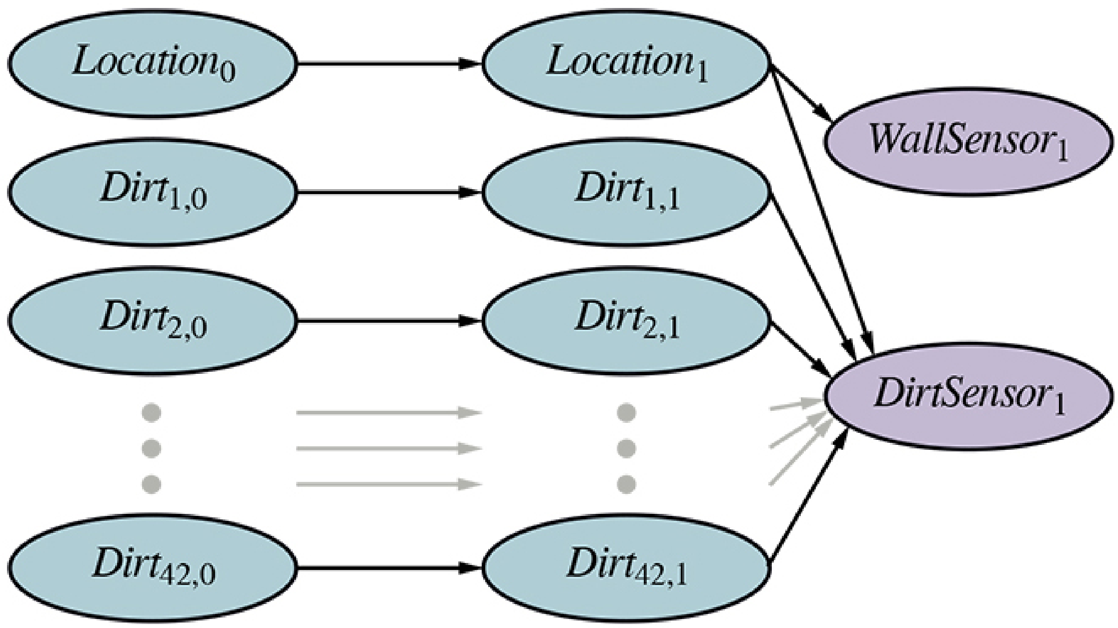


Figure 14.20 A dynamic Bayes net for simultaneous localization and mapping in the stochastic-dirt vacuum world. Dirty squares persist with probability p, and clean squares become dirty with probability 1 – p. The local dirt sensor is 90% accurate, for the square in which the robot is currently located.

For simplicity, we'll start by assuming that the robot has a perfect location sensor, rather than the noisy wall sensor. The algorithm's performance is shown in Figure 14.21(a), where its estimates for dirt are compared to the results of exact inference. (We'll see shortly how exact inference is possible.) For low values of the dirt persistence p, the error remains small— but this is no great achievement, because for every square the true posterior for dirt is close to 0.5 if the robot hasn't visited that square recently. For higher values of p, the dirt stays around longer, so visiting a square yields more useful information that is valid over a longer period. Perhaps surprisingly, particle filtering does *worse* for higher values of p. It fails completely when $p = 1$, even though that seems like the easiest case: the dirt arrives at time 0 and stays put forever, so after a few tours of the world, the robot should have a close-to-perfect dirt map. Why does particle filtering fail in this case?

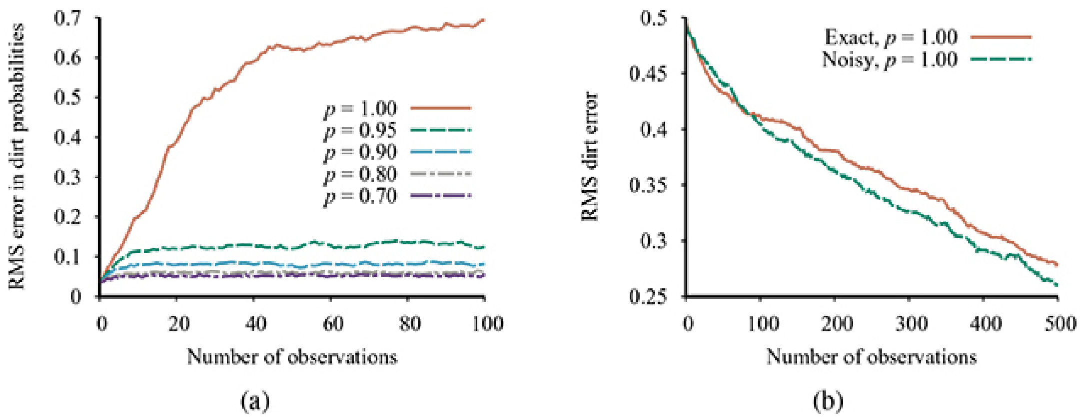


Figure 14.21 (a) Performance of the standard particle filtering algorithm with 1,000 particles, showing RMS error in marginal dirt probabilities compared to exact inference for different values of the dirt persistence p. (b) Performance of Rao-Blackwellized particle filtering (100 particles) compared to ground truth, for both exact location sensing and noisy wall sensing and with deterministic dirt. Data averaged over 20 runs.

It turns out that the theoretical condition requiring that "the probabilities in the transition and sensor models are strictly greater than 0 and less than 1" is more than mere mathematical pedantry. What happens is first each particle initially contains 42 guesses from $\mathbf{P}(\mathbf{X}_0)$ about which squares have dirt and which do not. Then, the state for each particle is projected forward in time according to the transition model. Unfortunately, the transition model for deterministic dirt is deterministic: the dirt stays exactly where it is. Thus, the initial guesses in each particle are never updated by the evidence.

The chance that the initial guesses are all correct is 2^{-42} or about 2×10^{-13}, so it is vanishingly unlikely that a thousand particles (or even a million particles) will include one with the correct dirt map. Typically, the best particle out of a thousand will get about 32 right and 10 wrong, and usually there will be only one such particle, or perhaps a handful. One of those best particles will come to dominate the total likelihood as time progresses and the diversity of the population of particles will collapse. Then, because all the particles agree on a single, incorrect map, the algorithm becomes convinced that that map is correct and never changes its mind.

Fortunately, the problem of simultaneous localization and mapping has a special structure: conditioned on the sequence of robot locations, the dirt statuses of the individual squares are independent (Exercise 14.RBPF). More specifically,

$$\mathbf{P}(Dirt_{1,0:t}, \ldots, Dirt_{42,0:t} | DirtSensor_{1:t}\ , WallSensor_{1:t}, Location_{1:t}$$
$$= \prod_i \mathbf{P}(Dirt_{i,0:t} | DirtSensor_{1:t}, Location_{1:t}). \tag{14.24}$$

This means it is useful to apply a statistical trick called **Rao-Blackwellization**, which is based on the simple idea that exact inference is always more accurate than sampling, even if it's only for a subset of the variables. (See Exercise 14.RAOB.) For the SLAM problem, we run particle filtering on the robot location and then, for each particle, we run exact HMM inference for each dirt square independently, conditioned on the location sequence in that particle. Each particle therefore contains a sampled location plus 42 exact marginal posteriors for the 42 squares—exact, that is, assuming that the hypothesized location trajectory followed by that particle is correct. This approach, called the **Rao-Blackwellized particle filter**, handles the case of deterministic dirt with no difficulty, gradually building an exact dirt map with either exact location sensing or noisy wall sensing, as shown in Figure 14.21(b).

In cases that do not satisfy the kind of conditional independence structure exemplified by Equation (14.24), Rao-Blackwellization is not applicable. The notes at the end of the chapter mention a number of algorithms that have been proposed to handle the general problem of filtering with static variables. None has the elegance and broad applicability of the particle filter, but several are effective in practice on certain classes of problems.

Summary

This chapter has addressed the general problem of representing and reasoning about probabilistic temporal processes. The main points are as follows:

- The changing state of the world is handled by using a set of random variables to represent the state at each point in time.
- Representations can be designed to (roughly) satisfy the **Markov property**, so that the future is independent of the past given the present. Combined with the assumption that the process is **time-homogeneous**, this greatly simplifies the representation.
- A temporal probability model can be thought of as containing a **transition model** describing the state evolution and a sensor model describing the observation process.
- The principal inference tasks in temporal models are **filtering** (**state estimation**), **prediction, smoothing**, and computing the **most likely explanation**. Each of these tasks can be achieved using simple, recursive algorithms whose run time is linear in the length of the sequence.
- Three families of temporal models were studied in more depth: **hidden Markov models**, **Kalman filters**, and **dynamic Bayesian networks** (which include the other two as special cases).
- Unless special assumptions are made, as in Kalman filters, exact inference with many state variables is intractable. In practice, the **particle filtering** algorithm and its descendants are an effective family of approximation algorithms.

Bibliographical and Historical Notes

Many of the basic ideas for estimating the state of dynamical systems came from the mathematician C. F. Gauss (1809), who formulated a deterministic least-squares algorithm for the problem of estimating orbits from astronomical observations. A. A. Markov (1913) developed what was later called the **Markov assumption** in his analysis of stochastic processes; he estimated a first-order Markov chain on letters from the text of *Eugene Onegin*. The general theory of Markov chains and their mixing times is covered by Levin *et al.* (2008).

Significant classified work on filtering was done during World War II by Wiener (1942) for continuous-time processes and by Kolmogorov (1941) for discrete-time processes. Although this work led to important technological developments over the next 20 years, its use of a frequency-domain representation made many calculations quite cumbersome. Direct state-space modeling of the stochastic process turned out to be simpler, as shown by Peter Swerling (1959) and Rudolf Kalman (1960). The latter paper described what is now known as the Kalman filter for forward inference in linear systems with Gaussian noise; Kalman's results had, however, been obtained previously by the Danish astronomer Thorvold Thiele (1880) and by the Russian physicist Ruslan Stratonovich (1959). After a visit to NASA Ames Research Center in 1960, Kalman saw the applicability of the method to the tracking of rocket trajectories, and the filter was later implemented for the Apollo missions.

Key results on smoothing were derived by Rauch *et al.* (1965), and the impressively named Rauch-Tung-Striebel smoother is still a standard technique today. Many early results are gathered in Gelb (1974). Bar-

Shalom and Fortmann (1988) give a more modern treatment with a Bayesian flavor, as well as many references to the vast literature on the subject. Chat- field (1989) and Box *et al.* (2016) cover the control theory approach to time series analysis.

The hidden Markov model and associated algorithms for inference and learning, including the forward-backward algorithm, were developed by Baum and Petrie (1966). The Viterbi algorithm first appeared in (Viterbi, 1967). Similar ideas also appeared independently in the Kalman filtering community (Rauch *et al.*, 1965).

The forward-backward algorithm was one of the main precursors of the general formulation of the EM algorithm (Dempster *et al.*, 1977); see also Chapter 21. Constant-space smoothing appears in Binder *et al.* (1997b), as does the divide-and-conquer algorithm developed in Exercise 14.ISLE. Constant-time fixed-lag smoothing for HMMs first appeared in Russell and Norvig (2003).

HMMs have found many applications in language processing (Charniak, 1993), speech recognition (Rabiner and Juang, 1993), machine translation (Och and Ney, 2003), computational biology (Krogh *et al.*, 1994; Baldi *et al.*, 1994), financial economics (Bhar and Hamori, 2004) and other fields. There have been several extensions to the basic HMM model: for example, the Hierarchical HMM (Fine *et al.*, 1998) and Layered HMM (Oliver *et al.*, 2004) introduce structure back into the model, replacing the single state variable of HMMs.

Dynamic Bayesian networks (DBNs) can be viewed as a sparse encoding of a Markov process and were first used in AI by Dean and Kanazawa (1989b), Nicholson and Brady (1992), and Kjaerulff (1992). The last work extends the HUGIN Bayes net system to accommodate dynamic Bayesian networks. The book by Dean and Wellman (1991) helped popularize DBNs and the probabilistic approach to planning and control within AI. Murphy (2002) provides a thorough analysis of DBNs.

Dynamic Bayesian networks have become popular for modeling a variety of complex motion processes in computer vision (Huang *et al.*, 1994; Intille and Bobick, 1999). Like HMMs, they have found applications in speech recognition (Zweig and Russell, 1998; Livescu *et al.*,2003), robot localization (Theocharous *et al.*, 2004), and genomics (Murphy and Mian, 1999; Li *et al.*, 2011). Other application areas include gesture analysis (Suk *et al.*, 2010), driver fatigue detection (Yang *et al.*, 2010), and urban traffic modeling (Hofleitner *et al.*, 2012).

The link between HMMs and DBNs, and between the forward-backward algorithm and Bayesian network propagation, was explicated by Smyth *et al.* (1997). A further unification with Kalman filters (and other statistical models) appears in Roweis and Ghahramani (1999). Procedures exist for learning the parameters (Binder *et al.*, 1997a; Ghahramani, 1998) and structures (Friedman *et al.*, 1998) of DBNs. **Continuous-time Bayesian networks** (Nodel- man *et al.*, 2002) are the discrete-state, continuous-time analog of DBNs, avoiding the need to choose a particular duration for time steps.

The first sampling algorithms for filtering (also called sequential Monte Carlo methods) were developed in the control theory community by Handschin and Mayne (1969), and the resampling idea that is the core of particle filtering appeared in a Russian control journal (Zarit- skii *et al.*, 1975). It was later reinvented in statistics as **sequential importance sampling with resampling,** or **SIR** (Rubin, 1988; Liu and Chen, 1998), in control theory as particle filtering (Gordon *et al.*, 1993; Gordon, 1994), in AI as **survival of the fittest** (Kanazawa *et al.*,1995), and in computer vision as **condensation** (Isard and Blake, 1996).

The paper by Kanazawa *et al.* (1995) includes an improvement called **evidence reversal** whereby the state at time t + 1 is sampled conditional on both the state at time t and the evidence at time t + 1. This allows the evidence to influence sample generation directly and was proved by Doucet (1997) and Liu and Chen (1998) to reduce the approximation error.

Particle filtering has been applied in many areas, including tracking complex motion patterns in video (Isard and Blake, 1996), predicting the stock market (de Freitas *et al.*, 2000), and diagnosing faults on planetary rovers (Verma *et al.*, 2004). Since its invention, tens of thousands of papers have been published on applications and variants of the algorithm. Scalable implementations on parallel hardware have become important; although one might think it straightforward to distribute N particles across up to N processor threads, the basic algorithm requires synchronized communication among threads for the resampling step (Hendeby *et al.*, 2010). The **particle cascade algorithm** (Paige *et al.*, 2015) removes the synchronization requirement, resulting in much faster parallel computation.

The **Rao-Blackwellized particle filter** is due to Doucet *et al.* (2000) and Murphy and Russell (2001); its application to practical localization and mapping problems in robotics is described in Chapter 26. Many other algorithms have been proposed to handle more general filtering problems with static or nearly-static variables, including the resample-move algorithm (Gilks and Berzuini, 2001), the Liu–West algorithm (Liu and West, 2001), the Storvik filter (Storvik, 2002), the extended parameter filter (Erol *et al.*, 2013), and the assumed parameter filter (Erol et *al.*, 2017). The latter is a hybrid of particle filtering with a much older idea called **assumed-density filter.** An assumed-density filter assumes that the posterior distribution over states at time t belongs to a particular finitely parameterized family; if the projection and update steps take it outside this family, the distribution is projected back to give the best approximation within the family. For DBNs, the Boyen-Koller algorithm (Boyen et *al.*, 1999) and the **factored frontier** algorithm (Murphy and Weiss, 2001) assume that the posterior distribution can be approximated well by a product of small factors.

MCMC methods (see Section 13.4.2) can be applied to the filtering problem; for example, Gibbs sampling can be applied directly to an unrolled DBN. The **particle MCMC** family of algorithms (Andrieu et *al.*, 2010; Lindsten et *al.*, 2014) combines MCMC on the unrolled temporal model with particle filtering to generate the MCMC proposals; although it provably converges to the correct posterior distribution in the general case (i.e., with both static and dynamic variables), it is an offline algorithm. To avoid the problem of increasing update times as the unrolled network grows, the **decayed MCMC** filter (Marthi et *al.*, 2002) prefers to sample more recent state variables, with a probability that decreases for variables further in the past.

The book by Doucet *et al.* (2001) collects many important papers on **sequential Monte Carlo** (SMC) algorithms, of which particle filtering is the most important instance. There are useful tutorials by Arulampalam *et al.* (2002) and Doucet and Johansen (2011). There are also several theoretical results concerning conditions under which SMC methods retain a bounded error indefinitely compared to the true posterior (Crisan and Doucet, 2002; Del Moral, 2004; Del Moral *et al.*, 2006).

---

[1] Uncertainty over *continuous* time can be modeled by **stochastic differential equations (SDEs)**. The models studied in this chapter can be viewed as discrete-time approximations to SDEs.

[2] The term “filtering” refers to the roots of this problem in early work on signal processing, where the problem is to filter out the noise in a signal by estimating its underlying properties.

[3] In particular, when tracking a moving object with inaccurate position observations, smoothing gives a smoother estimated trajectory than filtering—hence the name.

[4] If one picks an arbitrary day to be $t = 0$, then it makes sense to choose the prior $\mathbf{P}(Rain_0)$ to match the stationary distribution, which is why we picked (0.5,0.5) as the prior. Had we picked a different prior, the stationary distribution would still have worked out to (0.5,0.5).

[5] Notice that these are not quite the probabilities of the most likely paths to reach the states $\mathbf{X}_t$ given the evidence, which would be the conditional probabilities $\max_{\mathbf{x}_{1:t-1}} \mathbf{P}(\mathbf{x}_{1:t-1}, \mathbf{X}_t | \mathbf{e}_{1:t})$; but the two vectors are related by a constant factor $P(\mathbf{e}_{1:t})$. The difference is immaterial because the max operator doesn't care about constant factors. We get a slightly simpler recursion with $\mathbf{m}_{1:t}$ defined this way.

[6] The reader unfamiliar with basic operations on vectors and matrices might wish to consult Appendix A before proceeding with this section.

[7] Strictly speaking, a Gaussian distribution is problematic because it assigns nonzero probability to large negative charge levels. The **beta distribution** is sometimes a better choice for a variable whose range is restricted.

CHAPTER 15

MAKING SIMPLE DECISIONS

In which we see how an agent should make decisions so that it gets what it wants in an uncertain world—at least as much as possible and on average.

In this chapter, we fill in the details of how utility theory combines with probability theory to yield a decision-theoretic agent—an agent that can make rational decisions based on what it believes and what it wants. Such an agent can make decisions in contexts in which uncertainty and conflicting goals leave a logical agent with no way to decide. A goal-based agent has a binary distinction between good (goal) and bad (non-goal) states, while a decision-theoretic agent assigns a continuous range of values to states, and thus can more easily choose a better state even when no best state is available.

Section 15.1 introduces the basic principle of decision theory: the maximization of expected utility. Section 15.2 shows that the behavior of a rational agent can be modeled by maximizing a utility function. Section 15.3 discusses the nature of utility functions in more detail, and in particular their relation to individual quantities such as money. Section 15.4 shows how to handle utility functions that depend on several quantities. In Section 15.5, we describe the implementation of decision-making systems. In particular, we introduce a formalism called a **decision network** (also known as an **influence diagram**) that extends Bayesian networks by incorporating actions and utilities. Section 15.6 shows how a decision-theoretic agent can calculate the value of acquiring new information to improve its decisions.

While Sections 15.1–15.6 assume that the agent operates with a given, known utility function, Section 15.7 relaxes this assumption. We discuss the consequences of preference uncertainty on the part of the machine—the most important of which is deference to humans.

15.1 Combining Beliefs and Desires under Uncertainty

We begin with an agent that, like all agents, has to make a decision. It has available some actions a. There may be uncertainty about the current state, so we'll assume that the agent assigns a probability $P(s)$ to each possible current state s. There may also be uncertainty about the action outcomes; the transition model is given by $P(s' \mid s, a)$, the probability that action a in state s reaches state s'. Because we're primarily interested in the outcome s', we'll also use the abbreviated notation $P(\textsc{Result}(a) = s')$, the probability of reaching s' by doing a in the current state, whatever that is. The two are related as follows:

$$P(\text{RESULT}(a){=}s\prime) = \sum_{s} P(s)P(s\prime|s,a).$$

Decision theory, in its simplest form, deals with choosing among actions based on the desirability of their *immediate* outcomes; that is, the environment is assumed to be episodic in the sense defined on page 63. (This assumption is relaxed in Chapter 16.) The agent's preferences are captured by a **utility function,** $U(s)$, which assigns a single number to express the desirability of a state. The **expected utility** of an action given the evidence, $EU(a)$, is just the average utility value of the outcomes, weighted by the probability that the outcome occurs:

$$EU(a) = \sum_{s\prime} P(\text{RESULT}(a) = s\text{'})U(s\text{'}). \tag{15.1}$$

The principle of **maximum expected utility (MEU)** says that a rational agent should choose the action that maximizes the agent's expected utility:

$$action = \underset{a}{\operatorname{argmax}}\, EU(a).$$

In a sense, the MEU principle could be seen as a prescription for intelligent behavior. All an intelligent agent has to do is calculate the various quantities, maximize utility over its actions, and away it goes. But this does not mean that the AI problem is *solved* by the definition!

The MEU principle *formalizes* the general notion that an intelligent agent should "do the right thing," but does not *operationalize* that advice. Estimating the probability distribution $P(s)$ over possible states of the world, which folds into $P(\textsc{Result}(a) = s')$, requires perception, learning, knowledge representation, and inference. Computing $P(\textsc{Result}(a) = s')$ itself requires a causal model of the world. There may be many actions to consider, and computing the outcome utilities $U(s')$ may itself require further searching or planning because an agent may not know how good a state is until it knows where it can get to from that state. An AI system acting on behalf of a human may not know the human's true utility function, so there may be uncertainty about U. In summary, decision theory is not a panacea that solves the AI problem—but it does provide the beginnings of a basic mathematical framework that is general enough to define the AI problem.

The MEU principle has a clear relation to the idea of performance measures introduced in Chapter 2. The basic idea is simple. Consider the environments that could lead to an agent having a given percept history, and consider the different agents that we could design. *If an agent acts so as to maximize a utility function that correctly reflects the performance measure, then the agent will achieve the highest possible performance score (averaged over all the possible environments*). This is the central justification for the MEU principle itself. While the claim may seem tautological, it does in fact embody a very important transition from the external performance measure to an internal utility function. The performance measure gives a score for a history—a sequence of states. Thus it is applied retrospectively after an agent completes a sequence of actions. The utility function applies to the very next state, so it can be used to guide actions step by step.

15.2 The Basis of Utility Theory

Intuitively, the principle of Maximum Expected Utility (MEU) seems like a reasonable way to make decisions, but it is by no means obvious that it is the *only* rational way. After all, why should maximizing the *average* utility be so special? What's wrong with an agent that maximizes the weighted sum of the cubes of the possible utilities, or tries to minimize the worst possible loss? Could an agent act rationally just by expressing preferences between states, without giving them numeric values? Finally, why should a utility function with the required properties exist at all? We shall see.

15.2.1 Constraints on rational preferences

These questions can be answered by writing down some constraints on the preferences that a rational agent should have and then showing that the MEU principle can be derived from the constraints. We use the following notation to describe an agent's preferences:

$A \succ B$	the agent prefers A over B.
$A \sim B$	the agent is indifferent between A and B.
$A \succsim B$	the agent prefers A over B or is indifferent between them.

Now the obvious question is, what sorts of things are A and B? They could be states of the world, but more often than not there is uncertainty about what is really being offered. For example, an airline passenger who is offered "the pasta dish or the chicken" does not know what lurks beneath the tinfoil cover.[1] The pasta could be delicious or congealed, the chicken juicy or overcooked beyond recognition. We can think of the set of outcomes for each action as a **lottery**—think of each action as a ticket. A lottery L with possible outcomes S_1,..., S_n that occur with probabilities p_1,..., p_n is written

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

In general, each outcome S_i, of a lottery can be either an atomic state or another lottery. The primary issue for utility theory is to understand how preferences between complex lotteries are related to preferences between the underlying states in those lotteries. To address this issue we list six constraints that we require any reasonable preference relation to obey:

- **Orderability**: Given any two lotteries, a rational agent must either prefer one or else rate them as equally preferable. That is, the agent cannot avoid deciding. As noted on page 412, refusing to bet is like refusing to allow time to pass.

  Exactly one of $(A \succ B)$, $(B \succ A)$, or $(A \sim B)$ holds.

- **Transitivity:** Given any three lotteries, if an agent prefers A to B and prefers B to C, then the agent must prefer A to C.

$$(A \succ B) \wedge (B \succ C) \Rightarrow (A \succ C).$$

- **Continuity:** If some lottery B is between A and C in preference, then there is some probability p for which the rational agent will be indifferent between getting B for sure and the lottery that yields A with probability p and C with probability $1 - p$.

  $$A \succ B \succ C \Rightarrow \exists p\,[\mathrm{p}, A;\ 1 - p, C] \sim B.$$

- **Substitutability:** If an agent is indifferent between two lotteries A and B, then the agent is indifferent between two more complex lotteries that are the same except that B is substituted for A in one of them. This holds regardless of the probabilities and the other outcome(s) in the lotteries.

  $$A \sim B \Rightarrow [p, A;\ 1 - p, C] \sim [p, B;\ 1 - p, C].$$

  This also holds if we substitute $\succ$ for ~ in this axiom.

- **Monotonicity:** Suppose two lotteries have the same two possible outcomes, A and B. If an agent prefers A to B, then the agent must prefer the lottery that has a higher probability for A (and vice versa).

  $$A \succ B \Rightarrow (p > q \Leftrightarrow [p, A;\ 1 - p, B] \succ [q, A;\ 1 - q, B]).$$

- **Decomposability:** Compound lotteries can be reduced to simpler ones using the laws of probability. This has been called the “no fun in gambling” rule: as Figure 15.1(b) shows, it compresses two consecutive lotteries into a single equivalent lottery.[2]

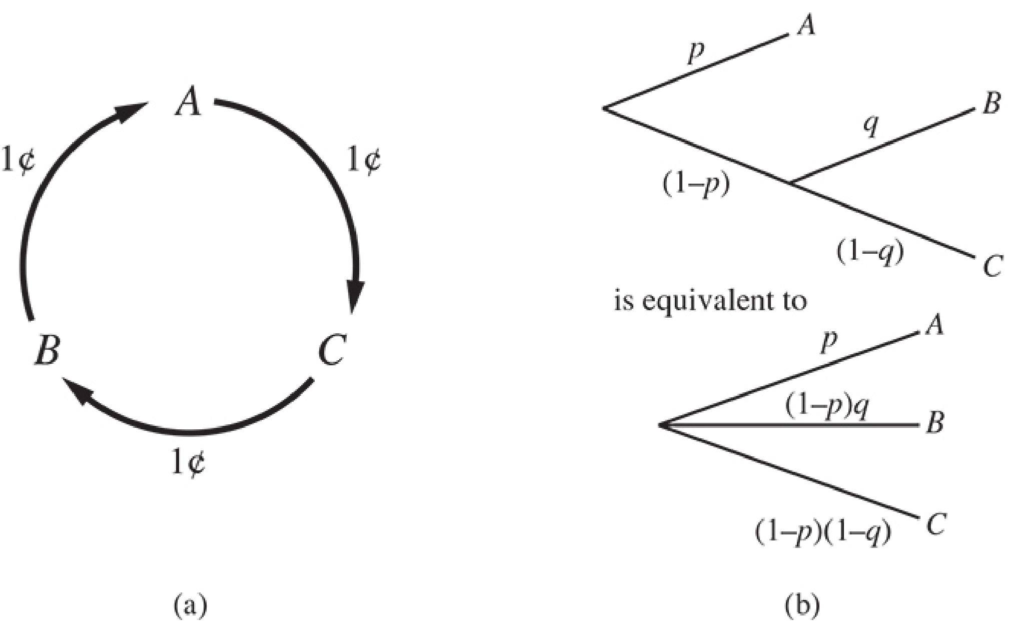


Figure 15.1 (a) Nontransitive preferences $A \succ B \succ C \succ A$ can result in irrational behavior: a cycle of exchanges each costing one cent. (b) The decomposability axiom.

$$[p, A;\ 1-p,\ [q, B;\ 1-q, C]]\ \sim\ [p, A;\ (1-p)q, B;\ (1-\ p)(1-q), C].$$

These constraints are known as the axioms of utility theory. Each axiom can be motivated by showing that an agent that violates it will exhibit patently irrational behavior in some situations. For example, we can motivate transitivity by making an agent with nontransitive preferences give us all its money. Suppose that the agent has the nontransitive preferences $A \succ B \succ C \succ A$, where A, B, and C are goods that can be freely exchanged. If the agent currently has A, then we could offer to trade C for A plus one cent. The agent prefers C, and so would be willing to make this trade. We could then offer to trade B for C, extracting another cent, and finally trade A for B. This brings us back where we started from, except that the agent has given us three cents (Figure 15.1(a)). We can keep going around the cycle until the agent has no money at all. Clearly, the agent has acted irrationally in this case.

15.2.2 Rational preferences lead to utility

Notice that the axioms of utility theory are really axioms about preferences—they say nothing about a utility function. But in fact from the axioms of utility we can derive the following consequences (for the proof, see von Neumann and Morgenstern, 1944):

- **Existence of Utility Function:** If an agent's preferences obey the axioms of utility, then there exists a function U such that $U(A) > U(B)$ if and only if A is preferred to B, and $U(A) = U(B)$ if and only if the agent is indifferent between A and B. That is,

  $$U(A) > U(B) \Leftrightarrow A \succ B \quad \text{and} \quad U(A) = U(B) \Leftrightarrow A \sim B.$$

- **Expected Utility of a Lottery**: The utility of a lottery is the sum of the probability of each outcome times the utility of that outcome.

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

In other words, once the probabilities and utilities of the possible outcome states are specified, the utility of a compound lottery involving those states is completely determined. Because the outcome of a nondeterministic action is a lottery, it follows that an agent can act rationally—that is, consistently with its preferences—only by choosing an action that maximizes expected utility according to Equation (15.1).

The preceding theorems establish that (assuming the constraints on rational preferences) a utility function *exists* for any rational agent. The theorems do not establish that the utility function is *unique*. It is easy to see, in fact, that an agent's behavior would not change if its utility function $U(S)$ were transformed according to

$$U'(S) = aU(S) + b, \qquad (15.2)$$

where a and b are constants and $a > 0$; a positive affine transformation.[3] This fact was noted in Chapter 6 (page 213) for two-player games of chance; here, we see that it applies to all kinds of decision scenarios.

As in game-playing, in a deterministic environment an agent needs only a preference ranking on states—the numbers don't matter. This is called a **value function** or **ordinal utility function**.

It is important to remember that the existence of a utility function that describes an agent's preference behavior does not necessarily mean that the agent is *explicitly* maximizing that utility function in its own deliberations. As we showed in Chapter 2, rational behavior can be generated in any number of ways. A rational agent might be implemented with a table lookup (if the number of possible states is small enough).

By observing a rational agent's behavior, an observer can learn about the utility function that represents what the agent is actually trying to achieve (even if the agent doesn't know it). We return to this point in Section 15.7.

15.3 Utility Functions

Utility functions map from lotteries to real numbers. We know they must obey the axioms of orderability, transitivity, continuity, substitutability, monotonicity, and decomposability. Is that all we can say about utility functions? Strictly speaking, that is it: an agent can have any preferences it likes. For example, an agent might prefer to have a prime number of dollars in its bank account; in which case, if it had \$16 it would give away \$3. This might be unusual, but we can't call it irrational. An agent might prefer a dented 1973 Ford Pinto to a shiny new Mercedes. The agent might prefer prime numbers of dollars only when it owns the Pinto, but when it owns the Mercedes, it might prefer more dollars to fewer. Fortunately, the preferences of real agents are usually more systematic and thus easier to deal with.

15.3.1 Utility assessment and utility scales

If we want to build a decision-theoretic system that helps a human make decisions or acts on his or her behalf, we must first work out what the human's utility function is. This process, often called **preference elicitation**, involves presenting choices to the human and using the observed preferences to pin down the underlying utility function.

Equation (15.2) says that there is no absolute scale for utilities, but it is helpful, nonetheless, to establish *some* scale on which utilities can be recorded and compared for any particular problem. A scale can be established by fixing the utilities of any two particular outcomes, just as we fix a temperature scale by fixing the freezing point and boiling point of water. Typically, we fix the utility of a "best possible prize" at $U(S) = u_{\top}$ and a "worst possible catastrophe" at $U(S) = u_{\perp}$. (Both of these should be finite.) **Normalized utilities** use a scale with $u_{\perp} = 0$ and $u_{\top} = 1$. With such a scale, an England fan might assign a utility of 1 to England winning the World Cup and a utility of 0 to England failing to qualify.

Given a utility scale between $u_\top$ and $u_\perp$, we can assess the utility of any particular prize S by asking the agent to choose between S and a **standard lottery** $[p, u_\top; (1-p), u_\perp]$. The probability p is adjusted until the agent is indifferent between S and the standard lottery. Assuming normalized utilities, the utility of S is given by p. Once this is done for each prize, the utilities for all lotteries involving those prizes are determined. Suppose, for example, we want to know how much our England fan values the outcome of England reaching the semi-final and then losing. We compare that outcome to a standard lottery with probability p of winning the trophy and probability $1 - p$ of an ignominious failure to qualify. If there is indifference at $p = 0.3$, then 0.3 is the value of reaching the semi-final and then losing.

In medical, transportation, environmental and other decision problems, people's lives are at stake. (Yes, there are things more important than England's fortunes in the World Cup.) In such cases, $u_\perp$ is the value assigned to immediate death (or in the really worst cases, many deaths). *Although nobody feels comfortable with putting a value on human life, it is a fact that tradeoffs on matters of life and death are made all the time*. Aircraft are given a complete overhaul at intervals, rather than after every trip. Cars are manufactured in a way that trades off costs against accident survival rates. We tolerate a level of air pollution that kills four million people a year.

Paradoxically, a refusal to put a monetary value on life can mean that life is *undervalued*. Ross Shachter describes a government agency that commissioned a study on removing asbestos from schools. The decision analysts performing the study assumed a particular dollar value for the life of a school-age child, and argued that the rational choice under that assumption was to remove the asbestos. The agency, morally outraged at the idea of setting the value of a life, rejected the report out of hand. It then decided against asbestos removal—implicitly asserting a lower value for the life of a child than that assigned by the analysts.

Currently several agencies of the U.S. government, including the Environmental Protection Agency, the Food and Drug Administration, and the Department of Transportation, use the **value of a statistical life** to determine the costs and benefits of regulations and interventions. Typical values in 2019 are roughly $10 million.

Some attempts have been made to find out the value that people place on their own lives. One common "currency" used in medical and safety analysis is the **micromort**, a one in a million chance of death. If you ask people how much they would pay to avoid a risk—for example, to avoid playing Russian roulette with a million-barreled revolver—they will respond with very large numbers, perhaps tens of thousands of dollars, but their actual behavior reflects a much lower monetary value for a micromort.

For example, in the UK, driving in a car for 230 miles incurs a risk of one micromort. Over the life of your car—say, 92,000 miles—that's 400 micromorts. People appear to be willing to pay about $12,000 more for a safer car that halves the risk of death. Thus, their car-buying action says they have a value of $60 per micromort. A number of studies have confirmed a figure in this range across many individuals and risk types. However, government agencies such as the U.S. Department of Transportation typically set a lower figure; they will spend only about $6 in road repairs per expected life saved. Of course, these calculations hold only for small risks. Most people won't agree to kill themselves, even for $60 million.

Another measure is the **QALY**, or quality-adjusted life year. Patients are willing to accept a shorter life expectancy to avoid a disability. For example, kidney patients on average are indifferent between living two years on dialysis and one year at full health.

15.3.2 The utility of money

Utility theory has its roots in economics, and economics provides one obvious candidate for a utility measure: money (or more specifically, an agent's total net assets). The almost universal exchangeability of money for all kinds of goods and services suggests that money plays a significant role in human utility functions.

It will usually be the case that an agent prefers more money to less, all other things being equal. We say that the agent exhibits a **monotonic preference** for more money. This does not mean that money behaves as a utility function, because it says nothing about preferences between *lotteries* involving money.

Suppose you have triumphed over the other competitors in a television game show. The host now offers you a choice: either you can take the \$1,000,000 prize or you can gamble it on the flip of a coin. If the coin comes up heads, you end up with nothing, but if it comes up tails, you get \$2,500,000. If you're like most people, you would decline the gamble and pocket the million. Are you being irrational?

Assuming the coin is fair, the **expected monetary value** (EMV) of the gamble is $\frac{1}{2}(\$0) + \frac{1}{2}(\$2,500,000) = \$1,250,000$, which is more than the original \$1,000,000. But that does not necessarily mean that accepting the gamble is a better decision. Suppose we use S_n to denote the state of possessing total wealth \$$n$, and that your current wealth is \$$k$. Then the expected utilities of the two actions of accepting and declining the gamble are

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

To determine what to do, we need to assign utilities to the outcome states. Utility is not directly proportional to monetary value, because the utility for your first million is very high (or so they say), whereas the utility for an additional million is smaller. Suppose you assign a utility of 5 to your current financial status (S_k), a 9 to the state $S_{k+\ 2,500,000}$, and an 8 to the state S_{k+}

$_{1{,}000{,}000}$. Then the rational action would be to decline, because the expected utility of accepting is only 7 (less than the 8 for declining). On the other hand, a billionaire would most likely have a utility function that is locally linear over the range of a few million more, and thus would accept the gamble.

In a pioneering study of actual utility functions, Grayson (1960) found that the utility of money was almost exactly proportional to the *logarithm* of the amount. (This idea was first suggested by Bernoulli (1738); see Exercise 15.STPT.) One particular utility curve, for a certain Mr. Beard, is shown in Figure 15.2(a). The data obtained for Mr. Beard's preferences are consistent with a utility function

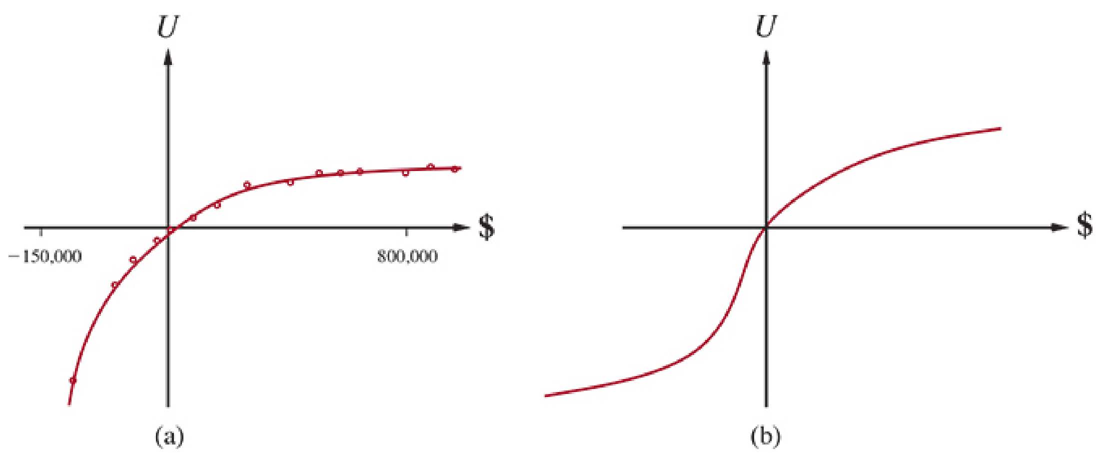


Figure 15.2 The utility of money. (a) Empirical data for Mr. Beard over a limited range. (b) A typical curve for the full range.

$$U(S_{k+n}) = -263.31 + 22.09 \log(n + 150{,}000)$$

for the range between $n = -\$150,000$ and $n = \$800,000$.

We should not assume that this is the definitive utility function for monetary value, but it is likely that most people have a utility function that is concave for positive wealth. Going into debt is bad, but preferences between different levels of debt can display a reversal of the concavity associated with positive wealth. For example, someone already \$10,000,000 in debt might well accept a gamble on a fair coin with a gain of \$10,000,000 for heads and a loss of \$20,000,000 for tails.[4] This yields the S-shaped curve shown in Figure 15.2(b).

If we restrict our attention to the positive part of the curves, where the slope is decreasing, then for any lottery L, the utility of being faced with that lottery is less than the utility of being handed the expected monetary value of the lottery as a sure thing:

$$U(L) < U(S_{EMV(L)}).$$

That is, agents with curves of this shape are **risk-averse**: they prefer a sure thing with a payoff that is less than the expected monetary value of a gamble. On the other hand, in the "desperate" region at large negative wealth in Figure 15.2(b), the behavior is **risk-seeking**. The value an agent will accept in lieu of a lottery is called the **certainty equivalent** of the lottery. Studies have shown that most people will accept about \$400 in lieu of a gamble that gives \$1000 half the time and \$0 the other half—that is, the certainty equivalent of the lottery is \$400, while the EMV is \$500.

The difference between the EMV of a lottery and its certainty equivalent is called the **insurance premium**. Risk aversion is the basis for the insurance industry, because it means that insurance premiums are positive. People would rather pay a small insurance premium than gamble the price of their house against the chance of a fire. From the insurance company's point of view, the price of the house is very small compared with the firm's total reserves. This means that the insurer's utility curve is approximately linear over such a small region, and the gamble costs the company almost nothing.

Notice that for *small* changes in wealth relative to the current wealth, almost any curve will be approximately linear. An agent that has a linear curve is said to be **risk-neutral**. For gambles with small sums, therefore, we expect risk neutrality. In a sense, this justifies the simplified procedure that proposed small gambles to assess probabilities and to justify the axioms of probability in Section 12.2.3.

15.3.3 Expected utility and post-decision disappointment

The rational way to choose the best action, $a*$, is to maximize expected utility:

$$a^* = \operatorname*{argmax}_a EU(a).$$

If we have calculated the expected utility correctly according to our probability model, and if the probability model correctly reflects the underlying stochastic processes that generate the outcomes, then, on average, we will get the utility we expect if the whole process is repeated many times.

In reality, however, our model usually oversimplifies the real situation, either because we don't know enough (e.g., when making a complex investment decision) or because the computation of the true expected utility is too difficult (e.g., when making a move in backgammon, needing to take into account all possible future dice rolls). In that case, we are really working with *estimates* $\widehat{EU}(a)$ of the true expected utility. We will assume, kindly perhaps, that the estimates are **unbiased**—that is, the expected value of the error, $E(\widehat{EU}(a) - EU(a))$, is zero. In that case, it still seems reasonable to choose the action with the highest estimated utility and to expect to receive that utility, on average, when the action is executed.

Unfortunately, the real outcome will usually be significantly *worse* than we estimated, even though the estimate was unbiased! To see why, consider a decision problem in which there are k choices, each of which has true estimated utility of 0. Suppose that the error in each utility estimate is independent and has a unit normal distribution—that is, a Gaussian with zero mean and standard deviation of 1, shown as the bold curve in Figure 15.3. Now, as we actually start to generate the estimates, some of the errors will be negative (pessimistic) and some will be positive (optimistic). Because we select the action with the *highest* utility estimate, we are favoring the overly optimistic estimates, and that is the source of the bias.

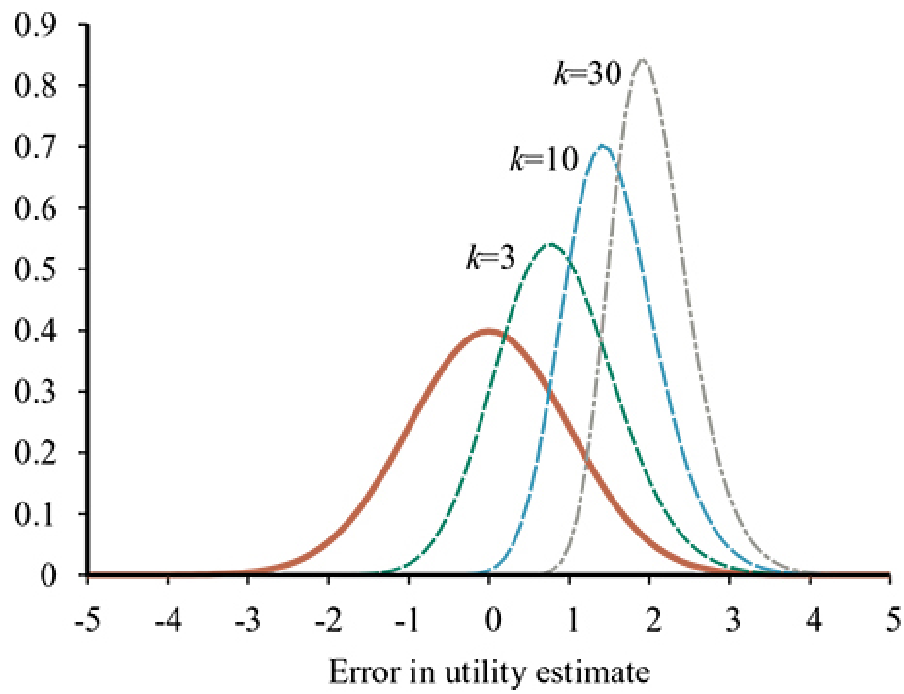


Figure 15.3 Unjustified optimism caused by choosing the best of k options: we assume that each option has a true utility of 0 but a utility estimate that is distributed according to a unit normal (brown

curve). The other curves show the distributions of the maximum of k estimates for k = 3, 10, and 30.

It is a straightforward matter to calculate the distribution of the maximum of the k estimates and hence quantify the extent of our disappointment. (This calculation is a special case of computing an **order statistic,** the distribution of any particular ranked element of a sample.) Suppose that each estimate X_i has a probability density function $f(x)$ and cumulative distribution $F(x)$. (As explained in Appendix A, the cumulative distribution F measures the probability that the cost is less than or equal to any given amount—that is, it integrates the original density f.) Now let X^* be the largest estimate, i.e., $\max\{X_1,...,X_k\}$. Then the cumulative distribution for X^* is

$$\begin{aligned} P(\max\{X_1,\ldots,X_k\} \leq x) &= P(X_1 \leq x,\ldots,X_k \leq x) \\ &= P(X_1 \leq x)\ldots P(X_k \leq x) = F(x)^k. \end{aligned}$$

The probability density function is the derivative of the cumulative distribution function, so the density for X^*, the maximum of k estimates, is

$$P(x) = \frac{d}{dx}(F(x)^k) = kf(x)(F(x))^{k-1}.$$

These densities are shown for different values of k in Figure 15.3 for the case where $f(x)$ is the unit normal. For k = 3, the density for X^* has a mean around 0.85, so the average disappointment will be about 85% of the standard deviation in the utility estimates. With more choices, extremely optimistic estimates are more likely to arise: for k = 30, the disappointment will be around twice the standard deviation in the estimates.

This tendency for the estimated expected utility of the best choice to be too high is called the **optimizer's curse** (Smith and Winkler, 2006). It afflicts even the most seasoned decision analysts and statisticians. Serious manifestations include believing that an exciting new drug that has cured 80%

of patients in a trial will cure 80% of patients (it's been chosen from k = thousands of candidate drugs) or that a mutual fund advertised as having above-average returns will continue to have them (it's been chosen to appear in the advertisement out of k = dozens of funds in the company's overall portfolio). It can even be the case that what appears to be the best choice may not be, if the variance in the utility estimate is high: a drug that has cured 9 of 10 patients and has been selected from thousands tried is probably *worse* than one that has cured 800 of 1000.

The optimizer's curse crops up everywhere because of the ubiquity of utility-maximizing selection processes, so taking the utility estimates at face value is a bad idea. We can avoid the curse with a Bayesian approach that uses an explicit probability model $\mathbf{P}(\widehat{EU} \mid EU)$ of the error in the utility estimates. Given this model and a prior on what we might reasonably expect the utilities to be, we treat the utility estimate as evidence and compute the posterior distribution for the true utility using Bayes' rule.

15.3.4 Human judgment and irrationality

Decision theory is a **normative theory:** it describes how a rational agent *should* act. A **descriptive theory**, on the other hand, describes how actual agents—for example, humans—really do act. The application of economic theory would be greatly enhanced if the two coincided, but there appears to be some experimental evidence to the contrary. The evidence suggests that humans are "predictably irrational" (Ariely, 2009).

The best-known problem is the Allais paradox (Allais, 1953). People are given a choice between lotteries A and B and then between C and D, which have the following prizes:

$$\begin{array}{ll} A: \ \ 80\% \text{ chance of } \$4000 & \qquad C: 20\% \text{ chance of } \$4000 \\ B: 100\% \text{ chance of } \$3000 & \qquad D: 25\% \text{ chance of } \$3000 \end{array}$$

Most people consistently prefer B over A (taking the sure thing), and C over D (taking the higher EMV). The normative analysis disagrees! We can see this most easily if we use the freedom implied by Equation (15.2) to set $U(\$0) = 0$. In that case, then $B \succ A$ implies that $U(\$3000) > 0.8\, U(\$4000)$, whereas $C \succ D$ implies exactly the reverse. In other words, there is no utility function that is consistent with these choices.

One explanation for the apparently irrational preferences is the **certainty effect** (Kahneman and Tversky, 1979): people are strongly attracted to gains that are certain. There are several reasons why this may be so.

First, people may prefer to reduce their computational burden; by choosing certain outcomes, they don't have to compute with probabilities. But the effect persists even when the computations involved are very easy ones.

Second, people may distrust the legitimacy of the stated probabilities. I trust that a coin flip is roughly 50/50 if I have control over the coin and the flip, but I may distrust the result if the flip is done by someone with a vested interest in the outcome.[5] In the presence of distrust, it might be better to go for the sure thing.[6]

Third, people may be accounting for their emotional state as well as their financial state. People know they would experience regret if they gave up a certain reward (B) for an 80% chance at a higher reward and then lost.

In other words, if A is chosen, there is a 20% chance of getting no money *and feeling like a complete idiot*, which is worse than just getting no money. So perhaps people who choose B over A and C over D are not irrational; they are willing to give up $200 of EMV to avoid a 20% chance of feeling like an idiot.

A related problem is the Ellsberg paradox. Here the prizes are fixed, but the probabilities are underconstrained. Your payoff will depend on the color of a ball chosen from an urn. You are told that the urn contains 1/3 red balls, and 2/3 either black or yellow balls, but you don't know how many black and how many yellow. Again, you are asked whether you prefer lottery *A* or *B*; and then *C* or *D*:

$$\begin{array}{ll} A: \$100 \text{ for a red ball} & C: \$100 \text{ for a red or yellow ball} \\ B: \$100 \text{ for a black ball} & D: \$100 \text{ for a black or yellow ball}: \end{array}$$

It should be clear that if you think there are more red than black balls then you should prefer *A* over *B* and *C* over *D*; if you think there are fewer red than black you should prefer the opposite. But it turns out that most people prefer *A* over *B* and also prefer *D* over *C*, even though there is no state of the world for which this is rational. It seems that people have **ambiguity aversion:** A gives you a 1/3 chance of winning, while *B* could be anywhere between 0 and 2/3. Similarly, *D* gives you a 2/3 chance, while *C* could be anywhere between 1/3 and 3/3. Most people elect the known probability rather than the unknown unknowns.

Yet another problem is that the exact wording of a decision problem can have a big impact on the agent's choices; this is called the **framing effect**. Experiments show that people like a medical procedure that is described as having a "90% survival rate" about twice as much as one described as having a "10% death rate," even though these two statements mean exactly the same thing. This discrepancy in judgment has been found in multiple experiments and is about the same whether the subjects are patients in a clinic, statistically sophisticated business school students, or experienced doctors.

People feel more comfortable making *relative* utility judgments rather than absolute ones. I may have little idea how much I might enjoy the various wines offered by a restaurant. The restaurant takes advantage of this by offering a \$200 bottle that nobody will buy, but which serves to skew upward the customer's estimate of the value of all wines, making a \$55 bottle seem like a bargain. This is called the **anchoring effect**.

If human informants insist on contradictory preference judgments, there is nothing that automated agents can do to be consistent with them. Fortunately, preference judgments made by humans are often open to revision in the light of further consideration. Paradoxes like the Allais and Ellsberg paradoxes are greatly reduced (but not eliminated) if the choices are explained better. In work at the Harvard Business School on assessing the utility of money, Keeney and Raiffa (1976, p. 210) found the following:

> Subjects tend to be too risk-averse in the small and therefore ... the fitted utility functions exhibit unacceptably large risk premiums for lotteries with a large spread. ... Most of the subjects, however, can reconcile their inconsistencies and feel that they have learned an important lesson about how they want to behave. As a consequence, some subjects cancel their automobile collision insurance and take out more term insurance on their lives.

The evidence for human irrationality is also questioned by researchers in the field of **evolutionary psychology**, who point to the fact that our brain's decision-making mechanisms did not evolve to solve word problems with probabilities and prizes stated as decimal numbers. Let us grant, for the sake of argument, that the brain has built-in neural mechanisms for computing with probabilities and utilities, or something functionally equivalent. If so, the required inputs would be obtained through accumulated experience of outcomes and rewards rather than through linguistic presentations of numerical values.

It is far from obvious that we can directly access the brain's built-in neural mechanisms by presenting decision problems in linguistic/numerical form. The very fact that different wordings of the *same decision problem* elicit different choices suggests that the decision problem itself is not getting through. Spurred by this observation, psychologists have tried presenting problems in uncertain reasoning and decision making in "evolutionarily appropriate" forms; for example, instead of saying "90% survival rate," the experimenter might show 100 stick-figure animations of the operation, where the patient dies in 10 of them and survives in 90. With decision problems posed in this way, people's behavior seems to be much closer to the standard of rationality.

15.4 Multiattribute Utility Functions

Decision making in the field of public policy involves high stakes, in both money and lives. For example, in deciding what levels of harmful emissions to allow from a power plant, policy makers must weigh the prevention of death and disability against the benefit of the power and the economic burden of mitigating the emissions. Picking a site for a new airport requires consideration of the disruption caused by construction; the cost of land; the distance from centers of population; the noise of flight operations; safety issues arising from local topography and weather conditions; and so on. Problems like these, in which outcomes are characterized by two or more attributes, are handled by **multiattribute utility theory**. In essence, it's the theory of comparing apples to oranges.

Let the attributes be $\mathbf{X} = X_1,...,X_n$ and let $\mathbf{x} = \langle x_1,...,x_n \rangle$ be a complete vector of assignments, where each x_i is either a numeric value or a discrete value with an assumed ordering on values. The analysis is easier if we arrange it so that higher values of an attribute always correspond to higher utilities: utilities are monotonically increasing. That means that we can't use, say, the number of deaths, d as an attribute; we would have to use $-d$. It also means that we can't use the room temperature, t, as an attribute. If the utility function for temperature has a peak at 70°F and falls off monotonically on either side, then we could split the attribute into two pieces. We could use $t - 70$ to measure whether the room is warm enough, and $70 - t$ to measure whether it is cool enough; both of these attributes would be monotonically increasing until they reach their maximum utility value at 0; the utility curve is flat from that point on, meaning that you dont't get any more "warm enough" above 70°F, nor any more "cool enough" below 70°F.

The attributes in the airport problem could be:

- *Throughput,* measured by the number of flights per day;
- *Safety,* measured by minus the expected number of deaths per year;
- *Quietness,* measured by minus the number of people living under the flight paths;
- *Frugality,* measured by the negative cost of construction.

We begin by examining cases in which decisions can be made *without* combining the attribute values into a single utility value. Then we look at cases in which the utilities of attribute combinations can be specified very concisely.

15.4.1 Dominance

Suppose that airport site S_1 costs less, generates less noise pollution, and is safer than site S_2. One would not hesitate to reject S_2. We then say that there is **strict dominance** of S_1 over S_2. In general, if an option is of lower value on all attributes than some other option, it need not be considered further. Strict dominance is often very useful in narrowing down the field of choices to the real contenders, although it seldom yields a unique choice. Figure 15.4(a) shows a schematic diagram for the two-attribute case.

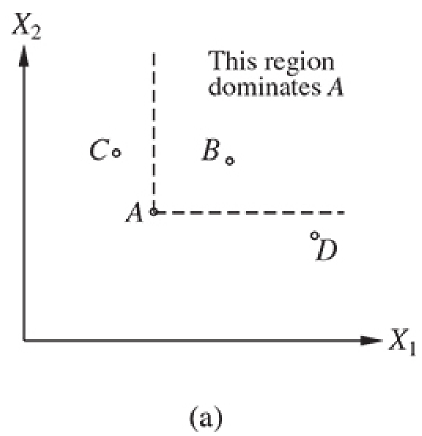


Figure 15.4 Strict dominance. (a) Deterministic: Option A is strictly dominated by B but not by C or D. (b) Uncertain: A is strictly dominated by B but not by C.

That is fine for the deterministic case, in which the attribute values are known for sure. What about the general case, where the outcomes are uncertain? A direct analog of strict dominance can be constructed, where, despite the uncertainty, all possible concrete outcomes for S_1 strictly dominate all possible outcomes for S_2. (See Figure 15.4(b).) Of course, this will probably occur even less often than in the deterministic case.

Fortunately, there is a more useful generalization called **stochastic dominance**, which occurs very frequently in real problems. Stochastic dominance is easiest to understand in the context of a single attribute. Suppose we believe that the cost of placing the airport at S_1 is uniformly distributed between \$2.8 billion and \$4.8 billion and that the cost at S_2 is uniformly distributed between \$3 billion and \$5.2 billion. Define the *Frugality* attribute to be the negative cost. Figure 15.5(a) shows the distributions for the frugality of sites S_1 and S_2. Then, given only the information that the more frugal choice is better (all other things being equal), we can say that S_1 stochastically dominates S_2 (i.e., S_2 can be discarded). It is important to note that this does *not* follow from comparing the expected costs. For example, if we knew the cost of S_1 to be *exactly* \$3.8 billion, then we would be *unable* to make a decision without additional information on the utility of money. (It might seem odd that *more* information on the cost of S_1 could make the agent *less* able to decide. The paradox is resolved by noting that in the absence of exact cost information, the decision is easier to make but is more likely to be wrong.)

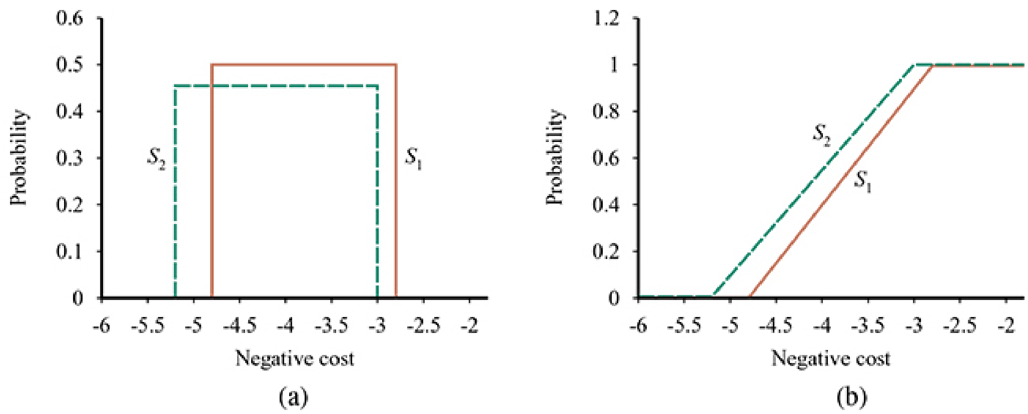


Figure 15.5 Stochastic dominance. (a) S_1 stochastically dominates S_2 on frugality (negative cost). (b) Cumulative distributions for the frugality of S_1 and S_2.

The exact relationship between the attribute distributions needed to establish stochastic dominance is best seen by examining the cumulative distributions, shown in Figure 15.5(b). If the cumulative distribution for S_1 is always to the right of the cumulative distribution for S_2, then, stochastically speaking, S_1 is cheaper than S_2. Formally, if two actions A_1 and A_2 lead to probability distributions $p_1(x)$ and $p_2(x)$ on attribute X, then A_1 stochastically dominates A_2 on X if

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

The relevance of this definition to the selection of optimal decisions comes from the following property: *if A_1 stochastically dominates A_2, then for any monotonically nondecreasing utility function $U(x)$, the expected utility of A_1 is at least as high as the expected utility of A_2.* To see why this is true, consider the two expected utilities, $\int p_1(x)U(x)dx$ and $\int p_2(x)U(x)dx$. Initially, it's not obvious why the first integral is bigger than the second, given that the stochastic dominance condition has a p_1-integral that is smaller than the p_2-integral.

Instead of thinking about the integral over x, however, think about the integral over y, the cumulative probability, as shown in Figure 15.5(b). For any value of y, the corresponding value of x (and hence of $U(x)$) is bigger for S_1 than for S_2; so if we integrate a bigger quantity over the whole range of y, we are bound to get a bigger result. Formally, it's just a substitution of $y = P_1(x)$ in the integral for S_1's expected value and $y = P_2(x)$ in the integral for S_2's. With these substitutions, we have $dy = \frac{d}{dx}(P_1(x))dx = p_1(x)dx$ for S_1 and $dy = p_2(x)dx$ for S_2, hence

$$\int_{-\infty}^{\infty} p1(x)U(x)dx = \int_{0}^{1} U(P_1^{-1}(y))dy \geq \int_{0}^{1} U(P_2^{-1}(y))dy = \int_{-\infty}^{\infty} p_2(x)U(x)dx.$$

This inequality allows us to prefer A_1 to A_2 in a single-attribute problem. More generally, if an action is stochastically dominated by another action on *all* attributes in a multiattribute problem, then it can be discarded.

The stochastic dominance condition might seem rather technical and perhaps not so easy to evaluate without extensive probability calculations. In fact, it can be decided very easily in many cases. For example, would you rather fall head-first onto concrete from 3 millimeters or 3 meters? Assuming you chose 3 millimeters—good choice! Why is it necessarily a better decision? There is a good deal of uncertainty about the degree of damage you will incur in both cases; but for any given level of damage, the probability that you'll incur at least that level of damage is higher when falling from 3 meters than from 3 millimeters. In other words, 3 millimeters stochastically dominates 3 meters on the *Safety* attribute.

This kind of reasoning comes as second nature to humans; it's so obvious we don't even think about it. Stochastic domination abounds in the airport problem too. Suppose, for example, that the construction transportation cost depends on the distance to the supplier. The cost itself is uncertain, but the greater the distance, the greater the cost. If S_1 is closer than S_2, then S_1 will dominate S_2 on frugality. Although we will not present them here, algorithms exist for propagating this kind of qualitative information among uncertain variables in **qualitative probabilistic networks**, enabling a system to make rational decisions based on stochastic dominance, without using any numeric values.

15.4.2 Preference structure and multiattribute utility

Suppose we have n attributes, each of which has d distinct possible values. To specify the complete utility function $U(x_1,...,x_n)$, we need d^n values in the worst case. Multiattribute utility theory aims to identify additional structure in human preferences so that we don't need to specify all d^n values individually. Having identified some regualarity in preference behavior, we then derive **representation theorems** to show that an agent with a certain kind of preference structure has a utility function

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

where F is (we hope) a simple function such as addition. Notice the similarity to the use of Bayesian networks to decompose the joint probability of several random variables.

As an example, suppose each x_i is the amount of money the agent has in a particular currency: dollars, euros, marks, lira, etc. The f_i functions could then convert each amount into a common currency, and F would then be simply addition.

Preferences without uncertainty

Let us begin with the deterministic case. On page 522 we noted that for deterministic environments, the agent has a value function, which we write here as $V(x_1,...,x_n)$; the aim is to represent this function concisely. The basic regularity that arises in deterministic preference structures is called **preference independence**. Two attributes X_1 and X_2 are preferentially independent of a third attribute X_3 if the preference between outcomes $\langle x_1, x_2, x_3 \rangle$ and $\langle x'_1, x'_2, x_3 \rangle$ does not depend on the particular value x_3 for attribute X_3.

Going back to the airport example, where we have (among other attributes) *Quietness, Frugality,* and *Safety* to consider, one may propose that *Quietness* and *Frugality* are preferentially independent of *Safety*. For example, if we prefer an outcome with 20,000 people residing in the flight path and a construction cost of $4 billion over an outcome with 70,000 people residing in the flight path and a cost of $3.7 billion when the safety level is 0.006 deaths per billion passenger miles in both cases, then we would have the same preference when the safety level is 0.012 or 0.003; and the same independence would hold for preferences between any other pair of values for *Quietness* and *Frugality*. It is also apparent that *Frugality* and *Safety* are preferentially independent of *Quietness* and that *Quietness* and *Safety* are preferentially independent of *Frugality*.

We say that the set of attributes {*Quietness, Frugality, Safety*} exhibits **mutual preferential independence (MPI)**. MPI says that, whereas each attribute may be important, it does not affect the way in which one trades off the other attributes against each other.

Mutual preferential independence is a complicated name, but it leads to a simple form for the agent's value function (Debreu, 1960): *If attributes X_1,..., X_n are mutually preferentially independent, then the agent's preferences can be represented by a value function*

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

where each V_i refers only to the attribute X_i. For example, it might well be the case that the airport decision can be made using a value function

$$V(quietness,\ frugality,\ safety) \ = \ quietness \times 10^4 + frugality + safety \times 10^{12}.$$

A value function of this type is called an **additive value function**. Additive functions are an extremely natural way to describe an agent's preferences and are valid in many real-world situations. For *n* attributes, assessing an additive value function requires assessing *n* separate one-dimensional value functions rather than one n-dimensional function; typically, this represents an exponential reduction in the number of preference experiments that are needed. Even when MPI does not strictly hold, as might be the case at extreme values of the attributes, an additive value function might still provide a good approximation to the agent's preferences. This is especially true when the violations of MPI occur in portions of the attribute ranges that are unlikely to occur in practice.

To understand MPI better, it helps to look at cases where it *doesn't* hold. Suppose you are at a medieval market, considering the purchase of some hunting dogs, some chickens, and some wicker cages for the chickens. The hunting dogs are very valuable, but if you don't have enough cages for the chickens, the dogs will eat the chickens; hence, the tradeoff between dogs and chickens depends strongly on the number of cages, and MPI is violated. The existence of these kinds of interactions among various attributes makes it much harder to assess the overall value function.

Preferences with uncertainty

When uncertainty is present in the domain, we also need to consider the structure of preferences between lotteries and to understand the resulting properties of utility functions, rather than just value functions. The mathematics of this problem can become quite complicated, so we present just one of the main results to give a flavor of what can be done.

The basic notion of **utility independence** extends preference independence to cover lotteries: a set of attributes **X** is utility independent of a set of attributes **Y** if preferences between lotteries on the attributes in **X** are independent of the particular values of the attributes in **Y**. A set of attributes is **mutually utility independent** (MUI) if each of its subsets is utility-independent of the remaining attributes. Again, it seems reasonable to propose that the airport attributes are MUI.

MUI implies that the agent's behavior can be described using a **multiplicative utility function** (Keeney, 1974). The general form of a multiplicative utility function is best seen by looking at the case for three attributes. For conciseness, we use U_i to mean $U_i(x_i)$:

$$\begin{aligned} U \ = \ & k_1U_1 + k_2U_2 + k_3U_3 + k_1k_2U_1U_2 + k_2k_3U_2U_3 + k_3k_1U_3U_1 \\ & + k_1k_2k_3U_1U_2U_3\ . \end{aligned}$$

Although this does not look very simple, it contains just three single-attribute utility functions and three constants. In general, an *n*-attribute problem exhibiting MUI can be modeled using *n* single-attribute utilities and *n* constants. Each of the single-attribute utility functions can be developed independently of the other attributes, and this combination will be guaranteed to generate the correct overall preferences. Additional assumptions are required to obtain a purely additive utility function.

15.5 Decision Networks

In this section, we look at a general mechanism for making rational decisions. The notation is often called an **influence diagram** (Howard and Matheson, 1984), but we will use the more descriptive term **decision network**. Decision networks combine Bayesian networks with additional node types for actions and utilities. We use the problem of picking an airport site as an example.

15.5.1 Representing a decision problem with a decision network

In its most general form, a decision network represents information about the agent's current state, its possible actions, the state that will result from the agent's action, and the utility of that state. It therefore provides a substrate for implementing utility-based agents of the type first introduced in Section 2.4.5. Figure 15.6 shows a decision network for the airport-siting problem. It illustrates the three types of nodes used:

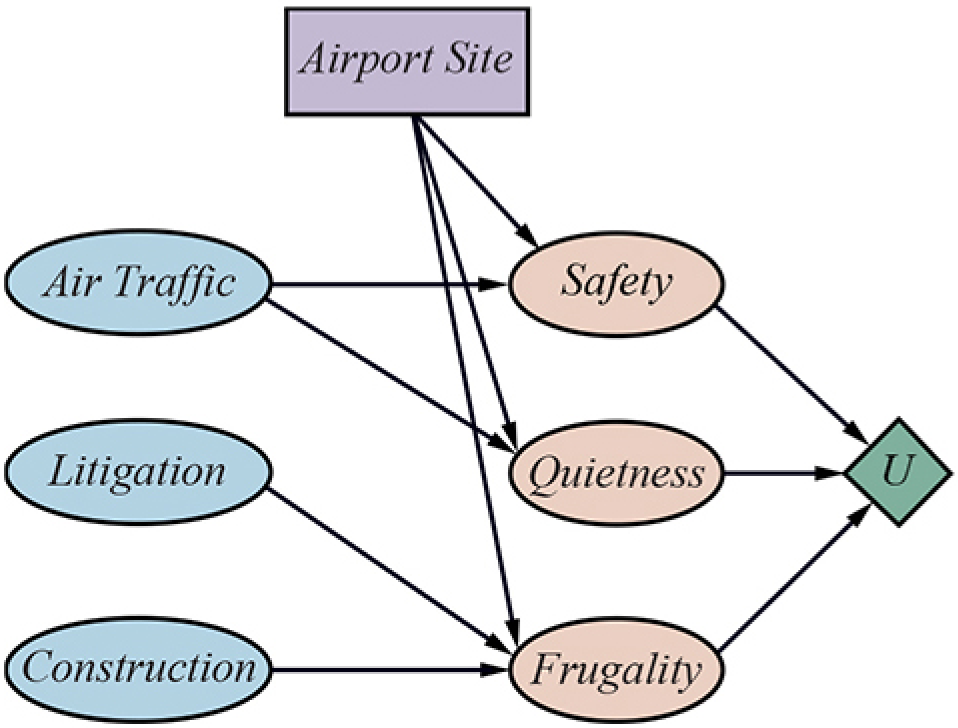


Figure 15.6 A decision network for the airport-siting problem.

- **Chance nodes** (ovals) represent random variables, just as they do in Bayesian networks. The agent could be uncertain about the construction cost, the level of air traffic and the potential for litigation, and the *Safety*, *Quietness,* and total *Frugality* variables, each of which also depends on the site chosen. Each chance node has associated with it a conditional distribution that is indexed by the state of the parent nodes. In decision networks, the parent nodes can include decision nodes as well as chance nodes. Note that each of the current-state chance nodes could be part of a large Bayesian network for assessing construction costs, air traffic levels, or litigation potentials.

- **Decision nodes** (rectangles) represent points where the decision maker has a choice of actions. In this case, the *AirportSite* action can take on a different value for each site under consideration. The choice influences the safety, quietness, and frugality of the solution. In this chapter, we assume that we are dealing with a single decision node. Chapter 16 deals with cases in which more than one decision must be made.
- **Utility nodes** (diamonds) represent the agent's utility function.[7] The utility node has as parents all variables describing the outcomes that directly affect utility. Associated with the utility node is a description of the agent's utility as a function of the parent attributes. The description could be just a tabulation of the function, or it might be a parameterized additive or linear function of the attribute values. For now, we will assume that the function is deterministic; that is, given the values of its parent variables, the value of the utility node is fully determined.

A simplified form is also used in many cases. The notation remains identical, but the chance nodes describing the outcome states are omitted. Instead, the utility node is connected directly to the current-state nodes and the decision node. In this case, rather than representing a utility function on outcome states, the utility node represents the *expected* utility associated with each action, as defined in Equation (15.1) on page 519; that is, the node is associated with an **action-utility function** (also known as a **Q-function** in reinforcement learning, as described in Chapter 23). Figure 15.7 shows the action-utility representation of the airport siting problem.

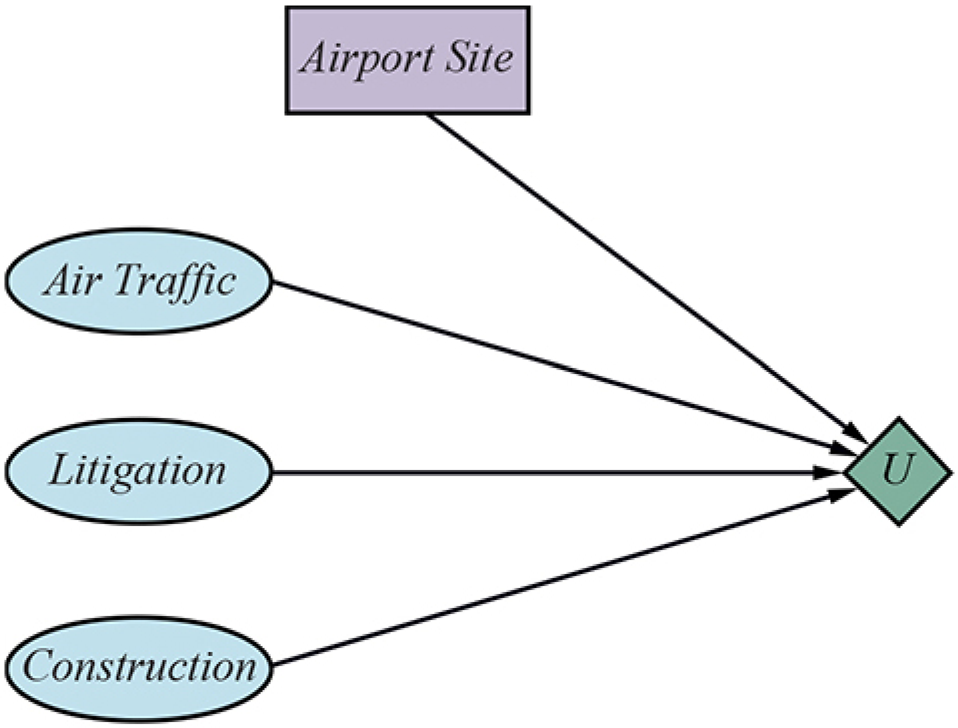


Figure 15.7 A simplified representation of the airport-siting problem. Chance nodes corresponding to outcome states have been factored out.

Notice that, because the *Quietness*, *Safety*, and *Frugality* chance nodes in Figure 15.6 refer to future states, they can never have their values set as evidence variables. Thus, the simplified version that omits these nodes can be used whenever the more general form can be used. Although the simplified form contains fewer nodes, the omission of an explicit description of the outcome of the siting decision means that it is less flexible with respect to changes in circumstances.

For example, in Figure 15.6, a change in aircraft noise levels can be reflected by a change in the conditional probability table associated with the

Quietness node, whereas a change in the weight accorded to noise pollution in the utility function can be reflected by a change in the utility table. In the action-utility diagram, Figure 15.7, on the other hand, all such changes have to be reflected by changes to the action-utility table. Essentially, the actionutility formulation is a *compiled* version of the original formulation, obtained by summing out the outcome state variables.

15.5.2 Evaluating decision networks

Actions are selected by evaluating the decision network for each possible setting of the decision node. Once the decision node is set, it behaves exactly like a chance node that has been set as an evidence variable. The algorithm for evaluating decision networks is the following:

1. Set the evidence variables for the current state.
2. For each possible value of the decision node:
    (a) Set the decision node to that value.
    (b) Calculate the posterior probabilities for the parent nodes of the utility node, using a standard probabilistic inference algorithm.
    (c) Calculate the resulting utility for the action.
3. Return the action with the highest utility.

This is a straightforward approach that can utilize any available Bayesian network algorithm and can be incorporated directly into the agent design given in Figure 12.1 on page 406. We will see in Chapter 16 that the possibility of executing several actions in sequence makes the problem much more interesting.

15.6 The Value of Information

In the preceding analysis, we have assumed that all relevant information, or at least all available information, is provided to the agent before it makes its decision. In practice, this is hardly ever the case. *One of the most important parts of decision making is knowing what questions to ask.* For example, a doctor cannot expect to be provided with the results of all possible diagnostic tests and questions at the time a patient first enters the consulting room. Tests are often expensive and sometimes hazardous (both directly and because of associated delays). Their importance depends on two factors: whether the test results would lead to a significantly better treatment plan, and how likely the various test results are.

This section describes **information value theory**, which enables an agent to choose what information to acquire. We assume that prior to selecting a "real" action represented by the decision node, the agent can acquire the value of any of the potentially observable chance variables in the model. Thus, information value theory involves a simplified form of sequential decision making—simplified because the observation actions affect only the agent's belief state, not the external physical state. The value of any particular observation must derive from the potential to affect the agent's eventual physical action; and this potential can be estimated directly from the decision model itself.

15.6.1 A simple example

Suppose an oil company is hoping to buy one of n indistinguishable blocks of ocean-drilling rights. Let us assume further that exactly one of the blocks contains oil that will generate net profits of C dollars, while the others are worthless. The asking price of each block is C/n dollars. If the company is risk-neutral, then it will be indifferent between buying a block and not buying one because the expected profit is zero in both cases.

Now suppose that a seismologist offers the company the results of a survey of block number 3, which indicates definitively whether the block contains oil. How much should the company be willing to pay for the information? The way to answer this question is to examine what the company would do if it had the information:

- With probability $1/n$, the survey will indicate oil in block 3. In this case, the company will buy block 3 for C/n dollars and make a profit of $C - C/n = (n-1)C/n$ dollars.
- With probability $(n-1)/n$, the survey will show that the block contains no oil, in which case the company will buy a different block. Now the probability of finding oil in one of the other blocks changes from $1/n$ to $1/(n-1)$, so the company makes an expected profit of $C/(n-1) - C/n = C/n(n-1)$ dollars.

Now we can calculate the expected profit, given access to the survey information:

$$\frac{1}{n} \times \frac{(n-1)C}{n} + \frac{n-1}{n} \times \frac{C}{n(n-1)} = C/n.$$

Thus, the information is worth *C/n* dollars to the company, and the company should be willing to pay the seismologist some significant fraction of this amount.

The value of information derives from the fact that *with* the information, one's course of action can be changed to suit the *actual* situation. One can discriminate according to the situation, whereas without the information, one has to do what's best on average over the possible situations. In general, the value of a given piece of information is defined to be the difference in expected value between best actions before and after information is obtained.

15.6.2 A general formula for perfect information

It is simple to derive a general mathematical formula for the value of information. We assume that exact evidence can be obtained about the value of some random variable E_j (that is, we learn $E_j = e_j$), so the phrase **value of perfect information** (VPI) is used.[8]

In the agent's initial information state, the value of the current best action a is, from Equation (15.1),

$$EU(\alpha) = \max_a \sum_{s'} P(\mathrm{RESULT}(a) = s')U(s'),$$

and the value of the new best action (after the new evidence $E_j = e_j$ is obtained) will be

$$EU(\alpha_{ej} \middle| e_j) = \max_a \sum_{s'} P(\mathrm{RESULT}(a) = s' \mid e_j)\, U(s').$$

But E_j is a random variable whose value is *currently* unknown, so to determine the value of discovering E_j we must average over all possible values e_j that we might discover for E_j, using our *current* beliefs about its value:

$$VPI(E_j) = \left(\sum_{e_j} P(E_j = e_j)\ EU(\alpha_{e_j} | E_j = e_j) \right) - EU(\alpha).$$

To get some intuition for this formula, consider the simple case where there are only two actions, a_1 and a_2, from which to choose. Their current expected utilities are U_1 and U_2. The information $E_j = e_j$ will yield some new expected utilities U'_1 and U'_2 for the actions, but before we obtain E_j, we will have some probability distributions over the possible values of U'_1 and U'_2 (which we assume are independent).

Suppose that a_1 and a_2 represent two different routes through a mountain range in winter: a_1 is a nice, straight highway through a tunnel, and a_2 is a winding dirt road over the top. Just given this information, a_1 is clearly preferable, because it is quite possible that a_2 is blocked by snow, whereas it is unlikely that anything blocks a_1. U_1 is therefore clearly higher than U_2. It is possible to obtain satellite reports E_j on the actual state of each road that would give new expectations, U'_1 and U'_2, for the two crossings. The distributions for these expectations are shown in Figure 15.8(a). Obviously, in this case, it is not worth the expense of obtaining satellite reports, because it is unlikely that the information derived from them will change the plan. With no change, information has no value.

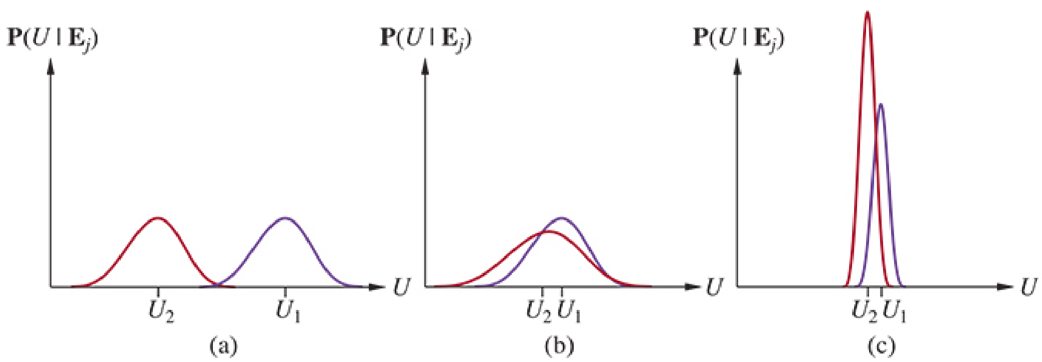


Figure 15.8 Three generic cases for the value of information. In (a), a_1 will almost certainly remain superior to a_2, so the information is not needed. In (b), the choice is unclear and the information is crucial. In (c), the choice is unclear, but because it makes little difference, the information is less valuable. (Note: The fact that U_2 has a high peak in (c) means that its expected value is known with higher certainty than U_1.)

Now suppose that we are choosing between two different winding dirt roads of slightly different lengths and we are carrying a seriously injured passenger. Then, even when U_1 and U_2 are quite close, the distributions of U'_1 and U'_2 are very broad. There is a significant possibility that the second route will turn out to be clear while the first is blocked, and in this case the difference in utilities will be very high. The VPI formula indicates that it might be worthwhile getting the satellite reports. Such a situation is shown in Figure 15.8(b).

Finally, suppose that we are choosing between the two dirt roads in summertime, when blockage by snow is unlikely. In this case, satellite reports might show one route to be more scenic than the other because of flowering alpine meadows, or perhaps wetter because of recent rain. It is therefore quite likely that we would change our plan if we had the information. In this case, however, the difference in value between the two routes is still likely to be very small, so we will not bother to obtain the reports. This situation is shown in Figure 15.8(c).

In sum, *information has value to the extent that it is likely to cause a change of plan and to the extent that the new plan will be significantly better than the old plan*.

15.6.3 Properties of the value of information

One might ask whether it is possible for information to be deleterious: can it actually have negative expected value? Intuitively, one should expect this to be impossible. After all, one could in the worst case just ignore the information and pretend that one has never received it. This is confirmed by the following theorem, which applies to any decision-theoretic agent using any decision network with possible observations E_j:

The expected value of information is nonnegative:

$$\forall j \; VPI(E_j) \; \geq \; 0.$$

The theorem follows directly from the definition of VPI, and we leave the proof as an exercise (Exercise 15.NNVP). It is, of course, a theorem about *expected* value, not *actual* value. Additional information can easily lead to a plan that *turns out to* be worse than the original plan if the information happens to be misleading. For example, a medical test that gives a false positive result may lead to unnecessary surgery; but that does not mean that the test shouldn't be done.

It is important to remember that VPI depends on the current state of information. It can change as more information is acquired. For any given piece of evidence E_j, the value of acquiring it can go down (e.g., if another variable strongly constrains the posterior for E_j) or up (e.g., if another variable provides a clue on which E_j builds, enabling a new and better plan to be devised). Thus, VPI is not additive. That is,

$$VPI(E_j, E_k) \; \neq \; VPI(E_j) + VPI(E_k) \quad \text{(in general)}.$$

VPI is, however, order-independent. That is,

$$VPI(E_j, E_k) \; = \; VPI(E_j) + VPI(E_k|E_j) \; = \; VPI(E_k) + VPI(E_j|E_k) \; = \; VPI(E_k, E_j)$$

where the notation $VPI(\bullet|E)$ denotes the VPI calculated according to the posterior distribution where E is already observed. Order independence distinguishes sensing actions from ordinary actions and simplifies the problem of calculating the value of a sequence of sensing actions. We return to this question in the next section.

15.6.4 Implementation of an information-gathering agent

A sensible agent should ask questions in a reasonable order, should avoid asking questions that are irrelevant, should take into account the importance of each piece of information in relation to its cost, and should stop asking questions when that is appropriate. All of these capabilities can be achieved by using the value of information as a guide.

Figure 15.9 shows the overall design of an agent that can gather information intelligently before acting. For now, we assume that with each observable evidence variable E_j, there is an associated cost, $C(E_j)$, which reflects the cost of obtaining the evidence through tests, consultants, questions, or whatever. The agent requests what appears to be the most efficient observation in terms of utility gain per unit cost. We assume that the result of the action Request(E_j) is that the next percept provides the value of E_j. If no observation is worth its cost, the agent selects a "real" action.

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

   integrate percept into D
   j ← the value that maximizes VPI(E_j) / C(E_j)
   if VPI(E_j) > C(E_j)
      then return Request(E_j)
   else return the best action from D
```

Figure 15.9 Design of a simple, myopic information-gathering agent. The agent works by repeatedly selecting the observation with the highest information value, until the cost of the next observation is greater than its expected benefit.

The agent algorithm we have described implements a form of information gathering that is called **myopic**. This is because it uses the VPI formula shortsightedly, calculating the value of information as if only a single evidence variable will be acquired. Myopic control is based on the same heuristic idea as greedy search and often works well in practice. (For example, it has been shown to outperform expert physicians in selecting diagnostic tests.) However, if there is no single evidence variable that will help a lot, a myopic agent might hastily take an action when it would have been better to request two or more variables first and then take action. The next section considers the possibility of obtaining multiple observations.

15.6.5 Nonmyopic information gathering

The fact that the value of a sequence of observations is invariant under permutations of the sequence is intriguing but doesn't, by itself, lead to efficient algorithms for optimal information gathering. Even if we restrict ourselves to choosing in advance a fixed subset of observations to collect, there are 2^n possible such subsets from n potential observations. In the general case, we face an even more complex problem of finding an optimal *conditional plan* (as described in Section 11.5.2) that chooses an observation and then acts or chooses more observations, depending on the outcome. Such plans form trees, and the number of such trees is superexponential in n.[9]

For observations of variables in a decision network, it turns out that this problem is intractable even when the network is a polytree. There are, however, special cases in which the problem can be solved efficiently. Here we present one such case: the **treasure hunt** problem (or the **least-cost testing sequence** problem, for the less romantically inclined). There are n locations $1,\ldots,n$; each location i contains treasure with independent probability $P(i)$; and it costs $C(i)$ to check location i. This corresponds to a decision network where all the potential evidence variables $Treasure_i$ are absolutely independent. The agent examines locations in some order until treasure is found; the question is, what is the optimal order?

To answer this question, we will need to consider the expected costs and success probabilities of various sequences of observations, assuming the agent stops when treasure is found. Let $\mathbf{x}$ be such a sequence; $\mathbf{xy}$ be the concatenation of sequences $\mathbf{x}$ and $\mathbf{y}$; $C(\mathbf{x})$ be the expected cost of $\mathbf{x}$; $P(\mathbf{x})$ be the probability that sequence $\mathbf{x}$ succeeds in finding treasure; and $F(\mathbf{x}) = 1 - P(\mathbf{x})$ be the probability that it fails. Given these definitions, we have

$$C(\mathbf{xy}) \ = C(\mathbf{x}) + F(\mathbf{x})C(\mathbf{y}), \qquad (15.3)$$

that is, the sequence $\mathbf{xy}$ will definitely incur the cost of $\mathbf{x}$ and, if $\mathbf{x}$ fails, it will also incur the cost of $\mathbf{y}$.

The basic idea in any sequence optimization problem is to look at the change in cost, defined by $\Delta = C(\mathbf{wxyz}) - C(\mathbf{wyxz})$, when two adjacent subsequences $\mathbf{x}$ and $\mathbf{y}$ in a general sequence $\mathbf{wxyz}$ are flipped. When the sequence is optimal, all such changes make the sequence worse. The first step is to show that the sign of the effect (increasing or decreasing the cost) doesn't depend on the context provided by $\mathbf{w}$ and $\mathbf{z}$. We have

$$\begin{aligned}
\Delta &= [C(\mathbf{w})+F(\mathbf{w})C(\mathbf{xyz})] - [C(\mathbf{w})+F(\mathbf{w})C(\mathbf{yxz})] \qquad \text{(by Equation (15.3))}\\
&= F(\mathbf{w})[C(\mathbf{xyz}) - C(\mathbf{yxz})]\\
&= F(\mathbf{w})[(C(\mathbf{xy}) + F(\mathbf{xy})C(\mathbf{z})) - (C(\mathbf{yx}) + F(\mathbf{yx})C(\mathbf{z}))] \quad \text{(by Equation (15.3))}\\
&= F(\mathbf{w})[C(\mathbf{xy}) - C(\mathbf{yx})] \qquad \text{(since } F(\mathbf{xy}) = F(\mathbf{yx})\text{)}\ .
\end{aligned}$$

So we have shown that the direction of the change in the cost of the whole sequence depends only on the direction of the change in cost of the pair of elements being flipped; the context of the pair doesn't matter. This gives us a way to sort the sequence by pairwise comparisons to obtain an optimal solution. Specifically, we now have

$$\begin{aligned}
\Delta &= F(\mathbf{w})[(C(\mathbf{x}) + F(\mathbf{x})C(\mathbf{y})) - (C(\mathbf{y}) + F(\mathbf{y})C(\mathbf{x}))] \qquad \text{(by Equation (15.3))}\\
&= F(\mathbf{w})[C(\mathbf{x})(1 - F(\mathbf{y})) - C(\mathbf{y})(1 - F(\mathbf{x}))] \ = \ F(\mathbf{w})[C(\mathbf{x})P(\mathbf{y}) - C(\mathbf{y})P(\mathbf{x})].
\end{aligned}$$

This holds for any sequences **x** and **y**, so it holds specifically when **x** and **y** are single observations of locations i and j, respectively. So we know that, for i and j to be adjacent in an optimal sequence, we must have $C(i)P(j) \leq C(j)P(i)$, or $\frac{P(i)}{C(i)} \geq \frac{P(j)}{C(j)}$. In other words, the optimal order ranks the locations according to the success probability per unit cost. Exercise 15.HUNT asks you to determine whether this is in fact the policy followed by the algorithm in Figure 15.9 for this problem.

15.6.6 Sensitivity analysis and robust decisions

The practice of **sensitivity analysis** is widespread in technological disciplines: it means analyzing how much the output of a process changes as the model parameters are tweaked. Sensitivity analysis in probabilistic and decision-theoretic systems is particularly important because the probabilities used are typically either learned from data or estimated by human experts, which means that they are themselves subject to considerable uncertainty. Only in rare cases, such as the dice rolls in backgammon, are the probabilities objectively known.

For a utility-driven decision-making process, you can think of the output as either the actual decision made or the expected utility of that decision. Consider the latter first: because expectation depends on probabilities from the model, we can compute the derivative of the expected utility of any given action with respect to each of those probability values. (For example, if all the conditional probability distributions in the model are explicitly tabulated, then computing the expectation involves computing a ratio of two sum-of-product expressions; for more on this, see Chapter 21.) Thus, one can determine which parameters in the model have the largest effect on the expected utility of the final decision.

If, instead, we are concerned about the actual decision made, rather than its utility according to the model, then we can simply vary the parameters systematically (perhaps using binary search) to see whether the decision changes, and, if so, what is the smallest perturbation that causes such a change. One might think it doesn't matter that much which decision is made, only what its utility is. That's true, but in practice there may be a very substantial difference between the *real* utility of a decision and the utility *according to the model*.

If all reasonable perturbations of the parameters leave the optimal decision unchanged, then it is reasonable to assume the decision is a good one, even if the utility estimate for that decision is substantially incorrect. if, on the other hand, the optimal decision changes considerably as the parameters of the model change, then there is a good chance that the model may produce a decision that is substantially suboptimal in reality. In that case, it is worth investing further effort to refine the model.

These intuitions have been formalized in several fields (control theory, decision analysis, risk management) that propose the notion of a **robust** or **minimax** decision—that is, one that gives the best result in the worst case. Here, "worst case" means worst with respect to all plausible variations in the parameter values of the model. Letting θ stand for all the parameters in the model, the robust decision is defined by

$$a^* = \underset{a}{\operatorname{argmax}} \min_{\theta} \; EU(a;\theta).$$

In many cases, particularly in control theory, the robust approach leads to designs that work very reliably in practice. In other cases, it leads to overly conservative decisions. For example, when designing a self-driving car, the robust approach would assume the worst case for the behavior of the other vehicles on the road—that is, they are all driven by homicidal maniacs. In that case, the optimal solution for the car is to stay in the garage.

Bayesian decision theory offers an alternative to robust methods: if there is uncertainty about the parameters of the model, then model that uncertainty using hyperparameters.

Whereas the robust approach might say that some probability θ_i in the model could be anywhere between 0.3 and 0.7, with the actual value chosen by an adversary to make things come out as badly as possible, the Bayesian approach would put a prior probability distribution on θ_i and then proceed as before. This requires more modeling effort—for example, the Bayesian modeler must decide if parameters θ_i and θ_j are independent—but often results in better performance in practice.

In addition to parametric uncertainty, applications of decision theory in the real world also suffer from *structural* uncertainty. For example, the assumption of independence of *AirTraffic*, *Litigation*, and *Construction* in Figure 15.6 may be incorrect, and there may be additional variables that the model simply omits. At present, we do not have a good understanding of how to take this kind of uncertainty into account. One possibility is to keep an ensemble of models, perhaps generated by machine learning algorithms, in the hope that the ensemble captures the significant variations that matter.

15.7 Unknown Preferences

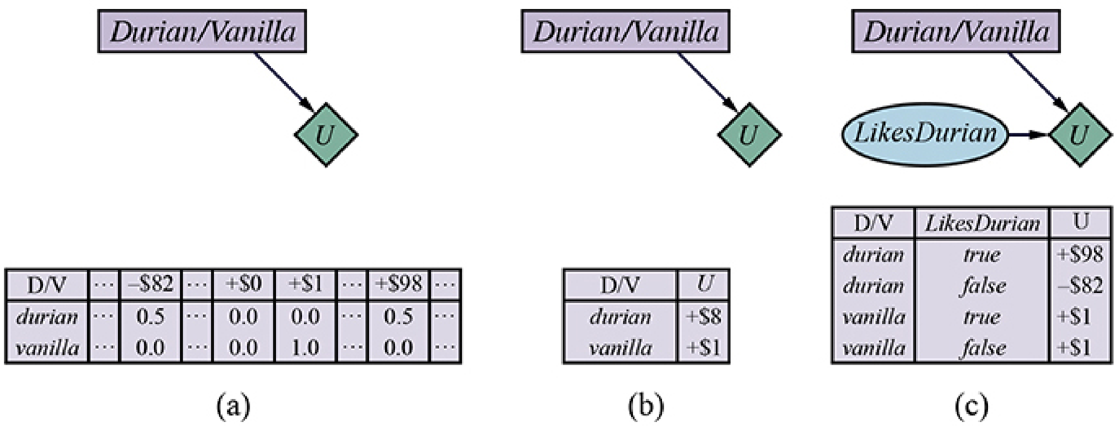
In this section we discuss what happens when there is uncertainty about the utility function whose expected value is to be optimized. There are two versions of this problem: one in which an agent (machine or human) is uncertain about its *own* utility function, and another in which a machine is supposed to help a human but is uncertain about what the human wants.

15.7.1 Uncertainty about one's own preferences

Imagine that you are at an ice-cream shop in Thailand and they have only two flavors left: vanilla and durian. Both cost $2. You know you have a moderate liking for vanilla and you'd be willing to pay up to $3 for a vanilla ice cream on such a hot day, so there is a net gain of $1 for choosing vanilla. On the other hand, you have no idea whether you like durian or not, but you've read on Wikipedia that the durian elicits different responses from different people: some find that "it surpasses in flavour all other fruits of the world" while others liken it to "sewage, stale vomit, skunk spray and used surgical swabs."

To put some numbers on this, let's say there's a 50% chance you'll find it sublime (+$100) and a 50% chance you'll hate it (-$80 if the taste lingers all afternoon). Here, there's no uncertainty about what prize you're going to win—it's the same durian ice cream either way—but there's uncertainty about your own preferences for that prize.

We could extend the decision network formalism to allow for uncertain utilities, as shown in Figure 15.10(a). If there is no more information to be obtained about your durian preferences, however—for example, if the shop won't let you taste it first—then the decision problem is identical to the one shown in Figure 15.10(b). We can simply replace the uncertain value of the durian with its expected net gain of (0.5 × \$100) – (0.5 × \$80) – \$2 = \$8 and your decision will remain unchanged.



(a)

D/V	⋯	–\$82	⋯	+\$0	+\$1	⋯	+\$98	⋯
durian	⋯	0.5	⋯	0.0	0.0	⋯	0.5	⋯
vanilla	⋯	0.0	⋯	0.0	1.0	⋯	0.0	⋯

(b)

D/V	*U*
durian	+\$8
vanilla	+\$1

(c)

D/V	*LikesDurian*	U
durian	*true*	+\$98
durian	*false*	–\$82
vanilla	*true*	+\$1
vanilla	*false*	+\$1

Figure 15.10 (a) A decision network for the ice cream choice with an uncertain utility function. (b) The network with the expected utility of each action. (c) Moving the uncertainty from the utility function into a new random variable.

If it’s possible for your beliefs about durian to change—perhaps you get a tiny taste, or you find out that all of your living relatives love durian—then the transformation in Figure 15.10(b) is not valid. It turns out, however, that we can still find an equivalent model in which the utility function is deterministic. Rather than saying there is uncertainty about the utility function, we move that uncertainty “into the world,” so to speak. That is, we create a new random variable *LikesDurian* with prior probabilities of 0.5 for *true* and *false*, as shown in Figure 15.10(c). With this extra variable, the utility function becomes deterministic, but we can still handle changing beliefs about your durian preferences.

The fact that unknown preferences can be modeled by ordinary random variables means that we can keep using the machinery and theorems developed for known preferences. On the other hand, it doesn't mean that we can always assume that preferences are known. The uncertainty is still there and still affects how agents should behave.

15.7.2 Deference to humans

Now let's turn to the second case mentioned above: a machine that is supposed to help a human but is uncertain about what the human wants. The full treatment of this case must be deferred to Chapter 17, where we discuss decisions involving more than one agent. Here, we ask one simple question: under what circumstances will such a machine defer to the human?

To study this question, let's consider a very simple scenario, as shown in Figure 15.11. Robbie is a software robot working for Harriet, a busy human, as her personal assistant. Harriet needs a hotel room for her next business meeting in Geneva. Robbie can act now—let's say he can book Harriet into a very expensive hotel near the meeting venue. He is quite unsure how much Harriet will like the hotel and its price; let's say he has a uniform probability for its net value to Harriet between –40 and +60, with an average of +10. He could also "switch himself off"—less melodramatically, take himself out of the hotel booking process altogether —which we define (without loss of generality) to have value 0 to Harriet. If those were his two choices, he would go ahead and book the hotel, incurring a significant risk of making Harriet unhappy. (If the range were –60 to +40, with average –10, he would switch himself off instead.) We'll give Robbie a third choice, however: explain his plan, wait, and let Harriet switch him off. Harriet can either switch him off or let him go ahead and book the hotel. What possible good could this do, one might ask, given that he could make both of those choices himself?

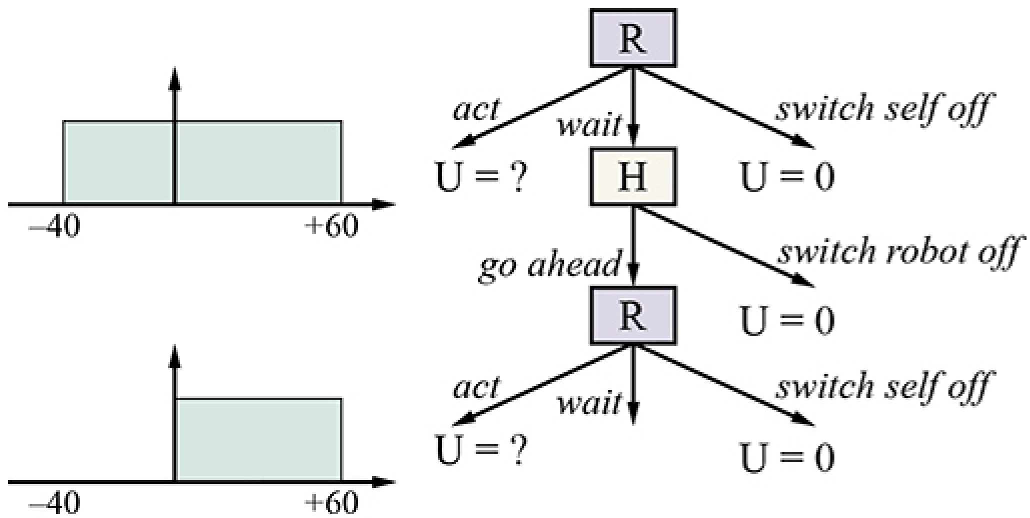


Figure 15.11 The off-switch game. *R*, the robot, can choose to act now, with a highly uncertain payoff; to switch itself off; or to defer to *H*, the human. *H* can switch *R* off or let it go ahead. *R* now has the same choice again. Acting still has an uncertain payoff, but now *R* knows the payoff is nonnegative.

The point is that Harriet's choice—to switch Robbie off or let him go ahead—provides Robbie with information about Harriet's preferences. We'll assume, for now, that Harriet is rational, so if Harriet lets Robbie go ahead, it means the value to Harriet is positive. Now, as shown in Figure

15.11, Robbie's belief changes: it is uniform between 0 and +60, with an average of +30.

So, if we evaluate Robbie's initial choices from his point of view:

1. Acting now and booking the hotel has an expected value of + 10.
2. Switching himself off has a value of 0.
3. Waiting and letting Harriet switch him off leads to two possible outcomes:
   (a) There is a 40% chance, based on Robbie's uncertainty about Harriet's preferences, that she will hate the plan and will switch Robbie off, with value 0.
   (b) There is a 60% chance Harriet will like the plan and allow Robbie to go ahead, with expected value +30.

Thus, waiting has expected value (0.4 × 0) + (0.6 × 30) = + 18, which is better than the +10 Robbie expects if he acts now.

The upshot is that Robbie has a positive incentive to defer to Harriet—that is, to allow himself to be switched off. This incentive comes directly from Robbie's uncertainty about Harriet's preferences. Robbie is aware that there's a chance (40% in this example) that he might be about to do something that will make Harriet unhappy, in which case being switched off would be preferable to going ahead. Were Robbie already certain about Harriet's preferences, he would just go ahead and make the decision (or switch himself off); there would be absolutely nothing to be gained from consulting Harriet, because, according to Robbie's definite beliefs, he can already predict exactly what she is going to decide.

In fact, it is possible to prove the same result in the general case: as long as Robbie is not completely certain that he's about to do what Harriet herself would do, he is better off allowing her to switch him off. Intuitively, her decision provides Robbie with information, and the expected value of information is always nonnegative. Conversely, if Robbie is certain about Harriet's decision, her decision provides no new information, and so Robbie has no incentive to allow her to decide.

Formally, let $P(u)$ be Robbie's prior probability density over Harriet's utility for the proposed action a. Then the value of going ahead with a is

$$EU(a) = \int_{-\infty}^{\infty} P(u) \cdot u du = \int_{-\infty}^{0} P(u) \cdot u du + \int_{0}^{\infty} P(u) \cdot u du.$$

(We will see shortly why the integral is split up in this way.) On the other hand, the value of action d, deferring to Harriet, is composed of two parts: if $u > 0$ then Harriet lets Robbie go ahead, so the value is u, but if $u < 0$ then Harriet switches Robbie off, so the value is 0:

$$EU(d) = \int_{-\infty}^{0} P(u) \cdot 0 du + \int_{0}^{\infty} P(u) \cdot u du.$$

Comparing the expressions for $EU(a)$ and $EU(d)$, we see immediately that

$$EU(d) \ \geq \ EU(a)$$

because the expression for $EU(d)$ has the negative-utility region zeroed out. The two choices have equal value only when the negative region has zero probability—that is, when Robbie is already certain that Harriet likes the proposed action.

There are some obvious elaborations on the model that are worth exploring immediately. The first elaboration is to impose a cost for Harriet's time. In that case, Robbie is less inclined to bother Harriet if the downside risk is small. This is as it should be. And if Harriet is really grumpy about being interrupted, she shouldn't be too surprised if Robbie occasionally does things she doesn't like.

The second elaboration is to allow for some probability of human error —that is, Harriet might sometimes switch Robbie off even when his proposed action is reasonable, and she might sometimes let Robbie go ahead even when his proposed action is undesirable. It is straightforward to fold this error probability into the model (see Exercise 15.OFFS). As one might expect, the solution shows that Robbie is less inclined to defer to an irrational Harriet who sometimes acts against her own best interests. The more randomly she behaves, the more uncertain Robbie has to be about her preferences before deferring to her. Again, this is as it should be: for example, if Robbie is a self-driving car and Harriet is his naughty two-year-old passenger, Robbie should not allow Harriet to switch him off in the middle of the highway.

Summary

This chapter shows how to combine utility theory with probability to enable an agent to select actions that will maximize its expected performance.

- **Probability theory** describes what an agent should believe on the basis of evidence, **utility theory** describes what an agent wants, and **decision theory** puts the two together to describe what an agent should do.
- We can use decision theory to build a system that makes decisions by considering all possible actions and choosing the one that leads to the best expected outcome. Such a system is known as a **rational agent**.
- Utility theory shows that an agent whose preferences between lotteries are consistent with a set of simple axioms can be described as possessing a utility function; furthermore, the agent selects actions as if maximizing its expected utility.
- **Multiattribute utility theory** deals with utilities that depend on several distinct attributes of states. **Stochastic dominance** is a particularly useful technique for making unambiguous decisions, even without precise utility values for attributes.
- **Decision networks** provide a simple formalism for expressing and solving decision problems. They are a natural extension of Bayesian networks, containing decision and utility nodes in addition to chance nodes.
- Sometimes, solving a problem involves finding more information before making a decision. The **value of information** is defined as the expected improvement in utility compared with making a decision

without the information; it is particularly useful for guiding the process of information-gathering prior to making a final decision.

- When, as is often the case, it is impossible to specify the human’s utility function completely and correctly, machines must operate under uncertainty about the true objective. This makes a significant difference when the possibility exists for the machine to acquire more information about human preferences. We showed by a simple argument that uncertainty about preferences ensures that the machine defers to the human, to the point of allowing itself to be switched off.

Bibliographical and Historical Notes

In the 17th century treatise *L'art de Penser,* or *Port-Royal Logic,* Arnauld (1662) states:

> To judge what one must do to obtain a good or avoid an evil, it is necessary to consider not only the good and the evil in itself, but also the probability that it happens or does not happen; and to view geometrically the proportion that all these things have together.

Modern texts talk of *utility* rather than good and evil, but this statement correctly notes that one should multiply utility by probability ("view geometrically") to give expected utility, and maximize that over all outcomes ("all these things") to "judge what one must do." It is remarkable how much Arnauld got right, more than 350 years ago, and only 8 years after Pascal and Fermat first showed how to use probability correctly.

Daniel Bernoulli (1738), investigating the St. Petersburg paradox (see Exercise 15.STPT), was the first to realize the importance of preference measurement for lotteries, writing "the *value* of an item must not be based on its *price,* but rather on the *utility* that it yields" (italics his). Utilitarian philosopher Jeremy Bentham (1823) proposed the **hedonic calculus** for weighing "pleasures" and "pains," arguing that all decisions (not just monetary ones) could be reduced to utility comparisons.

Bernoulli's introduction of utility—an internal, subjective quantity—to explain human behavior via a mathematical theory was an utterly remarkable proposal for its time. It was all the more remarkable for the fact that unlike monetary amounts, the utility values of various bets and prizes are not directly observable; instead, utilities are to be inferred from the preferences exhibited by an individual. It would be two centuries before the implications of the idea were fully worked out and it became broadly accepted by statisticians and economists.

The derivation of numerical utilities from preferences was first carried out by Ramsey (1931); the axioms for preference in the present text are closer in form to those rediscovered in *Theory of Games and Economic Behavior* (von Neumann and Morgenstern, 1944). Ramsey had derived subjective probabilities (not just utilities) from an agent's preferences; Savage (1954) and Jeffrey (1983) carry out more recent constructions of this kind. Beardon *et al*. (2002) show that a utility function does not suffice to represent nontransitive preferences and other anomalous situations.

In the post-war period, decision theory became a standard tool in economics, finance, and management science. A field of **decision analysis** emerged to aid in making policy decisions more rational in areas such as military strategy, medical diagnosis, public health, engineering design, and resource management. me process involves a **decision maker** who states preferences between outcomes and a **decision analyst** who enumerates the possible actions and outcomes and elicits preferences from the decision maker to determine the best course of action. Von Winterfeldt and Edwards (1986) provide a nuanced perspective on decision analysis and its relationship to human preference structures. Smith (1988) gives an overview of the methodology of decision analysis.

Until the 1980s, multivariate decision problems were handled by constructing "decision trees" of all possible instantiations of the variables. Influence diagrams or decision networks, which take advantage of the same conditional independence properties as Bayesian networks, were introduced by Howard and Matheson (1984), based on earlier work at SRI (Miller *et al*., 1976). Howard and Matheson's algorithm constructed the complete (exponentially large) decision tree from the decision network. Shachter (1986) developed a method for making decisions based directly on a decision network, without the creation of an intermediate decision tree. This algorithm was also one of the first to provide complete inference for multiply connected Bayesian networks. Nilsson and Lauritzen (2000) link algorithms for decision networks to ongoing developments in clustering algorithms for Bayesian networks. The collection by Oliver and Smith (1990) has a number of useful early articles on decision networks, as does the 1990 special issue of the journal *Networks*. The text by Fenton and Neil (2018) provides a hands-on guide to solving real-world decision problems using decision networks. Papers on decision networks and utility modeling also appear regularly in the journals *Management Science* and *Decision Analysis*.

Surprisingly few early AI researchers adopted decision-theoretic tools after the early applications in medical decision making described in Chapter 12. One of the few exceptions was Jerry Feldman, who applied decision theory to problems in vision (Feldman and Yakimovsky, 1974) and planning (Feldman and Sproull, 1977). Rule-based expert systems of the late 1970s and early 1980s concentrated on answering questions, rather than on making decisions. Those systems that did recommend actions generally did so using condition–action rules rather than explicit representations of outcomes and preferences.

Decision networks offer a far more flexible approach, for example by allowing preferences to change while keeping the transition model constant, or vice versa. They also allow a principled calculation of what information to seek next. In the late 1980s, partly due to Pearl's work on Bayes nets, decision-theoretic expert systems gained widespread acceptance (Horvitz *et al.*, 1988; Cowell *et al.*, 2002). In fact, from 1991 onward, the cover design of the journal *Artificial Intelligence* has depicted a decision network, although some artistic license appears to have been taken with the direction of the arrows.

Practical attempts to measure human utilities began with post-war decision analysis (see above). The micromort utility measure is discussed by Howard (1989). Thaler Thaler (1992) found that for a 1/1000 chance of death, a respondent wouldn't pay more than $200 to remove the risk, but wouldn't accept $50,000 to take on the risk.

The use of **QALY**s (quality-adjusted life years) to perform cost–benefit analyses of medical interventions and related social policies dates back at least to work by Klarman *et al*. (1968), although the term itself was first used by Zeckhauser and Shepard (1976). Like money, QALYs correspond directly to utilities only under fairly strong assumptions, such as risk neutrality, that are often violated (Beresniak *et al.*, 2015); nonetheless, QALYs are much widely used in practice, for example in forming National Health Service policies in the UK. See Russell (1990) for a typical example of an argument for a major change in public health policy on grounds of increased expected utility measured in QALYs.

Keeney and Raiffa (1976) give an introduction to **multiattribute utility theory**. They describe early computer implementations of methods for eliciting the necessary parameters for a multiattribute utility function and include extensive accounts of real applications of the theory. Abbas (2018) covers many advances since 1976. The theory was introduced to AI primarily by the work of Wellman (1985), who also investigated the use of stochastic dominance and qualitative probability models (Wellman, 1988, 1990a). Wellman and Doyle (1992) provide a preliminary sketch of how a complex set of utility-independence relationships might be used to provide a structured model of a utility function, in much the same way that

Bayesian networks provide a structured model of joint probability distributions. Bacchus and Grove (1995,1996) and La Mura and Shoham (1999) give further results along these lines. Boutilier *et al*. (2004) describe CP-nets, a fully worked out graphical model formalism for conditional *ceteribus paribus* preference statements.

The **optimizer's curse** was brought to the attention of decision analysts in a forceful way by Smith and Winkler (2006), who pointed out that the financial benefits to the client projected by analysts for their proposed course of action almost never materialized. They trace this directly to the bias introduced by selecting an optimal action and show that a more complete Bayesian analysis eliminates the problem.

The same underlying concept has been called **post-decision disappointment** by Harrison and March (1984) and was noted in the context of analyzing capital investment projects by Brown (1974). The optimizer's curse is also closely related to the **winner's curse** (Capen *et al.*, 1971; Thaler, 1992), which applies to competitive bidding in auctions: whoever wins the auction is very likely to have overestimated the value of the object in question. Capen *et al*. quote a petroleum engineer on the topic of bidding for oil-drilling rights: “If one wins a tract against two or three others he may feel fine about his good fortune. But how should he feel if he won against 50 others? Ill.”

The Allais paradox, due to Nobel Prize–winning economist Maurice Allais (1953), was tested experimentally to show that people are consistently inconsistent in their judgments (Tversky and Kahneman, 1982; Conlisk, 1989). The Ellsberg paradox on ambiguity aversion was introduced in the Ph.D. thesis of Daniel Ellsberg (1962).[10] Fox and Tversky (1995) describe a further study of ambiguity aversion. Machina (2005) gives an overview of choice under uncertainty and how it can vary from expected utility theory. See the classic text by Keeney and Raiffa (1976) and the more recent work by Abbas (2018) for an in-depth analysis of preferences with uncertainty.

2009 was a big year for popular books on human **irrationality**, including *Predictably Irrational* (Ariely, 2009), *Sway* (Brafman and Brafman, 2009), *Nudge* (Thaler and Sunstein, 2009), *Kluge* (Marcus, 2009), *How We Decide* (Lehrer, 2009) and *On Being Certain* (Burton, 2009). They complement the classic book *Judgment Under Uncertainty* (Kahneman *et al.*,1982) and the article that started it all (Kahneman and Tversky, 1979). Kahneman himself provides an insightful and readable account in *Thinking: Fast and Slow* (Kahneman, 2011).

The field of evolutionary psychology (Buss, 2005), on the other hand, has run counter to this literature, arguing that humans are quite rational in evolutionary appropriate contexts. Its adherents point out that irrationality is penalized by definition in an evolutionary context and show that in some cases it is an artifact of the experimental setup (Cummins and Allen, 1998). There has been a recent resurgence of interest in Bayesian models of cognition, overturning decades of pessimism (Elio, 2002; Chater and Oaksford, 2008; Griffiths *et al.*, 2008); this resurgence is not without its detractors, however (Jones and Love, 2011).

The theory of information value was explored first in the context of statistical experiments, where a quasi-utility (entropy reduction) was used (Lindley, 1956). The control theorist Ruslan Stratonovich (1965) developed the more general theory presented here, in which information has value by virtue of its ability to affect decisions. Stratonovich's work was not known in the West, where Ron Howard (1966) pioneered the same idea. His paper ends with the remark "If information value theory and associated decision theoretic structures do not in the future occupy a large part of the education of engineers, then the engineering profession will find that its traditional role of managing scientific and economic resources for the benefit of man has been forfeited to another profession.” To date, the implied revolution in managerial methods has not occurred.

The myopic information-gathering algorithm described in the chapter is ubiquitous in the decision analysis literature; its basic outlines can be discerned in the original paper on influence diagrams (Howard and Matheson, 1984). Efficient calculation methods are studied by Dittmer and Jensen (1997). Laskey (1995) and Nielsen and Jensen (2003) discuss methods for sensitivity analysis in Bayesian networks and decision networks, respectively. The classic text *Robust and Optimal Control* (Zhou *et al.,* 1995) provides thorough coverage and comparison of the robust and decision-theoretic approaches to decisions under uncertainty.

The treasure hunt problem was solved independently by many authors, dating back at least to papers on sequential testing by Gluss (1959) and Mitten (1960). The style of proof in this chapter draws on a basic result, due to Smith (1956), relating the value of a sequence to the value of the same sequence with two adjacent elements permuted. These results for independent tests were extended to more general tree and graph search problems (where the tests are partially ordered) by Kadane and Simon (1977). Results on the complexity of nonmyopic calculations of the value of information were obtained by Krause and Guestrin (2009). Krause *et al*. (2008) identified cases where submodularity leads to a tractable approximation algorithm, drawing on the seminal work of Nemhauser *et al*. (1978) on submodular functions; Krause and Guestrin (2005) identify cases where an exact dynamic programming algorithm gives an efficient solution for both evidence subset election and conditional plan generation.

Harsanyi (1967) studied the problem of *incomplete* information in game theory, where players may not know each others' payoff functions exactly. He showed that such games were identical to games with *imperfect* information, where players are uncertain about the state of the world, via the trick of adding state variables referring to players' payoffs. Cyert and de Groot (1979) developed a theory of **adaptive utility** in which an agent could be uncertain about its own utility function and could obtain more information through experience.

Work on Bayesian preference elicitation (Chajewska *et al.*, 2000; Boutilier, 2002) begins from the assumption of a prior probability over the agent's utility function. Fern *et al.*(2014) propose a decision-theoretic model of **assistance** in which a robot tries to ascertain and assist with a human goal about which it is initially uncertain. The off-switch example in Section 15.7.2 is adapted from Hadfield-Menell *et al.* (2017b). Russell (2019) proposes a general framework for benefical AI in which the off-switch game is a key example.

[1] We apologize to readers whose local airlines no longer offer food on long flights.

[2] We can account for the enjoyment of gambling by encoding gambling events into the state description; for example, "Have \$10 and gambled" could be preferred to "Have \$10 and didn't gamble."

[3] In this sense, utilities resemble temperatures: a temperature in Fahrenheit is 1.8 times the Celsius temperature plus 32, but converting from one to the other doesn't make you hotter or colder.

[4] Such behavior might be called desperate, but it is rational if one is already in a desperate situation.

[5] For example, the mathematician/magician Persi Diaconis can make a coin flip come out the way he wants every time (Landhuis, 2004).

[6] Even the sure thing may not be certain. Despite cast-iron promises, we have not yet received that $27,000,000 from the Nigerian bank account of a previously unknown deceased relative.

[7] These nodes are also called **value nodes** in the literature.

[8] There is no loss of expressiveness in requiring perfect information. Suppose we wanted to model the case in which we become somewhat more certain about a variable. We can do that by introducing *another* variable about which we learn perfect information. For example, suppose we initially have broad uncertainty about the variable *Temperature*. Then we gain the perfect knowledge *Thermometer* = 37; this gives us imperfect information about the true *Temperature,* and the uncertainty due to measurement error is encoded in the sensor model *P(Thermometer* | *Temperature*). See Exercise 15.VPIX for another example.

[9] The general problem of generating sequential behavior in a partially observable environment falls under the heading of **partially observable Markov decision processes**, which are described in Chapter 16.

[10] Ellsberg later became a military analyst at the RAND Corporation and leaked documents known as the Pentagon Papers, thereby contributing to the end of the Vietnam war.

CHAPTER 16

MAKING COMPLEX DECISIONS

In which we examine methods for deciding what to do today, given that we may face another decision tomorrow.

In this chapter, we address the computational issues involved in making decisions in a stochastic environment. Whereas Chapter 15 was concerned with one-shot or episodic decision problems, in which the utility of each action's outcome was well known, we are concerned here with **sequential decision problems,** in which the agent's utility depends on a sequence of decisions. Sequential decision problems incorporate utilities, uncertainty, and sensing, and include search and planning problems as special cases. Section 16.1 explains how sequential decision problems are defined, and Section 16.2 describes methods for solving them to produce behaviors that are appropriate for a stochastic environment. Section 16.3 covers **multiarmed bandit** problems, a specific and fascinating class of sequential decision problems that arise in many contexts. Section 16.4 explores decision problems in partially observable environments and Section 16.5 describes how to solve them.

16.1 Sequential Decision Problems

Suppose that an agent is situated in the 4×3 environment shown in Figure 16.1 (a). Beginning in the start state, it must choose an action at each time step. The interaction with the environment terminates when the agent reaches one of the goal states, marked +1 or −1. Just as for search problems, the actions available to the agent in each state are given by ACTIONS(s), sometimes abbreviated to $A(s)$; in the 4×3 environment, the actions in every state are *Up, Down, Left,* and *Right.* We assume for now that the environment is **fully observable,** so that the agent always knows where it is.

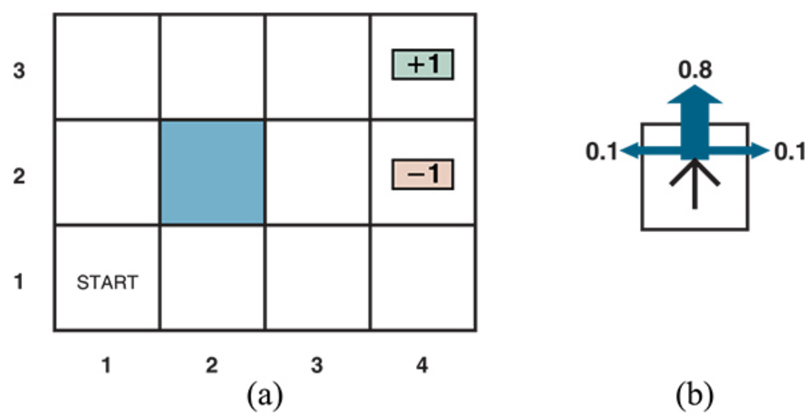


Figure 16.1 (a) A simple, stochastic 4×3 environment that presents the agent with a sequential decision problem. (b) Illustration of the transition model of the environment: the "intended" outcome occurs with probability 0.8, but with probability 0.2 the agent moves at right angles to the intended direction. A collision with a wall results in no movement. Transitions into the two terminal states have reward +1 and −1, respectively, and all other transitions have a reward of −0.04.

If the environment were deterministic, a solution would be easy: [*Up, Up, Right, Right, Right*], Unfortunately, the environment won't always go along with this solution, because the actions are unreliable. The particular model of stochastic motion that we adopt is illustrated in Figure 16.1(b). Each action achieves the intended effect with probability 0.8, but the rest of the time, the action moves the agent at right angles to the intended direction. Furthermore, if the agent bumps into a wall, it stays in the same square. For example, from the start square (1,1), the action *Up* moves the agent to (1,2) with probability 0.8, but with probability 0.1, it moves right to (2,1), and with probability 0.1, it moves left, bumps into the wall, and stays in (1,1). In such an environment, the sequence [*Up, Up,Right,Right,Right*] goes up around the barrier and reaches the goal state at (4,3) with probability 0.8^5 = 0.32768. There is also a small chance of accidentally reaching the goal by going the other way around with probability $0.1^4 \times 0.8$, for a grand total of 0.32776. (See also Exercise 16.MDPX.)

As in Chapter 3, the **transition model** (or just "model," when the meaning is clear) describes the outcome of each action in each state. Here, the outcome is stochastic, so we write $P(s' \mid s, a)$ for the probability of reaching state s' if action a is done in state s. (Some authors write $T(s, a, s')$ for the transition model.) We will assume that transitions are **Markovian**: the probability of reaching s' from s depends only on s and not on the history of earlier states.

To complete the definition of the task environment, we must specify the utility function for the agent. Because the decision problem is sequential, the utility function will depend on a sequence of states and actions—an **environment history**—rather than on a single state. Later in this section, we investigate the nature of utility functions on histories; for now, we simply stipulate that for every transition from s to s' via action a, the agent receives a **reward** $R(s, a, s')$. The rewards may be positive or negative, but they are bounded by $\pm R_{\max}$.[1]

For our particular example, the reward is –0.04 for all transitions except those entering terminal states (which have rewards +1 and –1). The utility of an environment history is just (for now) the sum of the rewards received. For example, if the agent reaches the +1 state after 10 steps, its total utility will be $(9 \times -0.04) + 1 = 0.64$. The negative reward of –0.04 gives the agent an incentive to reach (4,3) quickly, so our environment is a stochastic generalization of the search problems of Chapter 3. Another way of saying this is that the agent does not enjoy living in this environment and so it wants to leave as soon as possible.

To sum up: a sequential decision problem for a fully observable, stochastic environment with a Markovian transition model and additive rewards is called a **Markov decision process**, or **MDP**, and consists of a set of states (with an initial state s_0); a set ACTIONS(s) of actions in each state; a transition model $P(s' \mid s,a)$; and a reward function Markov decision process $R(s,a,s')$. Methods for solving MDPs usually involve **dynamic programming**: simplifying a problem by recursively breaking it into smaller pieces and remembering the optimal solutions to the pieces.

The next question is, what does a solution to the problem look like? No fixed action sequence can solve the problem, because the agent might end up in a state other than the goal. Therefore, a solution must specify what the agent should do for *any* state that the agent might reach. A solution of this kind is called a **policy**. It is traditional to denote a policy by π, and $\pi(s)$ is the action recommended by the policy π for state s. No matter what the outcome of the action, the resulting state will be in the policy, and the agent will know what to do next.

Each time a given policy is executed starting from the initial state, the stochastic nature of the environment may lead to a different environment history. The quality of a policy is therefore measured by the *expected* utility of the possible environment histories generated by that policy. An **optimal policy** is a policy that yields the highest expected utility. We use π^* to denote an optimal policy. Given π^*, the agent decides what to do by consulting its current percept, which tells it the current state s, and then executing the action $\pi^*(s)$. A policy represents the agent function explicitly and is therefore a description of a simple reflex agent, computed from the information used for a utility-based agent.

The optimal policies for the world of Figure 16.1 are shown in Figure 16.2(a). There are two policies because the agent is exactly indifferent between going left and going up from (3,1): going left is safer but longer, while going up is quicker but risks falling into (4,2) by accident. In general there will often be multiple optimal policies.

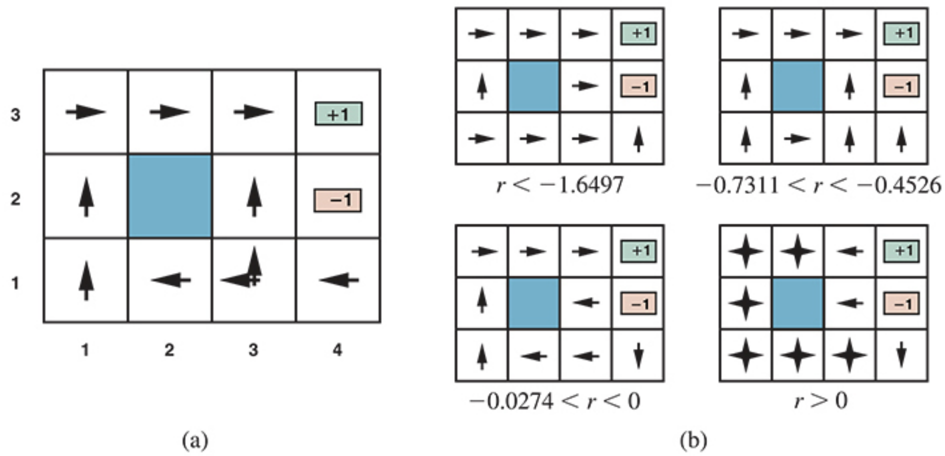


Figure 16.2 (a) The optimal policies for the stochastic environment with $r = -0.04$ for transitions between nonterminal states. There are two policies because in state (3,1) both *Left* and *Up* are optimal. (b) Optimal policies for four different ranges of r.

The balance of risk and reward changes depending on the value of $r = R(s, a, s')$ for transitions between nonterminal states. The policies shown in Figure 16.2(a) are optimal for $-0.0850 < r < -0.0273$. Figure 16.2(b) shows optimal policies for four other ranges of r. When $r < -1.6497$, life is so painful that the agent heads straight for the nearest exit, even if the exit is worth –1. When $-0.7311 < r < -0.4526$, life is quite unpleasant; the agent takes the shortest route to the +1 state from (2,1), (3,1), and (3,2), but from (4,1) the cost of reaching +1 is so high that the agent prefers to dive straight into –1. When life is only slightly dreary ($-0.0274 < r < 0$), the optimal policy takes *no risks at all.* In (4,1) and (3,2), the agent heads directly away from the –1 state so that it cannot fall in by accident, even though this means banging its head against the wall quite a few times. Finally, if $r > 0$, then life is positively enjoyable and the agent avoids *both* exits. As long as the actions in (4,1), (3,2), and (3,3) are as shown, every policy is optimal, and the agent obtains infinite total reward because it never enters a terminal state. It turns out that there are nine optimal policies in all for various ranges of r; Exercise 16.THRC asks you to find them.

The introduction of uncertainty brings MDPs closer to the real world than deterministic search problems. For this reason, MDPs have been studied in several fields, including AI, operations research, economics, and control theory. Dozens of solution algorithms have been proposed, several of which we discuss in Section 16.2. First, however, we spell out in more detail the definitions of utilities, optimal policies, and models for MDPs.

16.1.1 Utilities over time

In the MDP example in Figure 16.1, the performance of the agent was measured by a sum of rewards for the transitions experienced. This choice of performance measure is not arbitrary, but it is not the only possibility for the utility function[2] on environment histories, which we write as $U_h([s_0, a_0, s_1, a_1 ..., s_n])$.

The first question to answer is whether there is a **finite horizon** or an **infinite horizon** for decision making. A finite horizon means that there is a *fixed* time N after which nothing matters—the game is over, so to speak. Thus,

$$U_h([s_0, a_0, s_1, a_1, \ldots, s_{N+k}]) = U_h([s_0, a_0, s_1, a_1, \ldots, s_N])$$

for all $k > 0$. For example, suppose an agent starts at (3,1) in the 4 × 3 world of Figure 16.1, and suppose that N = 3. Then, to have any chance of reaching the +1 state, the agent must head directly for it, and the optimal action is to go *Up*. On the other hand, if N = 100, then there is plenty of time to take the safe route by going *Left. So, with a finite horizon, an optimal action in a given state may depend on how much time is left.* A policy that depends on the time is called **nonstationary**.

With no fixed time limit, on the other hand, there is no reason to behave differently in the same state at different times. Hence, an optimal action depends only on the current state, and the optimal policy is **stationary**. Policies for the infinite-horizon case are therefore simpler than those for the finite-horizon case, and we deal mainly with the infinite-horizon case in this chapter. (We will see later that for partially observable environments, the infinite-horizon case is not so simple.) Note that "infinite horizon" does not necessarily mean that all state sequences are infinite; it just means that there is no fixed deadline. There can be finite state sequences in an infinite-horizon MDP that contains a terminal state.

The next question we must decide is how to calculate the utility of state sequences. Throughout this chapter, we will **additive discounted rewards**: the utility of a history is

$$U_h\left([s_0, a_0,\ s_1, a_1, s_2, \ldots]\right) = R(s_0, a_0,\ s_1)\ +\ \gamma R(s_1, a_1, s_2) + \gamma^2 R(s_2, a_2, s_3) + \ldots\ ,$$

where the **discount factor** γ is a number between 0 and 1. The discount factor describes the preference of an agent for current rewards over future rewards. When γ is close to 0, rewards in the distant future are viewed as insignificant. When γ is close to 1, an agent is more willing to wait for long-term rewards. When γ is exactly 1, discounted rewards reduce to the special case of purely **additive rewards**. Notice that additivity was used implicitly in our use of path cost functions in heuristic search algorithms (Chapter 3).

There are several reasons why additive discounted rewards make sense. One is empirical: both humans and animals appear to value near-term rewards more highly than rewards in the distant future. Another is economic: if the rewards are monetary, then it really is better to get them sooner rather than later because early rewards can be invested and produce returns while you're waiting for the later rewards. In this context, a discount factor of γ is equivalent to an interest rate of $(1/\gamma) - 1$. For example, a discount factor of $\gamma = 0.9$ is equivalent to an interest rate of 11.1%.

A third reason is uncertainty about the true rewards: they may never arrive for all sorts of reasons that are not taken into account in the transition model. Under certain assumptions, a discount factor of *gamma* is equivalent to adding a probability $1 - \gamma$ of accidental termination at every time step, independent of the action taken.

A fourth justification arises from a natural property of preferences over histories. In the terminology of multiattribute utility theory (see Section 15.4), each transition $s_t \xrightarrow{a_t} s_{t+1}$ can be viewed as an **attribute** of the history $[s_0, a_0, s_1, a_1, s_2 ...]$. In principle, the utility function could depend in arbitrarily complex ways on these attributes. There is, however, a highly plausible preference-independence assumption that can be made, namely that the agent's preferences between state sequences are **stationary**.

Assume two histories $[s_0, a_0, s_1, a_1, s_2, ...]$ and $[s'_0, a'_0, s'_1, a'_1, s'_2, \ldots]$ begin with the same transition (i.e., $s_0 = s'_0$, $a_0 = a'_0$, and $s_1 = s'_1$). Then stationarity for preferences means that the two histories should be preference-ordered the same way as the histories $[s_1, a_1, s_2, ...]$ and $[s'_1, a'_1, s'_2, ...]$. In English, this means that if you prefer one future to another starting tomorrow, then you should still prefer that future if it were to start today instead. Stationarity is a fairly innocuous-looking assumption, but additive discounting is the only form of utility on histories that satisfies it.

A final justification for discounted rewards is that it conveniently makes some nasty infinities go away. With infinite horizons there is a potential difficulty: if the environment does not contain a terminal state, or if the agent never reaches one, then all environment histories will be infinitely long, and utilities with additive undiscounted rewards will generally be infinite. While we can agree that $+\infty$ is better than $-\infty$, comparing two state sequences with $+\infty$ utility is more difficult. There are three solutions, two of which we have seen already:

1. With discounted rewards, the utility of an infinite sequence is *finite*. In fact, if $\gamma < 1$ and rewards are bounded by $\pm R_{max}$, we have

$$U_h([s_0, a_0,\ s_1, \ldots]) = \sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) \leq \sum_{t=0}^{\infty} \gamma^t R_{\max} = \tfrac{R_{\max}}{1-\gamma}, \qquad (16.$$

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

2. If the environment contains terminal states *and if the agent is guaranteed to get to one eventually*, then we will never need to compare infinite sequences. A policy that is guaranteed to reach a terminal state is called a **proper policy**. With proper policies, we can use $\gamma = 1$ (i.e., additive undiscounted rewards). The first three policies shown in Figure 16.2(b) are proper, but the fourth is improper. It gains infinite total reward by staying away from the terminal states when the reward for transitions between nonterminal states is positive. The existence of improper policies can cause the standard algorithms for solving MDPs to fail with additive rewards, and so provides a good reason for using discounted rewards.
3. Infinite sequences can be compared in terms of the **average reward** obtained per time step. Suppose that transitions to square (1,1) in the 4×3 world have a reward of 0.1 while transitions to other nonterminal states have a reward of 0.01. Then a policy that does its best to stay in (1,1) will have higher average reward than one that stays elsewhere. Average reward is a useful criterion for some problems, but the analysis of average-reward algorithms is complex.

Additive discounted rewards present the fewest difficulties in evaluating histories, so we shall use them henceforth.

16.1.2 Optimal policies and the utilities of states

Having decided that the utility of a given history is the sum of discounted rewards, we can compare policies by comparing the *expected* utilities obtained when executing them. We assume the agent is in some initial state s and define S_t (a random variable) to be the state the agent reaches at time t when executing a particular policy π. (Obviously, $S_0 = s$, the state the agent is in now.) The probability distribution over state sequences $S_1, S_2, \ldots$, is determined by the initial state s, the policy π, and the transition model for the environment.

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

$$U^\pi(s) = E[\sum_{t=0}^{\infty} \gamma^t R(s_t, \pi(s_t), s_{t+1})], \qquad (16.2)$$

where the expectation E is with respect to the probability distribution over state sequences determined by s and π. Now, out of all the policies the agent could choose to execute starting in s, one (or more) will have higher expected utilities than all the others. We'll use π^*_s to denote one of these policies:

$$\pi_s^* = \operatorname*{argmax}_\pi U^\pi(s). \qquad (16.3)$$

Remember that π^*_s is a policy, so it recommends an action for every state; its connection with s in particular is that it's an optimal policy when s is the starting state. A remarkable consequence of using discounted utilities with infinite horizons is that the optimal policy is *independent* of the starting state. (Of course, the *action sequence* won't be independent; remember that a policy is a function specifying an action for each state.) This fact seems intuitively obvious: if policy π^*_a is optimal starting in a and policy π^*_b is optimal starting in b, then, when they reach a third state c, there's no good reason for them to disagree with each other, or with π^*_c, about what to do next.[3] So we can simply write π^* for an optimal policy.

Given this definition, the true utility of a state is just $U^{\pi^*}(s)$—that is, the expected sum of discounted rewards if the agent executes an optimal policy. We write this as $U(s)$, matching the notation used in Chapter 15 for the utility of an outcome. Figure 16.3 shows the utilities for the 4 × 3 world. Notice that the utilities are higher for states closer to the +1 exit, because fewer steps are required to reach the exit.

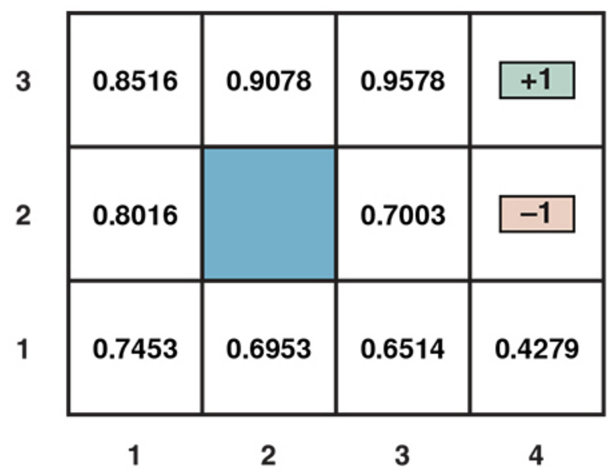


Figure 16.3 The utilities of the states in the 4 × 3 world with $\gamma = 1$ and $r = -0.04$ for transitions to nonterminal states.

The utility function $U(s)$ allows the agent to select actions by using the principle of maximum expected utility from Chapter 15—that is, choose the action that maximizes the reward for the next step plus the expected discounted utility of the subsequent state:

$$\pi^*(s) = \underset{a \in A(s)}{\operatorname{argmax}} \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma U(s')]. \tag{16.4}$$

We have defined the utility of a state, $U(s)$, as the expected sum of discounted rewards from that point onwards. From this, it follows that there is a direct relationship between the utility of a state and the utility of its neighbors: *the utility of a state is the expected reward for the next transition plus the discounted utility of the next state, assuming that the agent chooses the optimal action.* That is, the utility of a state is given by

$$U(s) = \max_{a \in A(s)} \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma U(s')]. \tag{16.5}$$

This is called the **Bellman equation**, after Richard Bellman (1957). The utilities of the states—defined by Equation (16.2) as the expected utility of subsequent state sequences—are solutions of the set of Bellman equations. In fact, they are the *unique* solutions, as we show in Section 16.2.1.

Let us look at one of the Bellman equations for the 4 × 3 world. The expression for $U(1,1)$ is

$$\begin{aligned}\max\{\quad & [0.8(-0.04+\gamma U(1{,}2))+0.1(-0.04+\gamma U(2{,}1))+0.1(-0.04+\gamma U(1{,}1))],\\ & [0.9(-0.04+\gamma U(1{,}1))+\ 0.1(-0.04+\gamma U(1{,}2))],\\ & [0.9(-0{:}04+\gamma U(1{,}1))+0.1(-0.04+\gamma U(2{,}1))],\\ & [0.8(-0.04+\gamma U(2{,}1))+0.1(-0.04+\gamma U(1{,}2))+0.1(-0.04+\gamma U(1{,}1))]\}\end{aligned}$$

where the four expressions correspond to *Up*, *Left*, *Down* and *Right* moves. When we plug in the numbers from Figure 16.3, with $\gamma = 1$, we find that *Up* is the best action.

Another important quantity is the **action–utility function**, or **Q–function**: $Q(s, a)$ is the expected utility of taking a given action in a given state. The Q–function is related to utilities in the obvious way:

$$U(s) = \max_a Q(s,a). \qquad (16.6)$$

Furthermore, the optimal policy can be extracted from the Q–function as follows:

$$\pi^*(s) = \operatorname*{argmax}_a Q(s,a). \qquad (16.7)$$

We can also develop a Bellman equation for Q-functions, noting that the expected total reward for taking an action is its immediate reward plus the discounted utility of the outcome state, which in turn can be expressed in terms of the Q-function:

$$\begin{aligned} Q(s,a) &= \sum_{s\prime} P(s\prime \mid s,a)[R(s,a,s') + \gamma U(s')]\\ &= \sum_{s\prime} P(s\prime| \ s,a)[R(s,a,s') + \gamma \max_{a\prime} Q(s',a')] \qquad (16.8)\end{aligned}$$

Solving the Bellman equations for U (or for Q) gives us what we need to find an optimal policy. The Q-function shows up again and again in algorithms for solving MDPs, so we shall use the following definition:

function Q-VALUE(mdp, s, a, U) **returns** a utility value
  return $\sum_{s\prime} P(s\prime| \ s,a)[R(s,a,s') + \gamma U[s']]$

16.1.3 Reward scales

Chapter 15 noted that the scale of utilities is arbitrary: an affine transformation leaves the optimal decision unchanged. We can replace $U(s)$ by $U'(s) = mU(s) + b$ where m and b are any constants such that $m > 0$. It is easy to see, from the definition of utilities as discounted sums of rewards, that a similar transformation of rewards will leave the optimal policy unchanged in an MDP:

$$R\prime(s,a,s\prime) = mR(s,a,s') + b.$$

It turns out, however, that the additive reward decomposition of utilities leads to significantly more freedom in defining rewards. Let $\Phi(s)$ be *any* function of the state s. Then, according to the **shaping theorem**, the following transformation leaves the optimal policy unchanged:

$$R\prime(s,a,s\prime) = R(s,a,s') + \gamma\Phi(s\prime) - \Phi(s). \qquad (16.9)$$

To show that this is true, we need to prove that two MDPs, M and M', have identical optimal policies as long as they differ only in their reward functions as specified in Equation (16.9). We start from the Bellman equation for Q, the Q-function for MDP M:

$$Q(s,a) = \sum_{s\prime} P(s\prime \Big| s,a)[R(s,a,s\prime) + \gamma \max_{a\prime} Q(s\prime,a\prime)].$$

Now let $Q'(s,a) = Q(s,a) - \Phi(s)$ and plug it into this equation; we get

$$Q\prime(s,a) \ + \ \Phi(s) = \sum_{s\prime} P(s\prime \middle| s,a)[R(s,a,s\prime) + \gamma \max_{a\prime} \left(Q\prime\left(s\prime,a\prime\right) \ + \ \Phi\left(s\prime\right)\right)].$$

which then simplifies to

$$\begin{aligned} Q\prime(s,a) \ &= \sum_{s\prime} P(s\prime \middle| s,a)[R(s,a,s\prime) + \gamma \Phi\left(s\prime\right) - \Phi\left(s\right) + \gamma \max_{a\prime} Q\prime\left(s\prime,a\prime\right)] \\ &= \sum_{s\prime} P(s\prime \middle| s,a)[R\prime(s,a,s\prime) + \gamma \max_{a\prime} Q\prime\left(s\prime,a\prime\right)]. \end{aligned}$$

In other words, $Q'(s, a)$ satisfies the Bellman equation for MDP M'. Now we can extract the optimal policy for M' using Equation (16.7):

$$\pi^*_{M\prime}(s) = \underset{a}{\operatorname{argmax}} Q\prime(s,a) = \underset{a}{\operatorname{argmax}} Q(s,a) - \Phi(s) = \underset{a}{\operatorname{argmax}} Q(s,a) = \pi^*_M(s).$$

The function $\Phi(s)$ is often called a **potential**, by analogy to the electrical potential (voltage) that gives rise to electric fields. The term $\gamma\Phi(s')$ — $\Phi(s)$ functions as a gradient of the potential. Thus, if $\Phi(s)$ has higher value in states that have higher utility, the addition of $\gamma\Phi(s')$ — $\Phi(s)$ to the reward has the effect of leading the agent "uphill" in utility.

At first sight, it may seem rather counterintuitive that we can modify the reward in this way without changing the optimal policy. It helps if we remember that *all policies are optimal* with a reward function that is zero everywhere. This means, according to the shaping theorem, that all policies are optimal for any potential-based reward of the form $R(s, a, s') = \gamma\Phi(s')$ — $\Phi(s)$. Intuitively, this is because with such a reward it doesn't matter which way the agent goes from A to B. (This is easiest to see when $\gamma = 1$: along any path the sum of rewards collapses to $\Phi(B) - \Phi(A)$, so all paths are equally good.) So adding a potential-based reward to any other reward shouldn't change the optimal policy.

The flexibility afforded by the shaping theorem means that we can actually help out the agent by making the immediate reward more directly reflect what the agent should do. In fact, if we set $\Phi(s) = U(s)$, then the greedy policy π_G with respect to the modified reward R' is also an optimal policy:

$$\begin{aligned} \pi_G(s) &= \underset{a}{\operatorname{argmax}} \sum_{s\prime} P(s\prime|s,a) R\prime(s,a,s\prime) \\ &= \underset{a}{\operatorname{argmax}} \sum_{s\prime} P(s\prime|s,a)[R(s,a,s\prime) + \gamma \Phi(s\prime) - \Phi\left(s\right)] \\ &= \underset{a}{\operatorname{argmax}} \sum_{s\prime} P(s\prime|s,a)[R(s,a,s\prime) + \gamma U(s\prime) - U(s)] \\ &= \underset{a}{\operatorname{argmax}} \sum_{s\prime} P(s\prime|s,a)[R(s,a,s\prime) + \gamma U(s\prime)] \\ &= \pi^*(s) \qquad (\text{by Equation (16.4)}) \,. \end{aligned}$$

Of course, in order to set $\Phi(s) = U(s)$, we would need to know $U(s)$; so there is no free lunch, but there is still considerable value in defining a reward function that is helpful to the extent possible. This is precisely what animal trainers do when they provide a small treat to the animal for each step in the target sequence.

16.1.4 Representing MDPs

The simplest way to represent $P(s' \mid s, a)$ and $R(s, a, s')$ is with big, three-dimensional tables of size $|S|^2|A|$. This is fine for small problems such as the 4×3 world, for which the tables have $11^2 \times 4 = 484$ entries each. In some cases, the tables are **sparse**—most entries are zero because each state s can transition to only a bounded number of states s'—which means the tables are of size $O(|S||A|)$. For larger problems, even sparse tables are far too big.

Just as in Chapter 15, where Bayesian networks were extended with action and utility nodes to create decision networks, we can represent MDPs by extending dynamic Bayesian networks (DBNs, see Chapter 14) with decision, reward, and utility nodes to create **dynamic decision networks**, or DDNs. DDNs are **factored representations** in the terminology of Chapter 2; they typically have an exponential complexity advantage over atomic representations and can model quite substantial real-world problems.

Figure 16.4, which is based on the DBN in Figure 14.13(b) (page 504), shows some elements of a slightly realistic model for a mobile robot that can charge itself. The state S_t is decomposed into four state variables:

- $\mathbf{X}_t$ consists of the two-dimensional location on a grid plus the orientation;
- $\mathbf{X}_t$ is the rate of change of $\mathbf{X}_t$;
- $Charging_t$ is true when the robot is plugged in to a power source;
- $Battery_t$ is the battery level, which we model as an integer in the range 0,..., 5.

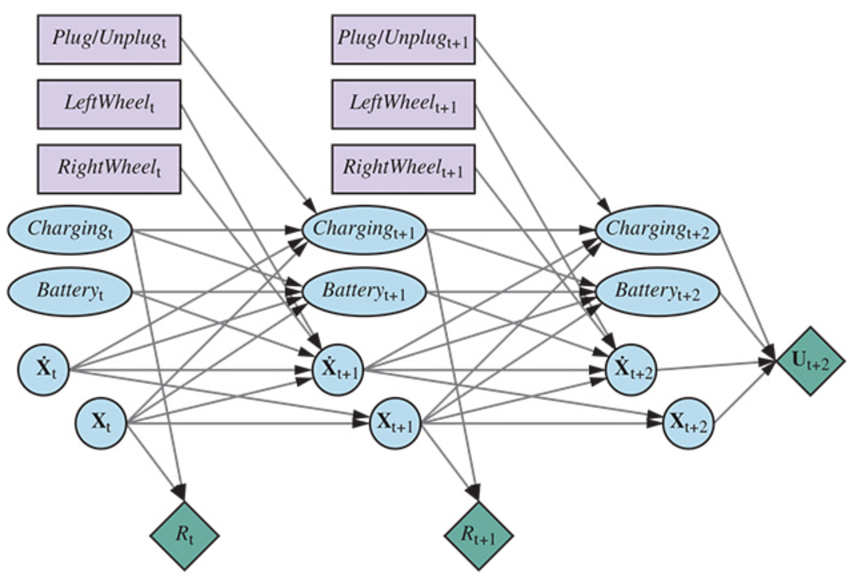


Figure 16.4 A dynamic decision network for a mobile robot with state variables for battery level, charging status, location, and velocity, and action variables for the left and right wheel motors and for charging.

The state space for the MDP is the Cartesian product of the ranges of these four variables. The action is now a set $\mathbf{A}_t$ of action variables, comprised of *Plug/Unplug*, which has three values (*plug*, *unplug*, and *noop*); *LeftWheel* for the power sent to the left wheel; and *RightWheel* for the power sent to the right wheel. The set of actions for the MDP is the Cartesian product of the ranges of these three variables. Notice that each action variable affects only a subset of the state variables.

The overall transition model is the conditional distribution $\mathbf{P}(\mathbf{X}_t+1 \mid \mathbf{X}_t, \mathbf{A}_t)$, which can be computed as a product of conditional probabilities from the DDN. The reward here is a single variable that depends only on the location $\mathbf{X}$ (for, say, arriving at a destination) and *Charging*, as the robot has to pay for electricity used; in this particular model, the reward doesn't depend on the action or the outcome state.

The network in Figure 16.4 has been projected two steps into the future. Notice that the network includes nodes for the *rewards* for times t and t +1, but the *utility* for time t + 2. This is because the agent must maximize the (discounted) sum of all future rewards, and $U(\mathbf{X}_{t+3})$ represents the reward for all rewards from t + 3 onwards. If a heuristic approximation to U is available, it can be included in the MDP representation in this way and used in lieu of further expansion. This approach is closely related to the use of bounded-depth search and heuristic evaluation functions for games in Chapter 6.

Another interesting and well-studied MDP is the game of Tetris (Figure 16.5(a)). The state variables for the game are the *CurrentPiece*, the *NextPiece*, and a bit-vector-valued variable *Filled* with one bit for each of the 10 × 20 board locations. Thus, the state space has $7 \times 7 \times 2^{200} \approx 10^{62}$ states. The DDN for Tetris is shown in Figure 16.5(b). Note that $Filled_{t+1}$ is a deterministic function of $Filled_t$ and A_t. It turns out that every policy for Tetris is proper (reaches a terminal state): eventually the board fills despite one's best efforts to empty it.

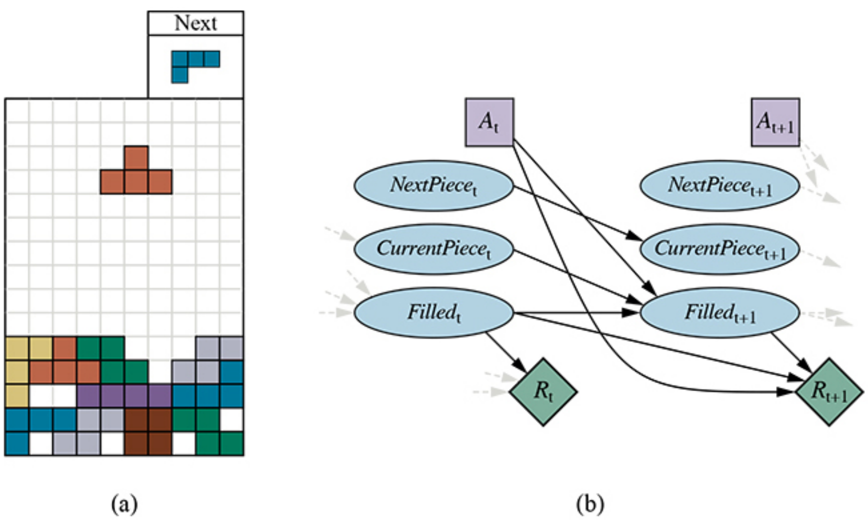


Figure 16.5 (a) The game of Tetris. The T-shaped piece at the top center can be dropped in any orientation and in any horizontal position. If a row is completed, that row disappears and the rows above it move down, and the agent receives one point. The next piece (here, the L-shaped piece at top right) becomes the current piece, and a new next piece appears, chosen at random from the seven piece types. The game ends if the board fills up to the top. (b) The DDN for the Tetris MDP.

16.2 Algorithms for MDPs

In this section, we present four different algorithms for solving MDPs. The first three, **value iteration**, **policy iteration**, and **linear programming**, generate exact solutions offline. The fourth is a family of online approximate algorithms that includes **Monte Carlo planning**.

16.2.1 Value Iteration

The Bellman equation (Equation (16.5)) is the basis of the **value iteration** algorithm for solving MDPs. If there are n possible states, then there are n Bellman equations, one for each state. The n equations contain n unknowns—the utilities of the states. So we would like to solve these simultaneous equations to find the utilities. There is one problem: the equations are *nonlinear,* because the "max" operator is not a linear operator. Whereas systems of linear equations can be solved quickly using linear algebra techniques, systems of nonlinear equations are more problematic. One thing to try is an *iterative* approach. We start with arbitrary initial values for the utilities, calculate the right-hand side of the equation, and plug it into the left-hand side—thereby updating the utility of each state from the utilities of its neighbors. We repeat this until we reach an equilibrium.

Let $U_i(s)$ be the utility value for state s at the ith iteration. The iteration step, called a **Bellman update**, looks like this:

$$U_{i+1}(s) \leftarrow \max_{a\in A(s)} \sum_{s\prime} P(s\prime|s,a)[R(s,a,s\prime) + \gamma U_i(s\prime)], \tag{16.10}$$

where the update is assumed to be applied simultaneously to all the states at each iteration. If we apply the Bellman update infinitely often, we are guaranteed to reach an equilibrium (see "convergence of value iteration" below), in which case the final utility values must be solutions to the Bellman equations. In fact, they are also the *unique* solutions, and the corresponding policy (obtained using Equation (16.4)) is optimal. The detailed algorithm, including a termination condition when the utilities are "close enough," is shown in Figure 16.6. Notice that we make use of the Q-VALUE function defined on page 559.

---

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

  repeat
    $U \leftarrow U'$; $\delta \leftarrow 0$
    for each state s **in** S **do**
      $U'[s] \leftarrow \max_{a\in A(s)}$ Q-VALUE(mdp, s, a, U)
      if $|U'[s] - U[s]| > \delta$ **then** $\delta \leftarrow |U'[s] - U[s]|$
  until $\delta \leq \epsilon(1-\gamma)/\gamma$
  return U

Figure 16.6 The value iteration algorithm for calculating utilities of states. The termination condition is from Equation (16.12).

We can apply value iteration to the 4×3 world in Figure 16.1(a). Starting with initial values of zero, the utilities evolve as shown in Figure 16.7(a). Notice how the states at different distances from (4,3) accumulate negative reward until a path is found to (4,3), whereupon the utilities start to increase. We can think of the value iteration algorithm as *propagating information* through the state space by means of local updates.

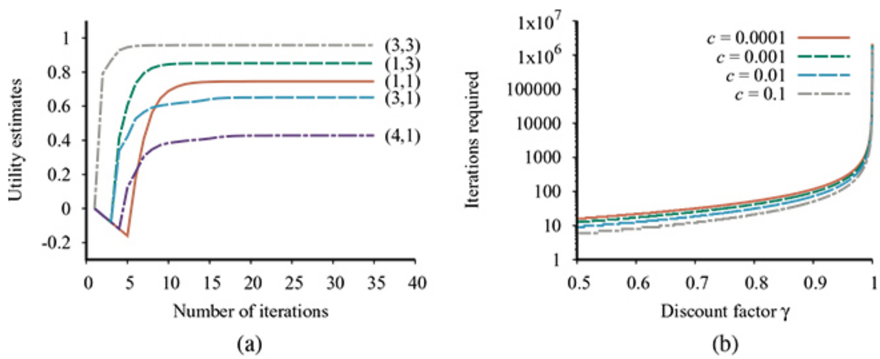


Figure 16.7 (a) Graph showing the evolution of the utilities of selected states using value iteration. (b) The number of value iterations required to guarantee an error of at most $\epsilon = c \, . \, R_{\max}$, for different values of c, as a function of the discount factor γ.

Convergence of value iteration

We said that value iteration eventually converges to a unique set of solutions of the Bellman equations. In this section, we explain why this happens. We introduce some useful mathematical ideas along the way, and we obtain some methods for assessing the error in the utility function returned when the algorithm is terminated early; this is useful because it means that we don’t have to run forever. This section is quite technical.

The basic concept used in showing that value iteration converges is the notion of a **contraction**. Roughly speaking, a contraction is a function of one argument that, when applied to two different inputs in turn, produces two output values that are “closer together,” by at least some constant factor, than the original inputs. For example, the function “divide by two” is a contraction, because, after we divide any two numbers by two, their difference is halved. Notice that the “divide by two” function has a fixed point, namely zero, that is unchanged by the application of the function. From this example, we can discern two important properties of contractions:

- A contraction has only one fixed point; if there were two fixed points they would not get closer together when the function was applied, so it would not be a contraction.
- When the function is applied to any argument, the value must get closer to the fixed point (because the fixed point does not move), so repeated application of a contraction always reaches the fixed point in the limit.

Now, suppose we view the Bellman update (Equation (16.10)) as an operator B that is applied simultaneously to update the utility of every state. Then the Bellman equation becomes $U = BU$ and the Bellman update equation can be written as

$$U_{i+1} \leftarrow BU_i.$$

Next, we need a way to measure distances between utility vectors. We will use the **max norm**, which measures the "length" of a vector by the absolute value of its biggest component:

$$\|U\| = \max_s \left|U(s)\right|.$$

With this definition, the "distance" between two vectors, $\|U - U'\|$, is the maximum difference between any two corresponding elements. The main result of this section is the following: Let U_i and U'_i be any two utility vectors. Then we have

$$\| BU_i - BU'_i \| \leq \gamma \| U_i - U'_i \|. \qquad (16.11)$$

That is, *the Bellman update is a contraction by a factor of* γ *on the space of utility vectors.*(Exercise 16.VICT provides some guidance on proving this claim.) Hence, from the properties of contractions in general, it follows that value iteration always converges to a unique solution of the Bellman equations whenever $\gamma < 1$.

We can also use the contraction property to analyze the *rate* of convergence to a solution. In particular, we can replace U'_i in Equation (16.11) with the *true* utilities U, for which $BU = U$. Then we obtain the inequality

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

If we view $\|U_i - U\|$ as the *error* in the estimate U_i, we see that the error is reduced by a factor of at least γ on each iteration. Thus, value iteration converges exponentially fast. We can calculate the number of iterations required as follows: First, recall from Equation (16.1) that the utilities of all states are bounded by $\pm R_{\max}/(1-\gamma)$. This means that the maximum initial error $\|U_0 - U\| \leq 2R_{\max}/(1-\gamma)$. Suppose we run for N iterations to reach an error of at most ϵ. Then, because the error is reduced by at least γ each time, we require $\gamma^N \cdot 2R_{\max}/(1-\gamma) \leq \epsilon$. Taking logs, we find that

$$N = \lceil \log(2R_{\max}/\epsilon(1-\gamma))/\log(1/\gamma) \rceil$$

iterations suffice. Figure 16.7(b) shows how N varies with γ, for different values of the ratio $\epsilon/R_{\max}$. The good news is that, because of the exponentially fast convergence, N does not depend much on the ratio $\epsilon/R_{\max}$. The bad news is that N grows rapidly as becomes close to 1. We can get fast convergence if we make γ small, but this effectively gives the agent a short horizon and could miss the long-term effects of the agent's actions.

The error bound in the preceding paragraph gives some idea of the factors influencing the run time of the algorithm, but is sometimes overly conservative as a method of deciding when to stop the iteration. For the latter purpose, we can use a bound relating the error to the size of the Bellman update on any given iteration. From the contraction property (Equation (16.11)), it can be shown that if the update is small (i.e., no state's utility changes by much), then the error, compared with the true utility function, also is small. More precisely,

$$\text{if} \quad \| U_{i+1} - U_i \| < \epsilon(1-\gamma)/\gamma \quad \text{then} \quad \| U_{i+1} - U \| < \epsilon. \qquad (16.12)$$

This is the termination condition used in the VALUE–ITERATION algorithm of Figure 16.6.

So far, we have analyzed the error in the utility function returned by the value iteration algorithm. *What the agent really cares about, however, is how well it will do if it makes its decisions on the basis of this utility function.* Suppose that after i iterations of value iteration, the agent has an estimate U_i of the true utility U and obtains the maximum expected utility (MEU) policy π_i based on one-step look-ahead using U_i (as in Equation (16.4)). Will the resulting behavior be nearly as good as the optimal behavior? This is a crucial question for any real agent, and it turns out that the answer is yes. $U^{\pi_i}(s)$ is the utility obtained if π_i is executed starting in s, and the **policy loss** $\| U^{\pi_i} - U \|$ is the most the agent can lose by executing π_i instead of the optimal policy π^*. The policy loss of π_i is connected to the error in U_i by the following inequality:

$$\text{if} \quad \| U_i - U \| < \varepsilon \quad \text{then} \quad \| U^{\pi_i} - U \| < 2\varepsilon. \tag{16.13}$$

In practice, it often occurs that π_i becomes optimal long before U_i has converged. Figure 16.8 shows how the maximum error in U_i and the policy loss approach zero as the value iteration process proceeds for the 4×3 environment with $\gamma = 0.9$. The policy π_i is optimal when $i = 5$, even though the maximum error in U_i is still 0.51.

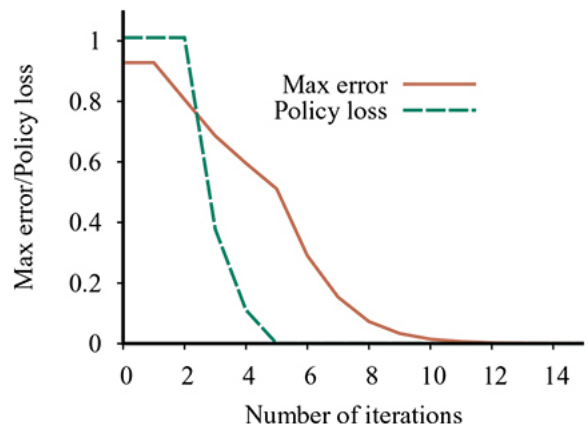


Figure 16.8 The maximum error $\| U_i - U \|$ of the utility estimates and the policy loss $\|U^{\pi_i} - U\|$, as a function of the number of iterations of value iteration on the 4×3 world.

Now we have everything we need to use value iteration in practice. We know that it converges to the correct utilities, we can bound the error in the utility estimates if we stop after a finite number of iterations, and we can bound the policy loss that results from executing the corresponding MEU policy. As a final note, all of the results in this section depend on discounting with $\gamma < 1$. If $\gamma = 1$ and the environment contains terminal states, then a similar set of convergence results and error bounds can be derived.

16.2.2 Policy iteration

In the previous section, we observed that it is possible to get an optimal policy even when the utility function estimate is inaccurate. If one action is clearly better than all others, then the exact magnitude of the utilities on the states involved need not be precise. This insight suggests an alternative way to find optimal policies. The **policy iteration** algorithm alternates the following two steps, beginning from some initial policy π_0:

- **Policy evaluation** : given a policy π_i, calculate $U_i = U^{\pi_i}$, the utility of each state if π_i were to be executed.

- **Policy improvement** : Calculate a new MEU policy π_{i+1}, using one-step look-ahead based on U_i (as in Equation (16.4)).

The algorithm terminates when the policy improvement step yields no change in the utilities. At this point, we know that the utility function U_i is a fixed point of the Bellman update, so it is a solution to the Bellman equations, and π_i must be an optimal policy. Because there are only finitely many policies for a finite state space, and each iteration can be shown to yield a better policy, policy iteration must terminate. The algorithm is shown in Figure 16.9. As with value iteration, we use the Q-VALUE function defined on page 559.

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

  repeat
     U ← POLICY-EVALUATION(π, U, mdp)
     unchanged? ← true
     for each state s in S do
        a* ← argmax_{a∈A(s)} Q-VALUE(mdp, s, a, U)
        if Q-VALUE(mdp, s, a*, U) > Q-VALUE(mdp, s, π[s], U) then
           π[s] ← a*; unchanged? ← false
  until unchanged?
  return π
```

Figure 16.9 The policy iteration algorithm for calculating an optimal policy.

How do we implement POLICY-EVALUATION? It turns out that doing so is simpler than solving the standard Bellman equations (which is what value iteration does), because the action in each state is fixed by the policy. At the ith iteration, the policy π_i specifies the action $\pi_i(s)$ in state s. This means that we have a simplified version of the Bellman equation (16.5) relating the utility of s (under π_i) to the utilities of its neighbors:

$$U_i(s) = \sum_{s\prime} P(s\prime|s, \pi_i(s))[R(s, \pi_i(s), s\prime) + \gamma U_i(s\prime)]. \tag{16.14}$$

For example, suppose π_i is the policy shown in Figure 16.2(a). Then we have $\pi_i(1,1) = Up, \pi_i(1,2) = Up$, and so on, and the simplified Bellman equations are

$$U_i(1,1) = 0.8[-0.04 + U_i(1,2)] + 0.1[-0.04 + U_i(2,1) + 0.1[-0.04 + U_i(1,1)]],$$
$$U_i(1,2) = 0.8[-0.04 + U_i(1,3)] + 0.2[-0.04 + U_i(1,2)],$$

and so on for all the states. The important point is that these equations are *linear,* because the “max” operator has been removed. For n states, we have n linear equations with n unknowns, which can be solved exactly in time $O(n^3)$ by standard linear algebra methods. If the transition model is sparse–that is, if each state transitions only to a small number of other states–then the solution process can be faster still.

For small state spaces, policy evaluation using exact solution methods is often the most efficient approach. For large state spaces, $O(n^3)$ time might be prohibitive. Fortunately, it is not necessary to do *exact* policy evaluation. Instead, we can perform some number of simplified value iteration steps (simplified because the policy is fixed) to give a reasonably good approximation of the utilities. The simplified Bellman update for this process is

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

and this is repeated several times to efficiently produce the next utility estimate. The resulting algorithm is called **modified policy iteration**.

The algorithms we have described so far require updating the utility or policy for all states at once. It turns out that this is not strictly necessary. In fact, on each iteration, we can pick *any subset* of states and apply *either* kind of updating (policy improvement or simplified value iteration) to that subset. This very general algorithm is called **asynchronous policy iteration**. Given certain conditions on the initial policy and initial utility function, asynchronous policy iteration is guaranteed to converge to an optimal policy. The freedom to choose any states to work on means that we can design much more efficient heuristic algorithms–for example, algorithms that concentrate on updating the values of states that are likely to be reached by a good policy. There's no sense planning for the results of an action you will never do.

16.2.3 Linear programming

Linear programming or LP, which was mentioned briefly in Chapter 4 (page 139), is a general approach for formulating constrained optimization problems, and there are many industrial-strength LP solvers available. Given that the Bellman equations involve a lot of sums and maxes, it is perhaps not surprising that solving an MDP can be reduced to solving a suitably formulated linear program.

The basic idea of the formulation is to consider as variables in the LP the utilities $U(s)$ of each state s, noting that the utilities for an optimal policy are the highest utilities attainable that are consistent with the Bellman equations. In LP language, that means we seek to minimize $U(s)$ for all s subject to the inequalities

$$U(s) \geq \sum_{s\prime} P(s' \mid s, a)[R(s, a, s\prime) + \gamma U(s\prime)]$$

for every state s and every action a.

This creates a connection from dynamic programming to linear programming, for which algorithms and complexity issues have been studied in great depth. For example, from the fact that linear programming is solvable in polynomial time, one can show that MDPs can be solved in time polynomial in the number of states and actions and the number of bits required to specify the model. In practice, it turns out that LP solvers are seldom as efficient as dynamic programming for solving MDPs. Moreover, polynomial time may sound good, but the number of states is often very large. Finally, it's worth remembering that even the simplest and most uninformed of the search algorithms in Chapter 3 runs in linear time in the number of states and actions.

16.2.4 Online algorithms for MDPs

Value iteration and policy iteration are *offline* algorithms: like the A* algorithm in Chapter 3, they generate an optimal solution for the problem, which can then be executed by a simple agent. For sufficiently large MDPs, such as the Tetris MDP with 10^{62} states, exact offline solution, even by a polynomial–time algorithm, is not possible. Several techniques have been developed for approximate offline solution of MDPs; these are covered in the notes at the end of the chapter and in Chapter 23 (Reinforcement Learning).

Here we will consider online algorithms, analogous to those used for game playing in Chapter 6, where the agent does a significant amount of computation at each decision point rather than operating primarily with precomputed information.

The most straightforward approach is actually a simplification of the EXPECTIMINIMAX algorithm for game trees with chance nodes: the EXPECTIMAX algorithm builds a tree of alternating max and chance nodes, as illustrated in Figure 16.10. (There is a slight difference from standard EXPECTIMINIMAX in that there are rewards on nonterminal as well as terminal transitions.) An evaluation function can be applied to the nonterminal leaves of the tree, or they can be given a default value. A decision can be extracted from the search tree by backing up the utility values from the leaves, taking an average at the chance nodes and taking the maximum at the decision nodes.

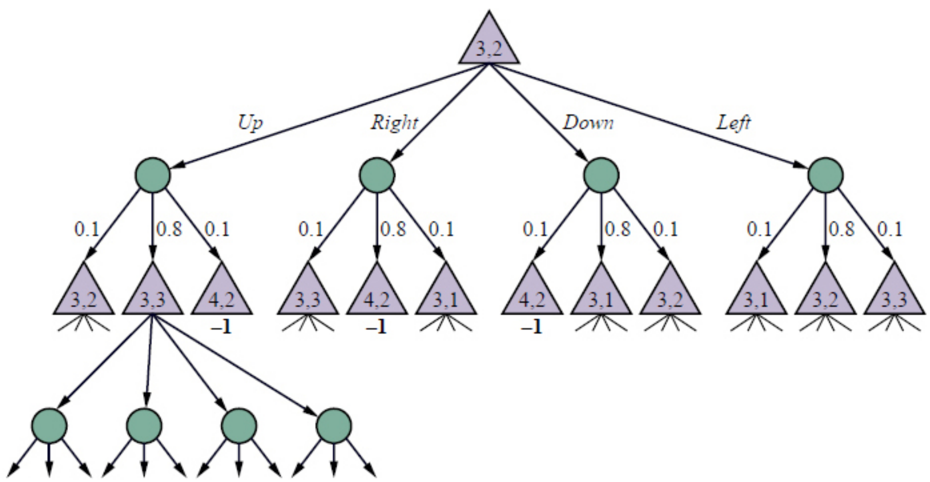


Figure 16.10 Part of an expectimax tree for the 4 × 3 MDP rooted at (3,2). The triangular nodes are max modes and the circular nodes are chance nodes.

For problems in which the discount factor γ is not too close to 1, the ϵ-horizon is a useful concept. Let ϵ be a desired bound on the absolute error in the utilities computed from an expectimax tree of bounded depth, compared to the exact utilities in the MDP. Then the ϵ-horizon is the tree depth H such that the sum of rewards beyond any leaf at that depth is less than e-roughly speaking, anything that happens after H is irrelevant because it's so far in the future. Because the sum of rewards beyond H is bounded by $\gamma^H R_{max}/(1-\gamma)$, a depth of $H = \lceil \log_\gamma \epsilon (1-\gamma)/R_{max} \rceil$ suffices. So, building a tree to this depth gives near-optimal decisions. For example, with $\gamma = 0.5$, $\epsilon = 0.1$, and $R_{max} = 1$, we find $H = 5$, which seems reasonable. On the other hand, if $\gamma = 0.9$, $H = 44$, which seems less reasonable!

In addition to limiting the depth, it is also possible to avoid the potentially enormous branching factor at the chance nodes. (For example, if all the conditional probabilities in a DBN transition model are nonzero, the transition probabilities, which are given by the product of the conditional probabilities, are also nonzero, meaning that every state has *some* probability of transitioning to every other state.)

As noted in Section 13.4, expectations with respect to a probability distribution P can be approximated by generating N samples from P and using the sample mean. In mathematical form, we have

$$\sum_x P(x)f(x) \approx \frac{1}{N}\sum_{i=1}^{N} f(x_i).$$

So, if the branching factor is very large, meaning that there are very many possible x values, a good approximation to the value of the chance node can be obtained by sampling a bounded number of outcomes from the action. Typically, the samples will focus on the *most likely* outcomes because those are most likely to be generated.

If you look closely at the tree in Figure 16.10, you will notice something: it isn't really a tree. For example, the root (3,2) is also a leaf, so one ought to consider this as a graph, and one ought to constrain the value of the leaf (3,2) to be the same as the value of the root (3,2), since they are the same state. In fact, this line of thinking quickly brings us back to the Bellman equations that relate the values of states to the values of neighboring states. The explored states actually constitute a sub-MDP of the original MDP, and this sub-MDP can be solved using any of the algorithms in this chapter to yield a decision for the current state. (Frontier states are typically given a fixed estimated value.)

This general approach is called **real–time dynamic programming (RTDP)** and is quite analogous to LRTA in Chapter 4. Algorithms of this kind can be quite effective in moderatesized domains such as grid worlds; in larger domains such as Tetris, there are two issues. First, the state space is such that any manageable set of explored states contains very few repeated states, so one might as well use a simple expectimax tree. Second, a simple heuristic for frontier nodes may not be enough to guide the agent, particularly if rewards are sparse.

One possible fix is to apply reinforcement learning to generate a much more accurate heuristic (see Chapter 23). Another approach is to look further ahead in the MDP using the Monte Carlo approach of Section 6.4. In fact, the UCT algorithm from Figure 6.10 was developed originally for MDPs rather than games. The changes required to solve MDPs rather than games are minimal: they arise primarily from the fact that the opponent (nature) is stochastic and from the need to keep track of rewards rather than just wins and losses.

When applied to the 4 × 3 world, the performance of UCT is not especially impressive. As Figure 16.11 shows, it takes 160 playouts on average to reach a total reward of 0.4, whereas an optimal policy has an expected total reward of 0.7453 from the initial state (see Figure 16.3). One reason UCT can have difficulty is that is builds a tree rather than a graph and uses (an approximation to) expectimax rather than dynamic programming. The 4 × 3 world is very “loopy”: although there are only 9 nonterminal states, UCT's playouts often continue for more than 50 actions.

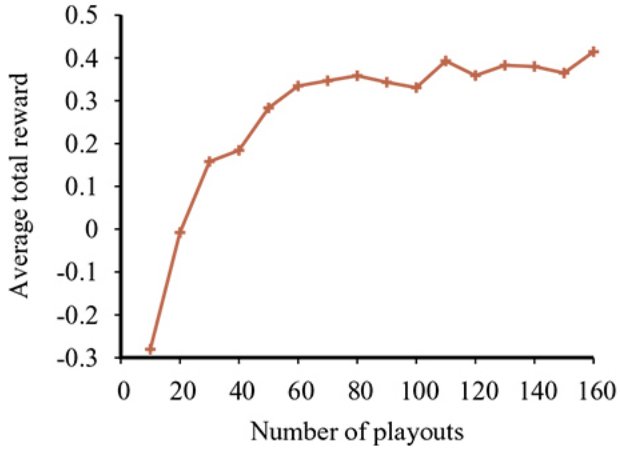


Figure 16.11 Performance of UCT as a function of the number of playouts per move for the 4×3 world using a random playout policy, averaged over 1000 runs per data point.

UCT seems better suited for Tetris, where the playouts go far enough into the future to give the agent a sense of whether a potentially risky move will work out in the end or cause a massive pile-up. Exercise 16.UCTT explores the application of UCT to Tetris. One particularly interesting question is how much a simple simulation policy can help–for example, one that avoids creating overhangs and puts pieces as low as possible.

16.3 Bandit Problems

In Las Vegas, a *one-armed bandit* is a slot machine. A gambler can insert a coin, pull the lever, and collect the winnings (if any). An **n-armed bandit** has *n* levers. Behind each lever is a fixed but unknown probability distribution of winnings; each pull samples from that unknown distribution.

The gambler must choose which lever to play on each successive coin–the one that has paid off best, or maybe one that has not been tried yet? This is an example of the ubiquitous tradeoff between **exploitation** of the current best action to obtain rewards and **exploration** of previously unknown states and actions to gain information, which can in some cases be converted into a better policy and better long-term rewards. In the real world, one constantly has to decide between continuing in a comfortable existence, versus striking out into the unknown in the hopes of a better life.

The *n*-armed bandit problem is a formal model for real problems in many vitally important areas, such as deciding which of *n* possible new treatments to try to cure a disease, which of *n* possible investments to put part of your savings into, which of *n* possible research projects to fund, or which of *n* possible advertisements to show when the user visits a particular web page.

Early work on the problem began in the U.S. during World War II; it proved so recalcitrant that Allied scientists proposed that "the problem be dropped over Germany, as the ultimate instrument of intellectual sabotage" (Whittle, 1979).

It turns out that the scientists, both during and after the war, were trying to prove "obviously true" facts about bandit problems that are, in fact, false. (As Bradt *et al.* (1956) put it, "There are many nice properties which optimal strategies do not possess.") For example, it was generally assumed that an optimal policy would eventually settle on the best arm in the long run; in fact, there is a finite probability that an optimal policy settles on a suboptimal arm. We now have a solid theoretical understanding of bandit problems as well as useful algorithms for solving them.

There are several different definitions of **bandit problems**; one of the cleanest and most general is as follows:

- Each arm M_i is a **Markov reward process** or MRP, that is, an MDP with only one Markov reward process possible action a_i. It has states S_i, transition model $P_i(s' \mid s,a_i)$, and reward $R_i(s,a_i, s')$. The arm defines a distribution over sequences of rewards $R_{i,0}$, $R_{i,1}$, $R_{i,2}$,..., where each $R_{i,t}$ is a random variable.
- The overall bandit problem is an MDP: the state space is given by the Cartesian product $S = S_1 \times ... \times S_n$; the actions are $a_1,...,a_n$, the transition model updates the state of whichever arm M_i is selected, according to its specific transition model, leaving the other arms unchanged; and the discount factor is Υ.

This definition is very general, covering a wide range of cases. The key property is that the arms are independent, coupled only by the fact that the agent can work on only one arm at a time. It's possible to define a still more general version in which fractional efforts can be applied to all arms simultaneously, but the total effort across all arms is bounded; the basic results described here carry over to this case.

We will see shortly how to formulate a typical bandit problem within this framework, but let's warm up with the simple special case of deterministic reward sequences. Let $\gamma = 0.5$, and suppose that there are two arms labeled M and M_1. Pulling M multiple times yields the sequence of rewards 0,2,0,7.2,0,0,..., while pulling M_1 yields 1,1,1,... (Figure 16.12(a)). If, at the beginning, one had to commit to one arm or the other and stick with it, the choice would be made by computing the utility (total discounted reward) for each arm:

$$U(M) = (1.0 \times 0) + (0.5 \times 2) + (0.5^2 \times 0) + (0.5^3 \times 7.2) = 1.9$$

$$U(M_1) = \sum_{t=0}^{\infty} 0.5^t = 2.0.$$

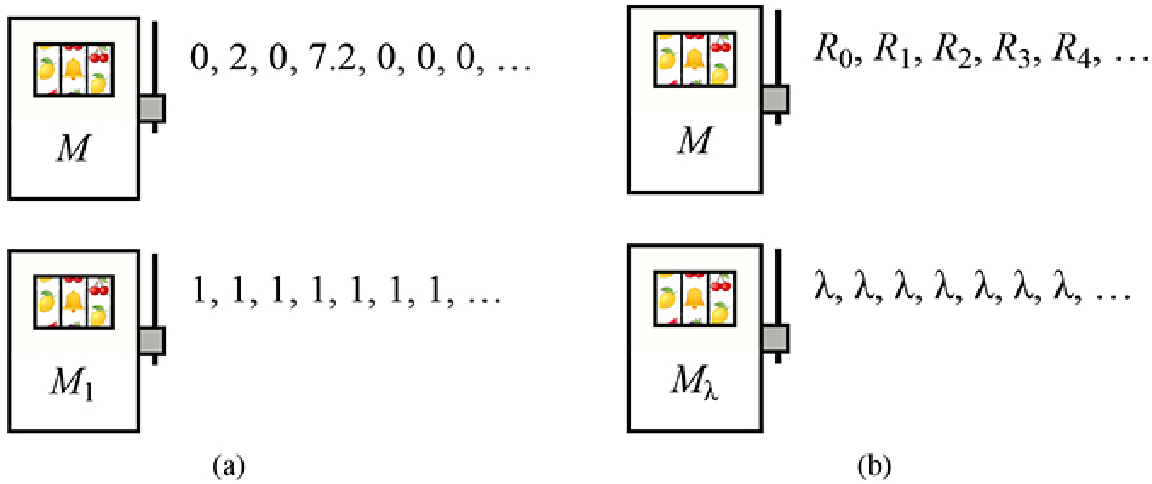


Figure 16.12 (a) A simple deterministic bandit problem with two arms. The arms can be pulled in any order, and each yields the sequence of rewards shown. (b) A more general case of the bandit in (a), where the first arm gives an arbitrary sequence of rewards and the second arm gives a fixed reward λ.

One might think the best choice is to go with M_1, but a moment's more thought shows that starting with M and then switching to M_1 after the fourth reward gives the sequence $S = 0,2,0,7.2,1,1,1,\ldots$, for which

$$U(S) = (1.0 \times 0) + (0.5 \times 2) + (0.5^2 \times 0) + (0.5^3 \times 7.2) + \sum_{t=4}^{\infty} 0.5^t = 2.025.$$

Hence the strategy S that switches from M to M_1 at the right time is better than either arm individually. In fact, S is optimal for this problem: all other switching times give less reward.

Let's generalize this case slightly, so that now the first arm M yields an arbitrary sequence $R_0,R_1,R_2,\ldots$ (which may be known or unknown) and the second arm M_λ yields $\lambda,\lambda,\lambda,\ldots$ for some known fixed constant λ (see Figure 16.12(b)). This is called a **one–armed bandit** in the literature, because it is formally equivalent to the case where there is one arm M that produces rewards R_0, R_1, R_2,... and costs λ for each pull. (Pulling arm M is equivalent to not pulling M_λ, so it gives up a reward of λ each time.) With just one arm, the only choice is to whether to pull again or to stop. If you pull the first arm T times (i.e., at times 0,1,..., $T - 1$ we say that the **stopping time** is T.

Going back to our version with M and M_λ, let's assume that after T pulls of the first arm, an optimal strategy eventually pulls the second arm for the first time. Since no information is gained from this move (we already know the payoff will be λ), at time $T + 1$ we will be in the same situation and thus an optimal strategy must make the same choice.

Equivalently, we can say that an optimal strategy is to run arm M up to time T and then switch to M_λ for the rest of time. It's possible that $T = 0$ if the strategy chooses M_λ immediately, or $T = \infty$ if the strategy never chooses M_λ, or somewhere in between. Now let's consider the value of λ such that an optimal strategy is *exactly indifferent* between (a) running M up to the best possible stopping time and then switching to M_λ forever, and (b) choosing M_λ immediately. At the tipping point we have

$$\max_{T>0} E\left[\left(\sum_{t=0}^{T-1}\gamma^t R_t\right) + \sum_{t=T}^{\infty}\gamma^t\lambda\right] = \sum_{t=0}^{\infty}\gamma^t\lambda,$$

which simplifies to

$$\lambda = \max_{T>0}\frac{E\left(\sum_{t=0}^{T-1}\gamma^t R_t\right)}{E\left(\sum_{t=0}^{T-1}\gamma^t\right)}. \qquad (16.15)$$

This equation defines a kind of "value" for M in terms of its ability to deliver a stream of timely rewards; the numerator of the fraction represents a utility while the denominator can be thought of as a "discounted time," so the value describes the maximum obtainable utility per unit of discounted time. (It's important to remember that T in the equation is a stopping time, which is governed by a rule for stopping rather than being a simple integer; it reduces to a simple integer only when

M is a deterministic reward sequence.) The value defined in Equation (16.15) is called the **Gittins index** of M.

The remarkable thing about the Gittins index is that it provides a very simple optimal policy for any bandit problem: *pull the arm that has the highest Gittins index, then update the Gittins indices.* Furthermore, because the index of arm M_i depends only on the properties of that arm, an optimal decision on the first iteration can be calculated in $O(n)$ time, where n is the number of arms. And because the Gittins indices of the arms that are not selected remain unchanged, each decision after the first one can be calculated in $O(1)$ time.

16.3.1 Calculating the Gittins index

To get more of a feel for the index, let's calculate the value of the numerator, denominator, and ratio in Equation (16.15) for different possible stopping times on the deterministic reward sequence 0,2,0,7.2,0,0,0,...:

Clearly, the ratio will decrease from here on, because the numerator remains constant while the denominator continues to increase. Thus, the Gittins index for this arm is 1.0133, the maximum value attained by the ratio. In combination with a fixed arm M_λ with $0 < \lambda \leq 1.0133$, the optimal policy collects the first four rewards from M and then switches to M_λ. For $\lambda > 1.0133$, the optimal policy always chooses M_λ.

To calculate the Gittins index for a general arm M with current state s, we simply make the following observation: at the tipping point where an optimal policy is indifferent between choosing arm M and choosing the fixed arm M_λ, the value of choosing M is the same as the value of choosing an infinite sequence of λ-rewards.

Suppose we augment M so that at each state in M, the agent has two choices: either continue with M as before, or quit and receive an infinite sequence of λ-rewards (see Figure 16.13(a)). This turns M into an MDP, whose optimal policy is just the optimal stopping rule for M. Hence the value of an optimal policy in this new MDP is equal to the value of an infinite sequence of λ–rewards, that is, $\lambda/(1 - \gamma)$. So we can just solve this MDP ... but, unfortunately, we don't know the value of γ to put into the MDP, as this is precisely what we are trying to calculate. But we *do* know that, at the tipping point, an optimal policy is indifferent between M and M_λ, so we could replace the choice to get an infinite sequence of λ-rewards with the choice to go back and restart M from its initial state s. (More precisely, we add a new action in every state that has the same rewards and outcomes as the action available in s; see Exercise 16.KATV.) This new MDP M^s, called a **restart MDP**, is illustrated in Figure 16.13(b).

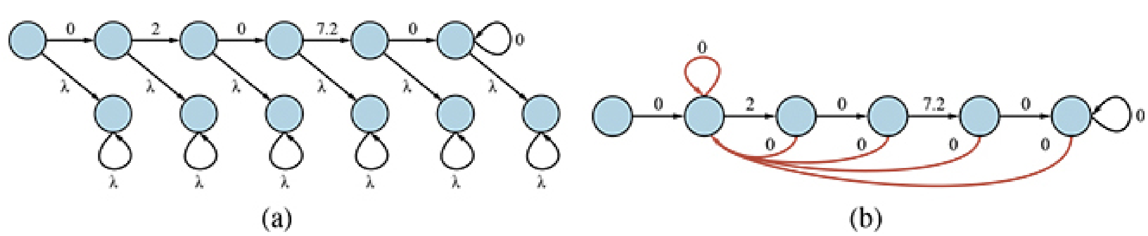


Figure 16.13 (a) The reward sequence M = 0,2,0,7.2,0,0,0,... augmented with a choice to switch permanently to a constant arm M_λ at each point. (b) An MDP whose optimal value is exactly equivalent to the optimal value for (a), at the point where the optimal policy is indifferent between M and M_λ.

We have the general result that the Gittins index for an arm M in state s is equal to 1 – γ times the value of an optimal policy for the restart MDP M^s. This MDP can be solved by any of the algorithms in Section 16.2. Value iteration applied to M^s in Figure 16.13(b) gives a value of 2.0266 for the start state, so we have λ = 2.0266 (1 – γ) = 1.0133 as before.

16.3.2 The Bernoulli bandit

Perhaps the simplest and best-known instance of a bandit problem is the **Bernoulli bandit**, where each arm M_i, produces a reward of 0 or 1 with a fixed but unknown probability μ_i. The state of arm M_i, is defined by s_i, and f_i, the counts of successes (1s) and failures (0s) so far for that arm; the transition probability predicts the next outcome to be 1 with probability $(s_i)/(s_i + f_i)$ and 0 with probability $(f_i)/(s_i + f_i)$. The counts are initialized to 1 so that the initial probabilities are 1/2 rather than 0/0.[4] The Markov reward process is shown in Figure 16.14(a).

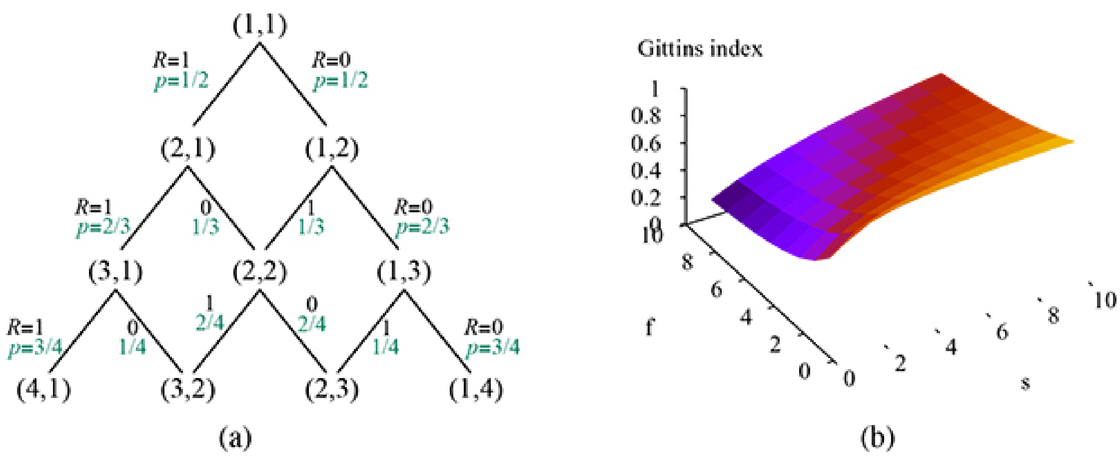


Figure 16.14 (a) States, rewards, and transition probabilities for the Bernoulli bandit. (b) Gittins indices for the states of the Bernoulli bandit process.

We cannot quite apply the transformation of the preceding section to calculate the Gittins index of the Bernoulli arm because it has infinitely many states. We can, however, obtain a very accurate approximation by solving the truncated MDP with states up to $s_i + f_i = 100$ and $\gamma = 0.9$. The results are shown in Figure 16.14(b). The results are intuitively reasonable: we see that, generally speaking, arms with higher payoff probabilities are preferred, but there is also an **exploration bonus** associated with arms that have only been tried a few times. For example, the index for the state

(3,2) is higher than the index for the state (7,4) (0.7057 vs. 0.6922), even though the estimated value at (3,2) is lower (0.6 vs. 0.6364).

16.3.3 Approximately optimal bandit policies

Calculating Gittins indices for more realistic problems is rarely easy. Fortunately, the general properties observed in the preceding section–namely, the desirability of some combination of estimated value and uncertainty–lend themselves to the creation of simple policies that turn out to be “nearly as good” as optimal policies.

The first class of methods uses the **upper confidence bound** or UCB heuristic, previously introduced for Monte Carlo tree search (Figure 6.11 on page 209). The basic idea is to use the samples from each arm to establish a **confidence interval** for the value of the arm, that is, a range within which the value can be estimated to lie with high confidence; then choose the arm with the highest upper bound on its confidence interval. The upper bound is the current mean value estimate $\widehat{\mu}_i$ plus some multiple of the standard deviation of the uncertainty in the value. The standard deviation is proportional to $\sqrt{1/N_i}$, where N_i is the number of times arm M_i has been sampled. So we have an approximate index value for arm M_i given by

$$UCB(M_i) = \hat{\mu}_i + g(N)/\sqrt{N_i},$$

where $g(N)$ is an appropriately chosen function of N, the total number of samples drawn from all arms. A UCB policy simply picks the arm with the highest UCB value. Notice that the UCB value is not strictly an index because it depends on N, the total number of samples drawn across all arms, and not just on the arm itself.

The precise definition of g determines the **regret** relative to the clairvoyant policy, which simply picks the best arm and yields average reward μ^*. A famous result due to Lai and Robbins (1985) shows that, for the undiscounted case, no possible algorithm can have regret that grows more slowly than $O(\log N)$. Several different choices of g lead to a UCB policy that matches this growth; for example, we can use $g(N) = (2\log(1 + N\log^2 N))^{1/2}$.

A second method, **Thompson sampling** (Thompson, 1933), chooses an arm randomly according to the probability that the arm is in fact optimal, given the samples so far. Suppose that $P_i(\mu_i)$ is the current probability distribution for the true value of arm M_i. Then a simple way to implement Thompson sampling is to generate one sample from each P_i and then pick the best sample. This algorithm also has a regret that grows as $O(\log N)$.

16.3.4 Non-indexable variants

Bandit problems were motivated in part by the task of testing new medical treatments on seriously ill patients. For this task, the goal of maximizing the total number of successes over time clearly makes sense: each successful test means a life saved, each failure a life lost.

If we change the assumptions slightly, however, a different problem emerges. Suppose that, instead of determining the best medical treatment for each new human patient, we are instead testing different drugs on samples of bacteria with the goal of deciding which drug is best. We will then put that drug into production and forgo the others. In this scenario there is no additional cost if the bacteria dies–there is a fixed cost for each test, but we don't have to minimize test failures; rather we are just trying to make a good decision as fast as possible.

The task of choosing the best option under these conditions is called a **selection problem**. Selection problems are ubiquitous in industrial and personnel contexts. One often must decide which supplier to use for a process; or which candidate employees to hire. Selection problems are superficially similar to the bandit problem but have different mathematical properties. In particular, *no index function exists for selection problems.* The proof of this fact requires showing any scenario where the optimal policy switches its preferences for two arms M_1 and M_2 when a third arm M_3 is added (see Exercise 16.SELC).

Chapter 6 introduced the concept of **metalevel** decision problems such as deciding what computations to make during a game-tree search prior to making a move. A metalevel decision of this kind is also a selection problem rather than a bandit problem. Clearly, a node expansion or evaluation *costs* the same amount of time whether it produces a high or a low output value. It is perhaps surprising, then, that the Monte Carlo tree search algorithm (see page 209) has been so successful, given that it tries to solve selection problems with the UCB heuristic, which was designed for bandit problems. Generally speaking, one expects optimal bandit algorithms to explore much less than optimal selection algorithms, because the bandit algorithm assumes that a failed trial costs real money.

An important generalization of the bandit process is the **bandit superprocess** or **BSP**, in which each arm is a full Markov decision process in its own right, rather than being a Markov reward process with only one possible action. All other properties remain the same: the arms are independent, only one (or a bounded number) can be worked on at a time, and there is a single discount factor.

Examples of BSPs include daily life, where one can attend to one task at a time, even though several tasks may need attention; project management with multiple projects; teaching with multiple pupils needing individual guidance; and so on. The ordinary term for this is **multitasking**. It is so ubiquitous as to be barely noticeable: when formulating a real-world decision problem, decision analysts rarely ask if their client has other, unrelated problems.

One might reason as follows: "If there are n disjoint MDPs then it is obvious that an optimal policy overall is built from the optimal solutions of the individual MDPs. Given its optimal policy π_i each MDP becomes a Markov reward process where there is only one action $\pi_i(s)$ in each state s. So we have reduced the n-armed bandit superprocess to an n-armed bandit process." For example, if a real–estate developer has one construction crew and several shopping centers to build, it seems to be just common sense that one should devise the optimal construction plan for each shopping center and then solve the bandit problem to decide where to send the crew each day.

While this sounds highly plausible, it is incorrect. In fact, the globally optimal policy for a BSP may include actions that are locally suboptimal from the point of view of the constituent MDP in which they are taken. The reason for this is that the availability of other MDPs in which to act changes the balance between short-term and long-term rewards in a component MDP. In fact, it tends to lead to greedier behavior in each MDP (seeking short-term rewards) because aiming for long-term reward in one MDP would delay rewards in all the other MDPs.

For example, suppose the locally optimal construction schedule for one shopping center has the first shop available for rent by week 15, whereas a suboptimal schedule costs more but has the first shop available by week 5. If there are four shopping centers to build, it might be better to use the locally suboptimal schedule in each so that rents start coming in from weeks 5, 10, 15, and 20, rather than weeks 15, 30,45, and 60. In other words, what would be only a 10-week delay for a single MDP turns into a 40-week delay for the fourth MDP. In general, the globally and locally optimal policies necessarily coincide only when the discount factor is 1; in that case, there is no cost to delaying rewards in any MDP.

The next question is how to solve BSPs. Obviously, the globally optimal solution for a BSP could be computed by converting it into a global MDP on the Cartesian–product state space. The number of states would be exponential in the number of arms of the BSP, so this would be horrendously impractical.

Instead, we can take advantage of the loose nature of the interaction between the arms. This interaction arises only from the agent's limited ability to attend to the arms simultaneously. To some extent, the interaction can be modeled by the notion of **opportunity cost**: how much utility is given up per time step by not devoting that time step to another arm. The higher the opportunity cost, the more necessary it is to generate early rewards in a given arm. In some cases, an optimal policy in a given arm is unaffected by the opportunity cost. (Trivially, this is true in a Markov reward process because there is only one policy.) In that case, an optimal policy can be applied, converting that arm into a Markov reward process.

Such an optimal policy, if it exists, is called a **dominating policy**. It turns out that by adding actions to states, it is always possible to create a relaxed version of an MDP (see Section 3.6.2) so that it has a dominating policy, which thus gives an upper bound on the value of acting in the arm. A lower bound can be computed by solving each arm separately (which may yield a suboptimal policy overall) and then computing the Gittins indices. If the lower bound for acting in one arm is higher than the upper bounds for all other actions, then the problem is solved; if not, then a combination of look–ahead search and recomputation of bounds is guaranteed to eventually identify an optimal policy for the BSP. With this approach, relatively large BSPs (10^{40} states or more) can be solved in a few seconds.

16.4 Partially Observable MDPs

The description of Markov decision processes in Section 16.1 assumed that the environment was **fully observable**. With this assumption, the agent always knows which state it is in. This, combined with the Markov assumption for the transition model, means that the optimal policy depends only on the current state.

When the environment is only **partially observable**, the situation is, one might say, much less clear. The agent does not necessarily know which state it is in, so it cannot execute the action $\pi(s)$ recommended for that state. Furthermore, the utility of a state s and the optimal action in s depend not just on s, but also on *how much the agent knows* when it is in s. For these reasons, **partially observable MDPs** (or **POMDPs**–pronounced "pom–dee–pees") are usually viewed as much more difficult than ordinary MDPs. We cannot avoid POMDPs, however, because the real world is one.

16.4.1 Definition of POMDPs

To get a handle on POMDPs, we must first define them properly. A POMDP has the same elements as an MDP–the transition model $P(s' \mid s,a)$, actions $A(s)$, and reward function $R(s, a, s')$–but, like the partially observable search problems of Section 4.4, it also has a **sensor model** $P(e \mid s)$. Here, as in Chapter 14, the sensor model specifies the probability of perceiving evidence e in state s.[5] For example, we can convert the 4×3 world of Figure 16.1 into a POMDP by adding a noisy or partial sensor instead of assuming that the agent knows its location exactly. The noisy four–bit sensor from page 494 could be used, which reports the presence or absence of a wall in each compass direction with accuracy $1 - \epsilon$.

As with MDPs, we can obtain compact representations for large POMDPs by using dynamic decision networks (see Section 16.1.4). We add sensor variables $\mathbf{E}_t$, assuming that the state variables $\mathbf{X}_t$ may not be directly observable. The POMDP sensor model is then given by $\mathbf{P}(\mathbf{E}_t \mid \mathbf{X}_t)$. For example, we might add sensor variables to the DDN in Figure 16.4 such as $BatteryMeter_t$ to estimate the actual charge $Battery_t$ and $Speedometer_t$ to estimate the magnitude of the velocity vector $\dot{\mathbf{X}}_t$. A sonar sensor $Walls_t$ might give estimated distances to the nearest wall in each of the four cardinal directions relative to the robot's current orientation; these values depends on the current position and orientation $\mathbf{X}_t$.

In Chapters 4 and 11, we studied nondeterministic and partially observable planning problems and identified the **belief state**–the set of actual states the agent might be in–as a key concept for describing and calculating solutions. In POMDPs, the belief state b becomes a *probability distribution* over all possible states, just as in Chapter 14. For example, the initial belief state for the 4×3 POMDP could be the uniform distribution over the nine nonterminal states along with 0s for the terminal states, that is, $\langle \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, 0, 0 \rangle$.

We use the notation $b(s)$ to refer to the probability assigned to the actual state s by belief state b. The agent can calculate its current belief state as the conditional probability distribution over the actual states given the sequence of percepts and actions so far. This is essentially the **filtering** task described in Chapter 14. The basic recursive filtering equation (14.5 on page 485) shows how to calculate the new belief state from the previous belief state and the new evidence. For POMDPs, we also have an action to consider, but the result is essentially the same. If b was the previous belief state, and the agent does action a and then perceives evidence e, then the new belief state is obtained by calculating the probability of now being in state s', for each s', with the following formula:

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

where α is a normalizing constant that makes the belief state sum to 1. By analogy with the update operator for filtering (page 485), we can write this as

$$b' = \alpha \, \text{FORWARD}(b, a, e) \; . \quad (16.16)$$

In the 4 × 3 POMDP, suppose the agent moves *Left* and its sensor reports one adjacent wall; then it's quite likely (although not guaranteed, because both the motion and the sensor are noisy) that the agent is now in (3,1). Exercise 16.POMD asks you to calculate the exact probability values for the new belief state.

The fundamental insight required to understand POMDPs is this: *the optimal action depends only on the agent's current belief state.* That is, an optimal policy can be described by a mapping $\pi^*(b)$ from belief states to actions. It does *not* depend on the *actual* state the agent is in. This is a good thing, because the agent does not know its actual state; all it knows is the belief state. Hence, the decision cycle of a POMDP agent can be broken down into the following three steps:

1. Given the current belief state b, execute the action $a = \pi^*(b)$.
2. Observe the percept e.
3. Set the current belief state to FORWARD(b, a, e) and repeat.

We can think of POMDPs as requiring a search in belief-state space, just like the methods for sensorless and contingency problems in Chapter 4. The main difference is that the POMDP belief-state space is *continuous*, because a POMDP belief state is a probability distribution. For example, a belief state for the 4 × 3 world is a point in an 11-dimensional continuous space. An action changes the belief state, not just the physical state, because it affects the percept that is received. Hence, the action is evaluated at least in part according to the information the agent acquires as a result. POMDPs therefore include the value of information (Section 15.6) as one component of the decision problem.

Let's look more carefully at the outcome of actions. In particular, let's calculate the probability that an agent in belief state b reaches belief state b' after executing action a. Now, if we knew the action *and the subsequent percept,* then Equation (16.16) would provide a *deterministic* update to the belief state: $b' =$ FORWARD(b, a, e). Of course, the subsequent percept is not yet known, so the agent might arrive in one of several possible belief states b', depending on the percept that is received. The probability of perceiving *e*, given that *a* was performed starting in belief state *b*, is given by summing over all the actual states $s^{'}$ that the agent might reach:

$$P\left(e \middle| a, b\right) = \sum_{s\prime} P(e|a,\ s\prime, b)P(s\prime|a, b)$$

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

$$= \sum_{s\prime} P\left(e \middle| s\prime\right) \sum_{s} P(s\prime|s, a)b(s).$$

Let us write the probability of reaching *b′* from *b*, given action a, as *P(b′* | *b,* a). This probability can be calculated as follows:

$$P\left(b\prime \middle| b, a\right) = \sum_{e} P(b\prime|e, a, b)P(e|a, b)$$

$$= \sum_{e} P\left(b\prime \middle| e, a, b\right) \sum_{s\prime} P\left(e \middle| s\prime\right) \sum_{s} P(s\prime|s, a)b(s), \qquad (16.17)$$

where P(*b*'|*e, a, b)* is 1 if *b′* = FORWARD(*b,a, e)* and 0 otherwise.

Equation (16.17) can be viewed as defining a transition model for the belief-state space. We can also define a reward function for belief-state transitions, which is derived from the expected reward of the real state transitions that might be occurring. Here, we use the simple form *ρ(b, a),* the expected reward if the agent does *a* in belief state *b:*

$$\rho(b, a) = \sum_{e} b(s) \sum_{s\prime} P(s\prime|s, a)R(s, a, s\prime).$$

Together, *P*(*b′* | *b*, *a*) and *ρ*(*b*, *a*) define an *observable* MDP on the space of belief states. Furthermore, it can be shown that an optimal policy for this MDP, *π***(*b*), is also an optimal policy for the original POMDP. In other words, *solving a POMDP on a physical state space can be reduced to solving an MDP on the* *corresponding belief–state space*. This fact is perhaps less surprising if we remember that the belief state is always observable to the agent, by definition.

16.5 Algorithms for Solving POMDPs

We have shown how to reduce POMDPs to MDPs, but the MDPs we obtain have a continuous (and usually high–dimensional) state space. This means we will have to redesign the dynamic programming algorithms from Sections 16.2.1 and 16.2.2, which assumed a finite state space and a finite number of actions. Here we describe a value iteration algorithm designed specifically for POMDPs, followed by an online decision-making algorithm similar to those developed for games in Chapter 6.

16.5.1 Value iteration for POMDPs

Section 16.2.1 described a value iteration algorithm that computed one utility value for each state. With infinitely many belief states, we need to be more creative. Consider an optimal policy π^* and its application in a specific belief state *b:* the policy generates an action, then, for each subsequent percept, the belief state is updated and a new action is generated, and so on. For this specific *b*, therefore, the policy is exactly equivalent to a **conditional plan**, as defined in Chapter 4 for nondeterministic and partially observable problems. Instead of thinking about policies, let us think about conditional plans and how the expected utility of executing a fixed conditional plan varies with the initial belief state. We make two observations:

1. Let the utility of executing a *fixed* conditional plan *p* starting in physical state *s* be $\alpha_p(s)$. Then the expected utility of executing *p* in belief state *b* is just $\Sigma_s\, b(s)\, \alpha_{\mathrm{p}}(s)$, or $b \cdot \alpha_p$ if we think of them both as vectors. Hence, the expected utility of a fixed conditional plan varies *linearly* with *b*; that is, it corresponds to a hyperplane in belief space.
2. At any given belief state *b*, an optimal policy will choose to execute the conditional plan with highest expected utility; and the expected utility of *b* under an optimal policy is just the utility of that conditional plan: $U(b) = U^{\pi^*}(b) = \max_p b \cdot \alpha_p$. If an optimal policy π^* chooses to execute *p* starting at *b*, then it is reasonable to expect that it might choose to execute *p* in belief states that are very close to *b*; in fact, if we bound the depth of the conditional plans, then there are only finitely many such plans and the continuous space of belief states will generally be divided into *regions*, each corresponding to a particular conditional plan that is optimal in that region.

From these two observations, we see that the utility function $U(b)$ on belief states, being the maximum of a collection of hyperplanes, will be *piecewise linear* and *convex*.

To illustrate this, we use a simple two-state world. The states are labeled *A* and *B* and there are two actions: *Stay* stays put with probability 0.9 and *Go* switches to the other state with probability 0.9. The rewards are $R(.,.,A) = 0$ and $R(.,,B) = 1$; that is, any transition ending in *A* has reward zero and any transition ending in *B* has reward 1. For now we will assume the discount factor $\gamma = 1$. The sensor reports the correct state with probability 0.6. Obviously, the agent should *Stay* when it's in state *B* and *Go* when it's in state *A*. The problem is that it doesn't know where it is!

The advantage of a two-state world is that the belief space can be visualized in one dimension, because the two probabilities $b(A)$ and $b(B)$ sum to 1. In Figure 16.15(a), the *x*-axis represents the belief state, defined by $b(B)$, the probability of being in state *B*. Now let us consider the one-step plans [*Stay*] and [*Go*], each of which receives the reward for one transition as follows:

$$\alpha[Stay](A) = 0.9R(A, Stay, A) + 0.1R(A, Stay, B) = 0.1$$
$$\alpha[Stay](B) = 0.1R(B, Stay, A) + 0.9R(B, Stay, B) = 0.9$$
$$\alpha[Go](A) = 0.1R(A, Go, A) + 0.9R(A, Go, B) = 0.9$$
$$\alpha[Go](B) = 0.9R(B, Go, A) + 0.1R(B, Go, B) = 0.1$$

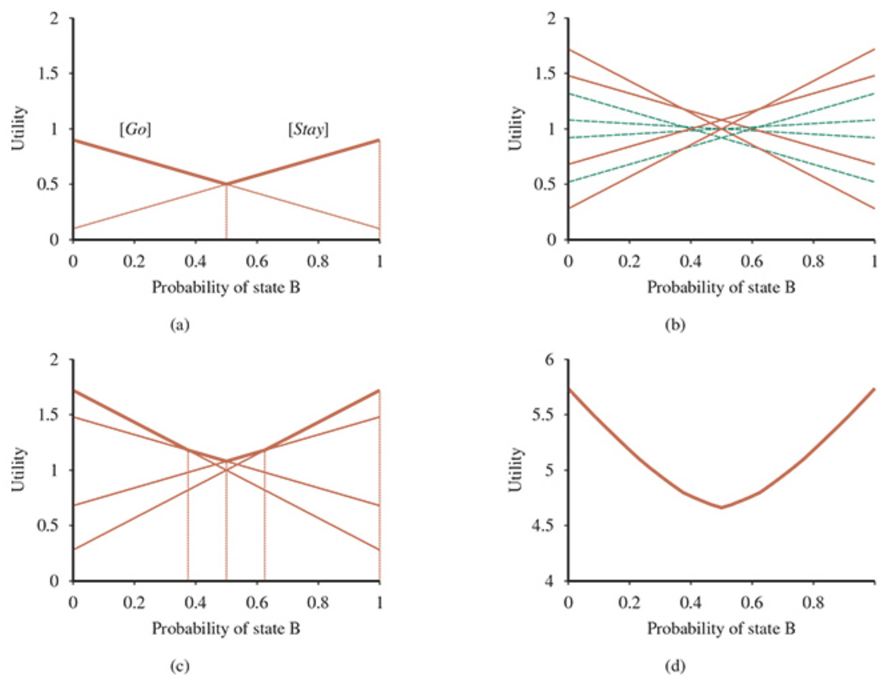


Figure 16.15 (a) Utility of two one-step plans as a function of the initial belief state *b(B)* for the two-state world, with the corresponding utility function shown in bold. (b) Utilities for 8 distinct two-step plans. (c) Utilities for four undominated two-step plans. (d) Utility function for optimal eight-step plans.

The hyperplanes (lines, in this case)for $b \cdot \alpha_{[Stay]}$ and $b \cdot \alpha_{[Go]}$ are shown in Figure 16.15(a)and their maximum is shown in bold. The bold line therefore represents the utility function for the finite-horizon problem that allows just one action, and in each "piece" of the piecewise linear utility function an optimal action is the first action of the corresponding conditional plan. In this case, the optimal one-step policy is to *Stay* when $b(B) > 0.5$ and *Go* otherwise.

Once we have utilities $\alpha_p(s)$ for all the conditional plans p of depth 1 in each physical state s, we can compute the utilities for conditional plans of depth 2 by considering each possible first action, each possible subsequent percept, and then each way of choosing a depth-1 plan to execute for each percept:

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

There are eight distinct depth-2 plans in all, and their utilities are shown in Figure 16.15(b). Notice that four of the plans, shown as dashed lines, are suboptimal across the entire belief space–we say these plans are **dominated**, and they need not be considered further. There are four undominated plans, each of which is optimal in a specific region, as shown in Figure 16.15(c). The regions partition the belief-state space.

We repeat the process for depth 3, and so on. In general, let p be a depth–d conditional plan whose initial action is a and whose depth-(d – 1) subplan for percept e is $p.e$; then

$$\alpha_p(s) = \sum_{s\prime} P(s\prime \Big| s,a)[R(s,a,s\prime) + \gamma \sum_{e} P(e|s\prime)\alpha_{p.e}(s\prime)]. \tag{16.18}$$

This recursion naturally gives us a value iteration algorithm, which is given in Figure 16.16. The structure of the algorithm and its error analysis are similar to those of the basic value iteration algorithm in Figure 16.6 on page 563; the main difference is that instead of computing one utility number for each state, POMDP–VALUE–ITERATION maintains a collection of undominated plans with their utility hyperplanes.

---

function POMDP-VALUE-ITERATION($pomdp, \epsilon$) **returns** a utility function
 inputs: *pomdp*, a POMDP with states S, actions $A(s)$, transition model $P(s' \mid s,a)$,
  sensor model $P(e \mid s)$, rewards $R(s,a,s')$, discount γ
  ϵ, the maximum error allowed in the utility of any state
 local variables: U, U', sets of plans p with associated utility vectors α_p

 $U' \leftarrow$ a set containing all one-step plans $[a]$, with $\alpha_{[a]}(s) = \sum_{s'} P(s' \mid s,a)\, R(s,a,s')$
 repeat
  $U \leftarrow U'$
  $U' \leftarrow$ the set of all plans consisting of an action and, for each possible next percept,
   a plan in U with utility vectors computed according to Equation (16.18)
  $U' \leftarrow$ REMOVE-DOMINATED-PLANS(U')
 until MAX-DIFFERENCE(U, U') $\leq \epsilon(1-\gamma)/\gamma$
 return U

Figure 16.16 A high-level sketch of the value iteration algorithm for POMDPs. The REMOVE-DOMINATED-P LANS step and MAX–DIFFERENCE test are typically implemented as linear programs.

The algorithm's complexity depends primarily on how many plans get generated. Given |A| actions and |E| possible observations, there are $|A|^{O(|E|^{d-1})}$ distinct depth-d plans. Even for the lowly two-state world with d = 8, that's 2^{255} plans. The elimination of dominated plans is essential for reducing this doubly exponential growth: the number of undominated plans with d = 8 is just 144. The utility function for these 144 plans is shown in Figure 16.15(d).

Notice that the intermediate belief states have lower value than state A and state B, because in the intermediate states the agent lacks the information needed to choose a good action. This is why information has value in the sense defined in Section 15.6 and optimal policies in POMDPS often include information-gathering actions.

Given such a utility function, an executable policy can be extracted by looking at which hyperplane is optimal at any given belief state b and executing the first action of the corresponding plan. In Figure 16.15(d), the corresponding optimal policy is still the same as for depth-1 plans: *Stay* when $b(B) > 0.5$ and Go otherwise.

In practice, the value iteration algorithm in Figure 16.16 is hopelessly inefficient for larger problems–even the 4×3 POMDP is too hard. The main reason is that given n undominated conditional plans at level d, the algorithm constructs $|A| \cdot n^{|\mathbf{E}|}$ conditional plans at level $d+1$ before eliminating the dominated ones. With the four-bit sensor, $|E|$ is 16, and n can be in the hundreds, so this is hopeless.

Since this algorithm was developed in the 1970s, there have been several advances, including more efficient forms of value iteration and various kinds of policy iteration algorithms. Some of these are discussed in the notes at the end of the chapter. For general POMDPs, however, finding optimal policies is very difficult (PSPACE–hard, in fact–that is, very hard indeed). The next section describes a different, approximate method for solving POMDPs, one based on look-ahead search.

16.5.2 Online algorithms for POMDPs

The basic design for an online POMDP agent is straightforward: it starts with some prior belief state; it chooses an action based on some deliberation process centered on its current belief state; after acting, it receives an observation and updates its belief state using a filtering algorithm; and the process repeats.

One obvious choice for the deliberation process is the expectimax algorithm from Section 16.2.4, except with belief states rather than physical states as the decision nodes in the tree. The chance nodes in the POMDP tree have branches labeled by possible observations and leading to the next belief state, with transition probabilities given by Equation (16.17). A fragment of the belief-state expectimax tree for the 4×3 POMDP is shown in Figure 16.17.

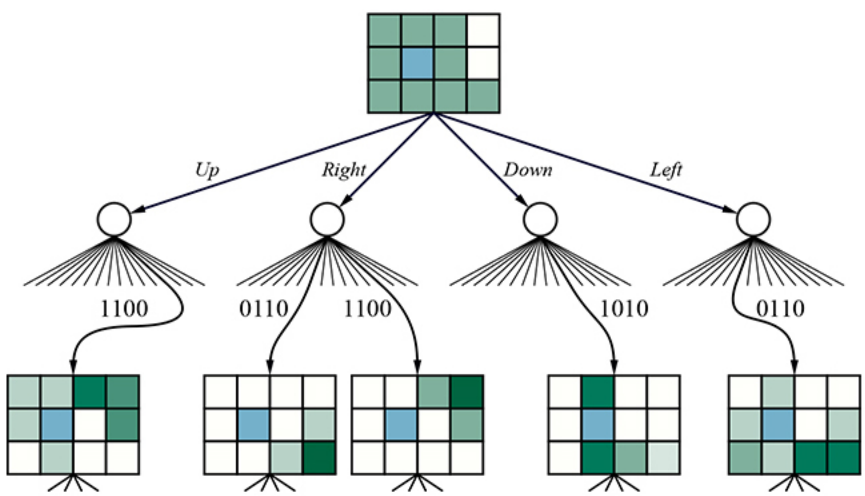


Figure 16.17 Part of an expectimax tree for the 4×3 POMDP with a uniform initial belief state. The belief states are depicted with shading proportional to the probability of being in each location.

The time complexity of an exhaustive search to depth d is $O(|A|^d \cdot |\mathbf{E}|^d)$, where $|A|$ is the number of available actions and $|\mathbf{E}|$ is the number of possible percepts. (Notice that this is far less than the number of possible depth-d conditional plans generated by value iteration.) As in the observable case, sampling at the chance nodes is a good way to cut down the branching factor without losing too much accuracy in the final decision. Thus, the complexity of approximate online decision making in POMDPs may not be drastically worse than that in MDPs.

For very large state spaces, exact filtering is infeasible, so the agent will need to run an approximate filtering algorithm such as particle filtering (see page 510). Then the belief states in the expectimax tree become collections of particles rather than exact probability distributions. For problems with long horizons, we may also need to run the kind of long–range playouts used in the UCT algorithm (Figure 6.11). The combination of particle filtering and UCT applied to POMDPs goes under the name of partially observable Monte Carlo planning or **POMCP**. With a DDN representation for the model, the POMCP algorithm is, at least in principle, applicable to very large and realistic POMDPs. Details of the algorithm are explored in Exercise 16.POMC. POMCP is capable of generating competent behavior in the 4×3 POMDP. A short (and somewhat fortunate) example is shown in Figure 16.18.

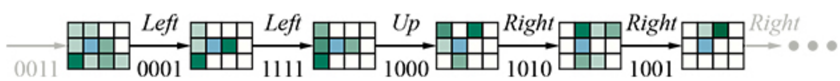


Figure 16.18 A sequence of percepts, belief states, and actions in the 4×3 POMDP with a wall-sensing error of $\epsilon = 0.2$. Notice how the early *Left* moves are safe—they are very unlikely to fall into (4,2)—and coerce the agent's location into a small number of possible locations. After moving *Up*, the agent thinks it is probably in (3,3), but possibly in (1,3). Fortunately, moving *Right* is a good idea in both cases, so it moves *Right*, finds out that it had been in (1,3) and is now in (2,3), and then continues moving *Right* and reaches the goal.

POMDP agents based on dynamic decision networks and online decision making have a number of advantages compared with other, simpler agent designs presented in earlier chapters. In particular, they handle partially observable, stochastic environments and can easily revise their "plans" to handle unexpected evidence. With appropriate sensor models, they can handle sensor failure and can plan to gather information. They exhibit "graceful degradation" under time pressure and in complex environments, using various approximation techniques.

So what is missing? The principal obstacle to real-world deployment of such agents is the inability to generate successful behavior over long time-scales. Random or near-random playouts have no hope of gaining any positive reward on, say, the task of laying the table for dinner, which might take tens of millions of motor–control actions. It seems necessary to borrow some of the hierarchical planning ideas described in Section 11.4. At the time of writing, there are not yet satisfactory and efficient ways to apply these ideas in stochastic, partially observable environments.

Summary

This chapter shows how to use knowledge about the world to make decisions even when the outcomes of an action are uncertain and the rewards for acting might not be reaped until many actions have passed. The main points are as follows:

- Sequential decision problems in stochastic environments, also called **Markov decision processes**, or MDPs, are defined by a **transition model** specifying the probabilistic outcomes of actions and a **reward function** specifying the reward in each state.
- The utility of a state sequence is the sum of all the rewards over the sequence, possibly discounted over time. The solution of an MDP is a **policy** that associates a decision with every state that the agent might reach. An optimal policy maximizes the utility of the state sequences encountered when it is executed.
- The utility of a state is the expected sum of rewards when an optimal policy is executed from that state. The value iteration algorithm iteratively solves a set of equations relating the utility of each state to those of its neighbors.
- **Policy iteration** alternates between calculating the utilities of states under the current policy and improving the current policy with respect to the current utilities.
- Partially observable MDPs, or POMDPs, are much more difficult to solve than are MDPs. They can be solved by conversion to an MDP in the continuous space of belief states; both value iteration and policy iteration algorithms have been devised. Optimal behavior in POMDPs

includes information gathering to reduce uncertainty and therefore make better decisions in the future.

- A decision-theoretic agent can be constructed for POMDP environments. The agent uses a **dynamic decision network** to represent the transition and sensor models, to update its belief state, and to project forward possible action sequences.

We shall return MDPs and POMDPs in Chapter 23, which covers **reinforcement learning** methods that allow an agent to improve its behavior from experience.

Bibliographical and Historical Notes

Richard Bellman developed the ideas underlying the modern approach to sequential decision problems while working at the RAND Corporation beginning in 1949. According to his autobiography (Bellman, 1984), he coined the term "dynamic programming" to hide from a research-phobic Secretary of Defense, Charles Wilson, the fact that his group was doing mathematics. (This cannot be strictly true, because his first paper using the term (Bellman, 1952) appeared before Wilson became Secretary of Defense in 1953.) Bellman's book, *Dynamic Programming* (1957), gave the new field a solid foundation and introduced the value iteration algorithm.

Shapley (1953b) actually described the value iteration algorithm independently of Bellman, but his results were not widely appreciated in the operations research community, perhaps because they were presented in the more general context of Markov games. Although the original formulations included discounting, its analysis in terms of stationary preferences was suggested by Koopmans (1972). The shaping theorem is due to Ng *et al.* (1999).

Ron Howard's Ph.D. thesis (1960) introduced policy iteration and the idea of average reward for solving infinite-horizon problems. Several additional results were introduced by Bellman and Dreyfus (1962). The use of contraction mappings in analyzing dynamic programming algorithms is due to Denardo (1967). Modified policy iteration is due to van Nunen (1976) and Puterman and Shin (1978). Asynchronous policy iteration was analyzed by Williams and Baird (1993), who also proved the policy loss bound in Equation (16.13). The general family of prioritized sweeping algorithms aims to speed up convergence to optimal policies by heuristically ordering the value and policy update calculations (Moore and Atkeson, 1993; Andre *et al.*, 1998; Wingate and Seppi, 2005).

The formulation of MDP-solving as a linear program is due to de Ghellinck (1960), Manne (1960), and D'Épenoux (1963). Although linear programming has traditionally been considered inferior to dynamic programming as an exact solution method for MDPs, de Farias and Roy (2003) show that it is possible to use linear programming and a linear representation of the utility function to obtain provably good approximate solutions to very large MDPs. Papadimitriou and Tsitsiklis (1987) and Littman *et al.* (1995) provide general results on the computational complexity of MDPs. Yinyu Ye (2011) analyzes the relationship between policy iteration and the simplex method for linear programming and proves that for fixed γ, the runtime of policy iteration is polynomial in the number of states and actions.

Seminal work by Sutton (1988) and Watkins (1989) on reinforcement learning methods for solving MDPs played a significant role in introducing MDPs into the AI community. (Earlier work by Werbos (1977) contained many similar ideas, but was not taken up to the same extent.) AI researchers have pushed MDPs in the direction of more expressive representations that can accommodate much larger problems than the traditional atomic representations based on transition matrices.

The basic ideas for an agent architecture using dynamic decision networks were proposed by Dean and Kanazawa (1989a). Tatman and Shachter (1990) showed how to apply dynamic programming algorithms to DDN models. Several authors made the connection between MDPs and AI planning problems, developing probabilistic forms of the compact STRIPS representation for transition models (Wellman, 1990b; Koenig, 1991). The book *Planning and Control* by Dean and Wellman (1991) explores the connection in great depth.

Later work on **factored MDPs** (Boutilier *et al.,* 2000; Koller and Parr, 2000; Guestrin *et al*., 2003b) uses structured representations of the value function as well as the transition model, with provable improvements in complexity. Relational MDPs (Boutilier *et al.,* 2001; Guestrin *et al*., 2003a) go one step further, using structured representations to handle domains with many related objects. Open-universe MDPs and POMDPs (Srivastava *et al.*, 2014b) also allow for uncertainty over the existence and identity of objects and actions.

Many authors have developed approximate online algorithms for decision making in MDPs, often borrowing explicitly from earlier AI approaches to real-time search and gameplaying (Werbos, 1992; Dean *et al.,* 1993; Tash and Russell, 1994). The work of Barto *et al.*(1995) on RTDP (real-time dynamic programming) provided a general framework for understanding such algorithms and their connection to reinforcement learning and heuristic search. The analysis of depth-bounded expectimax with sampling at chance nodes is due to Kearns *et al.* (2002). The UCT algorithm described in the chapter is due to Kocsis and Szepes- vari (2006) and borrows from earlier work on random playouts for estimating the values of states (Abramson, 1990; Briigmann, 1993; Chang *et al.,* 2005).

Bandit problems were introduced by Thompson (1933) but came to prominence after World War II through the work of Herbert Robbins (1952). Bradt *et al.* (1956) proved the first results concerning stopping rules for one-armed bandits, which led eventually to the breakthrough results of John Gittins (Gittins and Jones, 1974; Gittins, 1989). Katehakis and Veinott (1987) suggested the restart MDP as a method of computing Gittins indices. The text by Berry and Fristedt (1985) covers many variations on the basic problem, while the pellucid online text by Ferguson (2001) connects bandit problems with stopping problems.

Lai and Robbins (1985) initiated the study of the asymptotic regret of optimal bandit policies. The UCB heuristic was introduced and analyzed by Auer *et al.* (2002). Bandit superprocesses (BSPs) were first studied by Nash (1973) but have remained largely unknown in AI. Hadfield-Menell and Russell (2015) describe an efficient branch-and-bound algorithm capable of solving relatively large BSPs. Selection problems were introduced by Bechhofer (1954). Hay *et al.* (2012) developed a formal framework for metareasoning problems, showing that simple instances mapped to selection rather than bandit problems. They also proved the satisfying result that expected computation cost of the optimal computational strategy is never higher than the expected gain in decision quality–although there are cases where the optimal policy may, with some probability, keep computing long past the point where any possible gain has been used up.

The observation that a partially observable MDP can be transformed into a regular MDP over belief states is due to Astrom (1965) and Aoki (1965). The first complete algorithm for the exact solution of POMDPs–essentially the value iteration algorithm presented in this chapter–was proposed by Edward Sondik (1971) in his Ph.D. thesis. (A later journal paper by Smallwood and Sondik (1973) contains some errors, but is more accessible.) Lovejoy (1991) surveyed the first twenty-five years of POMDP research, reaching somewhat pessimistic conclusions about the feasibility of solving large problems.

The first significant contribution within AI was the Witness algorithm (Cassandra *et al.*, 1994; Kaelbling *et al.*, 1998), an improved version of POMDP value iteration. Other algorithms soon followed, including an approach due to Hansen (1998) that constructs a policy incrementally in the form of a finite-state automaton whose states define the possible belief states of the agent.

More recent work in AI has focused on **point-based** value iteration methods that, at each iteration, generate conditional plans and α -vectors for a finite set of belief states rather than for the entire belief space. Lovejoy (1991) proposed such an algorithm for a fixed grid of points, an approach taken also by Bonet (2002). An influential paper by Pineau *et al.* (2003) suggested generating reachable points by simulating trajectories in a somewhat greedy fashion; Spaan and Vlassis (2005) observe that one need generate plans for only a small, randomly selected subset of points to improve on the plans from the previous iteration for all points in the set. Shani *et al.* (2013) survey these and other developments in point-based algorithms, which have led to good solutions for problems with thousands of states. Because POMDPs are PSPACE-hard (Papadimitriou and Tsitsiklis, 1987), further progress on offline solution methods may require taking advantage of various kinds of structure in value functions arising from a factored representation of the model.

The online approach for POMDPs—using look-ahead search to select an action for the current belief state—was first examined by Satia and Lave (1973). The use of sampling at chance nodes was explored analytically by Kearns *et al.* (2000) and Ng and Jordan (2000). The POMCP algorithm is due to Silver and Veness (2011).

With the development of reasonably effective approximation algorithms for POMDPs, their use as models for real-world problems has increased, particularly in education (Rafferty *et al.*, 2016), dialog systems (Young *et al*, 2013), robotics (Hsiao *et al.*, 2007; Huynh and Roy, 2009), and self-driving cars (Forbes *et al.*, 1995; Bai *et al.*, 2015). An important large- scale application is the Airborne Collision Avoidance System X

(ACAS X), which keeps airplanes and drones from colliding midair. The system uses POMDPs with neural networks to do function approximation. ACAS X significantly improves safety compared to the legacy TCAS system, which was built in the 1970s using expert system technology (Kochenderfer, 2015; Julian *et al.*, 2018).

Complex decision making has also been studied by economists and psychologists. They find that decision makers are not always rational, and may not be operating exactly as described by the models in this chapter. For example, when given a choice, a majority of people prefer $100 today over a guarantee of $200 in two years, but those same people prefer $200 in eight years over $100 in six years. One way to interpret this result is that people are not using additive exponentially discounted rewards; perhaps they are using **hyperbolic rewards** (the hyperbolic function dips more steeply in the near term than does the exponential decay function). This and other possible interpretations are discussed by Rubinstein (2003).

The texts by Bertsekas (1987) and Puterman (1994) provide rigorous introductions to sequential decision problems and dynamic programming. Bertsekas and Tsitsiklis (1996) include coverage of reinforcement learning. Sutton and Barto (2018) cover similar ground but in a more accessible style. Sigaud and Buffet (2010), Mausam and Kolobov (2012) and Kochenderfer (2015) cover sequential decision making from an AI perspective. Krishnamurthy (2016) provides thorough coverage of POMDPs.

[1] It is also possible to use costs $c(s, a, s')$, as we did in the definition of search problems in Chapter 3. The use of rewards is, however, standard in the literature on sequential decisions under uncertainty.

[2] In this chapter we use U for the utility function (to be consistent with the rest of the book), but many works about MDPs use V (for *value)* instead.

[3] Although this seems obvious, it does not hold for finite-horizon policies or for other ways of combining rewards over time, such as taking the max. The proof follows directly from the uniqueness of the utility function on states, as shown in Section 16.2.1.

[4] The probabilities are those of a Bayesian updating process with a Beta(1,1) prior (see Section 21.2.5).

[5] The sensor model can also depend on the action and outcome state, but this change is not fundamental.

CHAPTER 17

MULTIAGENT DECISION MAKING

In which we examine what to do when more than one agent inhabits the environment.

17.1 Properties of Multiagent Environments

So far, we have largely assumed that only one agent has been doing the sensing, planning, and acting. But this represents a huge simplifying assumption, which fails to capture many real-world AI settings. In this chapter, therefore, we will consider the issues that arise when an agent must make decisions in environments that contain multiple actors. Such environments are called **multiagent systems**, and agents in such a system face a **multiagent planning problem**. However, as we will see, the precise nature of the multiagent planning problem—and the techniques that are appropriate for solving it—will depend on the relationships among the agents in the environment.

17.1.1 One decision maker

The first possibility is that while the environment contains multiple *actors*, it contains only one *decision maker*. In such a case, the decision maker develops plans for the other agents, and tells them what to do. The assumption that agents will simply do what they are told is called the **benevolent agent assumption**. However, even in this setting, plans involving multiple actors will require actors to *synchronize* their actions. Actors A and B will have to act at the same time for joint actions (such as singing a duet), at different times for mutually exclusive actions (such as recharging batteries when there is only one plug), and sequentially when one establishes a precondition for the other (such as A washing the dishes and then B drying them).

One special case is where we have a single decision maker with multiple effectors that can operate concurrently—for example, a human who can walk and talk at the same time. Such an agent needs to do **multieffector planning** to manage each effector while handling positive and negative interactions among the effectors. When the effectors are physically decoupled into detached units—as in a fleet of delivery robots in a factory—multieffector planning becomes **multibody planning**.

A multibody problem is still a “standard” single-agent problem as long as the relevant sensor information collected by each body can be pooled—either centrally or within each body—to form a common estimate of the world state that then informs the execution of the overall plan; in this case, the multiple bodies can be thought of as acting as a single body. When communication constraints make this impossible, we have what is sometimes called a **decentralized planning** problem; this is perhaps a misnomer, because the planning phase is centralized but the execution phase is at least partially decoupled. In this case, the subplan constructed for each body may need to include explicit communicative actions with other bodies. For example, multiple reconnaissance robots covering a wide area may often be out of radio contact with each other and should share their findings during times when communication is feasible.

17.1.2 Multiple decision makers

The second possibility is that the other actors in the environment are also decision makers: they each have preferences and choose and execute their own plan. We call them **counterparts**. In this case, we can distinguish two further possibilities.

- The first is that, although there are multiple decision makers, they are all pursuing a **common goal**. This is roughly the situation of workers in a company, in which different decision makers are pursuing, one hopes, the same goals on behalf of the company. The main problem faced by the decision makers in this setting is the **coordination problem**: they need to ensure that they are all pulling in the same direction, and not accidentally fouling up each other's plans.
- The second possibility is that the decision makers each have their own personal preferences, which they each will pursue to the best of their abilities. It could be that the preferences are diametrically opposed, as is the case in zero-sum games such as chess (see Chapter 6). But most multiagent encounters are more complicated than that, with more complex preferences.

When there are multiple decision makers, each pursuing their own preferences, an agent must take into account the preferences of other agents, as well as the fact that these other agents are *also* taking into account the preferences of other agents, and so on. This brings us into the realm of **game theory**: the theory of strategic decision making. It is this *strategic* aspect of reasoning—players each taking into account how other players may act—that distinguishes game theory from decision theory. In the same way that decision theory provides the theoretical foundation for decision making in single-agent AI, game theory provides the theoretical foundation for decision making in multiagent systems.

The use of the word "game" here is also not ideal: a natural inference is that game theory is mainly concerned with recreational pursuits, or artificial scenarios. Nothing could be further from the truth. Game theory is the theory of **strategic decision making**. It is used in decision making situations including the auctioning of oil drilling rights and wireless frequency spectrum rights, bankruptcy proceedings, product development and pricing decisions, and national defense—situations involving billions of dollars and many lives. Game theory in AI can be used in two main ways:

1. **Agent design**: Game theory can be used by an agent to analyze its possible decisions and compute the expected utility for each of these (under the assumption that other agents are acting rationally, according to game theory). In this way, game-theoretic techniques can determine the best strategy against a rational player and the expected return for each player.
2. **Mechanism design**: When an environment is inhabited by many agents, it might be possible to define the rules of the environment (i.e., the game that the agents must play) so that the collective good of all agents is maximized when each agent adopts the game-theoretic solution that maximizes its own utility. For example, game theory can help design the protocols for a collection of Internet traffic routers so that each router has an incentive to act in such a way that global throughput is maximized. Mechanism design can also be used to construct intelligent multiagent systems that solve complex problems in a distributed fashion.

Game theory provides a range of different models, each with its own set of underlying assumptions; it is important to choose the right model for each setting. The most important distinction is whether we should consider it a cooperative game or not:

- In a **cooperative game**, it is possible to have a binding agreement between agents, thereby enabling robust cooperation. In the human world, legal contracts and social norms help establish such binding agreements. In the world of computer programs, it may be possible to inspect source code to make sure it will follow an agreement. We use cooperative game theory to analyze this situation.
- If binding agreements are not possible, we have a **non-cooperative game**. Although this term suggests that the game is inherently competitive, and that cooperation is not possible, that need not be the case: non-cooperative simply means that there is no central agreement that binds all agents and guarantees cooperation. But it could well be that agents independently decide to cooperate, because it is in their own best interests. We use non-cooperative game theory to analyze this situation.

Some environments will combine multiple different dimensions. For example, a package delivery company may do centralized, offline planning for the routes of its trucks and planes each day, but leave some aspects open for autonomous decisions by drivers and pilots who can respond individually to traffic and weather situations. Also, the goals of the company and its employees are brought into alignment, to some extent, by the payment of **incentives** (salaries and bonuses)—a sure sign that this is a true multiagent system.

17.1.3 Multiagent planning

For the time being, we will treat the multieffector, multibody, and multiagent settings in the same way, labeling them generically as **multiactor** settings, using the generic term **actor** to cover effectors, bodies, and agents. The goal of this section is to work out how to define transition models, correct plans, and efficient planning algorithms for the multiactor setting. A correct plan is one that, if executed by the actors, achieves the goal. (In the true multiagent setting, of course, the agents may not agree to execute any particular plan, but at least they will know what plans *would* work if they *did* agree to execute them.)

A key difficulty in attempting to come up with a satisfactory model of multiagent action is that we must somehow deal with the thorny issue of **concurrency**, by which we simply mean that the plans of each agent may be executed simultaneously. If we are to reason about the execution of multiactor plans, then we will first need a model of multiactor plans that embodies a satisfactory model of concurrent action.

In addition, multiactor action raises a whole set of issues that are not a concern in singleagent planning. In particular, *agents must take into account the way in which their own actions interact with the actions of other agents*. For example, an agent will need to consider whether the actions performed by other agents might clobber the preconditions of its own actions, whether the resources it makes use of while executing its policy are sharable, or may be depleted by other agents; whether actions are mutually exclusive; and a helpfully inclined agent could consider how its actions might facilitate the actions of others.

To answer these questions we need a model of concurrent action within which we can properly formulate them. Models of concurrent action have been a major focus of research in the mainstream computer science community for decades, but no definitive, universally accepted model has prevailed. Nevertheless, the following three approaches have become widely established.

The first approach is to consider the **interleaved execution** of the actions in respective plans. For example, suppose we have two agents, *A* and *B*, with plans as follows:

$$A \ : \ [a_1, a_2]$$
$$B \ : \ [b_1, b_2].$$

The key idea of the interleaved execution model is that the only thing we can be certain about in the execution of the two agents' plans is that the order of actions in the respective plans will be preserved. If we further assume that actions are atomic, then there are six different ways in which the two plans above might be executed concurrently:

$$[a_1, a_2, b_1, b_2]$$
$$[b_1, b_2, a_1, a_2]$$
$$[a_1, b_1, a_2, b_2]$$
$$[b_1, a_1, b_2, a_2]$$
$$[a_1, b_1, b_2, a_2]$$
$$[b_1, a_1, a_2, b_2]$$

For a plan to be correct in the interleaved execution model, *it must be correct with respect to all possible interleavings of the plans*. The interleaved execution model has been widely adopted within the concurrency community, because it is a reasonable model of the way multiple threads take turns running on a single CPU. However, it does not model the case where two actions actually happen at the same time. Furthermore, the number of interleaved sequences will grow exponentially with the number of agents and actions: as a consequence, checking the correctness of a plan, which is computationally straightforward in single-agent settings, is computationally difficult with the interleaved execution model.

The second approach is **true concurrency**, in which we do not attempt to create a full serialized ordering of the actions, but leave them *partially ordered:* we know that a_1 will occur before a_2, but with respect to the ordering of a_1 and b_1, for example, we can say nothing; one may occur before the other, or they could occur concurrently. We can always "flatten" a partial-order model of concurrent plans into an interleaved model, but in doing so, we lose the partial-order information. While partial-order models are arguably more satisfying than interleaved models as a theoretical account of concurrent action, they have not been as widely adopted in practice.

The third approach is to assume perfect **synchronization**: there is a global clock that each agent has access to, each action takes the same amount of time, and actions at each point in the joint plan are simultaneous. Thus, the actions of each agent are executed synchronously, in lockstep with each other (it may be that some agents execute a no-op action when they are waiting for other actions to complete). Synchronous execution is not a very complete model of concurrency in the real world, but it has a simple semantics, and for this reason, it is the model we will work with here.

We begin with the transition model; for the single-agent deterministic case, this is the function RESULT(s, a), which gives the state that results from performing the action a when the environment is in state s. In the single-agent setting, there might be b different choices for the action; b can be quite large, especially for first-order representations with many objects to act on, but action schemas provide a concise representation nonetheless.

In the multiactor setting with n actors, the single action a is replaced by a **joint action** $\langle a_1, ..., a_n \rangle$, where a_i is the action taken by the ith actor. Immediately, we see two problems: first, we have to describe the transition model for b^n different joint actions; second, we have a joint planning problem with a branching factor of b^n.

Having put the actors together into a multiactor system with a huge branching factor, the principal focus of research on multiactor planning has been to *decouple* the actors to the extent possible, so that (ideally) the complexity of the problem grows linearly with n rather than exponentially with b^n.

If the actors have no interaction with one another—for example, n actors each playing a game of solitaire—then we can simply solve n separate problems. If the actors are **loosely coupled**, can we attain something close to this exponential improvement? This is, of course, a central question in many areas of AI. We have seen successful solution methods for loosely coupled systems in the context of CSPs, where "tree like" constraint graphs yielded efficient solution methods (see page 186), as well as in the context of disjoint pattern databases (page 119) and additive heuristics for planning (page 374).

The standard approach to loosely coupled problems is to pretend the problems are completely decoupled and then fix up the interactions. For the transition model, this means writing action schemas as if the actors acted independently.

Let's see how this works for a game of doubles tennis. Here, we have two human tennis players who form a doubles team with the common goal of winning the match against an opponent team. Let's suppose that at one point in the game, the team has the goal of returning the ball that has been hit to them and ensuring that at least one of them is covering the net. Figure 17.1 shows the initial conditions, goal, and action schemas for this problem. It is easy to see that we can get from the initial conditions to the goal with a two-step **joint plan** that specifies what each player has to do: A should move over to the right baseline and hit the ball, while B should just stay put at the net:

$$
\begin{aligned}
&Actors(A, B)\\
&Init(At(A, LeftBaseline) \land At(B, RightNet) \land\\
&\qquad Approaching(Ball, RightBaseline) \land Partner(A, B) \land Partner(B, A)\\
&Goal(Returned(Ball) \land (At(x, RightNet) \lor At(x, LeftNet))\\
&Action(Hit(actor, Ball),\\
&\qquad \text{PRECOND:}Approaching(Ball, loc) \land At(actor, loc)\\
&\qquad \text{EFFECT:}Returned(Ball))\\
&Action(Go(actor, to),\\
&\qquad \text{PRECOND:}At(actor, loc) \land to \neq loc,\\
&\qquad \text{EFFECT:}At(actor, to) \land \neg At(actor, loc))
\end{aligned}
$$

Figure 17.1 The doubles tennis problem. Two actors, *A* and *B*, are playing together and can be in one of four locations: *LeftBaseline*, *RightBaseline*, *LeftNet*, and *RightNet*. The ball can be returned only if a player is in the right place. The *NoOp* action is a dummy, which has no effect. Note that each action must include the actor as an argument.

$$
\begin{aligned}
\text{PLAN 1:}\quad & A : [Go(A, RightBaseline), Hit(A, Ball)]\\
& B : [NoOp(B), NoOp(B)].
\end{aligned}
$$

Problems arise, however, when a plan dictates that both agents hit the ball at the same time. In the real world, this won't work, but the action schema for *Hit* says that the ball will be returned successfully. The difficulty is that preconditions constrain the state in which an action by itself can be executed successfully, but do not constrain other concurrent actions that might mess it up.

We solve this problem by augmenting action schemas with one new feature: a **concurrent action constraint** stating which actions must or must not be executed concurrently. For example, the *Hit* action could be described as follows:

$$
\begin{aligned}
&Action(Hit(actor,\ Ball),\\
&\qquad \text{CONCURRENT}: \forall b\ b \neq actor \Rightarrow \neg Hit(b, Ball)\\
&\qquad \text{PRECOND}: Approaching(Ball, loc) \land At(actor, loc)\\
&\qquad \text{EFFECT:}Returned(Ball))\ .
\end{aligned}
$$

In other words, the *Hit* action has its stated effect only if no other *Hit* action by another agent occurs at the same time. (In the SATPLAN approach, this would be handled by a partial **action exclusion axiom**.) For some actions, the desired effect is achieved *only* when another action occurs concurrently. For example, two agents are needed to carry a cooler full of beverages to the tennis court:

$$
\begin{aligned}
&Action(Carry(actor,\ cooler, here,\ there),\\
&\quad \text{CONCURRENT}: \exists b\ b\ \neq\ actor\ \wedge Carry(b, cooler, here,\ there)\\
&\quad \text{PRECOND}: At(actor,\ here) \wedge\ At\,(cooler, here) \wedge Cooler(cooler)\\
&\quad \text{EFFECT}: At(actor, there) \wedge At(cooler, there) \wedge \neg At(actor, here) \wedge \neg At(cooler, here)).
\end{aligned}
$$

With these kinds of action schemas, any of the planning algorithms described in Chapter 11 can be adapted with only minor modifications to generate multiactor plans. To the extent that the coupling among subplans is loose—meaning that concurrency constraints come into play only rarely during plan search—one would expect the various heuristics derived for singleagent planning to also be effective in the multiactor context.

17.1.4 Planning with multiple agents: Cooperation and coordination

Now let us consider a true multiagent setting in which each agent makes its own plan. To start with, let us assume that the goals and knowledge base are shared. one might think that this reduces to the multibody case—each agent simply computes the joint solution and executes its own part of that solution. Alas, the “the” in “the joint solution” is misleading. Here is a second plan that also achieves the goal:

$$
\begin{aligned}
\text{PLAN 2:}\qquad & A\ :\ [Go(A,\ LeftNet), NoOp(A)]\\
& B\ :\ [Go(B,\ RightBaseline),\ Hit(B,\ Ball)].
\end{aligned}
$$

If both agents can agree on either plan 1 or plan 2, the goal will be achieved. But if *A* chooses plan 2 and *B* chooses plan 1, then nobody will return the ball. Conversely, if *A* chooses 1 and *B* chooses 2, then they will both try to hit the ball and that too will fail. The agents know this, but how can they coordinate to make sure they agree on the plan?

One option is to adopt a **convention** before engaging in joint activity. A convention is any constraint on the selection of joint plans. For example, the convention “stick to your side of the court” would rule out plan 1, causing both partners to select plan 2. Drivers on a road face the problem of not colliding with each other; this is (partially) solved by adopting the convention “stay on the right-hand side of the road” in most countries; the alternative, “stay on the left-hand side,” works equally well as long as all agents in an environment agree. Similar considerations apply to the development of human language, where the important thing is not which language each individual should speak, but the fact that a community all speaks the same language. When conventions are widespread, they are called **social laws**.

In the absence of a convention, agents can use **communication** to achieve common knowledge of a feasible joint plan. For example, a tennis player could shout “Mine!” or “Yours!” to indicate a preferred joint plan. Communication does not necessarily involve a verbal exchange. For example, one player can communicate a preferred joint plan to the other simply by executing the first part of it. If agent *A* heads for the net, then agent *B* is obliged to go back to the baseline to hit the ball, because plan 2 is the only joint plan that begins with A’s heading for the net. This approach to coordination, sometimes called **plan recognition**, works when a single action (or short sequence of actions) by one agent is enough for the other to determine a joint plan unambiguously.

17.2 Non-Cooperative Game Theory

We will now introduce the key concepts and analytical techniques of game theory—the theory that underpins decision making in multiagent environments. Our tour will start with noncooperative game theory.

17.2.1 Games with a single move: Normal form games

The first game model we will look at is one in which all players take action simultaneously and the result of the game is based on the profile of actions that are selected in this way. (Actually, it is not crucial that the actions take place at the same time; what matters is that no player has knowledge of the other players' choices.) These games are called **normal form games**. A normal form game is defined by three components:

- **Players** or agents who will be making decisions. Two-player games have received the most attention, although n-player games for $n > 2$ are also common. We give players capitalized names, like *Ali* and *Bo* or *O* and *E*.
- **Actions** that the players can choose. We will give actions lowercase names, like *one* or *testify*. The players may or may not have the same set of actions available.
- A **payoff function** that gives the utility to each player for each combination of actions by all the players. For two-player games, the payoff function for a player can be represented by a matrix in which there is a row for each possible action of one player, and a column for each possible choice of the other player: a chosen row and a chosen column define a matrix cell, which is labeled with the payoff for the relevant player. In the two-player case, it is conventional to combine the two matrices into a single **payoff matrix**, in which each cell is labeled with payoffs for both players.

To illustrate these ideas, let's look at an example game, called **two-finger Morra**. In this game, two players, *O* and *E*, simultaneously display one or two fingers. Let the total number of fingers displayed be *f*. If *f* is odd, *O* collects *f* dollars from *E*; and if *f* is even, *E* collects *f* dollars from *O*.[1] The payoff matrix for two-finger Morra is as follows:
We say that *E* is the **row player** and *O* is the **column player**. So, for example, the lower-right corner shows that when player *O* chooses action *two* and *E* also chooses *two*, the payoff is +4 for *E* and –4 for *O*.

Before analyzing two-finger Morra, it is worth considering why game-theoretic ideas are needed at all: why can't we tackle the challenge facing (say) player *E* using the apparatus of decision theory and utility maximization that we've been using elsewhere in the book? To see why something else is needed, let's suppose *E* is trying to find the best action to perform. The alternatives are *one* or two. If *E* chooses *one,* then the payoff will be either +2 or –3. Which payoff *E* will *actually* receive, however, will depend on the choice made by *O*: the most that *E* can do, as the row player, is to force the outcome of the game to be in a particular row. Similarly, *O* chooses only the column.

To choose optimally between these possibilities, *E* must take into account how *O* will act as a rational decision maker. But *O*, in turn, should take into account the fact that *E* is a rational decision maker. Thus, decision making in multiagent settings is quite different in character to decision making in single-agent settings, because the players need to take each other's reasoning into account. The role of **solution concepts** in game theory is to try to make this kind of reasoning precise.

The term **strategy** is used in game theory to denote what we have previously called a *policy*. A **pure strategy** is a deterministic policy; for a single-move game, a pure strategy is just a single action. As we will see below, for many games an agent can do better with a **mixed strategy**, which is a randomized policy that selects actions according to a probability distribution. The mixed strategy that chooses action *a* with probability *p* and action *b* otherwise is written [p: a; $(1 - p)$: b]. For example, a mixed strategy for two-finger Morra might be [0.5: *one*;0.5: *two*]. A **strategy profile** is an assignment of a strategy to each player; given the strategy profile, the game's **outcome** is a numeric value for each player—if players use mixed strategies, then we must use expected utility.

So, how should agents decide act in games like Morra? Game theory provides a range of solution concepts that attempt to define rational action with respect to an agent's beliefs about the other agent's beliefs. Unfortunately, there is no one perfect solution concept: it is problematic to define what "rational" means when each agent chooses only part of the strategy profile that determines the outcome.

We introduce our first solution concept through what is probably the most famous game in the game theory canon—the **prisoner's dilemma**. This game is motivated by the following story: Two alleged burglars, Ali and Bo, are caught red-handed near the scene of a burglary and are interrogated separately. A prosecutor offers each a deal: if you testify against your partner as the leader of a burglary ring, you'll go free for being the cooperative one, while your partner will serve 10 years in prison. However, if you both testify against each other, you'll both get 5 years. Ali and Bo also know that if both refuse to testify they will serve only 1 year each for the lesser charge of possessing stolen property. Now Ali and Bo face the so-called prisoner's dilemma: should they testify or refuse? Being rational agents, Ali and Bo each want to maximize their own expected utility, which means minimizing the number of years in prison—each is indifferent about the welfare of the other player. The prisoner's dilemma is captured in the following payoff matrix:

Now, put yourself in Ali's place. She can analyze the payoff matrix as follows:

- Suppose Bo plays *testify*. Then I get 5 years if I testify and 10 years if I don't, so in that case testifying is better.
- On the other hand, if Bo plays *refuse*, then I go free if I testify and I get 1 year if I refuse, so testifying is also better in that case.
- So *no matter what Bo chooses to do,* it would be better for me to testify.

Ali has discovered that *testify* is a **dominant strategy** for the game. We say that a strategy s for player p **strongly dominates** strategy s' if the outcome for s is better for p than the outcome for s', for every choice of strategies by the other player(s). Strategy s **weakly dominates** s' if s is better than s' on at least one strategy profile and no worse on any other. A dominant strategy is a strategy that dominates all others. A common assumption in game theory is that *a rational player will always choose a dominant strategy and avoid a dominated strategy*. Being rational—or at least not wishing to be thought irrational—Ali chooses the dominant strategy.

It is not hard to see that Bo's reasoning will be identical: he will also conclude that *testify* is a dominant strategy for him, and will choose to play it. The solution of the game, according to dominant strategy analysis, will be that both players choose *testify,* and as a consequence both will serve 5 years in prison.

In a situation like this, where all players choose a dominant strategy, then the outcome that results is said to be a **dominant strategy equilibrium**. It is an "equilibrium" because no player has any incentive to deviate from their part of it: by definition, if they did so, they could not do better, and may do worse. In this sense, dominant strategy equilibrium is a very strong solution concept.

Going back to the prisoner's dilemma, we can see that the *dilemma* is that the dominant strategy equilibrium outcome in which both players *testify* is worse for both players than the outcome they would get if they both refused to testify. The (*refuse*, *refuse*) outcome would give both players just one year in prison, which would be better for *both* of them than the 5 years that each would serve if they chose the dominant strategy equilibrium.

Is there any way for Ali and Bo to arrive at the *(refuse, refuse)* outcome? It is certainly an *allowable* option for both of them to refuse to testify, but it is hard to see how rational agents could make this choice, given the way the game is set up. Remember, this is a noncooperative game: they aren't allowed to talk to each other, so they cannot make a binding agreement to *refuse*.

It is, however, possible to get to the (*refuse, refuse)* solution if we change the game. We could change it to a cooperative game where the agents are allowed to form a binding agreement. Or we could change to a **repeated game** in which the players know that they will meet again—we will see how this works below. Alternatively, the players might have moral beliefs that encourage cooperation and fairness. But that would mean they have different utility functions, and again, they would be playing a different game.

The presence of a dominant strategy for a particular player greatly simplifies the decision making process for that player. Once Ali has realized that testifying is a dominant strategy, she doesn't need to invest any effort in trying to figure out what Bo will do, because she knows that *no matter what Bo does,* testifying would be her **best response**. However, most games have neither dominant strategies nor dominant strategy equilibria. It is rare that a single strategy is the best response to all possible counterpart strategies.

The next solution concept we consider is weaker than dominant strategy equilibrium, but it is much more widely applicable. It is called **Nash equilibrium**, and is named for John Forbes Nash, Jr. (1928–2015), who studied it in his 1950 Ph.D. thesis—work for which he was awarded a Nobel Prize in 1994.

A strategy profile is a Nash equilibrium if no player could unilaterally change their strategy and as a consequence receive a higher payoff, under the assumption that the other players stayed with their strategy choices. Thus, in a Nash equilibrium, every player is simultaneously playing a best response to the choices of their counterparts. A Nash equilibrium represents a stable point in a game: stable in the sense that there is no rational incentive for any player to deviate. However, Nash equilibria are *local* stable points: as we will see, a game may contain multiple Nash equilibria.

Since a dominant strategy is a best response to *all* counterpart strategies, it follows that any dominant strategy equilibrium must also be a Nash equilibrium (Exercise 17.EQIB). In the prisoner's dilemma, therefore, there is a unique dominant strategy equilibrium, which is also the unique Nash equilibrium.

The following example game demonstrates, first, that sometimes games have no dominant strategies, and second, that some games have multiple Nash equilibria.

It is easy to verify that there are no dominant strategies in this game, for either player, and hence no dominant strategy equilibrium. However, the strategy profiles (*t*, *l*) and (*b*, *r*) are both Nash equilibria. Now, clearly it is in the interests of both agents to aim for the *same* Nash equilibrium—either (*t*, *l*) or (*b*, *r*)—but since we are in the domain of *non-cooperative* game theory, players must make their choices independently, without any knowledge of the choices of the others, and without any way of making an agreement with them. This is an example of a **coordination problem**: the players want to coordinate their actions globally, so that they both choose actions leading to the same equilibrium, but must do so using only local decision making.

A number of approaches to resolving coordination problems have been proposed. One idea is that of **focal points**. A focal point in a game is an outcome that in some way stands out to players as being an "obvious" outcome upon which to coordinate their choices. This is of course not a precise definition—what it means will depend on the game at hand. In the example above, though, there is one obvious focal point: the outcome (*t*, *l*) would give both players substantially higher utility than they would obtain if they coordinated on (*b*, *r*). From the point of view of game theory, both outcomes are Nash equilibria—but it would be a perverse player indeed who expected to coordinate on (*b, r*).

Some games have no Nash equilibria in pure strategies, as the following game, called **matching pennies**, illustrates. In this game, Ali and Bo simultaneously choose one side of a coin, either heads of tails: if they make the same choices, then Bo gives Ali $1, while if they make different choices, then Ali gives Bo $1:

We invite the reader to check that the game contains no dominant strategies, and that no outcome is a Nash equilibrium in pure strategies: in every outcome, one player regrets their choice, and would rather have chosen differently, given the choice of the other player.

To find a Nash equilibrium, the trick is to use mixed strategies—to allow players to randomize over their choices. Nash proved that *every game has at least one Nash equilibrium in mixed strategies*. This explains why Nash equilibrium is such an important solution concept: other solution concepts, such as dominant strategy equilibrium, are not guaranteed to exist for every game, but we always get a solution if we look for Nash equilibria with mixed strategies.

In the case of matching pennies, we have a Nash equilibrium in mixed strategies if both players choose *heads* and *tails* with equal probability. To see that this outcome is indeed a Nash equilibrium, suppose one of the players chose an outcome with a probability other than 0.5. Then the other player would be able to exploit that, putting all their weight behind a particular strategy. For example, suppose Bo played *heads* with probability 0.6 (and so *tails* with probability 0.4). Then Ali would do best to play *heads* with certainty. It is then easy to see that Bo playing *heads* with probability 0.6 could not form part of any Nash equilibrium.

17.2.2 Social welfare

The main perspective in game theory is that of players within the game, trying to obtain the best outcomes for themselves that they can. However, it is sometimes instructive to adopt a different perspective. Suppose you were a benevolent, omniscient entity looking down on the game, and you were able to *choose* the outcome. Being benevolent, you want to choose the best overall outcome—the outcome that would be best for *society as a whole*, so to speak. How should you choose? What criteria might you apply? This is where the notion of **social welfare** comes in.

Probably the most important and least contentious social welfare criterion is that you should avoid outcomes that *waste* utility. This requirement is captured in the concept of **Pareto optimality**, which is named for the Italian economist Vilfredo Pareto (1848–1923). An outcome is Pareto optimal if there is no other outcome that would make one player better off without making someone else worse off. If you choose an outcome that is not Pareto optimal, then it wastes utility in the sense that you could have given more utility to at least one agent, without taking any from other agents.

Utilitarian social welfare is a measure of how good an outcome is in the aggregate. The utilitarian social welfare of an outcome is simply the sum of utilities given to players by that outcome. There are two key difficulties with utilitarian social welfare. The first is that it considers the sum but not the *distribution* of utilities among players, so it could lead to a very unequal distribution if that happens to maximize the sum. The second difficulty is that it assumes a *common scale* for utilities. Many economists argue that this is impossible to establish because utility (unlikely money) is a subjective quantity. If we're trying to decide how to divide up a batch of cookies, should we give them all to the utility monster who says, "I love cookies a thousand times more than anyone else?" That would maximize the total self-reported utility, but doesn't seem right.

The question of how utility is distributed among players is addressed by research in **egalitarian social welfare**. For example, one proposal suggests that we should maximize the expected utility of the worst–off member of society—a maximin approach. Other metrics are possible, including the **Gini coefficient**, which summarizes how evenly utility is spread among the players. The main difficulties with such proposals is that they may sacrifice a great deal of total welfare for small distributional gains, and, like plain utilitarianism, they are still at the mercy of the utility monster.

Applying these concepts to the prisoner's dilemma game, introduced above, explains why it is called a dilemma. Recall that (*testify*, *testify*) is a dominant strategy equilibrium, and the only Nash equilibrium. However, this is the only outcome that is *not* Pareto optimal. The outcome (*refuse*, *refuse*) maximizes both utilitarian and egalitarian social welfare. The dilemma in the prisoner's dilemma thus arises because a very strong solution concept (dominant strategy equilibrium) leads to an outcome that essentially fails every test of what counts as a reasonable outcome from the point of view of the "society." Yet there is no clear way for the individual players to arrive at a better solution.

Computing equilibria

Let's now consider the key computational questions associated with the concepts discussed above. First we will consider pure strategies, where randomization is not permitted.

If players have only a finite number of possible choices, then exhaustive search can be used to find equilibria: iterate through each possible strategy profile, and check whether any player has a beneficial deviation from that profile; if not, then it is a Nash equilibrium in pure strategies. Dominant strategies and dominant strategy equilibria can be computed by similar algorithms. Unfortunately, the number of possible strategy profiles for n players each with m possible actions, is m^n, i.e., infeasibly large for an exhaustive search.

An alternative approach, which works well in some games, is **myopic best response** (also known as **iterated best response**): start by choosing a strategy profile at random; then, if some player is not playing their optimal choice given the choices of others, flip their choice to an optimal one, and repeat the process. The process will converge if it leads to a strategy profile in which every player is making an optimal choice, given the choices of the others—a Nash equilibrium, in other words. For some games, myopic best response does not converge, but for some important classes of games, it is guaranteed to converge.

Computing mixed–strategy equilibria is algorithmically much more intricate. To keep things simple, we will focus on methods for zero–sum games and comment briefly on their extension to other games at the end of this section.

In 1928, von Neumann developed a method lor finding the *optimal* mixed strategy lor two-player, **zero-sum games**—games in which the payoffs always add up to zero (or a constant, as explained on page 193). Clearly, Morra is such a game. For two-player, zero-sum games, we know that the payoffs are equal and opposite, so we need consider the payoffs of only one player, who will be the maximizer (just as in Chapter 6). For Morra, we pick the even player E to be the maximizer, so we can define the payoff matrix by the values $U_E(e, o)$—the payoff to E if E does e and O does o. (For convenience we call player E "her" and O "him.") Von Neumann's method is called the **maximin** technique, and it works as follows:

- Suppose we change the rules as follows: first E picks her strategy and reveals it to O. Then O picks his strategy, with knowledge of E's strategy. Finally, we evaluate the expected payoff of the game based on the chosen strategies. This gives us a turn-taking game to which we can apply the standard **minimax** algorithm from Chapter 6. Let's suppose this gives an outcome $U_{E,O}$. Clearly, this game favors O, so the true utility U of the original game (from E's point of view) is *at least* $U_{E,O}$. For example, if we just look at pure strategies, the minimax game tree has a root value of -3 (see Figure 17.2(a)), so we know that $U \geq -3$.

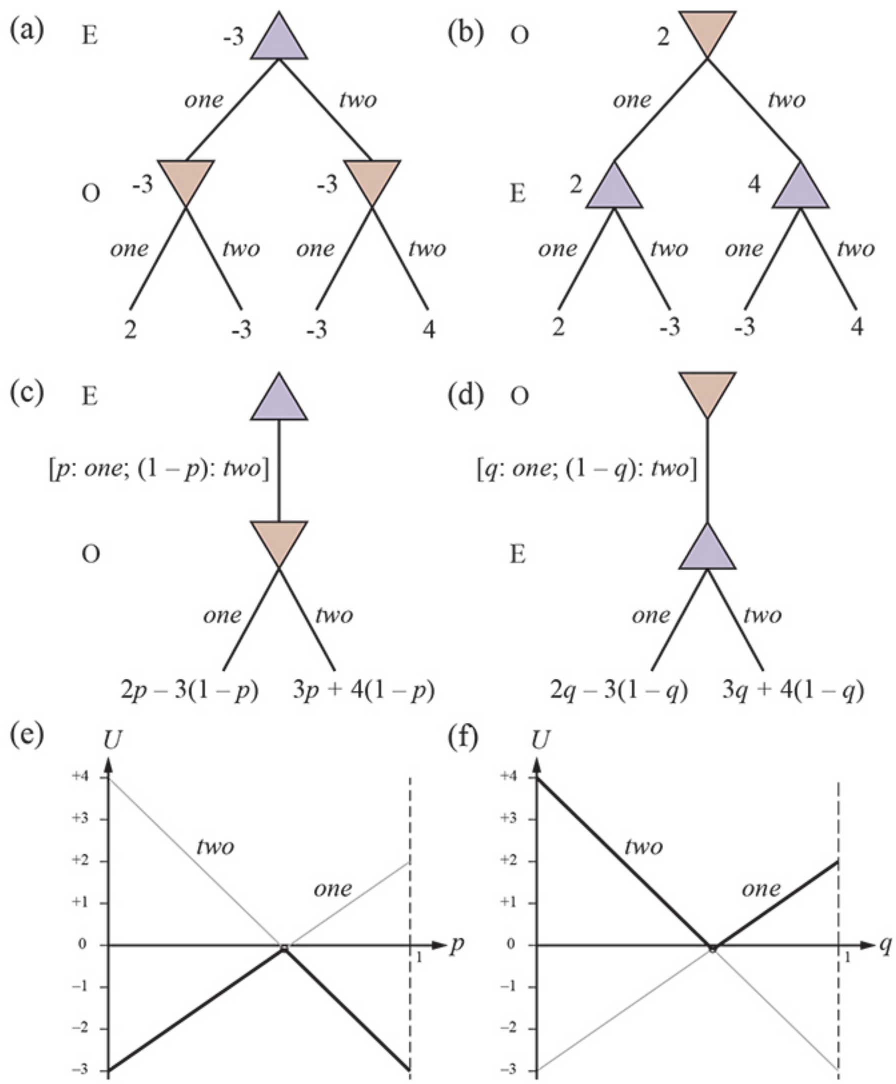


Figure 17.2 (a) and (b): Minimax game trees for two-finger Morra if the players take turns playing pure strategies. (c) and (d): Parameterized game trees where the first player plays a mixed strategy. The payoffs depend on the probability parameter (p or q) in the mixed strategy. (e) and (f): For any particular value of the probability parameter, the second player will choose the "better" of the two actions, so the value of the first player's mixed strategy is given by the heavy lines. The first player will choose the probability parameter for the mixed strategy at the intersection point.

- Now suppose we change the rules to force O to reveal his strategy first, followed by E. Then the minimax value of this game is $U_{O,E}$, and because this game favors E we know that U is at most $U_{O,E}$. With pure strategies, the value is +2 (see Figure 17.2(b)), so we know $U \leq +2$.

Combining these two arguments, we see that the true utility U of the solution to the original game must satisfy

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

To pinpoint the value of U, we need to turn our analysis to mixed strategies. First, observe the following: *once the first player has revealed a strategy, the second player might as well choose a pure strategy*. The reason is simple: if the second player plays a mixed strategy, [p: *one*; $(1-p)$:*two*], its expected utility is a linear combination ($p \cdot U_{one} + (1 - p) \cdot U_{two}$) of the utilities of the pure strategies, U_{one} and U_{two}. This linear combination can never be better than the better of U_{one} and U_{two}, so the second player can just choose the better one.

With this observation in mind, the minimax trees can be thought of as having infinitely many branches at the root, corresponding to the infinitely many mixed strategies the first player can choose. Each of these leads to a node with two branches corresponding to the pure strategies for the second player. We can depict these infinite trees finitely by having one "parameterized" choice at the root:

- If E chooses first, the situation is as shown in Figure 17.2(c). E chooses the strategy [p: *one;* $(1-p)$: *two*] at the root, and then O chooses a pure strategy (and hence a move) given the value of p. If O chooses *one*, the expected payoff (to E) is $2p - 3(1-p) = 5p - 3$; if O chooses two, the expected payoff is $-3p + 4(1-p) = 4 - 7p$. We can draw these two payoffs as straight lines on a graph, where p ranges from 0 to 1 on the x–axis, as shown in Figure 17.2(e). O, the minimizer, will always choose the lower of the two lines, as shown by the heavy lines in the figure. Therefore, the best that E can do at the root is to choose p to be at the intersection point, which is where

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

  The utility for E at this point is $U_{E,O} = -1/12$.
- If O moves first, the situation is as shown in Figure 17.2(d). O chooses the strategy [q: *one*; $(1-q)$: *two*] at the root, and then E chooses a move given the value of q. The payoffs are $2q - 3(1-q) = 5q - 3$ and $-3q + 4(1-q) = 4 - 7q$.[2] Again, Figure 17.2(f) shows that the best O can do at the root is to choose the intersection point:

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

  The utility for E at this point is $U_{O,E} = -1/12$.

Now we know that the true utility of the original game lies between –1/12 and –1/12; that is, it is exactly – 1/12! (The conclusion is that it is better to be O than E if you are playing this game.) Furthermore, the true utility is attained by the mixed strategy [7/12:*one*;5/12:*two*], which should be played by both players. This strategy is called the **maximin equilibrium** of the game, and is a Nash equilibrium. Note that each component strategy in an equilibrium mixed strategy has the same expected utility. In this case, both *one* and *two* have the same expected utility, –1/12, as the mixed strategy itself.

Our result for two-finger Morra is an example of the general result by von Neumann: *every two-player zero-sum game has a maximin equilibrium when you allow mixed strategies*. Furthermore, every Nash equilibrium in a zero–sum game is a maximin for both players. A player who adopts the maximin strategy has two guarantees: First, no other strategy can do better against an opponent who plays well (although some other strategies might be better at exploiting an opponent who makes irrational mistakes). Second, the player continues to do just as well even if the strategy is revealed to the opponent.

The general algorithm for finding maximin equilibria in zero–sum games is somewhat more involved than Figures 17.2(e) and (f) might suggest. When there are n possible actions, a mixed strategy is a point in n-dimensional space and the lines become hyperplanes. It's also possible for some pure strategies for the second player to be dominated by others, so that they are not optimal against *any* strategy for the first player. After removing all such strategies (which might have to be done repeatedly), the optimal choice at the root is the highest (or lowest) intersection point of the remaining hyperplanes.

Finding this choice is an example of a **linear programming** problem: maximizing an objective function subject to linear constraints. Such problems can be solved by standard techniques in time polynomial in the number of actions (and in the number of bits used to specify the reward function, if you want to get technical).

The question remains, what should a rational agent actually *do* in playing a single game of Morra? The rational agent will have derived the fact that [7/12: *one*;5/12: *two*] is the maximin equilibrium strategy, and will assume that this is mutual knowledge with a rational opponent. The agent could use a 12-sided die or a random number generator to pick randomly according to this mixed strategy, in which case the expected payoff would be -1/12 for *E*. Or the agent could just decide to play *one*, or two. In either case, the expected payoff remains -1/12 for *E*. Curiously, unilaterally choosing a particular action does not harm one's expected payoff, but allowing the other agent to know that one has made such a unilateral decision *does* affect the expected payoff, because then the opponent can adjust strategy accordingly.

Finding equilibria in non-zero-sum games is somewhat more complicated. The general approach has two steps: (1) Enumerate all possible subsets of actions that might form mixed strategies. For example, first try all strategy profiles where each player uses a single action, then those where each player uses either one or two actions, and so on. This is exponential in the number of actions, and so only applies to relatively small games. (2) For each strategy profile enumerated in (1), check to see if it is an equilibrium. This is done by solving a set of equations and inequalities that are similar to the ones used in the zero–sum case. For two players these equations are linear and can be solved with basic linear programming techniques, but for three or more players they are nonlinear and may be very difficult to solve.

17.2.3 Repeated games

So far, we have looked only at games that last a single move. The simplest kind of multiple– move game is the **repeated game** (also called an **iterated game**), in which players repeatedly play rounds of a single–move game, called the **stage game**. A strategy in a repeated game specifies an action choice for each player at each time step for every possible history of previous choices of players.

First, let's look at the case where the stage game is repeated a fixed, finite, and mutually known number of rounds—all of these conditions are required for the following analysis to work. Let's suppose Ali and Bo are playing a repeated version of the prisoner's dilemma, and that both they know that they must play exactly 100 rounds of the game. On each round, they will be asked whether to *testify* or *refuse,* and will receive a payoff for that round according to the rules of the prisoner's dilemma that we saw above.

At the end of 100 rounds, we find the overall payoff for each player by summing that player's payoffs in the 100 rounds. What strategies should Ali and Bo choose to play this game? Consider the following argument. They both know that the 100th round will not be a repeated game—that is, its outcome can have no effect on future rounds. So, on the 100th round, they are in effect playing a single prisoner's dilemma game.

As we saw above, the outcome of the 100th round will be (*testify*, *testify*), the dominant equilibrium strategy for both players. But once the 100th round is determined, the 99th round can have no effect on subsequent rounds, so it too will yield (*testify*, *testify*). By this inductive argument, both players will choose *testify* on every round, earning a total jail sentence of 500 years each. This type of reasoning is known as **backward induction**, and plays a fundamental role in game theory.

However, if we drop one of the three conditions—fixed, finite, or mutually known—then the inductive argument doesn't hold. Suppose that the game is repeated an *infinite* number of times. Mathematically, a strategy for a player in an infinitely repeated game is a function that maps every possible finite history of the game to a choice in the stage game for that player in the corresponding round. Thus, a strategy looks at what happened previously in the game, and decides what choice to make in the current round. But we can't store an infinite table in a finite computer. We need a *finite* model of strategies for games that will be played an *infinite* number of rounds. For this reason, it is standard to represent strategies for infinitely repeated games as finite state machines (FSMs) with output.

Figure 17.3 illustrates a number of FSM strategies for the iterated prisoner's dilemma. Consider the **Tit-for-Tat** strategy. Each oval is a state of the machine, and inside the oval is the choice that would be made by the strategy if the machine was in that state. From each state, we have one outgoing edge for every possible choice of the counterpart agent: we follow the outgoing edge corresponding to the choice made by the other to find the next state of the machine. Finally, one state is labeled with an incoming arrow, indicating that it is the initial state. Thus, with TIT-FOR-TAT, the machine starts in the *refuse* state; if the counterpart agent plays *refuse*, then it stays in the *refuse* state, while if the counterpart plays *testify* it transitions to the *testify* state. it will remain in the *testify* state as long its counterpart plays *testify*, but if ever its counterpart plays *refuse*, it will transition back to the *refuse* state. In sum, TIT-FOR-TAT will start by choosing *refuse*, and will then simply copy whatever its counterpart did on the previous round.

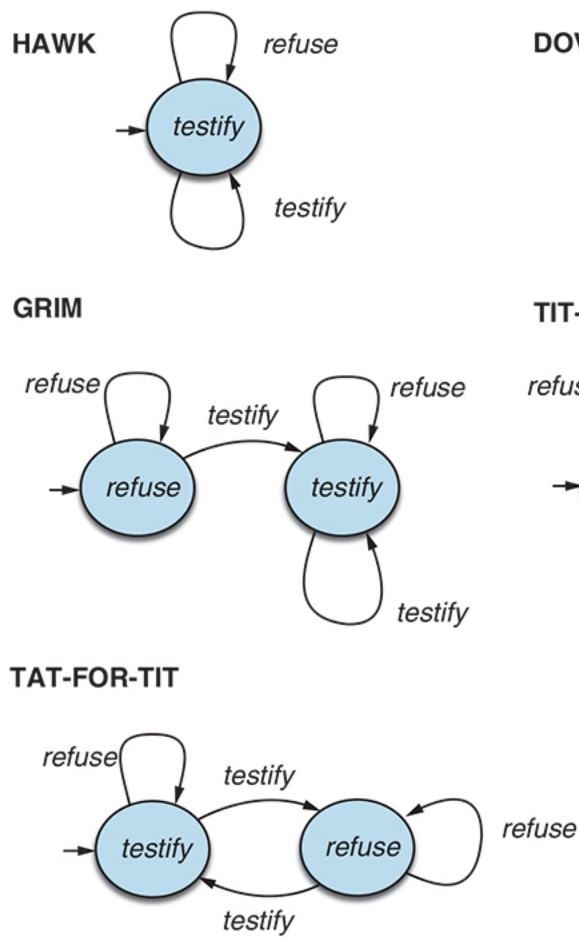


Figure 17.3 Some common, colorfully named finite-state machine strategies for the infinitely repeated prisoner's dilemma.

The HAWK and DOVE strategies are simpler: HAWK simply plays *testify* on every round, while DOVE simply plays *refuse* on every round. The GRIM strategy is somewhat similar to TIT-FOR-TAT, but with one important difference: if ever its counterpart plays *testify,* then it essentially turns into HAWK: it plays *testify* forever. While TIT-FOR-TAT is forgiving, in the sense that it will respond to a subsequent *refuse* by reciprocating the same, with GRIM there is no way back. Just playing *testify* once will result in punishment (playing *testify*) that goes on forever. (Can you see what TAT-FOR-TIT does?)

The next issue with infinitely repeated games is how to measure the utility of an infinite sequence of payoffs. Here, we will focus on the **limit of means** approach—essentially, this means taking the average of utilities received over the infinite sequence. With this approach, given an infinite sequence of payoffs (U_0, U_1, U_2,...), we define the utility of the sequence to the corresponding player to be:

$$\lim_{T \to \infty} \frac{1}{T} \sum_{t=0}^{T} U_t.$$

This value cannot be guaranteed to converge for arbitrary sequences of utilities, but it *is* guaranteed to do so for the utility sequences that are generated if we use FSM strategies. To see this, observe that if FSM strategies play against each other, then *eventually, the FSMs will reenter a configuration that they were in previously, at which point they will start to repeat themselves*. More precisely, any utility sequence generated by FSM strategies will consist of a finite (possibly empty) non-repeating sequence, followed by a nonempty finite sequence that repeats infinitely often. To compute the average utility received by a player over that infinite sequence, we simply have to compute the average over the finite repeating sequence.

In what follows, we will assume that players in an infinitely repeated game simply choose a finite state machine to play the game on their behalf. We don't impose any constraints on these machines: they can be as big and elaborate as players want. When all players have chosen a finite state machine to play on their behalf, then we can compute the payoffs for each player using the limit of means approach as described above. In this way, an infinitely repeated game reduces to a normal form game, albeit one with infinitely many possible strategies for each player.

Let's see what happens when we play the infinitely repeated prisoner's dilemma using some strategies from Figure 17.3. First, suppose Ali and Bo both pick DOVE.

$$\begin{array}{llccccccclcr}
 & & 0 & 1 & 2 & 3 & 4 & 5 & \ldots & & & \\
\text{Ali:} & \text{DOVE} & refuse & refuse & refuse & refuse & refuse & refuse & \ldots & \text{utility} & = & -1 \\
\text{Bo:} & \text{DOVE} & refuse & refuse & refuse & refuse & refuse & refuse & \ldots & \text{utility} & = & -1
\end{array}$$

It is not hard to see that this strategy pair does not form a Nash equilibrium: either player would have done better to alter their choice to HAWK. So, suppose Ali switches to HAWK:

$$\begin{array}{llccccccclc}
 & & 0 & 1 & 2 & 3 & 4 & 5 & \ldots & & \\
\text{Ali:} & \text{HAWK} & testify & testify & testify & testify & testify & testify & \ldots & \text{utility} & = \\
\text{Bo:} & \text{DOVE} & refuse & refuse & refuse & refuse & refuse & refuse & \ldots & \text{utility} & =
\end{array}$$

This is the worst possible outcome for Bo; and this strategy pair is again not a Nash equilibrium. Bo would have done better by also choosing HAWK:

$$\begin{array}{llccccccclc}
 & & 0 & 1 & 2 & 3 & 4 & 5 & \ldots & & \\
\text{Ali:} & \text{HAWK} & testify & testify & testify & testify & testify & testify & \ldots & \text{utility} & = \\
\text{Bo:} & \text{HAWK} & refuse & testify & testify & testify & testify & testify & \ldots & \text{utility} & =
\end{array}$$

This strategy pair *does* form a Nash equilibrium, but not a very interesting one—it takes us more or less back to where we started in the one-shot version of the game, with both players testifying against each other. It illustrates a key property of infinitely repeated games: *Nash equilibria of the stage game will be sustained as equilibria in an infinitely repeated version of the game*.

However, our story is not over yet. Suppose that Bo switched to GRIM:

$$\begin{array}{llccccccclc}
 & & 0 & 1 & 2 & 3 & 4 & 5 & \ldots & & \\
\text{Ali:} & \text{HAWK} & testify & testify & testify & testify & testify & testify & \ldots & \text{utility} & = \\
\text{Bo:} & \text{GRIM} & refuse & testify & testify & testify & testify & testify & \ldots & \text{utility} & =
\end{array}$$

Here, Bo does no worse than playing HAWK: on the first round, Ali plays *testify* while Bo plays *refuse*, but this triggers Bo into testifying forever after: the loss of utility on the first round disappears in the limit. Overall, the two players get the same utility as if they had both played HAWK. But here is the thing: these strategies do not form a Nash equilibrium because this time, Ali has a beneficial deviation—to GRIM. If both players choose GRIM, then this is what happens:

		0	1	2	3	4	5	...			
Ali:	GRIM	$refuse$	$refuse$	$refuse$	$refuse$	$refuse$	$refuse$	...	utility	=	-1
Bo:	GRIM	$refuse$	$refuse$	$refuse$	$refuse$	$refuse$	$refuse$	...	utility	=	-1

The outcomes and payoffs are the same as if both players had chosen DOVE, but unlike that case, GRIM playing against GRIM forms a Nash equilibrium, and Ali and Bo are able to rationally achieve an outcome that is impossible in the one-shot version of the game.

To see that these strategies form a Nash equilibrium, suppose for the sake of contradiction that they do not. Then one player—assume without loss of generality that it is Ali—has a beneficial deviation, in the form of an FSM strategy that would yield a higher payoff than GRIM. Now, at some point this strategy would have to do something different from GRIM— otherwise it would obtain the same utility. So, at some point it must play *testify*. But then Bo's GRIM strategy would flip to punishment mode, by permanently testifying in response. At that point, Ali would be doomed to receive a payoff of no more than –5: worse than the –1 she would have received by choosing GRIM. Thus, both players choosing GRIM forms a Nash equilibrium in the infinitely repeated prisoner's dilemma, giving a rationally sustained outcome that is impossible in the one-shot version of the game.

This is an instance of a general class of results called the **Nash folk theorems**, which characterize the outcomes that can be sustained by Nash equilibria in infinitely repeated games. Let's say a player's *security value* is the best payoff that the player could guarantee to obtain. Then the general form of the Nash folk theorems is roughly that *every outcome in which every player receives at least their security value can be sustained as a Nash equilibrium in an infinitely repeated game*. GRIM strategies are the key to the folk theorems: the mutual threat of punishment if any agent fails to play their part in the desired outcome keeps players in line. But it works as a deterrent only if the other player believes you have adopted this strategy—or at least that you might have adopted it.

We can also get different solutions by changing the agents, rather than changing the rules of engagement. Suppose the agents are finite state machines with n states and they are playing a game with $m > n$ total steps. The agents are thus incapable of representing the number of remaining steps, and must treat it as an unknown. Therefore, they cannot do the backward induction, and are free to arrive at the more favorable (*refuse*, *refuse*) equilibrium in the iterated Prisoner's Dilemma. In this case, ignorance *is* bliss—or rather, having your opponent believe that you are ignorant is bliss. Your success in these repeated games depends to a significant extent on the other player's *perception* of you as a bully or a simpleton, and not on your actual characteristics.

17.2.4 Sequential games: The extensive form

In the general case, a game consists of a sequence of turns that need not be all the same. Such games are best represented by a game tree, which game theorists call the **extensive form**. The tree includes all the same information we saw in Section 6.1: an initial state S_0 a function PLAYER(s) that tells which player has the move, a function ACTIONS(s) enumerating the possible actions, a function RESULT(s, a) that defines the transition to a new state, and a partial function UTILITY(s, p), which is defined only on terminal states, to give the payoff for each player. Stochastic games can be captured by introducing a distinguished player, *Chance*, that can take random actions. *Chance's* "strategy" is part of the definition of the game, specified as a probability distribution over actions (the other players get to choose their own strategy). To represent games with nondeterministic actions, such as billiards, we break the action into two pieces: the player's action itself has a deterministic result, and then *Chance* has a turn to react to the action in its own capricious way.

For the moment, we will make one simplifying assumption: we assume players have **perfect information**. Roughly, perfect information means that, when the game calls upon them to make a decision, they know precisely where they are in the game tree: they have no uncertainty about what has happened previously in the game. This is, of course, the situation in games like chess or Go, but not in games like poker or Kriegspiel. In the following section, we will show how the extensive form can be used to capture **imperfect information** in games, but for the moment, we will assume perfect information.

A strategy in an extensive-form game of perfect information is a function for a player that for every one of its decision states *s* dictates which action in ACTIONS(*s*) the player should choose to execute. When each player has selected a strategy, then the resulting strategy profile will trace a path in the game tree from the initial state S_0 to a terminal state, and the UTILITY function defines the utilities that each player will then receive.

Given this setup, we can directly apply the apparatus of Nash equilibria that we introduced above to analyze extensive-form games. To compute Nash equilibria, we can use a straightforward generalization of the minimax search technique that we saw in Chapter 6. In the literature on extensive-form games, the technique is called backward induction—we already saw backward induction informally used to analyze the finitely repeated prisoner's dilemma. Backward induction uses dynamic programming, working backwards from terminal states back to the initial state, progressively labeling each state with a payoff profile (an assignment of payoffs to players) that would be obtained if the game was played optimally from that point on.

In more detail, for each nonterminal state *s*, if all the children of *s* have been labeled with a payoff profile, then label *s* with a payoff profile from the child state that maximizes the payoff of the player making the decision at state *s*. (If there is a tie, then choose arbitrarily; if we have chance nodes, then compute expected utility.) The backward induction algorithm is guaranteed to terminate, and moreover runs in time polynomial in the size of the game tree.

As the algorithm does its work, it traces out strategies for each player. As it turns out, these strategies are Nash equilibrium strategies, and the payoff profile labeling the initial state is a payoff profile that would be obtained by playing Nash equilibrium strategies. Thus, Nash equilibrium strategies for extensive-form games can be computed in polynomial time using backward induction; and since the algorithm is guaranteed to label the initial state with a payoff profile, it follows that every extensive-form game has at least one Nash equilibrium in pure strategies.

These are attractive results, but there are several caveats. Game trees very quickly get very large, so polynomial running time should be understood in that context. But more problematically, Nash equilibrium itself has some limitations when it is applied to extensive-form games. Consider the game in Figure 17.4. Player 1 has two moves available: *above* or *below*. If she moves *below,* then both players receive a payoff of 0 (regardless of the move selected by player 2). If she moves *above,* then player 2 is presented with a choice of moving *up* or *down:* if she moves *down*, then both players receive a payoff of 0, while if she moves *up*, then they both receive 1.

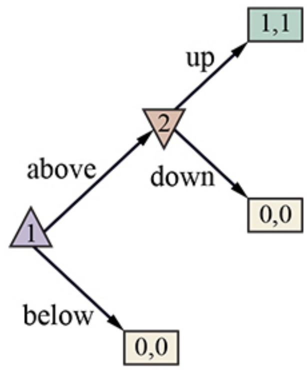


Figure 17.4 An extensive-form game with a counterintuitive Nash equilibrium.

Backward induction immediately tells us that (*above, up*) is a Nash equilibrium, resulting in both players receiving a payoff of 1. However, (*below, down*) is also a Nash equilibrium, which would result in both players receiving a payoff of 0. Player 2 is threatening player 1, by indicating that if called upon to make a decision she will choose *down*, resulting in a payoff of 0 for player 1; in this case, player 1 has no better alternative than choosing *below*. The problem is that player 2's threat (to play *down*) is not a **credible threat**, because if player 2 is actually called upon to make the choice, then she will choose *up*.

A refinement of Nash equilibrium called **subgame perfect Nash equilibrium** deals with this problem. To define it, we need the idea of a **subgame**. Every decision state in a game tree (including the initial state) defines a subgame—the game in Figure 17.4 therefore contains two subgames, one rooted at player 1's decision state, one rooted at player 2's decision state. *A profile of strategies then forms a subgame perfect Nash equilibrium in a game G if it is a Nash equilibrium in every subgame of G*. Applying this definition to the game of Figure 17.4, we find that (*above, up*) is subgame perfect, but (*below, down*) is not, because choosing *down* is not a Nash equilibrium of the subgame rooted at player 2's decision state.

Although we needed some new terminology to define subgame perfect Nash equilibrium, we don't need any new algorithms. The strategies computed through backward induction will be subgame perfect Nash equilibria, and it follows that every extensive-form game of perfect information has a subgame perfect Nash equilibrium, which can be computed in time polynomial in the size of the game tree.

Chance and simultaneous moves

To represent stochastic games, such as backgammon, in extensive form, we add a player called *Chance*, whose choices are determined by a probability distribution.

To represent simultaneous moves, as in the prisoner's dilemma or two-finger Morra, we impose an arbitrary order on the players, but we have the option of asserting that the earlier player's actions are not observable to the subsequent players: e.g., Ali must choose *refuse* or *testify* first, then Bo chooses, but Bo does not know what choice Ali made at that time (we can also represent the fact that the move is revealed later). However, we assume the players always remember all their *own* previous actions; this assumption is called **perfect recall**.

Capturing imperfect information

A key feature of extensive form that sets it apart from the game trees that we saw in Chapter 6 is that it can capture partial observability. Game theorists use the term **imperfect information** to describe situations where players are uncertain about the actual state of the game. Unfortunately, backward induction does not work with games of imperfect information, and in general, they are considerably more complex to solve than games of perfect information.

We saw in Section 6.6 that a player in a partially observable game such as Kriegspiel can create a game tree over the space of **belief states**. With that tree, we saw that in some cases a player can find a sequence of moves (a strategy) that leads to a forced checkmate regardless of what actual state we started in, and regardless of what strategy the opponent uses. However, the techniques of Chapter 6 could not tell a player what to do when there is no guaranteed checkmate. If the player's best strategy depends on the opponent's strategy and vice versa, then minimax (or alpha–beta) by itself cannot find a solution. The extensive form *does* allow us to find solutions because it represents the belief states (game theorists call them **information sets**) of *all* players at once. From that representation we can find equilibrium solutions, just as we did with normal-form games.

As a simple example of a sequential game, place two agents in the 4 × 3 world of Figure 16.1 and have them move simultaneously until one agent reaches an exit square and gets the payoff for that square. If we specify that no movement occurs when the two agents try to move into the same square simultaneously (a common problem at many traffic intersections), then certain pure strategies can get stuck forever. Thus, agents need a mixed strategy to perform well in this game: randomly choose between moving ahead and staying put. This is exactly what is done to resolve packet collisions in Ethernet networks.

Next we'll consider a very simple variant of poker. The deck has only four cards, two aces and two kings. One card is dealt to each player. The first player then has the option to *raise* the stakes of the game from 1 point to 2, or to *check*. If player 1 checks, the game is over. If player 1 raises, then player 2 has the option to *call,* accepting that the game is worth 2 points, or *fold,* conceding the 1 point. If the game does not end with a fold, then the payoff depends on the cards: it is zero for both players if they have the same card; otherwise the player with the king pays the stakes to the player with the ace.

The extensive-form tree for this game is shown in Figure 17.5. Player 0 is *Chance;* players 1 and 2 are depicted by triangles. Each action is depicted as an arrow with a label, corresponding to a *raise, check, call,* or *fold,* or, for *Chance,* the four possible deals ("AK" means that player 1 gets an ace and player 2 a king). Terminal states are rectangles labeled by their payoff to player 1 and player 2. Information sets are shown as labeled dashed boxes; for example, $I_{1,1}$ is the information set where it is player 1's turn, and he knows he has an ace (but does not know what player 2 has). In information set $I_{2,1}$, it is player 2's turn and she knows that she has an ace and that player 1 has raised, but does not know what card player 1 has. (Due to the limits of two-dimensional paper, this information set is shown as two boxes rather than one.)

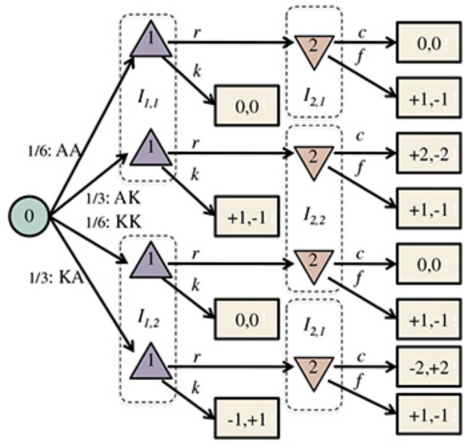


Figure 17.5 Extensive form of a simplified version of poker with two players and only four cards. The moves are r (raise), f (fold), c (call), and k (check).

One way to solve an extensive game is to convert it to a normal-form game. Recall that the normal form is a matrix, each row of which is labeled with a pure strategy for player 1, and each column by a pure strategy for player 2. In an extensive game a pure strategy for player *i* corresponds to an action for each information set involving that player. So in Figure 17.5, one pure strategy for player 1 is “raise when in $I_{1,1}$ (that is, when I have an ace), and check when in $I_{1,2}$ (when I have a king).” In the payoff matrix below, this strategy is called *rk*. Similarly, strategy *cf* for player 2 means “call when I have an ace and fold when I have a king.” Since this is a zero–sum game, the matrix below gives only the payoff for player 1; player 2 always has the opposite payoff:
This game is so simple that it has two pure-strategy equilibria, shown in bold: *cf* for player 2 and *rk* or *kk* for player 1. But in general we can solve extensive games by converting to normal form and then finding a solution (usually a mixed strategy) using standard linear programming methods. That works in theory. But if a player has I information sets and a actions per set, then that player will have a^I pure strategies. In other words, the size of the normal-form matrix is exponential in the number of information sets, so in practice the approach works only for tiny game trees—a dozen states or so. A game like two-player Texas hold ’em poker has about 10^{18} states, making this approach completely infeasible.

What are the alternatives? In Chapter 6 we saw how alpha–beta search could handle games of perfect information with huge game trees by generating the tree incrementally, by pruning some branches, and by heuristically evaluating nonterminal nodes. But that approach does not work well for games with imperfect information, for two reasons: first, it is harder to prune, because we need to consider mixed strategies that combine multiple branches, not a pure strategy that always chooses the best branch. Second, it is harder to heuristically evaluate a nonterminal node, because we are dealing with information sets, not individual states.

Koller *et al*. (1996) came to the rescue with an alternative representation of extensive games, called the **sequence form**, that is only linear in the size of the tree, rather than exponential. Rather than represent strategies, it represents paths through the tree; the number of paths is equal to the number of terminal nodes. Standard linear programming methods can again be applied to this representation. The resulting system can solve poker variants with 25,000 states in a minute or two. This is an exponential speedup over the normal-form approach, but still falls far short of handling, say, two-player Texas hold ’em, with 10^{18} states.

If we can't handle 10^{18} states, perhaps we can simplify the problem by changing the game to a simpler form. For example, if I hold an ace and am considering the possibility that the next card will give me a pair of aces, then I don't care about the suit of the next card; under the rules of poker any suit will do equally well. This suggests forming an **abstraction** of the game, one in which suits are ignored. The resulting game tree will be smaller by a factor of 4! = 24. Suppose I can solve this smaller game; how will the solution to that game relate to the original game? If no player is considering going for a flush (the only hand where the suits matter), then the solution for the abstraction will also be a solution for the original game. However, if any player is contemplating a flush, then the abstraction will be only an approximate solution (but it is possible to compute bounds on the error).

There are many opportunities for abstraction. For example, at the point in a game where each player has two cards, if I hold a pair of queens, then the other players' hands could be abstracted into three classes: *better* (only a pair of kings or a pair of aces), *same* (pair of queens) or *worse* (everything else). However, this abstraction might be too coarse. A better abstraction would divide *worse* into, say, *medium pair* (nines through jacks), *low pair,* and *no pair*. These examples are abstractions of states; it is also possible to abstract actions. For example, instead of having a bet action for each integer from 1 to 1000, we could restrict the bets to 10^0, 10^1, 10^2 and 10^3. Or we could cut out one of the rounds of betting altogether. We can also abstract over chance nodes, by considering only a subset of the possible deals. This is equivalent to the rollout technique used in Go programs. Putting all these abstractions together, we can reduce the 10^{18} states of poker to 10^7 states, a size that can be solved with current techniques.

We saw in Chapter 6 how poker programs such as Libratus and DeepStack were able to defeat champion human players at heads up (two-player) Texas hold 'em poker. More recently, the program Pluribus was able to defeat human champions at six-player poker in two formats: five copies of the program at the table with one human, and one copy of the program with five humans. There is a huge leap in complexity here. With one opponent, there are $\binom{50}{2 = 1225}$ possibilities for the opponent's hidden cards. But with five opponents there are $50\mathit{choose}10 \approx 10$ billion possibilities. Pluribus develops a baseline strategy entirely from self–play, then modifies the strategy during actual game play to react to a specific situation. Pluribus uses a combination of techniques, including Monte Carlo tree search, depth–limited search, and abstraction.

The extensive form is a versatile representation: it can handle partially observable, multiagent, stochastic, sequential, real-time environments—most of the hard cases from the list of environment properties on page 61. However, there are two limitations to the extensive form in particular and game theory in general. First, it does not deal well with continuous states and actions (although there have been some extensions to the continuous case; for example, the theory of **Cournot competition** uses game theory to solve problems where two companies choose prices for their products from a continuous space). Second, game theory assumes the game is *known*. Parts of the game may be specified as unobservable to some of the players, but it must be known what parts are unobservable. In cases in which the players learn the unknown structure of the game over time, the model begins to break down. Let's examine each source of uncertainty, and whether each can be represented in game theory.

Actions: There is no easy way to represent a game where the players have to discover what actions are available. Consider the game between computer virus writers and security experts. Part of the problem is anticipating what action the virus writers will try next.

Strategies: Game theory is very good at representing the idea that the other players' strategies are initially unknown—as long as we assume all agents are rational. The theory does not say what to do when the other players are less than fully rational. The notion of a **Bayes–Nash equilibrium** partially addresses this point: it is an equilibrium with respect to a player's prior probability distribution over the other players' strategies—in other words, it expresses a player's beliefs about the other players' likely strategies.

Chance: If a game depends on the roll of a die, it is easy enough to model a chance node with uniform distribution over the outcomes. But what if it is possible that the die is unfair? We can represent that with another chance node, higher up in the tree, with two branches for "die is fair" and "die is unfair," such that the corresponding nodes in each branch are in the same information set (that is, the players don't know if the die is fair or not). And what if we suspect the other opponent does know? Then we add *another* chance node, with one branch representing the case where the opponent does know, and one where the opponent doesn't.

Utilities: What if we don't know our opponent's utilities? Again, that can be modeled with a chance node, such that the other agent knows its own utilities in each branch, but we don't. But what if we don't know our *own* utilities? For example, how do I know if it is rational to order the chef's salad if I don't know how much I will like it? We can model that with yet another chance node specifying an unobservable "intrinsic quality" of the salad.

Thus, we see that game theory is good at representing most sources of uncertainty—but at the cost of doubling the size of the tree every time we add another node; a habit that quickly leads to intractably large trees. Because of these and other problems, game theory has been used primarily to *analyze* environments that are at equilibrium, rather than to *control* agents within an environment.

17.2.5 Uncertain payoffs and assistance games

In Chapter 1 (page 22), we noted the importance of designing AI systems that can operate under uncertainty about the true human objective. Chapter 15 (page 543) introduced a simple model for uncertainty about one's *own* preferences, using the example of durian-flavored ice cream. By the simple device of adding a new latent variable to the model to represent the unknown preferences, together with an appropriate sensor model (e.g., observing the taste of a small sample of the ice cream), uncertain preferences can be handled in a natural way.

Chapter 15 also studied the **off-switch problem**: we showed that a robot with uncertainty about human preferences will defer to the human and allow itself to be switched off. In that problem, Robbie the robot is uncertain about Harriet the human's preferences, but we model Harriet's decision (whether or not to switch Robbie off) as a simple, deterministic consequence of her own preferences for the action that Robbie proposes. Here, we generalize this idea into a full two-person game called an **assistance game**, in which both Harriet and Robbie are players. We assume that Harriet observes her own preferences θ and acts in accordance with them, while Robbie has a prior probability $P(\theta)$ over Harriet's preferences. The payoff is defined by θ and is identical for both players: both Harriet and Robbie are maximizing Harriet's payoff. In this way, the assistance game provides a formal model of the idea of provably beneficial AI introduced in Chapter 1.

In addition to the deferential behavior exhibited by Robbie in the off-switch problem—which is a restricted kind of assistance game—other behaviors that emerge as equilibrium strategies in general assistance games include actions on Harriet's part that we would describe as teaching, rewarding, commanding, correcting, demonstrating, or explaining, as well as actions on Robbie's part that we would describe as asking permission, learning from demonstrations, preference elicitation, and so on. The key point is that these behaviors need not be scripted: by solving the game, Harriet and Robbie work out for themselves how to convey preference information from Harriet to Robbie, so that Robbie can be more useful to Harriet. We need not stipulate in advance that Harriet is to "give rewards" or that Robbie is to "follow instructions," although these may be reasonable interpretations of how they end up behaving.

To illustrate assistance games, we'll use the **paperclip game**. It's a very simple game in which Harriet the human has an incentive to "signal" to Robbie the robot some information about her preferences. Robbie is able to interpret that signal because he can solve the game and therefore he can understand what would have to be true about Harriet's preferences in order for her to signal in that way.

The steps of the game are depicted in Figure 17.6. It involves making paperclips and staples. Harriet's preferences are expressed by a payoff function that depends on the number of paperclips and the number of staples produced, with a certain "exchange rate" between the two. Harriet's preference parameter θ denotes the relative value (in dollars) of a paperclip; for example, she might value paperclips at $\theta = 0.45$ dollars, which means staples are worth $1 - \theta = 0.55$ dollars. So, if p paperclips and s staples are produced, Harriet's payoff will be $p\theta + s(1 - \theta)$ dollars in all. Robbie's prior is $P(\theta) = Uniform(\theta;0,1)$. In the game itself, Harriet goes first, and can choose to make two paperclips, two staples, or one of each. Then Robbie can choose to make 90 paperclips, 90 staples, or 50 of each.

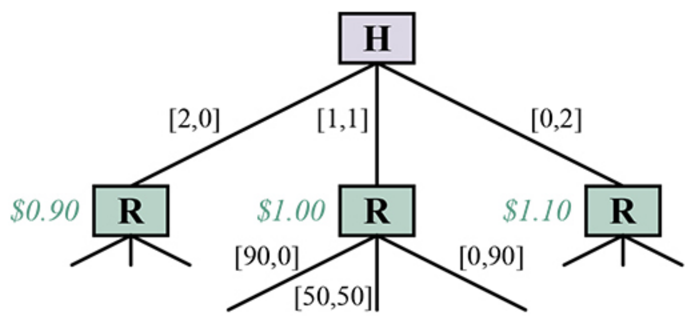


Figure 17.6 The paperclip game. Each branch is labeled [p, s] denoting the number of paperclips and staples manufactured on that branch. Harriet the human can choose to make two paperclips, two staples, or one of each. (The values in green italics are the values for Harriet if the game ended there, assuming $\theta = 0.45$.) Robbie the robot then has a choice to make 90 paperclips, 90 staples, or 50 of each.

Notice that if she were doing this by herself, Harriet would just make two staples, with a value of $1.10. (See the annotations at the first level of the tree in Figure 17.6.) But Robbie is watching, and he learns from her choice. What exactly does he learn? Well, that depends on how Harriet makes her choice. How does Harriet make her choice? That depends on how Robbie is going to interpret it. We can resolve this circularity by finding a Nash equilibrium. In this case, it is unique and can be found by applying myopic best response: pick any strategy for Harriet; pick the best strategy for Robbie, given Harriet's strategy; pick the best strategy for Harriet, given Robbie's strategy; and so on. The process unfolds as follows:

1. Start with the greedy strategy for Harriet: make two paperclips if she prefers paperclips; make one of each if she is indifferent; make two staples if she prefers staples.
2. There are three possibilities Robbie has to consider, given this strategy for Harriet:
    (a) If Robbie sees Harriet make two paperclips, he infers that she prefers paperclips, so he now believes the value of a paperclip is uniformly distributed between 0.5 and 1.0, with an average of 0.75. In that case, his best plan is to make 90 paperclips with an expected value of $67.50 for Harriet.
    (b) If Robbie sees Harriet make one of each, he infers that she values paperclips and staples at 0.50, so the best choice is to make 50 of each.
    (c) If Robbie sees Harriet make two staples, then by the same argument as in (a), he should make 90 staples.

3. Given this strategy for Robbie, Harriet's best strategy is now somewhat different from the greedy strategy in step 1. If Robbie is going to respond to her making one of each by making 50 of each, then she is better off making one of each not just if she is exactly indifferent, but if she is anywhere close to indifferent. In fact, the optimal policy is now to make one of each if she values paperclips anywhere between about 0.446 and 0.554.
4. Given this new strategy for Harriet, Robbie's strategy remains unchanged. For example, if she chooses one of each, he infers that the value of a paperclip is uniformly distributed between 0.446 and 0.554, with an average of 0.50, so the best choice is to make 50 of each. Because Robbie's strategy is the same as in step 2, Harriet's best response will be the same as in step 3, and we have found the equilibrium.

With her strategy, Harriet is, in effect, teaching Robbie about her preferences using a simple code—a language, if you like—that emerges from the equilibrium analysis. Note also that Robbie never learns Harriet's preferences exactly, but he learns enough to act optimally on her behalf—i.e., he acts just as he would if he did know her preferences exactly. He is provably beneficial to Harriet under the assumptions stated, and under the assumption that Harriet is playing the game correctly.

Myopic best response works for this example and others like it, but not for more complex cases. It is possible to prove that provided there are no ties that cause coordination problems, finding an optimal strategy profile for an assistance game is reducible to solving a POMDP whose state space is the underlying state space of the game plus the human preference parameters θ. POMDPs in general are very hard to solve (Section 16.5), but the POMDPs that represent assistance games have additional structure that enables more efficient algorithms.

Assistance games can be generalized to allow for multiple human participants, multiple robots, imperfectly rational humans, humans who don't know their own preferences, and so on. By providing a factored or structured action space, as opposed to the simple atomic actions in the paperclip game, the opportunities for communication can be greatly enhanced. Few of these variations have been explored so far, but we expect the key property of assistance games to remain true: the more intelligent the robot, the better the outcome for the human.

17.3 Cooperative Game Theory

Recall that cooperative games capture decision making scenarios in which agents can form binding agreements with one another to cooperate. They can then benefit from receiving extra value compared to what they would get by acting alone.

We start by introducing a model for a class of **ooperative games**. Formally, these games are called "cooperative games with transferable utility in characteristic function form." The idea of the model is that when a group of agents cooperate, the group as a whole obtains some utility value, which can then be split among the group members. The model does not say what actions the agents will take, nor does the game structure itself specify how the value obtained will be split up (that will come later).

Formally, we use the formula $G = (N, v)$ to say that a cooperative game, G, is defined by a set of players $N = \{1,...,n\}$ and a **characteristic function**, v, which for every subset of players $C \subseteq N$ gives the value that the group of players could obtain, should they choose to work together.

Typically, we assume that the empty set of players achieves nothing ($v(\{\ \}) = 0$), and that the function is nonnegative ($v(C) \geq 0$ for all C). In some games we make the further assumption that players achieve nothing by working alone: $v(\{i\}) = 0$ for all $i \in N$.

17.3.1 Coalition structures and outcomes

It is conventional to refer to a subset of players C as a **coalition**. In everyday use the term "coalition" implies a collection of people with some common cause (such as the Coalition to Stop Gun Violence), but we will refer to *any* subset of players as a coalition. The set of all players N is known as the **grand coalition**.

In our model, every player must choose to join exactly one coalition (which could be a coalition of just the single player alone). Thus, the coalitions form a

partition of the set of players. We call the partition a **coalition structure.** Formally, a coalition structure over a set of players N is a set of coalitions $\{C_1,...,C_k\}$ such that:

$$C_i \neq \{\}$$
$$C_i \subseteq N$$
$$C_i \cap C_j = \{\} \text{ for all } i \neq j \in N$$
$$C_1 \cup \cdots \cup C_k = N.$$

For example, if we have $N = \{1,2,3\}$, then there are seven possible coalitions:

$$\{1\}, \{2\}, \{3\}, \{1,2\}, \{2,3\}, \{3,1\}, \text{and } \{1,2,3\}$$

and five possible coalition structures:

$$\{\{1\}, \{2\}, \{3\}\}, \{\{1\}, \{2,3\}\}, \{\{2\},\{1,3\}\}, \{\{3\}, \{1,2\}\}, \text{and } \{\{1,2,3\}\}.$$

We use the notation **CS**(N) to denote the set of all coalition structures over player set N, and $CS(i)$ to denote the coalition that player i belongs to.

The **outcome** of a game is defined by the choices the players make, in deciding which coalitions to form, and in choosing how to divide up the $v(C)$ value that each coalition receives. Formally, given a cooperative game defined by (N,v), the outcome is a pair (CS, **x**) consisting of a coalition structure and a **payoff vector x** = $(x_1,...,x_n)$ where x_i is the value that goes to player i. The payoff must satisfy the constraint that each coalition C splits up all of its value $v(C)$ among its members:

$$\sum_{i \in C} x_i = v(C) \qquad \text{for all } C \in CS$$

For example, given the game $(\{1,2,3\}, v)$ where $v(\{1\}) = 4$ and $v(\{2,3\}) = 10$, a possible outcome is:

$$(\{\{1\},\{2,3\}\}, (4,5,5)).$$

That is, player 1 stays alone and accepts a value of 4, while players 2 and 3 team up to receive a value of 10, which they choose to split evenly.

Some cooperative games have the feature that when two coalitions merge together, they do no worse than if they had stayed apart. This property is called **superadditivity**. Formally, a game is superadditive if its characteristic function satisfies the following condition:

$$v(C \cup D) \geq v(C) + v(D) \qquad \text{for all } C, D \subseteq N$$

If a game is superadditive, then the grand coalition receives a value that is at least as high as or higher than the total received by any other coalition structure. However, as we will see shortly, superadditive games do not always end up with a grand coalition, for much the same reason that the players do not always arrive at a collectively desirable Pareto–optimal outcome in the prisoner's dilemma.

17.3.2 Strategy in cooperative games

The basic assumption in cooperative game theory is that players will make strategic decisions about who they will cooperate with. Intuitively, players will not desire to work with unproductive players—they will naturally seek out players that collectively yield a high coalitional value. But these sought-after players will be doing their own strategic reasoning. Before we can describe this reasoning, we need some further definitions.

An **imputation** for a cooperative game (N, v) is a payoff vector that satisfies the following two conditions:

$$\textstyle\sum_{i=1}^{n} x_i = v(N)$$

$$x_i \geq v(\{i\}) \text{ for all } i \in N.$$

The first condition says that an imputation must distribute the total value of the grand coalition; the second condition, known as **individual rationality**, says that each player is at least as well off as if it had worked alone.

Given an imputation $\mathbf{x} = (x_1,\ldots,x_n)$ and a coalition $C \subseteq N$, we define $x(C)$ to be the sum $\sum_{i \in C} x_i$—the total amount disbursed to C by the imputation $\mathbf{x}$.

Next, we define the **core** of a game (N, v) as the set of all imputations $\mathbf{x}$ that satisfy the condition $x(C) \geq v(C)$ for every possible coalition $C \subset N$. Thus, if an imputation **x** is *not* in the core, then there exists some coalition $C \subset N$ such that $v(C) > x(C)$. The players in C would refuse to join the grand coalition because they would be better off sticking with C.

The core of a game therefore consists of all the possible payoff vectors that no coalition could object to on the grounds that they could do better by not joining the grand coalition. Thus, if the core is empty, then the grand coalition cannot form, because no matter how the grand coalition divided its payoff, some smaller coalition would refuse to join. The main computational questions around the core relate to whether or not it is empty, and whether a particular payoff distribution is in the core.

The definition of the core naturally leads to a system of linear inequalities, as follows (the unknowns are variables $x_1,\ldots, x_n$, and the values $v(C)$ are constants):

$$x_i \geq v(\{i\}) \text{ for all } i \in N$$
$$\sum_{i \in N} x_i = v(N)$$
$$\sum_{i \in C} x_i \geq v(N) \text{ for all } C \subseteq N$$

Any solution to these inequalities will define an imputation in the core. We can formulate the inequalities as a linear program by using a dummy objective function (for example, maximizing $\sum_{i \in N} x_i$), which will allow us to compute imputations in time polynomial in the number of inequalities. The difficulty is that this gives an exponential number of inequalities (one for each of the 2^n possible coalitions). Thus, this approach yields an algorithm for checking non-emptiness of the core that runs in exponential time. Whether we can do better than this depends on the game being studied: for many classes of cooperative game, the problem of checking non-emptiness of the core is co-NP-complete. We give an example below.

Before proceeding, let's see an example of a superadditive game with an empty core. The game has three players $N = \{1,2,3\}$, and has a characteristic function defined as follows:

$$v(C) = \begin{cases} 1 \text{ if } |C| \geq 2 \\ 0 \text{ otherwise.} \end{cases}$$

Now consider any imputation (x_1, x_2, x_3) for this game. Since $v(N) = 1$, it must be the case that at least one player i has $x_i > 0$, and the other two get a total payoff less than 1. Those two could benefit by forming a coalition without player i and sharing the value 1 among themselves. But since this holds for all imputations, the core must be empty.

The core formalizes the idea of the grand coalition being *stable*, in the sense that no coalition can profitably defect from it. However, the core may contain imputations that are *unreasonable*, in the sense that one or more players might feel they were unfair. Suppose $N = \{1,2\}$, and we have a characteristic function v defined as follows:

$$v(\{1\}) = v(\{2\}) = 5$$
$$v(\{1,2\}) = 20.$$

Here, cooperation yields a surplus of 10 over what players could obtain working in isolation, and so intuitively, cooperation will make sense in this scenario. Now, it is easy to see that the imputation (6,14) is in the core of this game: neither player can deviate to obtain a higher utility. But from the point of view of player 1, this might appear unreasonable, because it gives 9/10 of the surplus to player 2. Thus, the notion of the core tells us when a grand coalition can form, but it does not tell us how to distribute the payoff.

The **Shapley value** is an elegant proposal for how to divide the $v(N)$ value among the players, given that the grand coalition N formed. Formulated by Nobel laureate Lloyd Shapley in the early 1950s, the Shapley value is intended to be a *fair* distribution scheme.

What does *fair* mean? It would be unfair to distribute $v(N)$ based on the eye color of players, or their gender, or skin color. Students often suggest that the value $v(N)$ should be divided equally, which seems like it might be fair, until we consider that this would give the same reward to players that contribute a lot and players that contribute nothing. Shapley's insight was to suggest that the only fair way to divide the value $v(N)$ was to do so according to how much each player *contributed* to creating the value $v(N)$.

First we need to define the notion of a player's **marginal contribution**. The marginal contribution that a player i makes to a coalition C is the value that i would add (or remove), should i join the coalition C. Formally, the marginal contribution that player i makes to C is denoted by $mc_i(C)$:

$$mc_i(C) \;=\; v(C \cup \{i\}) - v(C).$$

Now, a first attempt to define a payoff division scheme in line with Shapley's suggestion that players should be rewarded according to their contribution would be to pay each player i the value that they would add to the coalition containing all other players:

$$mc_i(N - \{i\}).$$

The problem is that this implicitly assumes that player i is the *last* player to enter the coalition. So, Shapley suggested, we need to consider all possible ways that the grand coalition could form, that is, all possible orderings of the players N, and consider the value that i adds to the preceding players in the ordering. Then, a player should be rewarded by being paid *the average marginal contribution that player i makes, over all possible orderings of the players, to the set of players preceding i in the ordering*.

We let P denote all possible permutations (e.g., orderings) of the players N, and denote members of P by p, p',... etc. Where $P \in P$ and $i \in N$, we denote by p_i the set of players that precede i in the ordering p. Then the Shapley value for a game G is the imputation $\phi(G) = (\phi_1(G), \ldots, \phi_n(G))$ defined as follows:

$$\phi_i(G) = \tfrac{1}{n!} \sum_{p \in P} mc_i(p_i). \qquad (17.1)$$

This should convince you that the Shapley value is a reasonable proposal. But the remarkable fact is that it is the *unique* solution to a set of axioms that characterizes a "fair" payoff distribution scheme. We'll need some more definitions before defining the axioms.

We define a **dummy player** as a player i that never adds any value to a coalition–that is, $mc_i(C) = 0$ for all $C \subseteq N - \{i\}$. We will say that two players i and j are **symmetric players** if they always make *identical* contributions to coalitions–that is, $mc_i(C) = mc_j(C)$ for all $C \subseteq N - \{i, j\}$. Finally, where $G = (N, v)$ and $G' = (N, v')$ are games with the same set of players, the game $G + G'$ is the game with the same player set, and a characteristic function v'' defined by $v''(C) = v(C) + v'(C)$.

Given these definitions, we can define the fairness axioms satisfied by the Shapley value:

- *Efficiency:* $\sum_{i \in N} \phi_i(G) = v(N)$. (All the value should be distributed.)
- *Dummy Player:* If i is a dummy player in G then $\phi_i(G) = 0$. (Players who never contribute anything should never receive anything.)
- *Symmetry:* If i and j are symmetric in G then $\phi_i(G) = \phi_j(G)$. (Players who make identical contributions should receive identical payoffs.)
- *Additivity:* The value is additive over games: For all games $G = (N, v)$ and $G' = (N, v')$, and for all players $i \in N$, we have $\phi_i(G + G') = \phi_i(G) + \phi_i(G) + \phi_i(G')$.

The additivity axiom is admittedly rather technical. If we accept it as a requirement, however, we can establish the following key property: *the Shapley value is the* only *way to distribute coalitional value so as to satisfy these fairness axioms*.

17.3.3 Computation in cooperative games

From a theoretical point of view, we now have a satisfactory solution. But from a computational point of view, we need to know how to *compactly represent* cooperative games, and how to *efficiently compute* solution concepts such as the core and the Shapley value.

The obvious representation for a characteristic function would be a table listing the value $v(C)$ for all 2^n coalitions. This is infeasible for large n. A number of approaches to compactly representing cooperative games have been developed, which can be distinguished by whether or not they are *complete*. A complete representation scheme is one that is capable of representing *any* cooperative game. The drawback with complete representation schemes is that there will always be some games that cannot be represented compactly. An alternative is to use a representation scheme that is guaranteed to be compact, but which is not complete.

Marginal contribution nets

We now describe one representation scheme, called **marginal contribution nets** (MC-nets). We will use a slightly simplified version to facilitate presentation, and the simplification makes it incomplete—the full version of MC-nets is a complete representation.

The idea behind marginal contribution nets is to represent the characteristic function of a game (N, v) as a set of coalition-value rules, of the form: (C_i, x_i), where $C_i \subseteq N$ is a coalition and x_i is a number. To compute the value of a coalition C, we simply sum the values of all rules (C_i, x_i) such that $C_i \subseteq C$. Thus, given a set of rules $R = \{(C_1, x_1),...,C_k, x_k)\}$, the corresponding characteristic function is:

$$v(C) = \sum\{x_i \mid (C_i, x_i) \in R \text{ and } C_i \subseteq C\}.$$

Suppose we have a rule set R containing the following three rules:

$$\{(\{1,2\},5), \quad (\{2\},2), \quad (\{3\},4)\}.$$

Then, for example, we have:

- $v(\{1\}) = 0$ (because no rules apply),
- $v(\{3\}) = 4$ (third rule),
- $v(\{1,3\}) = 4$ (third rule),
- $v(\{2,3\}) = 6$ (second and third rules), and

- $v(\{1,2,3\}) = 11$ (first, second, and third rules).

With this representation we can compute the Shapley value in polynomial time. The key insight is that each rule can be understood as defining a game on its own, in which the players are symmetric. By appealing to Shapley's axioms of additivity and symmetry, therefore, the Shapley value $\phi_i(R)$ of player i in the game associated with the rule set R is then simply:

$$\phi_i(R) = \sum_{(C,x)\in R} \begin{cases} \frac{x}{|C|} & \text{if } i \in C \\ 0 & \text{otherwise.} \end{cases}$$

The version of marginal contribution nets that we have presented here is not a *complete* representation scheme: there are games whose characteristic function cannot be represented using rule sets of the form described above. A richer type of marginal contribution networks allows for rules of the form (ϕ, x), where ø is a propositional logic formula over the players N: a coalition C satisfies the condition ø if it corresponds to a satisfying assignment for ø. This scheme is a complete representation—in the worst case, we need a rule for every possible coalition. Moreover, the Shapley value can be computed in polynomial time with this scheme; the details are more involved than for the simple rules described above, although the basic principle is the same; see the notes at the end of the chapter for references.

Coalition structures for maximum social welfare

We obtain a different perspective on cooperative games if we assume that the agents share a common purpose. For example, if we think of the agents as being workers in a company, then the strategic considerations relating to coalition formation that are addressed by the core, for example, are not relevant. Instead, we might want to organize the workforce (the agents) into teams so as to maximize their overall productivity. More generally, the task is to find a coalition that maximizes the *social welfare* of the system, defined as the sum of the values of the individual coalitions. We write the social welfare of a coalition structure CS as $sw(CS)$, with the following definition:

$$sw(CS) = \sum_{C \in CS} v(C).$$

Then a socially optimal coalition structure CS^* with respect to G maximizes this quantity. Finding a socially optimal coalition structure is a very natural computational problem, which has been studied beyond the multiagent systems community: it is sometimes called the **set partitioning problem**. Unfortunately, the problem is NP–hard, because the number of possible coalition structures grows exponentially in the number of players.

Finding the optimal coalition structure by naive exhaustive search is therefore infeasible in general. An influential approach to optimal coalition structure formation is based on the idea of searching a subspace of the **coalition structure graph**. The idea is best explained with reference to an example.

Suppose we have a game with four agents, $N = \{1,2,3,4\}$. There are fifteen possible coalition structures for this set of agents. We can organize these into a coalition structure graph as shown in Figure 17.7, where the nodes at level ℓ of the graph correspond to all the coalition structures with exactly ℓ coalitions. An upward edge in the graph represents the division of a coalition in the lower node into two separate coalitions in the upper node. For example, there is an edge from $\{\{1\}, \{2,3,4\}\}$ to $\{\{1\}, \{2\}, \{3,4\}\}$ because this latter coalition structure is obtained from the former by dividing the coalition $\{2,3,4\}$ into the coalitions $\{2\}$ and $\{3,4\}$.

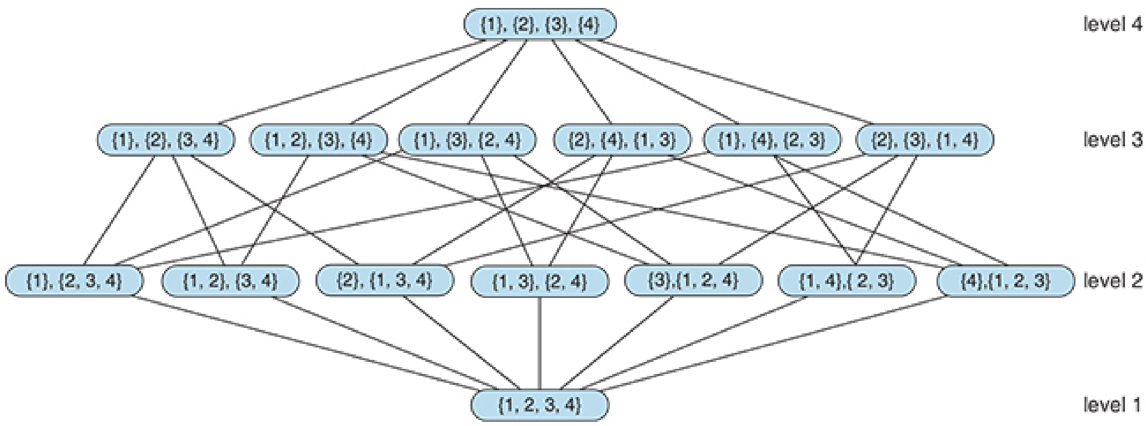


Figure 17.7 The coalition structure graph for $N = \{1,2,3,4\}$. Level 1 has coalition structures containing a single coalition; level 2 has coalition structures containing two coalitions, and so on.

The optimal coalition structure CS^* lies somewhere within the coalition structure graph, and so to find this, it seems we would have to evaluate every node in the graph. But consider the bottom two rows of the graph—levels 1 and 2. Every possible coalition (excluding the empty coalition) appears in these two levels. (Of course, not every possible coalition structure appears in these two levels.) Now, suppose we restrict our search for a possible coalition structure to *just* these two levels—we go no higher in the graph. Let CS' be the best coalition structure that we find in these two levels, and let CS^* be the best coalition structure overall. Let C^* be a coalition with the highest value of all possible coalitions:

$$C^* \in \arg\max_{C \subseteq N} v(C).$$

The value of the best coalition structure we find in the first two levels of the coalition structure graph must be at least as much as the value of the best possible coalition: $sw(CS') \geq v(C^*)$. This is because every possible coalition appears in at least one coalition structure in the first two levels of the graph. So assume the worst case, that is, $sw(CS') = v(C^*)$.

Compare the value of $sw(CS')$ to $sw(CS^*)$. Since $sw(CS')$ is the highest possible value of any coalition structure, and there are n agents ($n = 4$ in the case of Figure 17.7), then the highest possible value of $sw(CS^*)$ would be $nv(C^*) = n \cdot sw(CS')$. In other words, in the worst possible case, the value of the best coalition structure we find in the first two levels of the graph would be $\frac{1}{n}$ the value of the best, where n is the number of agents. Thus, although searching the first two levels of the graph does not guarantee to give us the *optimal* coalition structure, it *does* guarantee to give us one that is no worse that $\frac{1}{n}$ of the optimal. In practice it will often be much better than that.

17.4 Making Collective Decisions

We will now turn from agent design to **mechanism design**—the problem of designing the right game for a collection of agents to play. Formally, a **mechanism** consists of

1. A language for describing the set of allowable strategies that agents may adopt.
2. A distinguished agent, called the **center**, that collects reports of strategy choices from the agents in the game. (For example, the auctioneer is the center in an auction.)
3. An outcome rule, known to all agents, that the center uses to determine the payoffs to each agent, given their strategy choices.

This section discusses some of the most important mechanisms.

17.4.1 Allocating tasks with the contract net

The **contract net protocol** is probably the oldest and most important multiagent problemsolving technique studied in AI. It is a high-level protocol for task sharing. As the name suggests, the contract net was inspired from the way that companies make use of contracts.

The overall contract net protocol has four main phases—see Figure 17.8. The process starts with an agent identifying the need for cooperative action with respect to some task. The need might arise because the agent does not have the capability to carry out the task in isolation, or because a cooperative solution might in some way be better (faster, more efficient, more accurate).

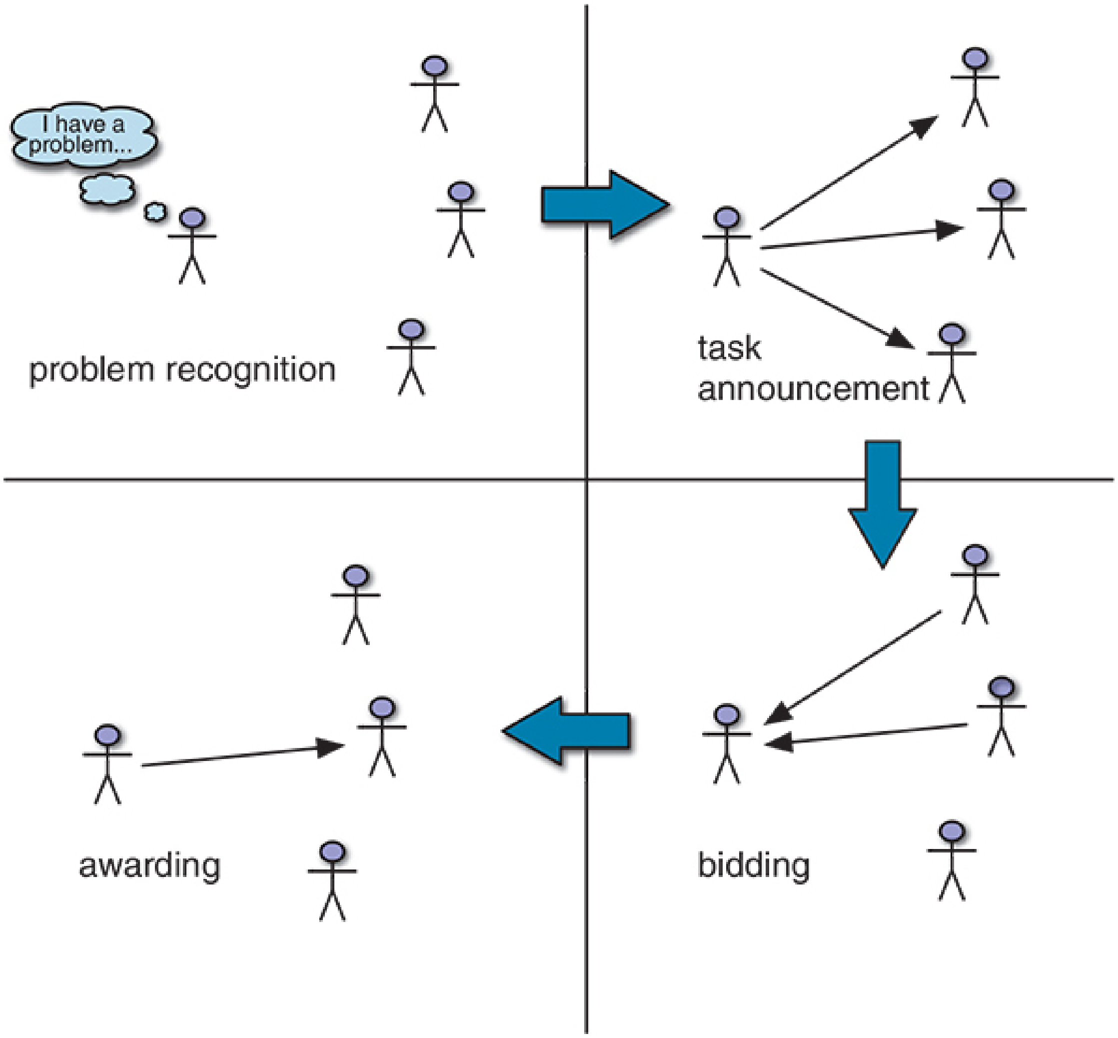


Figure 17.8 The contract net task allocation protocol.

The agent advertises the task to other agents in the net with a **task announcement** message, and then acts as the **manager** of that task for its duration. The task announcement message must include sufficient information for recipients to judge whether or not they are willing and able to bid for the task. The precise information included in a task announcement will depend on the application area. It might be some code that needs to be executed; or it might be a logical specification of a goal to be achieved. The task announcement might also include other information that might be required by recipients, such as deadlines, quality-of-service requirements, and so on.

When an agent receives a task announcement, it must evaluate it with respect to its own capabilities and preferences. In particular, each agent must determine, whether it has the capability to carry out the task, and second, whether or not it desires to do so. On this basis, it may then submit a **bid** for the task. A bid will typically indicate the capabilities of the bidder that are relevant to the advertised task, and any terms and conditions under which the task will be carried out.

In general, a manager may receive multiple bids in response to a single task announcement. Based on the information in the bids, the manager selects the most appropriate agent (or agents) to execute the task. Successful agents are notified through an award message, and become contractors for the task, taking responsibility for the task until it is completed.

The main computational tasks required to implement the contract net protocol can be summarized as follows:

- *Task announcement processing*. On receipt of a task announcement, an agent decides if it wishes to bid for the advertised task.
- *Bid processing*. On receiving multiple bids, the manager must decide which agent to award the task to, and then award the task.
- *Award processing*. Successful bidders (contractors) must attempt to carry out the task, which may mean generating new subtasks, which are advertised via further task announcements.

Despite (or perhaps because of) its simplicity, the contract net is probably the most widely implemented and best-studied framework for cooperative problem solving. It is naturally applicable in many settings—a variation of it is enacted every time you request a car with Uber, for example.

17.4.2 Allocating scarce resources with auctions

One of the most important problems in multiagent systems is that of allocating scarce resources; but we may as well simply say "allocating resources," since in practice most useful resources are scarce in some sense. The **auction** is the most important mechanism for allocating resources. The simplest setting for an auction is where there is a single resource and there are multiple possible **bidders**. Each bidder i has a utility value v_i for the item.

In some cases, each bidder has a **private value** for the item. For example, a tacky sweater might be attractive to one bidder and valueless to another.

In other cases, such as auctioning drilling rights for an oil tract, the item has a **common value**—the tract will produce some amount of money, X, and all bidders value a dollar equally—but there is uncertainty as to what the actual value of X is. Different bidders have different information, and hence different estimates of the item's true value. In either case, bidders end up with their own v_i. Given v_i, each bidder gets a chance, at the appropriate time or times in the auction, to make a bid b_i. The highest bid, b_{max}, wins the item, but the price paid need not be b_{max}; that's part of the mechanism design.

The best-known auction mechanism is the **ascending–bid auction**,[3] or **English auction**, in which the center starts by asking for a minimum (or **reserve**) bid b_{min}. If some bidder is willing to pay that amount, the center then asks for $b_{min} + d$, for some increment d, and continues up from there. The auction ends when nobody is willing to bid anymore; then the last bidder wins the item, paying the price bid.

How do we know if this is a good mechanism? One goal is to maximize expected revenue for the seller. Another goal is to maximize a notion of global utility. These goals overlap to some extent, because one aspect of maximizing global utility is to ensure that the winner of the auction is the agent who values the item the most (and thus is willing to pay the most). We say an auction is **efficient** if the goods go to the agent who values them most. The ascending– bid auction is usually both efficient and revenue maximizing, but if the reserve price is set too high, the bidder who values it most may not bid, and if the reserve is set too low, the seller may get less revenue.

Probably the most important things that an auction mechanism can do is encourage a sufficient number of bidders to enter the game and discourage them from engaging in **collusion**. Collusion is an unfair or illegal agreement by two or more bidders to manipulate prices. It can happen in secret backroom deals or tacitly, within the rules of the mechanism. For example, in 1999, Germany auctioned ten blocks of cellphone spectrum with a simultaneous auction (bids were taken on all ten blocks at the same time), using the rule that any bid must be a minimum of a 10% raise over the previous bid on a block. There were only two credible bidders, and the first, Mannesman, entered the bid of 20 million deutschmark on blocks 1-5 and 18.18 million on blocks 6-10. Why 18.18M? One of T-Mobile’s managers said they “interpreted Mannesman’s first bid as an offer.” Both parties could compute that a 10% raise on 18.18M is 19.99M; thus Mannesman’s bid was interpreted as saying “we can each get half the blocks for 20M; let’s not spoil it by bidding the prices up higher.” And in fact T-Mobile bid 20M on blocks 6-10 and that was the end of the bidding.

The German government got less than they expected, because the two competitors were able to use the bidding mechanism to come to a tacit agreement on how not to compete. From the government’s point of view, a better result could have been obtained by any of these changes to the mechanism: a higher reserve price; a sealed-bid first-price auction, so that the competitors could not communicate through their bids; or incentives to bring in a third bidder. Perhaps the 10% rule was an error in mechanism design, because it facilitated the precise signaling from Mannesman to T-Mobile.

In general, both the seller and the global utility function benefit if there are more bidders, although global utility can suffer if you count the cost of wasted time of bidders that have no chance of winning. One way to encourage more bidders is to make the mechanism easier for them. After all, if it requires too much research or computation on the part of the bidders, they may decide to take their money elsewhere.

So it is desirable that the bidders have a **dominant strategy**. Recall that “dominant” means that the strategy works against all other strategies, which in turn means that an agent can adopt it without regard for the other strategies. An agent with a dominant strategy can just bid, without wasting time contemplating other agents’ possible strategies. A mechanism by which agents have a dominant strategy is called a **strategy-proof** mechanism. If, as is usually the case, that strategy involves the bidders revealing their true value, v_i, then it is called a **truth-revealing**, or **truthful**, auction; the term **incentive compatible** is also used. The **revelation principle** states that any mechanism can be transformed into an equivalent truth-revealing mechanism, so part of mechanism design is finding these equivalent mechanisms.

It turns out that the ascending-bid auction has most of the desirable properties. The bidder with the highest value v_i gets the goods at a price of $b_o + d$, where b_o is the highest bid among all the other agents and d is the auctioneer's increment.[4] Bidders have a simple dominant strategy: keep bidding as long as the current cost is below your v_i. The mechanism is not quite truth-revealing, because the winning bidder reveals only that his $v_i \geq b_o + d$; we have a lower bound on v_i but not an exact amount.

A disadvantage (from the point of view of the seller) of the ascending–bid auction is that it can discourage competition. Suppose that in a bid for cellphone spectrum there is one advantaged company that everyone agrees would be able to leverage existing customers and infrastructure, and thus can make a larger profit than anyone else. Potential competitors can see that they have no chance in an ascending-bid auction, because the advantaged company can always bid higher. Thus, the competitors may not enter at all, and the advantaged company ends up winning at the reserve price.

Another negative property of the English auction is its high communication costs. Either the auction takes place in one room or all bidders have to have high-speed, secure communication lines; in either case they have to have time to go through several rounds of bidding.

An alternative mechanism, which requires much less communication, is the **sealed-bid auction**. Each bidder makes a single bid and communicates it to the auctioneer, without the other bidders seeing it. With this mechanism, there is no longer a simple dominant strategy. If your value is v_i, and you believe that the maximum of all the other agents' bids will be b_o, then you should bid $b_o + \epsilon$, for some small ϵ, if that is less than v_i,. Thus, your bid depends on your estimation of the other agents' bids, requiring you to do more work. Also, note that the agent with the highest v_i, might not win the auction. This is offset by the fact that the auction is more competitive, reducing the bias toward an advantaged bidder.

A small change in the mechanism for sealed-bid auctions leads to the **sealed-bid second-price auction**, also known as a **Vickrey auction**.[5] In such auctions, the winner pays the price of the *second*-highest bid, b_o, rather than paying his own bid. This simple modification completely eliminates the complex deliberations required for standard (or **first-price**) sealed-bid auctions, because the dominant strategy is now simply to bid v_i; the mechanism is truth-revealing. Note that the utility of agent i in terms of his bid b_i, his value v_i, and the best bid among the other agents, b_o, is

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

To see that $b_i = v_i$ is a dominant strategy, note that when $(v_i - b_o)$ is positive, any bid that wins the auction is optimal, and bidding v_i, in particular wins the auction. On the other hand, when $(v_i - b_o)$ is negative, any bid that loses the auction is optimal, and bidding v_i in particular loses the auction. So bidding v_i is optimal for all possible values of b_o, and in fact, v_i is the only bid that has this property. Because of its simplicity and the minimal computation requirements for both seller and bidders, the Vickrey auction is widely used in distributed AI systems.

Internet search engines conduct several trillion auctions each year to sell advertisements along with their search results, and online auction sites handle \$100 billion a year in goods, all using variants of the Vickrey auction. Note that the expected value to the seller is b_o, which is the same expected return as the limit of the English auction as the increment d goes to zero. This is actually a very general result: the **revenue equivalence theorem** states that, with a few minor caveats, any auction mechanism in which bidders have values v_i known only to themselves (but know the probability distribution from which those values are sampled), will yield the same expected revenue. This principle means that the various mechanisms are not competing on the basis of revenue generation, but rather on other qualities.

Although the second-price auction is truth-revealing, it turns out that auctioning n goods with an $n + 1$ price auction is not truth-revealing. Many Internet search engines use a mechanism where they auction n slots for ads on a page. The highest bidder wins the top spot, the second highest gets the second spot, and so on. Each winner pays the price bid by the next-lower bidder, with the understanding that payment is made only if the searcher actually clicks on the ad. The top slots are considered more valuable because they are more likely to be noticed and clicked on.

Imagine that three bidders, b_1, b_2 and b_3, have valuations for a click of $v_1 = 200$, $v_2 = 180$, and $v_3 = 100$, and that $n = 2$ slots are available; and it is known that the top spot is clicked on 5% of the time and the bottom spot 2%. If all bidders bid truthfully, then b_1 wins the top slot and pays 180, and has an expected return of $(200 - 180) \times 0.05 = 1$. The second slot goes to b_2. But b_1 can see that if she were to bid anything in the range 101–179, she would concede the top slot to b_2, win the second slot, and yield an expected return of $(200 - 100) \times .02 = 2$. Thus, b_1 can double her expected return by bidding less than her true value in this case.

In general, bidders in this $n + 1$ price auction must spend a lot of energy analyzing the bids of others to determine their best strategy; there is no simple dominant strategy.

Aggarwal *et al*. (2006) show that there is a unique truthful auction mechanism for this multislot problem, in which the winner of slot j pays the price for slot j just for those additional clicks that are available at slot j and not at slot $j + 1$. The winner pays the price for the lower slot for the remaining clicks. In our example, b_1 would bid 200 truthfully, and would pay 180 for the additional .05 – .02 = .03 clicks in the top slot, but would pay only the cost of the bottom slot, 100, for the remaining .02 clicks. Thus, the total return to b_1 would be $(200 - 180) \times .03 + (200 - 100) \times .02 = 2.6$.

Another example of where auctions can come into play within AI is when a collection of agents are deciding whether to cooperate on a joint plan. Hunsberger and Grosz (2000) show that this can be accomplished efficiently with an auction in which the agents bid for roles in the joint plan.

Common goods

Now let's consider another type of game, in which countries set their policy for controlling air pollution. Each country has a choice: they can reduce pollution at a cost of -10 points for implementing the necessary changes, or they can continue to pollute, which gives them a net utility of -5 (in added health costs, etc.) and also contributes -1 points to every other country (because the air is shared across countries). Clearly, the dominant strategy for each country is "continue to pollute," but if there are 100 countries and each follows this policy, then each country gets a total utility of -104, whereas if every country reduced pollution, they would each have a utility of -10. This situation is called the **tragedy of the commons**: if nobody has to pay for using a common resource, then it may be exploited in a way that leads to a lower total utility for all agents. It is similar to the prisoner's dilemma: there is another solution to the game that is better for all parties, but there appears to be no way for rational agents to arrive at that solution under the current game.

One approach for dealing with the tragedy of the commons is to change the mechanism to one that charges each agent for using the commons. More generally, we need to ensure that all **externalities**—effects on global utility that are not recognized in the individual agents' transactions—are made explicit.

Setting the prices correctly is the difficult part. In the limit, this approach amounts to creating a mechanism in which each agent is effectively required to maximize global utility, but can do so by making a local decision. For this example, a carbon tax would be an example of a mechanism that charges for use of the commons in a way that, if implemented well, maximizes global utility.

It turns out there is a mechanism design, known as the **Vickrey-Clarke-Groves** or VCG mechanism, which has two favorable properties. First, it is utility maximizing—that is, it maximizes the global utility, which is the sum of the utilities for all parties, $\sum_i v_i$. Second, the mechanism is truth-revealing—the dominant strategy for all agents is to reveal their true value. There is no need for them to engage in complicated strategic bidding calculations.

We will give an example using the problem of allocating some common goods. Suppose a city decides it wants to install some free wireless Internet transceivers. However, the number of transceivers available is less than the number of neighborhoods that want them. The city wants to maximize global utility, but if it says to each neighborhood council "How much do you value a free transceiver (and by the way we will give them to the parties that value them the most)?" then each neighborhood will have an incentive to report a very high value. The VCG mechanism discourages this ploy and gives them an incentive to report their true value. It works as follows:

1. The center asks each agent to report its value for an item, v_i.
2. The center allocates the goods to a set of winners, W, to maximize $\sum_{i \in W} v_i$.

3. The center calculates for each winning agent how much of a loss their individual presence in the game has caused to the losers (who each got 0 utility, but could have got v_j if they were a winner).
4. Each winning agent then pays to the center a tax equal to this loss.

For example, suppose there are 3 transceivers available and 5 bidders, who bid 100, 50, 40, 20, and 10. Thus the set of 3 winners, W, are the ones who bid 100, 50, and 40 and the global utility from allocating these goods is 190. For each winner, it is the case that had they not been in the game, the bid of 20 would have been a winner. Thus, each winner pays a tax of 20 to the center.

All winners should be happy because they pay a tax that is less than their value, and all losers are as happy as they can be, because they value the goods less than the required tax. That's why the mechanism is truth-revealing. In this example, the crucial value is 20; it would be irrational to bid above 20 if your true value was actually below 20, and vice versa. Since the crucial value could be anything (depending on the other bidders), that means that is always irrational to bid anything other than your true value.

The VCG mechanism is very general, and can be applied to all sorts of games, not just auctions, with a slight generalization of the mechanism described above. For example, in a **combinatorial auction** there are multiple different items available and each bidder can place multiple bids, each on a subset of the items. For example, in bidding on plots of land, one bidder might want either plot X or plot Y but not both; another might want any three adjacent plots, and so on. The VCG mechanism can be used to find the optimal outcome, although with 2^N subsets of N goods to contend with, the computation of the optimal outcome is NP– complete. With a few caveats the VCG mechanism is unique: every other optimal mechanism is essentially equivalent.

17.4.3 Voting

The next class of mechanisms that we look at are voting procedures, of the type that are used for political decision making in democratic societies. The study of voting procedures derives from the domain of **social choice theory**.

The basic setting is as follows. As usual, we have a set $N = \{1,..., n\}$ of agents, who in this section will be the voters. These voters want to make decisions with respect to a set $\Omega = \{\omega_1, \omega_2, \ldots\}$ of possible outcomes. In a political election, each element of Ω could stand for a different candidate winning the election.

Each voter will have preferences over Ω. These are usually expressed not as quantitative utilities but rather as qualitative comparisons: we write $\omega \succ_i \omega'$ to mean that outcome ω is ranked above outcome ω' by agent i. In an election with three candidates, agent i might have $\omega_2 \succ_i \omega_3 \succ_i \omega_1$.

The fundamental problem of social choice theory is to combine these preferences, using a **social welfare function**, to come up with a **social preference order**: a ranking of the candidates, from most preferred down to least preferred. In some cases, we are only interested in a **social outcome**—the most preferred outcome by the group as a whole. We will write $\omega \succ^* \omega'$ to mean that ω is ranked above ω' in the social preference order

A simpler setting is where we are not concerned with obtaining an entire ordering of candidates, but simply want to choose a set of winners. A **social choice function** takes as input a preference order for each voter, and produces as output a set of winners.

Democratic societies want a social outcome that reflects the preferences of the voters. Unfortunately, this is not always straightforward. Consider **Condorcet's Paradox**, a famous example posed by the Marquis de Condorcet (1743–1794). Suppose we have three outcomes,

$\Omega = \{\omega_a, \omega_b, \omega_c,\}$, and three voters, $N = \{1,2,3\}$, with preferences as follows.

$$\begin{aligned}
&\omega_a \succ_1 \omega_b \succ_1 \omega_c \\
&\omega_c \succ_2 \omega_a \succ_2 \omega_b \\
&\omega_b \succ_3 \omega_c \succ_3 \omega_a \qquad (17.2)
\end{aligned}$$

Now, suppose we have to choose one of the three candidates on the basis of these preferences. The paradox is that:

- 2/3 of the voters prefer ω_3 over ω_1.
- 2/3 of the voters prefer ω_1 over ω_2.
- 2/3 of the voters prefer ω_2 over ω_3.

So, for each possible winner, we can point to another candidate who would be preferred by at least 2/3 of the electorate. It is obvious that in a democracy we cannot hope to make *every* voter happy. This demonstrates that there are scenarios in which *no matter which outcome we choose, a majority of voters will prefer a different outcome*. A natural question is whether there is any "good" social choice procedure that really reflects the preferences of voters. To answer this, we need to be precise about what we mean when we say that a rule is "good." We will list some properties we would like a good social welfare function to satisfy:

- *The Pareto Condition:* The Pareto condition simply says that if every voter ranks ω_i above ω_j, then $\omega_i \succ^* \omega_j$.
- *The Condorcet Winner Condition:* An outcome is said to be a Condorcet winner if a majority of candidates prefer it over all other outcomes. To put it another way, a Condorcet winner is a candidate that would beat every other candidate in a pairwise election. The Condorcet winner condition says that if ω_i is a Condorcet winner, then ω_i should be ranked first.

- *Independence of Irrelevant Alternatives (IIA):* Suppose there are a number of candidates, including $\omega_{i;}$ and ω_j, and voter preferences are such that $\omega_i \succ^* \omega_j$. Now, suppose one voter changed their preferences in some way, but *not* about the relative ranking of ω_i and ω_j. The IIA condition says that, $\omega_i \succ^* \omega_j$ should not change.
- *No Dictatorships:* It should not be the case that the social welfare function simply outputs one voter's preferences and ignores all other voters.

These four conditions seem reasonable, but a fundamental theorem of social choice theory called **Arrow's theorem** (due to Kenneth Arrow) tells us that it is impossible to satisfy all four conditions (for cases where there are at least three outcomes). That means that for any social choice mechanism we might care to pick, there will be some situations (perhaps unusual or pathological) that lead to controversial outcomes. However, it does not mean that democratic decision making is hopeless in most cases. We have not yet seen any actual voting procedures, so let's now look at some.

- With just two candidates, **simple majority vote** (the standard method in the US and UK) is the favored mechanism. We ask each voter which of the two candidates they prefer, and the one with the most votes is the winner.
- With more than two outcomes, **plurality voting** is a common system. We ask each voter for their top choice, and select the candidate(s) (more than one in the case of ties) who get the most votes, even if nobody gets a majority. While it is common, plurality voting has been criticized for delivering unpopular outcomes. A key problem is that it only takes into account the top-ranked candidate in each voter's preferences.

- The **Borda count** (after Jean-Charles de Borda, a contemporary and rival of Condorcet) is a voting procedure that takes into account all the information in a voter's preference ordering. Suppose we have k candidates. Then for each voter i, we take their preference ordering $\succ_i$, and give a score of k to the top ranked candidate, a score of $k - 1$ to the second-ranked candidate, and so on down to the least-favored candidate in i's ordering. The total score for each candidate is their Borda count, and to obtain the social outcome $\succ^*$, outcomes are ordered by their Borda count—highest to lowest. One practical problem with this system is that it asks voters to express preferences on all the candidates, and some voters may only care about a subset of candidates.
- In **approval voting**, voters submit a subset of the candidates that they approve of. The winner(s) are those who are approved by the most voters. This system is often used when the task is to choose multiple winners.
- In **instant runoff voting**, voters rank all the candidates, and if a candidate has a majority of first-place votes, they are declared the winner. If not, the candidate with the fewest first-place votes is eliminated. That candidate is removed from all the preference rankings (so those voters who had the eliminated candidate as their first choice now have another candidate as their new first choice) and the process is repeated. Eventually, some candidate will have a majority of first-place votes (unless there is a tie).
- In **true majority rule voting**, the winner is the candidate who beats every other candidate in pairwise comparisons. Voters are asked for a full preference ranking of all candidates. We say that ω beats ω', if more voters have $\omega \succ \omega'$ than have $\omega' \succ \omega$. This system has the nice

property that the majority always agrees on the winner, but it has the bad property that not every election will be decided: in the Condorcet paradox, for example, no candidate wins a majority.

Strategic manipulation

Besides Arrow's Theorem, another important negative results in the area of social choice theory is the **Gibbard–Satterthwaite Theorem**. This result relates to the circumstances under which a voter can benefit from *misrepresenting their preferences*.

Recall that a social choice function takes as input a preference order for each voter, and gives as output a set of winning candidates. Each voter has, of course, their own true preferences, but there is nothing in the definition of a social choice function that requires voters to report their preferences *truthfully;* they can declare whatever preferences they like.

In some cases, it can make sense for a voter to misrepresent their preferences. For example, in plurality voting, voters who think their preferred candidate has no chance of winning may vote for their second choice instead. That means plurality voting is a game in which voters have to think strategically (about the other voters) to maximize their expected utility.

This raises an interesting question: can we design a voting mechanism that is immune to such manipulation—a mechanism that is truth-revealing? The Gibbard–Satterthwaite Theorem tells us that we can not: *Any social choice function that satisfies the Pareto condition for a domain with more than two outcomes is either manipulable or a dictatorship*. That is, for any "reasonable" social choice procedure, there will be some circumstances under which a voter can in principle benefit by misrepresenting their preferences. However, it does not tell us *how* such manipulation might be done; and it does not tell us that such manipulation is likely *in practice*.

17.4.4 Bargaining

Bargaining, or negotiation, is another mechanism that is used frequently in everyday life. It has been studied in game theory since the 1950s and more recently has become a task for automated agents. Bargaining is used when agents need to reach agreement on a matter of common interest. The agents make offers (also called proposals or deals) to each other under specific protocols, and either accept or reject each offer.

Bargaining with the alternating offers protocol

One influential bargaining protocol is the **alternating offers bargaining model**. For simplicity we'll again assume just two agents. Bargaining takes place in a sequence of rounds. A_1 begins, at round 0, by making an offer. If A_2 accepts the offer, then the offer is implemented. If A_2 rejects the offer, then negotiation moves to the next round. This time A_2 makes an offer and A_1 chooses to accept or reject it, and so on. If the negotiation never terminates (because agents reject every offer) then we define the outcome to be the **conflict deal**. A convenient simplifying assumption is that both agents prefer to reach an outcome—any outcome—in finite time rather than being stuck in the infinitely time-consuming conflict deal.

We will use the scenario of **dividing a pie** to illustrate alternating offers. The idea is that there is some resource (the "pie") whose value is 1, which can be divided into two parts, one part for each agent. Thus an offer in this scenario is a pair $(x, 1-x)$, where x is the amount of the pie that A_1 gets, and $1-x$ is the amount that A_2 gets. The space of possible deals (the **negotiation set**) is thus:

$$\{(x, 1-x) : 0 \le x \le 1\}.$$

Now, how should agents negotiate in this setting? To understand the answer to this question, we will first look at a few simpler cases.

First, suppose that we allow *just one round* to take place. Thus, A_1 makes a proposal; A_2 can either accept it (in which case the deal is implemented), or reject it (in which case the conflict deal is implemented). This is an **ultimatum game**. In this case, it turns out that A_1—the **first mover**—has all the power. Suppose that A_1 proposes to get all the pie, that is, proposes the deal (1,0). If A_2 rejects, then the conflict deal is implemented; since by definition A_2 would prefer to get 0 rather than the conflict deal, A_2 would be better off accepting. Of course, A_1 cannot do better than getting the whole pie. Thus, these two strategies—A_1 proposes to get the whole pie, and A_2 accepts—form a Nash equilibrium.

Now consider the case where we permit exactly *two* rounds of negotiation. Now the power has shifted: A_2 can simply reject the first offer, thereby turning the game into a one-round game in which A_2 is the first mover and thus will get the whole pie. In general, if the number of rounds is a fixed number, then whoever moves last will get all the pie.

Now let's move on to the general case, where there is *no* bound on the number of rounds. Suppose that A_1 uses the following strategy:

> Always propose (1,0), and always reject any counteroffer.

What is A_2's best response to this? If A_2 continually rejects the proposal, then the agents will negotiate forever, which by definition is the worst outcome for A_2 (as well as for A_1). So A_2 can do no better than accepting the first proposal that A_1 makes. Again, this is a Nash equilibrium. But what if A_1 uses the strategy:

> Always propose (0.8,0.2), and always reject any offer.

By a similar argument we can see that for this offer or *for any possible deal* $(x, 1 - x)$ *in the negotiation set, there is a Nash equilibrium pair of* *negotiation strategies such that the outcome will be agreement on the deal in the first time period*.

Impatient agents

This analysis tells us that if no constraints are placed on the number of rounds then there will be an infinite number of Nash equilibria. So let's add an assumption:

> For any outcome x and times t_1 and t_2, where $t_1 < t_2$, both agents would prefer outcome x at time t_1 over outcome x at time t_2.

In other words, agents are **impatient**. A standard approach to impatience is to use a **discount factor** γ_i (see page 555) for each agent ($0 \leq \gamma_i \leq 1$). Suppose that at some point in the negotiation agent i is offered a slice of the pie of size x. The value of the slice x at time t is $\gamma_i^t x$. Thus on the first negotiation step (time 0), the value is $\gamma_i^0 x = x$, and at any subsequent point in time the value of the same offer will be less. A larger value for γ_i (closer to 1) thus implies more patience; a smaller value means less patience.

To analyze the general case, let's first consider bargaining over fixed periods of time, as above. The 1-round case has the same analysis as given above: we simply have an ultimatum game. With *two* rounds the situation changes, because the value of the pie reduces in accordance with discount factors γ_i. Suppose A_2 rejects A_1's initial proposal. Then A_2 will get the whole pie with an ultimatum in the second round. But the *value* of that whole pie has reduced: it is only worth γ_2 to A_2. Agent A_1 can take this fact into account by offering $(1 - \gamma_2, \gamma_2)$, an offer that A_2 may as well accept because A_2 can do no better than γ_2 at this point in time. (If you are worried about what happens with ties, just make the offer be $(1 - (\gamma_2 + \epsilon), \gamma_2 + \epsilon)$ for some small value of ϵ.)

So, the two strategies of A_1 offering $(1 - \gamma_2, \gamma_2)$, and A_2 accepting that offer are in Nash equilibrium. Patient players (those with a larger γ_2) will be able to obtain larger pieces of the pie under this protocol: in this setting, patience truly is a virtue.

Now consider the general case, where there are no bounds on the number of rounds. As in the 1-round case, A_1 can craft a proposal that A_2 should accept, because it gives A_2 the maximal achievable amount, given the discount factors. It turns out that A_1 will get

$$\frac{1-\gamma 2}{1-\gamma 1\,\gamma 2}$$

and A_2 will get the remainder.

Negotiation in task-oriented domains

In this section, we consider negotiation for **task-oriented domains**. In such a domain, a set of tasks must be carried out, and each task is initially assigned to a set of agents. The agents may be able to benefit by negotiating on who will carry out which tasks. For example, suppose some tasks are done on a lathe machine and others on a milling machine, and that any agent using a machine must incur a significant setup cost. Then it would make sense for one agent to offer another "I have to set up on the milling machine anyway; how about if I do all your milling tasks, and you do all my lathe tasks?"

Unlike the bargaining scenario, we start with an initial allocation, so if the agents fail to agree on any offers, they perform the tasks T_i^0 that they were originally allocated.

To keep things simple, we will again assume just two agents. Let T be the set of all tasks and let $\left(T_1^0, T_2^0\right)$ denote the initial allocation of tasks to the two agents at time 0. Each task in T must be assigned to exactly one agent. We assume we have a cost function c, which for every set of tasks T'

gives a positive real number $c(T')$ indicating the cost to any agent of carrying out the tasks T'. (Assume the cost depends only on the tasks, not on the agent carrying out the task.) The cost function is monotonic—adding more tasks never reduces the cost–and the cost of doing nothing is zero: $c(\{\}) = 0$. As an example, suppose the cost of setting up the milling machine is 10 and each milling task costs 1, then the cost of a set of two milling tasks would be 12, and the cost of a set of five would be 15.

An offer of the form (T_1, T_2) means that agent i is committed to performing the set of tasks T_i, at cost $c(T_i)$. The utility to agent i is the amount they have to gain from accepting the offer–the difference between the cost of doing this new set of tasks versus the originally assigned set of tasks:

$$U_i((T_1, T_2)) = c(T_i) - c(T_i^0).$$

An offer (T_1, T_2) is **individually rational** if $U_i((T_1, T_2)) \geq 0$ for both agents. If a deal is not individually rational, then at least one agent can do better by simply performing the tasks it was originally allocated.

The negotiation set for task-oriented domains (assuming rational agents) is the set of offers that are both individually rational and Pareto optimal. There is no sense making an individually irrational offer that will be refused, nor in making an offer when there is a better offer that improves one agent's utility without hurting anyone else.

The monotonic concession protocol

The negotiation protocol we consider for task-oriented domains is known as the **monotonic concession protocol**. The rules of this protocol are as follows.

- Negotiation proceeds in a series of rounds.

- On the first round, both agents *simultaneously* propose a deal, D_i. = (T_1, T_2), from the negotiation set. (This is different from the alternating offers we saw before.)
- An agreement is reached if the two agents propose deals D_1 and D_2, respectively, such that either (i) $U_1(D_2) \geq U_1(D_1)$ or (ii) $U_2(D_1) \geq U_2(D_2)$, that is, if one of the agents finds that the deal proposed by the other is at least as good or better than the proposal it made. If agreement is reached, then the rule for determining the agreement deal is as follows: If each agent's offer matches or exceeds that of the other agent, then one of the proposals is selected at random. If only one proposal exceeds or matches the other's proposal, then this is the agreement deal.
- If no agreement is reached, then negotiation proceeds to another round of simultaneous proposals. In round $t + 1$, each agent must either repeat the proposal from the previous round or make a **concession**–a proposal that is more preferred by the other agent (i.e., has higher utility).
- If neither agent makes a concession, then negotiation terminates, and both agents implement the conflict deal, carrying out the tasks they were originally assigned.

Since the set of possible deals is finite, the agents cannot negotiate indefinitely: either the agents will reach agreement, or a round will occur in which neither agent concedes. However, the protocol does not guarantee that agreement will be reached *quickly*: since the number of possible deals is $O(2^{|T|})$, it is conceivable that negotiation will continue for a number of rounds exponential in the number of tasks to be allocated.

The Zeuthen strategy

So far, we have said nothing about how negotiation participants might or should behave when using the monotonic concession protocol for task–oriented domains. One possible strategy is the **Zeuthen strategy**.

The idea of the Zeuthen strategy is to measure an agent's *willingness to risk conflict*. Intuitively, an agent will be more willing to risk conflict if the difference in utility between its current proposal and the conflict deal is low. In this case, the agent has little to lose if negotiation fails and the conflict deal is implemented, and so is more willing to risk conflict, and less willing to concede. In contrast, if the difference between the agent's current proposal and the conflict deal is high, then the agent has more to lose from conflict and is therefore less willing to risk conflict—and thus more willing to concede.

Agent *i*'s willingness to risk conflict at round *t*, denoted $risk_i^t$, is measured as follows:

$$risk_i^t = \frac{\text{utility } i \text{ loses by conceding and accepting } j\text{'s offer}}{\text{utility } i \text{ loses by not conceding and causing conflict}}.$$

Until an agreement is reached, the value of $risk_i^t$ will be a value between 0 and 1. Higher values of $risk_i^t$ (nearer to 1) indicate that *i* has less to lose from conflict, and so is more willing to risk conflict.

The Zeuthen strategy says that each agent's first proposal should be a deal in the negotiation set that maximizes its own utility (there may be more than one). After that, the agent who should concede on round *t* of negotiation should be the one with the smaller value of risk—the one with the most to lose from conflict if neither concedes.

The next question to answer is how much should be conceded? The answer provided by the Zeuthen strategy is, "Just enough to change the balance of risk to the other agent." That is, an agent should make the *smallest* concession that will make the other agent concede on the next round.

There is one final refinement to the Zeuthen strategy. Suppose that at some point both agents have *equal* risk. Then, according to the strategy, both should concede. But, knowing this, one agent could potentially “defect” by not conceding, and so benefit. To avoid the possibility of both conceding at this point, we extend the strategy by having the agents “flip a coin” to decide who should concede if ever an equal risk situation is reached.

With this strategy, agreement will be Pareto optimal and individually rational. However, since the space of possible deals is exponential in the number of tasks, following this strategy may require $O(2^{|T|})$ computations of the cost function at each negotiation step. Finally, the Zeuthen strategy (with the coin flipping rule) is in Nash equilibrium.

Summary

- **Multiagent** planning is necessary when there are other agents in the environment with which to cooperate or compete. Joint plans can be constructed, but must be augmented with some form of coordination if two agents are to agree on which joint plan to execute.
- **Game theory** describes rational behavior for agents in situations in which multiple agents interact. Game theory is to multiagent decision making as decision theory is to single-agent decision making.
- **Solution concepts** in game theory are intended to characterize rational outcomes of a game—outcomes that might occur if every agent acted rationally.
- **Non-cooperative game theory** assumes that agents must make their decisions independently. **Nash equilibrium** is the most important solution concept in non-cooperative game theory. A Nash equilibrium is a strategy profile in which no agent has an incentive to deviate from its specified strategy. We have techniques for dealing with repeated games and sequential games.
- **Cooperative game theory** considers settings in which agents can make binding agreements to form coalitions in order to cooperate. Solution concepts in cooperative game attempt to formulate which coalitions are stable (the **core**) and how to fairly divide the value that a coalition obtains (the **Shapley value**).
- Specialized techniques are available for certain important classes of multiagent decision: the contract net for task sharing; auctions are used to efficiently allocate scarce resources; bargaining for reaching

agreements on matters of common interest; and voting procedures for aggregating preferences.

Bibliographical and Historical Notes

It is a curiosity of the field that researchers in AI did not begin to seriously consider the issues surrounding interacting agents until the 1980s—and the multiagent systems field did not really become established as a distinctive subdiscipline of AI until a decade later. Nevertheless, ideas that hint at multiagent systems were present in the 1970s. For example, in his highly influential *Society of Mind* theory, Marvin Minsky (1986, 2007) proposed that human minds are constructed from an ensemble of agents. Doug Lenat had similar ideas in a framework he called BEINGS (Lenat, 1975). In the 1970s, building on his PhD work on the PLANNER system, Carl Hewitt proposed a model of computation as interacting agents called the **actor model**, which has become established as one of the fundamental models in concurrent computation (Hewitt, 1977; Agha, 1986).

The prehistory of the multiagent systems field is thoroughly documented in a collection of papers entitled *Readings in Distributed Artificial Intelligence* (Bond and Gasser, 1988). The collection is prefaced with a detailed statement of the key research challenges in multiagent systems, which remains remarkably relevant today, more than thirty years after it was written. Early research on multiagent systems tended to assume that all agents in a system were acting with common interest, with a single designer. This is now recognized as a special case of the more general multiagent setting—the special case is known as **cooperative distributed problem solving**. A key system of this time was the Distributed Vehicle Monitoring Testbed (DVMT), developed under the supervision of Victor Lesser at the University of Massachusetts (Lesser and Corkill, 1988). The DVMT modeled a scenario in which a collection of geographically distributed acoustic sensor agents cooperate to track the movement of vehicles.

The contemporary era of multiagent systems research began in the late 1980s, when it was widely realized that agents with differing preferences are the norm in AI and society—from this point on, game theory began to be established as the main methodology for studying such agents.

Multiagent planning has leaped in popularity in recent years, although it does have a long history. Konolige (1982) formalizes multiagent planning in first-order logic, while Pednault (1986) gives a STRIPS-style description. The notion of joint intention, which is essential if agents are to execute a joint plan, comes from work on communicative acts (Cohen and Perrault, 1979; Cohen and Levesque, 1990; Cohen *et al.*, 1990). Boutilier and Brafman (2001) show how to adapt partial-order planning to a multiactor setting. Brafman and Domshlak (2008) devise a multiactor planning algorithm whose complexity grows only linearly with the number of actors, provided that the degree of coupling (measured partly by the tree width of the graph of interactions among agents) is bounded.

Multiagent planning is hardest when there are adversarial agents. As Jean-Paul Sartre (1960) said, “In a football match, everything is complicated by the presence of the other team.” General Dwight D. Eisenhower said, “In preparing for battle I have always found that plans are useless, but planning is indispensable,” meaning that it is important to have a conditional plan or policy, and not to expect an unconditional plan to succeed.

The topic of distributed and multiagent reinforcement learning (RL) was not covered in this chapter but is of great current interest. In distributed RL, the aim is to devise methods by which multiple, coordinated agents learn to optimize a common utility function. For example, can we devise methods whereby separate subagents for robot navigation and robot obstacle avoidance could cooperatively achieve a combined control system that is globally optimal? Some basic results in this direction have been obtained (Guestrin *et al.*, 2002; Russell and Zimdars, 2003). The basic idea is that each subagent learns its own Q-function (a kind of utility function; see Section 23.3.3) from its own stream of rewards. For example, a robotnavigation component can receive rewards for making progress towards the goal, while the obstacle-avoidance component receives negative rewards for every collision. Each global decision maximizes the sum of Q-functions and the whole process converges to globally optimal solutions.

The roots of game theory can be traced back to proposals made in the 17th century by Christiaan Huygens and Gottfried Leibniz to study competitive and cooperative human interactions scientifically and mathematically. Throughout the 19th century, several leading economists created simple mathematical examples to analyze particular examples of competitive situations.

The first formal results in game theory are due to Zermelo (1913) (who had, the year before, suggested a form of minimax search for games, albeit an incorrect one). Emile Borel (1921) introduced the notion of a mixed strategy. John von Neumann (1928) proved that every two-person, zero-sum game has a maximin equilibrium in mixed strategies and a well-defined value. Von Neumann’s collaboration with the economist Oskar Morgenstern led to the publication in 1944 of the *Theory of Games and Economic Behavior*, the defining book for game theory. Publication of the book was delayed by the wartime paper shortage until a member of the Rockefeller family personally subsidized its publication.

In 1950, at the age of 21, John Nash published his ideas concerning equilibria in general (non-zero-sum) games. His definition of an equilibrium solution, although anticipated in the work of Cournot (1838), became known as Nash equilibrium. After a long delay because of the schizophrenia he suffered from 1959 onward, Nash was awarded the Nobel Memorial Prize in Economics (along with Reinhart Selten and John Harsanyi) in 1994. The Bayes–Nash equilibrium is described by Harsanyi (1967) and discussed by Kadane and Larkey (1982). Some issues in the use of game theory for agent control are covered by Binmore (1982). Aumann and Brandenburger (1995) show how different equilibria can be reached depending on the knowleedge each player has.

The prisoner's dilemma was invented as a classroom exercise by Albert W. Tucker in 1950 (based on an example by Merrill Flood and Melvin Dresher) and is covered extensively by Axelrod (1985) and Poundstone (1993). Repeated games were introduced by Luce and Raiffa (1957), and Abreu and Rubinstein (1988) discuss the use of finite state machines for repeated games—technically, **Moore machines**. The text by Mailath and Samuelson (2006) concentrates on repeated games.

Games of partial information in extensive form were introduced by Kuhn (1953). The sequence form for partial-information games was invented by Romanovskii (1962) and independently by Koller *et al*. (1996); the paper by Koller and Pfeffer (1997) provides a readable introduction to the field and describes a system for representing and solving sequential games.

The use of abstraction to reduce a game tree to a size that can be solved with Koller's technique was introduced by Billings *et al*. (2003). Subsequently, improved methods for equilibrium–finding enabled solution of abstractions with 10^{12} states (Gilpin *et al.,* 2008; Zinkevich *et al.,* 2008).

Bowling *et al*. (2008) show how to use importance sampling to get a better estimate of the value of a strategy. Waugh et *al*. (2009) found that the abstraction approach is vulnerable to making systematic errors in approximating the equilibrium solution: it works for some games but not others. Brown and Sandholm (2019) showed that, at least in the case of multiplayer Texas hold ’em poker, these vulnerabilities can be overcome by sufficient computing power. They used a 64-core server running for 8 days to compute a baseline strategy for their Pluribus program. With that strategy they were able to defeat human champion opponents.

Game theory and MDPs are combined in the theory of Markov games, also called stochastic games (Littman, 1994; Hu and Wellman, 1998). Shapley (1953b) actually described the value iteration algorithm independently of Bellman, but his results were not widely appreciated, perhaps because they were presented in the context of Markov games. Evolutionary game theory (Smith, 1982; Weibull, 1995) looks at strategy drift over time: if your opponent’s strategy is changing, how should you react?

Textbooks on game theory from an economics point of view include those by Myerson (1991), Fudenberg and Tirole (1991), Osborne (2004), and Osborne and Rubinstein (1994). From an AI perspective we have Nisan et *al*. (2007) and Leyton–Brown and Shoham (2008). See (Sandholm, 1999) for a useful survey of multiagent decision making.

Multiagent RL is distinguished from distributed RL by the presence of agents who cannot coordinate their actions (except by explicit communicative acts) and who may not share the same utility function. Thus, multiagent RL deals with sequential game-theoretic problems or **Markov games**, as defined in Chapter 16. What causes problems is the fact that, while an agent is learning to defeat its opponent’s policy, the opponent is changing its policy to defeat the agent. Thus, the environment is **nonstationary** (see page 555).

Littman (1994) noted this difficulty when introducing the first RL algorithms for zero-sum Markov games. Hu and Wellman (2003) present a Q-learning algorithm for general-sum games that converges when the Nash equilibrium is unique; when there are multiple equilibria, the notion of convergence is not so easy to define (Shoham et *al.*, 2004).

Assistance games were introduced under the heading of **cooperative inverse reinforcement learning** by Hadfield-Menell *et al*. (2017a). Malik *et al*. (2018) introduced an efficient POMDP solver designed specifically for assistance games. They are related to **principal–agent games** in economics, in which a principal (e.g., an employer) and an agent (e.g., an employee) need to find a mutually beneficial arrangement despite having widely different preferences. The primary differences are that (1) the robot has no preferences of its own, and (2) the robot is uncertain about the human preferences it needs to optimize.

Cooperative games were first studied by von Neumann and Morgenstern (1944). The notion of the core was introduced by Donald Gillies (1959), and the Shapley value by Lloyd Shapley (1953a). A good introduction to the mathematics of cooperative games is Peleg and Sudholter (2002). Simple games in general are discussed in detail by Taylor and Zwicker (1999). For an introduction to the computational aspects of cooperative game theory, see Chalkiadakis *et al*. (2011).

Many compact representation schemes for cooperative games have been developed over the past three decades, starting with the work of Deng and Papadimitriou (1994). The most influential of these schemes is the marginal contribution networks model, which was introduced by Ieong and Shoham (2005). The approach to coalition formation that we describe was developed by Sandholm *et al*. (1999); Rahwan *et al*. (2015) survey the state of the art.

The contract net protocol was introduced by Reid Smith for his PhD work at Stanford University in the late 1970s (Smith, 1980). The protocol seems to be so natural that it is regularly reinvented to the present day. The economic foundations of the protocol were studied by Sandholm (1993).

Auctions and mechanism design have been mainstream topics in computer science and AI for several decades: see Nisan (2007) for a mainstream computer science perspective, Krishna (2002) for an introduction to the theory of auctions, and Cramton *et al*. (2006) for a collection of articles on computational aspects of auctions.

The 2007 Nobel Memorial Prize in Economics went to Hurwicz, Maskin, and Myerson “for having laid the foundations of mechanism design theory” (Hurwicz, 1973). The tragedy of the commons, a motivating problem for the field, was analyzed by William Lloyd (1833) but named and brought to public attention by Garrett Hardin (1968). Ronald Coase presented a theorem that if resources are subject to private ownership and if transaction costs are low enough, then the resources will be managed efficiently (Coase, 1960). He points out that, in practice, transaction costs are high, so this theorem does not apply, and we should look to other solutions beyond privatization and the marketplace. Elinor Ostrom’s *Governing the Commons* (1990) described solutions for the problem based on placing management control over the resources into the hands of the local people who have the most knowledge of the situation. Both Coase and Ostrom won the Nobel Prize in economics for their work.

The revelation principle is due to Myerson (1986), and the revenue equivalence theorem was developed independently by Myerson (1981) and Riley and Samuelson (1981). Two economists, Milgrom (1997) and

Klemperer (2002), write about the multibillion-dollar spectrum auctions they were involved in.

Mechanism design is used in multiagent planning (Hunsberger and Grosz, 2000; Stone *et al.,* 2009) and scheduling (Rassenti *et al.,* 1982). Varian (1995) gives a brief overview with connections to the computer science literature, and Rosenschein and Zlotkin (1994) present a book-length treatment with applications to distributed AI. Related work on distributed AI goes under several names, including collective intelligence (Tumer and Wolpert, 2000; Segaran, 2007) and market-based control (Clearwater, 1996). Since 2001 there has been an annual Trading Agents Competition (TAC), in which agents try to make the best profit on a series of auctions (Wellman *et al.,* 2001; Arunachalam and Sadeh, 2005).

The social choice literature is enormous, and spans the gulf from philosophical considerations on the nature of democracy through to highly technical analyses of specific voting procedures. Campbell and Kelly (2002) provide a good starting point for this literature. The *Handbook of Computational Social Choice* provides a range of articles surveying research topics and methods in this field (Brandt *et al*., 2016). Arrow’s theorem lists desired properties of a voting system and proves that is impossible to achieve all of them (Arrow, 1951). Das– gupta and Maskin (2008) show that majority rule (not plurality rule, and not ranked choice voting) is the most robust voting system. The computational complexity of manipulating elections was first studied by Bartholdi *et al*. (1989).

We have barely skimmed the surface of work on negotiation in multiagent planning. Durfee and Lesser (1989) discuss how tasks can be shared out among agents by negotiation. Kraus *et al*. (1991) describe a system for playing Diplomacy, a board game requiring negotiation, coalition formation, and dishonesty. Stone (2000) shows how agents can cooperate as teammates in the competitive, dynamic, partially observable environment of robotic soccer. In a later article, Stone (2003) analyzes two competitive multiagent environments—RoboCup, a robotic soccer competition, and TAC, the auction-based Trading Agents Competition—and finds that the computational intractability of our current theoretically well-founded approaches has led to many multiagent systems being designed by *ad hoc* methods. Sarit Kraus has developed a number of agents that can negotiate with humans and other agents—see Kraus (2001) for a survey. The monotonic concession protocol for automated negotiation was proposed by Jeffrey S. Rosenschein and his students (Rosenschein and Zlotkin, 1994). The alternating offers protocol was developed by Rubinstein (1982).

Books on multiagent systems include those by Weiss (2000a), Young (2004), Vlassis (2008), Shoham and Leyton–Brown (2009), and Wooldridge (2009). The primary conference for multiagent systems is the International Conference on Autonomous Agents and MultiAgent Systems (AAMAS); there is also a journal by the same name. The ACM Conference on Electronic Commerce (EC) also publishes many relevant papers, particularly in the area of auction algorithms. The principal journal for game theory is *Games and Economic Behavior*.

[1] Morra is a recreational version of an **inspection game**. In such games, an inspector chooses a day to inspect a facility (such as a restaurant or a biological weapons plant), and the facility operator chooses a day to hide all the nasty stuff. The inspector wins if the days are different, and the facility operator wins if they are the same.

[2] It is a coincidence that these equations are the same as those for p; the coincidence arises because $U_E(one, two) = U_E(two, one) = -3$. This also explains why the optimal strategy is the same for both players.

[3] The word “auction” comes from the Latin *augeo*, to increase.

[4] There is actually a small chance that the agent with highest v_i fails to get the goods, in the case in which $b_o < v_i < b_o + d$. The chance of this can be made arbitrarily small by decreasing the increment d.

[5] Named after William Vickrey (1914–1996), who won the 1996 Nobel Prize in economics for this work and died of a heart attack three days later.

CHAPTER 18

PROBABILISTIC PROGRAMMING

In which we explain the idea of universal languages for probabilistic knowledge representation and inference in uncertain domains.

The spectrum of representations—atomic, factored, and structured—has been a persistent theme in AI. For deterministic models, search algorithms assume only an atomic representation; CSPs and propositional logic provide factored representations; and first-order logic and planning systems take advantage of structured representations. The expressive power afforded by structured representations yields models that are vastly more concise than the equivalent factored or atomic descriptions.

For probabilistic models, Bayesian networks as described in Chapters 13 and 14 are factored representations: the set of random variables is fixed and finite, and each has a fixed range of possible values. This fact limits the applicability of Bayesian networks, because the Bayesian network representation for a complex domain is simply too large. This makes it infeasible to construct such representations by hand and infeasible to learn them from any reasonable amount of data.

The problem of creating an expressive formal language for probabilistic information has taxed some of the greatest minds in history, including Gottfried Leibniz (the co-inventor of calculus), Jacob Bernoulli (discoverer of *e*, the calculus of variations, and the Law of Large Numbers), Augustus De Morgan, George Boole, Charles Sanders Peirce (one of the principal logicians of the 19th century), John Maynard Keynes (the leading economist of the 20th century), and Rudolf Carnap (one of the greatest analytical philosophers of the 20th century). The problem resisted these and many other efforts until the 1990s.

Thanks in part to the development of Bayesian networks, there are now mathematically elegant and eminently practical formal languages that allow the creation of probabilistic models for very complex domains. These languages are *universal* in the same sense that Turing machines are universal: they can represent any computable probability model, just as Turing machines can represent any computable function. In addition, these languages come with general-purpose inference algorithms, roughly analogous to sound and complete logical inference algorithms such as resolution.

There are two routes to introducing expressive power into probability theory. The first is via logic: to devise a language that defines probabilities over first-order possible worlds, rather than the propositional possible worlds of Bayes nets. This route is covered in Sections 18.1 and 18.2, with Section 18.3 covering the specific case of temporal reasoning. The second route is via traditional programming languages: we introduce stochastic elements— random choices, for example—into such languages, and view programs as defining probability distributions over their own execution traces. This approach is covered in Section 18.4.

Both routes lead to a **probabilistic programming language (PPL)**. The first route leads to declarative PPLs, which bear roughly the same relationship to general PPLs as logic programming (Chapter 9) does to general programming languages.

18.1 Relational Probability Models

Recall from Chapter 12 that a probability model defines a set Ω of possible worlds with a probability $P(\omega)$ for each world ω. For Bayesian networks, the possible worlds are assignments of values to variables; for the Boolean case in particular, the possible worlds are identical to those of propositional logic.

For a first-order probability model, then, it seems we need the possible worlds to be those of first-order logic—that is, a set of objects with relations among them and an interpretation that maps constant symbols to objects, predicate symbols to relations, and function symbols to functions on those objects. (See Section 8.2.) The model also needs to define a probability for each such possible world, just as a Bayesian network defines a probability for each assignment of values to variables.

Let us suppose, for a moment, that we have figured out how to do this. Then, as usual (see page 407), we can obtain the probability of any first-order logical sentence ϕ (phi) as a sum over the possible worlds where it is true:

$$P(\phi) = \sum_{\omega:\phi \text{ is true in } \omega} P(\omega). \tag{18.1}$$

Conditional probabilities $P(\phi \mid \mathbf{e})$ can be obtained similarly, so we can, in principle, ask any question we want of our model—and get an answer. So far, so good.

There is, however, a problem: the set of first-order models is infinite. We saw this explicitly in Figure 8.4 on page 277, which we show again in Figure 18.1 (top). This means that (1) the summation in Equation (18.1) could be infeasible, and (2) specifying a complete, consistent distribution over an infinite set of worlds could be very difficult.

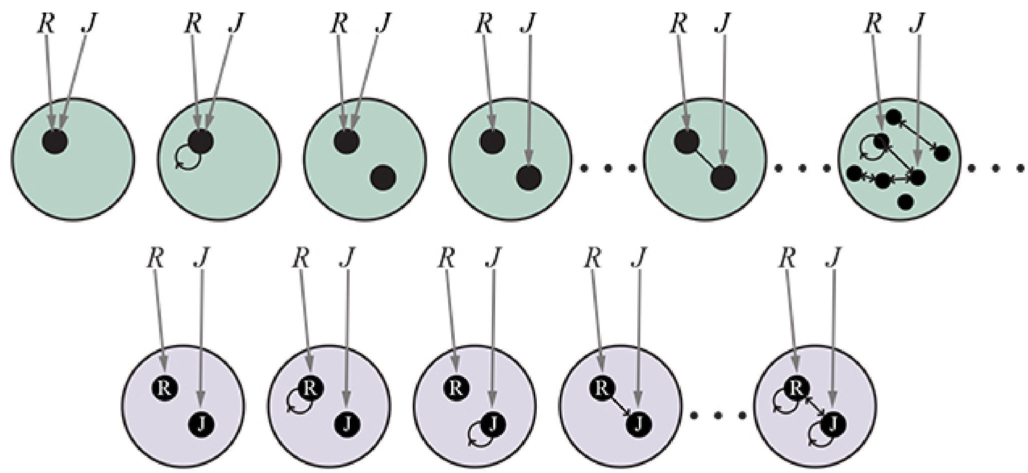


Figure 18.1 Top: Some members of the set of all possible worlds for a language with two constant symbols, R and J, and one binary relation symbol, under the standard semantics for first-order logic. Bottom: the possible worlds under database semantics. The interpretation of the constant symbols is fixed, and there is a distinct object for each constant symbol.

In this section, we avoid this issue by considering the **database semantics** defined in Section 8.2.8 (page 282). The database semantics makes the **unique names assumption—** here, we adopt it for the constant symbols. It also assumes **domain closure**—there are no more objects beyond those that are named. We can then guarantee a finite set of possible worlds by making the set of objects in each world be exactly the set of constant symbols that are used; as shown in Figure 18.1 (bottom), there is no uncertainty about the mapping from symbols to objects or about the objects that exist.

We will call models defined in this way **relational probability models**, or RPMs.[1] The most significant difference between the semantics of RPMs and the database semantics introduced in Section 8.2.8 is that RPMs do not make the closed-world assumption—in a probabilistic reasoning system we can't just assume that every unknown fact is false.

18.1.1 Syntax and semantics

Let us begin with a simple example: suppose that an online book retailer would like to provide overall evaluations of products based on recommendations received from its customers. The evaluation will take the form of a posterior distribution over the quality of the book, given the available evidence. The simplest solution is to base the evaluation on the average recommendation, perhaps with a variance determined by the number of recommendations, but this fails to take into account the fact that some customers are kinder than others and some are less honest than others. Kind customers tend to give high recommendations even to fairly mediocre books, while dishonest customers give very high or very low recommendations for reasons other than quality—they might be paid to promote some publisher's books.[2]

For a single customer C_1 recommending a single book B_1, the Bayes net might look like the one shown in Figure 18.2(a). (Just as in Section 9.1, expressions with parentheses such as *Honest*(C_1) are just fancy symbols—in this case, fancy names for random variables.) With two customers and two books, the Bayes net looks like the one in Figure 18.2(b). For larger numbers of books and customers, it is quite impractical to specify a Bayes net by hand.

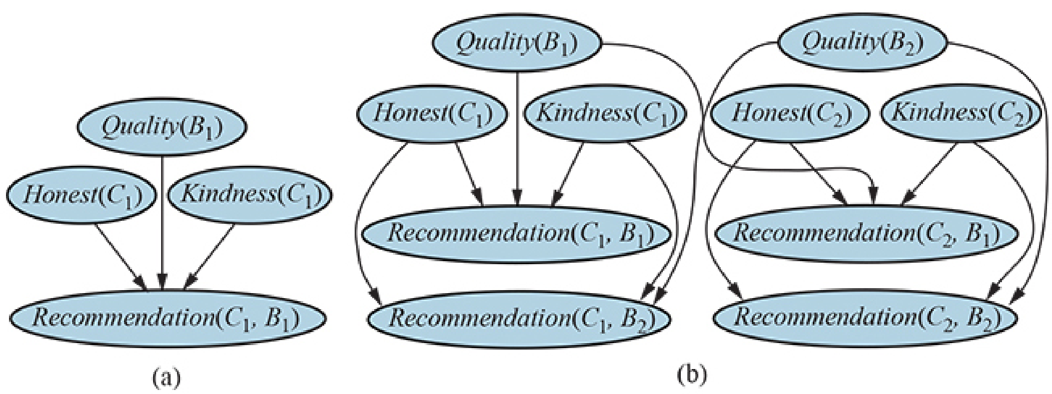


Figure 18.2 (a) Bayes net for a single customer C_1 recommending a single book B_1. *Honest*(C_1) is Boolean, while the other variables have integer values from 1 to 5. (b) Bayes net with two customers and two books.

Fortunately, the network has a lot of repeated structure. Each *Recommendation*(c, b) variable has as its parents the variables *Honest*(c), *Kindness*(c), and *Quality*(b). Moreover, the conditional probability tables (CPTs) for all the *Recommendation*(c, b) variables are identical, as are those for all the *Honest*(c) variables, and so on. The situation seems tailor-made for a first-order language. We would like to say something like

$$Recommendation(c, b) \sim RecCPT(Honest(c), Kindness(c), Quality(b))$$

which means that a customer's recommendation for a book depends probabilistically on the customer's honesty and kindness and the book's quality according to a fixed CPT.

Like first-order logic, RPMs have constant, function, and predicate symbols. We will also assume a **type signature** for each function—that is, a specification of the type of each argument and the function's value. (If the type of each object is known, many spurious possible worlds are eliminated by this mechanism; for example, we need not worry about the kindness of each book, books recommending customers, and so on.) For the book-recommendation domain, the types are *Customer* and *Book*, and the type signatures for the functions and predicates are as follows:

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

The constant symbols will be whatever customer and book names appear in the retailer's data set. In the example given in Figure 18.2(b), these were C_1, C_2 and B_1, B_2.

Given the constants and their types, together with the functions and their type signatures, the **basic random variables** of the RPM are obtained by instantiating each function with each possible combination of objects. For the book recommendation model, the basic random variables include *Honest*(C_1), *Quality*(B_2), *Recommendation*(C_1, B_2), and so on. These are exactly the variables appearing in Figure 18.2(b). Because each type has only finitely many instances (thanks to the domain closure assumption), the number of basic random variables is also finite.

To complete the RPM, we have to write the dependencies that govern these random variables. There is one dependency statement for each function, where each argument of the function is a logical variable (i.e., a variable that ranges over objects, as in first-order logic). For example, the following dependency states that, for every customer *c*, the prior probability of honesty is 0.99 *true* and 0.01 *false:*

$$Honest(c) \sim \langle 0.99, 0.01 \rangle$$

Similarly, we can state prior probabilities for the kindness value of each customer and the quality of each book, each on the 1-5 scale:

$$
\begin{aligned}
&Kindness(c) \sim \langle 0.1, 0.1, 0.2, 0.3, 0.3 \rangle\\
&Quality(b) \sim \langle 0.05, 0.2, 0.4, 0.2, 0.15 \rangle
\end{aligned}
$$

Finally, we need the dependency for recommendations: for any customer *c* and book *b*, the score depends on the honesty and kindness of the customer and the quality of the book:

$$Recommendation(c,b) \;\sim\; RecCPT(Honest(c), Kindness(c), Quality(b))$$

where *RecCPT* is a separately defined conditional probability table with $2 \times 5 \times 5 = 50$ rows, each with 5 entries. For the purposes of illustration, we'll assume that an honest recommendation for a book of quality q from a person of kindness k is uniformly distributed in the range $\left[\lfloor \frac{q+k}{2} \rfloor, \lceil \frac{q+k}{2} \rceil\right]$.

The semantics of the RPM can be obtained by instantiating these dependencies for all known constants, giving a Bayesian network (as in Figure 18.2(b)) that defines a joint distribution over the RPM's random variables.[3]

The set of possible worlds is the Cartesian product of the ranges of all the basic random variables, and, as with Bayesian networks, the probability for each possible world is the product of the relevant conditional probabilities from the model. With C customers and B books, there are C *Honest* variables, C *Kindness* variables, B *Quality* variables, and BC *Recommendation* variables, leading to $2^C 5^{C+B+BC}$ possible worlds. With ten million books and a billion customers, that's about $10^{7 \times 10}$ worlds. Thanks to the expressive power of RPMs, the complete probability model still has only fewer than 300 parameters—most of them in the *RecCPT* table.

We can refine the model by asserting a **context-specific independence** (see page 438) to reflect the fact that dishonest customers ignore quality when giving a recommendation; moreover, kindness plays no role in their decisions. Thus, *Recommendation*(c, b) is independent of *Kindness*(c) and *Quality*(b) when *Honest*(c) = *false:*

$$\begin{aligned} Recommendation(c,b) \;\sim\; & \textbf{if } Honest(c) \textbf{ then} \\ & \quad HonestRecCPT(Kindness(c), Quality(b)) \\ & \textbf{else } \langle 0.4, 0.1, 0.0, 0.1, 0.4 \rangle \,. \end{aligned}$$

This kind of dependency may look like an ordinary if-then-else statement in a programming language, but there is a key difference: the inference engine *doesn't necessarily know the value of the conditional test* because *Honest*(c) is a random variable.

We can elaborate this model in endless ways to make it more realistic. For example, suppose that an honest customer who is a fan of a book's author always gives the book a 5, regardless of quality:

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

Again, the conditional test *Fan*(c, *Author*(b)) is unknown, but if a customer gives only 5s to a particular author's books and is not otherwise especially kind, then the posterior probability that the customer is a fan of that author will be high. Furthermore, the posterior distribution will tend to discount the customer's 5s in evaluating the quality of that author's books.

In this example, we implicitly assumed that the value of *Author*(b) is known for every b, but this may not be the case. How can the system reason about whether, say, C_1 is a fan of *Author*(B_2) when *Author*(B_2) is unknown? The answer is that the system may have to reason about *all possible authors*. Suppose (to keep things simple) that there are just two authors, A_1 and A_2. Then *Author*(B_2) is a random variable with two possible values, A_1 and A_2, and it is a parent of *Recommendation*(C_1, B_2). The variables *Fan*(C_1, A_1) and *Fan*(C_1, A_2) are parents too. The conditional distribution for *Recommendation*(C_1, B_2) is then essentially a **multiplexer** in which the *Author*(B_2) parent acts as a selector to choose which of *Fan*(C_1, A_1) and *Fan*(C_1, A_2) actually gets to influence the recommendation. A fragment of the equivalent Bayes net is shown in Figure 18.3. Uncertainty in the value of *Author*(B_2), which affects the dependency structure of the network, is an instance of **relational uncertainty**.

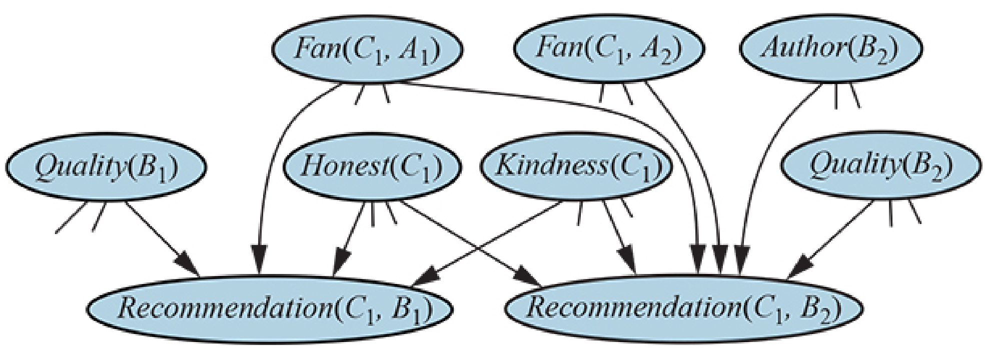


Figure 18.3 Fragment of the equivalent Bayes net for the book recommendation RPM when *Author*(B_2) is unknown.

In case you are wondering how the system can possibly work out who the author of B_2 is: consider the possibility that three other customers are fans of A_1 (and have no other favorite authors in common) and all three have given B_2 a 5, even though most other customers find it quite dismal. In that case, it is extremely likely that A_1 is the author of B_2. The emergence of sophisticated reasoning like this from an RPM model of just a few lines is an intriguing example of how probabilistic influences spread through the web of interconnections among objects in the model. As more dependencies and more objects are added, the picture conveyed by the posterior distribution often becomes clearer and clearer.

18.1.2 Example: Rating player skill levels

Many competitive games have a numerical measure of players' skill levels, sometimes called a **rating**. Perhaps the best-known is the Elo rating for chess players, which rates a typical beginner at around 800 and the world champion usually somewhere above 2800. Although Elo ratings have a statistical basis, they have some ad hoc elements. We can develop a Bayesian rating scheme as follows: each player i has an underlying skill level *Skill*(i); in each game g, i's actual performance is *Performance*(i, g), which may vary from the underlying skill level; and the winner of g is the player whose performance in g is better. As an RPM, the model looks like this:

$$\begin{aligned}
&Skill(i) \ \sim \ N(\mu, \sigma^2)\\
&Performance(i, g) \ \sim \ N(Skill(i), \beta^2)\\
&Win(i,\ j, g) \ = \ \mathbf{if}\ Game(g,\ i,\ j)\ \mathbf{then}\ (Performance(i, g) \ > \ Performance(\ j, g))
\end{aligned}$$

where β^2 is the variance of a player's actual performance in any specific game relative to the player's underlying skill level. Given a set of players and games, as well as outcomes for some of the games, an RPM inference engine can compute a posterior distribution over the skill of each player and the probable outcome of any additional game that might be played.

For team games, we'll assume, as a first approximation, that the overall performance of team t in game g is the sum of the individual performances of the players on t:

$$TeamPerformance(t, g) = \textstyle\sum_{i \in t}\ Performance(i, g).$$

Even though the individual performances are not visible to the ratings engine, the players' skill levels can still be estimated from the results of several games, as long as the team compositions vary across games. Microsoft's TrueSkill™ ratings engine uses this model, along with an efficient approximate inference algorithm, to serve hundreds of millions of users every day.

This model can be elaborated in numerous ways. For example, we might assume that weaker players have higher variance in their performance; we might include the player's role on the team; and we might consider specific kinds of performance and skill—e.g., defending and attacking—in order to improve team composition and predictive accuracy.

18.1.3 Inference in relational probability models

The most straightforward approach to inference in RPMs is simply to construct the equivalent Bayesian network, given the known constant symbols belonging to each type. With B books and C customers, the basic model given previously could be constructed with simple loops:[4]

for $b = 1$ **to** B *do*

  add node $Quality_b$ with no parents, prior $\langle$ 0.05,0.2,0.4,0.2,0.15 $\rangle$

for $c = 1$ **to** C *do*

  add node $Honest_c$ with no parents, prior $\langle$ 0.99,0.01 $\rangle$

  add node $Kindness_c$ with no parents, prior $\langle$ 0.1,0.1,0.2,0.3,0.3 $\rangle$

for $b = 1$ **to** B *do*

  add node $\mathrm{Re}\,commendation_{c,b}$ *with parents* $Honest_c, Kindness_c,\ Quality_b$

    and conditional distribution $RecCPT(Honest_c, Kindness_c,\ Quality_b)$

This technique is called **grounding** or **unrolling**; it is the exact analog of **propositionalization** for first-order logic (page 298). The obvious drawback is that the resulting Bayes net may be very large. Furthermore, if there are many candidate objects for an unknown relation or function—for example, the unknown author of B_2—then some variables in the network may have many parents.

Fortunately, it is often possible to avoid generating the entire implicit Bayes net. As we saw in the discussion of the variable elimination algorithm on page 451, every variable that is not an ancestor of a query variable or evidence variable is irrelevant to the query. Moreover, if the query is conditionally independent of some variable given the evidence, then that variable is also irrelevant. So, by chaining through the model starting from the query and evidence, we can identify just the set of variables that are relevant to the query. These are the only ones that need to be instantiated to create a potentially tiny fragment of the implicit Bayes net. Inference in this fragment gives the same answer as inference in the entire implicit Bayes net.

Another avenue for improving the efficiency of inference comes from the presence of repeated substructure in the unrolled Bayes net. This means that many of the factors constructed during variable elimination (and similar kinds of tables constructed by clustering algorithms) will be identical; effective caching schemes have yielded speedups of three orders of magnitude for large networks.

Third, MCMC inference algorithms have some interesting properties when applied to RPMs with relational uncertainty. MCMC works by sampling complete possible worlds, so in each state the relational structure is completely known. In the example given earlier, each MCMC state would specify the value of $Author(B_2)$, and so the other potential authors are no longer parents of the recommendation nodes for B_2. For MCMC, then, relational uncertainty causes no increase in network complexity; instead, the MCMC process includes transitions that change the relational structure, and hence the dependency structure, of the unrolled network.

Finally, it may be possible in some cases to avoid grounding the model altogether. Resolution theorem provers and logic programming systems avoid propositionalizing by instantiating the logical variables only as needed to make the inference go through; that is, they *lift* the inference process above the level of ground propositional sentences and make each lifted step do the work of many ground steps.

The same idea can be applied in probabilistic inference. For example, in the variable elimination algorithm, a lifted factor can represent an entire set of ground factors that assign probabilities to random variables in the RPM, where those random variables differ only in the constant symbols used to construct them. The details of this method are beyond the scope of this book, but references are given at the end of the chapter.

18.2 Open-Universe Probability Models

We argued earlier that database semantics was appropriate for situations in which we know exactly the set of relevant objects that exist and can identify them unambiguously. (In particular, all observations about an object are correctly associated with the constant symbol that names it.) In many real-world settings, however, these assumptions are simply untenable. For example, a book retailer might use an ISBN (International Standard Book Number) as a constant symbol to name each book, even though a given "logical" book (e.g., "Gone With the Wind") may have several ISBNs corresponding to hardcover, paperback, large print, reissues, and so on. It would make sense to aggregate recommendations across multiple ISBNs, but the retailer may not know for sure which ISBNs are really the same book. (Note that we are not reifying the *individual copies* of the book, which might be necessary for used-book sales, car sales, and so on.) Worse still, each customer is identified by a login ID, but a dishonest customer may have thousands of IDs! In the computer security field, these multiple IDs are called **sybils** and their use to confound a reputation system is called a **sybil attack**.[5] Thus, even a simple application in a relatively well-defined, online domain involves both **existence uncertainty** (what are the real books and customers underlying the observed data) and **identity uncertainty** (which logical terms really refer to the same object).

The phenomena of existence and identity uncertainty extend far beyond online booksellers. In fact they are pervasive:

- A vision system doesn't know what exists, if anything, around the next corner, and may not know if the object it sees now is the same one it saw a few minutes ago.
- A text-understanding system does not know in advance the entities that will be featured in a text, and must reason about whether phrases such as

“Mary,” “Dr. Smith,” “she,” “his cardiologist,” “his mother,” and so on refer to the same object.

- An intelligence analyst hunting for spies never knows how many spies there really are and can only guess whether various pseudonyms, phone numbers, and sightings belong to the same individual.

Indeed, a major part of human cognition seems to require learning what objects exist and being able to connect observations—which almost never come with unique IDs attached—to hypothesized objects in the world.

Thus, we need to be able to define an **open universe probability model (OUPM)** based on the standard semantics of first-order logic, as illustrated at the top of Figure 18.1. A language for OUPMs provides a way of easily writing such models while guaranteeing a unique, consistent probability distribution over the infinite space of possible worlds.

18.2.1 Syntax and semantics

The basic idea is to understand how ordinary Bayesian networks and RPMs manage to define a unique probability model and to transfer that insight to the first-order setting. In essence, a Bayes net *generates* each possible world, event by event, in the topological order defined by the network structure, where each event is an assignment of a value to a variable. An RPM extends this to entire sets of events, defined by the possible instantiations of the logical variables in a given predicate or function. OUPMs go further by allowing generative steps that *add objects* to the possible world under construction, where the number and type of objects may depend on the objects that are already in that world and their properties and relations. That is, the event being generated is not the assignment of a value to a variable, but the very *existence* of objects.

One way to do this in OUPMs is to provide **number statements** that specify conditional distributions over the numbers of objects of various kinds. For example, in the book- recommendation domain, we might want to distinguish between *customers* (real people) and their *login IDs*. (It's actually login IDs that make recommendations, not customers!) Suppose (to keep things simple) the number of customers is uniform between 1 and 3 and the number of books is uniform between 2 and 4:

$$\begin{aligned} \#Customer &\sim UniformInt(1{,}3) \\ \#Book &\sim UniformInt(2{,}4). \end{aligned} \qquad (18.2)$$

We expect honest customers to have just one ID, whereas dishonest customers might have anywhere between 2 and 5 IDs:

$$\begin{aligned} \#LoginID(Owner = c) \sim\ &\textbf{if } Honest(c) \textbf{ then } Exactly(1) \\ &\textbf{else } UniformInt(2{,}5). \end{aligned} \qquad (18.3)$$

This number statement specifies the distribution over the number of login IDs for which customer *c* is the *Owner*. The *Owner* function is called an **origin function** because it says where each object generated by this number statement came from.

The example in the preceding paragraph uses a uniform distribution over the integers between 2 and 5 to specify the number of logins for a dishonest customer. This particular distribution is bounded, but in general there may not be an a priori bound on the number of objects. The most commonly used distribution over the nonnegative integers is the **Poisson distribution**. The Poisson has one parameter, λ, which is the expected number of objects, and a variable *X* sampled from Poisson(A) has the following distribution:

$$P(X = k) \ = \ \lambda^k e^{-\lambda}/k!.$$

The variance of the Poisson is also λ, so the standard deviation is $\sqrt{\lambda}$. This means that for large values of λ, the distribution is narrow relative to the mean —for example, if the number of ants in a nest is modeled by a Poisson with a mean of one million, the standard deviation is only a thousand, or 0.1%. For large numbers, it often makes more sense to use the **discrete log-normal distribution**, which is appropriate when the log of the number of objects is normally distributed. A particularly intuitive form, which we call the **order-of-magnitude distribution**, uses logs to base 10: thus, a distribution OM(3,1) has a mean of 10^3 and a standard deviation of one order of magnitude, i.e., the bulk of the probability mass falls between 10^2 and 10^4.

The formal semantics of OUPMs begins with a definition of the objects that populate possible worlds. In the standard semantics of typed first-order logic, objects are just numbered tokens with types. In OUPMs, each object is a generation history; for example, an object might be "the fourth login ID of the seventh customer." (The reason for this slightly baroque construction will become clear shortly.) For types with no origin functions—e.g., the *Customer* and *Book* types in Equation (18.2)—the objects have an empty origin; for example, $\langle Customer, , 2\rangle$ refers to the second customer generated from that number statement. For number statements with origin functions—e.g., Equation (18.3)—each object records its origin; for example, the object $\langle LoginID, \langle Owner, \langle Customer, , 2\rangle\rangle, 3\rangle$ is the third login belonging to the second customer.

The **number variables** of an OUPM specify how many objects there are of each type with each possible origin in each possible world; thus $\#LoginlD_{\langle Owner,\langle Customer, ,2\rangle\rangle}(\omega) = 4$ means that in world ω, customer 2 owns 4 login IDs. As in relational probability models, the **basic random variables** determine the values of predicates and functions for all tuples of objects; thus, $Honest_{\langle \mathrm{Customer}, ,2\rangle}(\omega) = true$ means that in world ω, customer 2 is honest. A possible world is defined by the values of all the number variables and basic random variables. A world may be generated from the model by sampling in topological order; Figure 18.4 shows an example. The probability of a world so constructed is the product of the probabilities for all the sampled values; in this case, 1.2672×10^{-11}. Now it becomes clear why each object contains its origin: this property ensures that every world can be constructed by exactly one generation sequence. If this were not the case, the probability of a world would be an unwieldy combinatorial sum over all possible generation sequences that create it.

Variable	Value	Probability
$\#Customer$	2	0.3333
$\#Book$	3	0.3333
$Honest_{\langle Customer,,1\rangle}$	*true*	0.99
$Honest_{\langle Customer,,2\rangle}$	*false*	0.01
$Kindness_{\langle Customer,,1\rangle}$	4	0.3
$Kindness_{\langle Customer,,2\rangle}$	1	0.1
$Quality_{\langle Book,,1\rangle}$	1	0.05
$Quality_{\langle Book,,2\rangle}$	3	0.4
$Quality_{\langle Book,,3\rangle}$	5	0.15
$\#LoginID_{\langle Owner,\langle Customer,,1\rangle\rangle}$	1	1.0
$\#LoginID_{\langle Owner,\langle Customer,,2\rangle\rangle}$	2	0.25
$Recommendation_{\langle LoginID,\langle Owner,\langle Customer,,1\rangle\rangle,1\rangle,\langle Book,,1\rangle}$	2	0.5
$Recommendation_{\langle LoginID,\langle Owner,\langle Customer,,1\rangle\rangle,1\rangle,\langle Book,,2\rangle}$	4	0.5
$Recommendation_{\langle LoginID,\langle Owner,\langle Customer,,1\rangle\rangle,1\rangle,\langle Book,,3\rangle}$	5	0.5
$Recommendation_{\langle LoginID,\langle Owner,\langle Customer,,2\rangle\rangle,1\rangle,\langle Book,,1\rangle}$	5	0.4
$Recommendation_{\langle LoginID,\langle Owner,\langle Customer,,2\rangle\rangle,1\rangle,\langle Book,,2\rangle}$	5	0.4
$Recommendation_{\langle LoginID,\langle Owner,\langle Customer,,2\rangle\rangle,1\rangle,\langle Book,,3\rangle}$	1	0.4
$Recommendation_{\langle LoginID,\langle Owner,\langle Customer,,2\rangle\rangle,2\rangle,\langle Book,,1\rangle}$	5	0.4
$Recommendation_{\langle LoginID,\langle Owner,\langle Customer,,2\rangle\rangle,2\rangle,\langle Book,,2\rangle}$	5	0.4
$Recommendation_{\langle LoginID,\langle Owner,\langle Customer,,2\rangle\rangle,2\rangle,\langle Book,,3\rangle}$	1	0.4

Figure 18.4 One particular world for the book recommendation OUPM. The number variables and basic random variables are shown in topological order, along with their chosen values and the probabilities for those values.

Open-universe models may have infinitely many random variables, so the full theory involves nontrivial measure-theoretic considerations. For example, number statements with Poisson or order-of-magnitude distributions allow for unbounded numbers of objects, leading to unbounded numbers of random variables for the properties and relations of those objects. Moreover, OUPMs can have recursive dependencies and infinite types (integers, strings, etc.). Finally, well-formedness disallows cyclic dependencies and infinitely receding ancestor chains; these conditions are undecidable in general, but certain syntactic sufficient conditions can be checked easily.

18.2.2 Inference in open-universe probability models

Because of the potentially huge and sometimes unbounded size of the implicit Bayes net that corresponds to a typical OUPM, unrolling it fully and performing exact inference is quite impractical. Instead, we must consider approximate inference algorithms such as MCMC (see Section 13.4.2).

Roughly speaking, an MCMC algorithm for an OUPM is exploring the space of possible worlds defined by sets of objects and relations among them, as illustrated in Figure 18.1(top). A move between adjacent states in this space can not only alter relations and functions but also add or subtract objects and change the interpretations of constant symbols. Even though each possible world may be huge, the probability computations required for each step—whether in Gibbs sampling or Metropolis-Hastings—are entirely local and in most cases take constant time. This is because the probability ratio between neighboring worlds depends on a subgraph of constant size around the variables whose values are changed. Moreover, a logical query can be evaluated *incrementally* in each world visited, usually in constant time per world, rather than being recomputing from scratch.

Some special consideration needs to be given to the fact that a typical OUPM may have possible worlds of infinite size. As an example, consider the multitarget tracking model in Figure 18.9: the function $X(a,t)$, denoting the state of aircraft *a* at time *t*, corresponds to an infinite sequence of variables for an unbounded number of aircraft at each step. For this reason, MCMC for OUPMs samples not completely specified possible worlds but *partial* worlds, each corresponding to a disjoint set of complete worlds. A partial world is a *minimal self-supporting instantiation*[6] of a subset of the *relevant* variables—that is, ancestors of the evidence and query variables. For example, variables $X(a, t)$ for values of t greater than the last observation time (or the query time, whichever is greater) are irrelevant, so the algorithm can consider just a finite prefix of the infinite sequence.

18.2.3 Examples

The standard "use case" for an OUPM has three elements: the *model*, the *evidence* (the known facts in a given scenario), and the *query*, which may be any expression, possibly with free logical variables. The answer is a posterior joint probability for each possible set of substitutions for the free variables, given the evidence, according to the model.[7] Every model includes type declarations, type signatures for the predicates and functions, one or more number statements for each type, and one dependency statement for each predicate and function. (In the examples below, declarations and signatures are omitted where the meaning is clear.) As in RPMs, dependency statements use an if-then-else syntax to handle context- specific dependencies.

Citation matching

Millions of academic research papers and technical reports are to be found online in the form of pdf files. Such papers usually contain a section near the end called "References" or "Bibliography," in which citations—strings of characters—are provided to inform the reader of related work. These strings can be located and "scraped" from the pdf files with the aim of creating a database-like representation that relates papers and researchers by authorship and citation links. Systems such as CiteSeer and Google Scholar present such a representation to their users; behind the scenes, algorithms operate to find papers, scrape the citation strings, and identify the actual papers to which the citation strings refer. This is a difficult task because these strings contain no object identifiers and include errors of syntax, spelling, punctuation, and content. To illustrate this, here are two relatively benign examples:

1. [Lashkari et al 94] Collaborative Interface Agents, Yezdi Lashkari, Max Metral, and Pattie Maes, Proceedings of the Twelfth National Conference on Articial Intelligence, MIT Press, Cambridge, MA, 1994.
2. Metral M. Lashkari, Y. and P. Maes. Collaborative interface agents. In Conference of the American Association for Artificial Intelligence, Seattle, WA, August 1994.

The key question is one of identity: are these citations of the same paper or different papers? Asked this question, even experts disagree or are unwilling to decide, indicating that reasoning under uncertainty is going to be an important part of solving this problem.[8] Ad hoc approaches—such as methods based on a textual similarity metric—often fail miserably. For example, in 2002, CiteSeer reported over 120 distinct books written by Russell and Norvig.

In order to solve the problem using a probabilistic approach, we need a generative model for the domain. That is, we ask how these citation strings come to be in the world. The process begins with researchers, who have names. (We don't need to worry about how the researchers came into existence; we just need to express our uncertainty about how many there are.) These researchers write some papers, which have titles; people cite the papers, combining the authors' names and the paper's title (with errors) into the text of the citation according to some grammar. The basic elements of this model are shown in Figure 18.5, covering the case where papers have just one author.[9]

type Researcher, Paper, Citation
random String Name(Researcher)
random String Title(Paper)
random Paper PubCited(Citation)
random String Text(Citation)
random Boolean Professor(Researcher)
origin Researcher Author(Paper)

$\#Researcher \sim OM(3,1)$
$Name(r) \sim NamePrior()$
$Professor(r) \sim Boolean(0.2)$
$\#Paper(Author = r) \sim \textbf{if } Professor(r) \textbf{ then } OM(1.5, 0.5) \textbf{ else } OM(1, 0.5)$
$Title(p) \sim PaperTitlePrior()$
$CitedPaper(c) \sim UniformChoice(\{Paper\ p\})$
$Text(c) \sim HMMGrammar(Name(Author(CitedPaper(c))), Title(CitedPaper(c)))$

Figure 18.5 An OUPM for citation information extraction. For simplicity the model assumes one author per paper and omits details of the grammar and error models.

Given just citation strings as evidence, probabilistic inference on this model to pick out the most likely explanation for the data produces an error rate 2 to 3 times lower than CiteSeer's (Pasula *et al.*, 2003). The inference process also exhibits a form of collective, knowledge-driven disambiguation: the more citations for a given paper, the more accurately each of them is parsed, because the parses have to agree on the facts about the paper.

Nuclear treaty monitoring

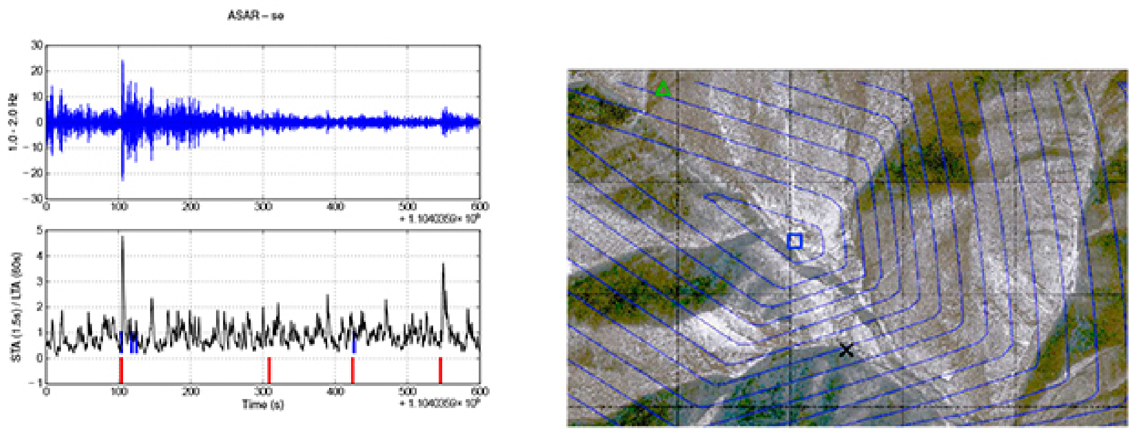
Verifying the Comprehensive Nuclear-Test-Ban Treaty requires finding all seismic events on Earth above a minimum magnitude. The UN CTBTO maintains a network of sensors, the International Monitoring System (IMS); its automated processing software, based on 100 years of seismology research, has a detection failure rate of about 30%. The NET-VISA system (Arora *et al.*, 2013), based on an OUPM, significantly reduces detection failures.

The NET-VISA model (Figure 18.6) expresses the relevant geophysics directly. It describes distributions over the number of events in a given time interval (most of which are naturally occurring) as well as over their time, magnitude, depth, and location. The locations of natural events are distributed according to a spatial prior that is trained (like other parts of the model) from historical data; man-made events are, by the treaty rules, assumed to occur uniformly over the surface of the Earth. At every station *s*, each phase (seismic wave type) p from an event e produces either ϕ or 1 detections (above-threshold signals); the detection probability depends on the event magnitude and depth and its distance from the station. “False alarm” detections also occur according to a station-specific rate parameter. The measured arrival time, amplitude, and other properties of a detection d from a real event depend on the properties of the originating event and its distance from the station.

$$
\begin{aligned}
&\#SeismicEvents \sim Poisson(T * \lambda_e) \\
&Time(e) \sim UniformReal(0, T) \\
&EarthQuake(e) \sim Boolean(0.999) \\
&Location(e) \sim \textbf{if } Earthquake(e) \textbf{ then } SpatialPrior() \textbf{ else } UniformEarth() \\
&Depth(e) \sim \textbf{if } Earthquake(e) \textbf{ then } UniformReal(0, 700) \textbf{ else } Exactly(0) \\
&Magnitude(e) \sim Exponential(log(10)) \\
&Detected(e, p, s) \sim Logistic(weights(s, p), Magnitude(e),\ Depth(e),\ Dist(e, s)) \\
&\#Detections(site = s) \sim Poisson(T * \lambda_f(s)) \\
&\#Detections(event{=}e,\ phase{=}p,\ station{=}s) = \textbf{if } Detected(e, p, s) \textbf{ then } 1 \textbf{ else } 0 \\
&OnsetTime(a, s) \textbf{ if } (event(a) = null) \textbf{ then } \sim UniformReal(0, T) \\
&\quad \textbf{else} = Time(event(a)) + GeoTT(Dist(event(a), s), Depth(event(a)), phase(a)) \\
&\qquad\qquad + Laplace(\mu_t(s), \sigma_t(s)) \\
&Amplitude(a, s) \textbf{ if } (event(a) = null) \textbf{ then } \sim NoiseAmpModel(s) \\
&\quad \textbf{else} = AmpModel(Magnitude(event(a)), Dist(event(a), s), Depth(event(a)), phase(a)) \\
&Azimuth(a, s) \textbf{ if } (event(a) = null) \textbf{ then } \sim UniformReal(0,\ 360) \\
&\quad \textbf{else} = GeoAzimuth(Location(event(a)), Depth(event(a)), phase(a), Site(s)) \\
&\qquad\qquad + Laplace(0, \sigma_a(s)) \\
&Slowness(a, s) \textbf{ if } (event(a) = null) \textbf{ then } \sim UniformReal(0, 20) \\
&\quad \textbf{else} = GeoSlowness(Location(event(a)), Depth(event(a)), phase(a), Site(s)) \\
&\qquad\qquad + Laplace(0, \sigma_s(s)) \\
&ObservedPhase(a, s) \sim CategoricalPhaseModel(phase(a))
\end{aligned}
$$

Figure 18.6 A simplified version of the NET-VISA model (see text).

Once trained, the model runs continuously. The evidence consists of detections (90% of which are false alarms) extracted from raw IMS waveform data, and the query typically asks for the most likely event history, or *bulletin*, given the data. Results so far are encouraging; for example, in 2009 the UN's SEL3 automated bulletin missed 27.4% of the 27294 events in the magnitude range 3-4 while NET-VISA missed 11.1%. Moreover, comparisons with dense regional networks show that NET-VISA finds up to 50% more real events than the final bulletins produced by the UN's expert seismic analysts. NET-VISA also tends to associate more detections with a given event, leading to more accurate location estimates (see Figure 18.7). As of January 1, 2018, NET-VISA has been deployed as part of the CTBTO monitoring pipeline.



(a) (b)

Figure 18.7 (a) Top: Example of seismic waveform recorded at Alice Springs, Australia. Bottom: the waveform after processing to detect the arrival times of seismic waves. Blue lines are the automatically detected arrivals; red lines are the true arrivals. (b) Location estimates for the DPRK nuclear test of February 12, 2013: UN CTBTO Late Event Bulletin (green triangle at top left); NET-VISA (blue square in center). The entrance to the underground test facility (small “x”) is 0.75km from NET-VISA’s estimate. Contours show NET-VISA’s posterior location distribution. Courtesy of CTBTO Preparatory Commission.

Despite superficial differences, the two examples are structurally similar: there are unknown objects (papers, earthquakes) that generate percepts according to some physical process (citation, seismic propagation). The percepts are ambiguous as to their origin, but when multiple percepts are hypothesized to have originated with the same unknown object, that object's properties can be inferred more accurately.

The same structure and reasoning patterns hold for areas such as database deduplication and natural language understanding. In some cases, inferring an object's existence involves grouping percepts together—a process that resembles the clustering task in machine learning. In other cases, an object may generate no percepts at all and still have its existence inferred— as happened, for example, when observations of Uranus led to the discovery of Neptune. The existence of the unobserved object follows from its effects on the behavior and properties of observed objects.

18.3 Keeping Track of a Complex World

Chapter 14 considered the problem of keeping track of the state of the world, but covered only the case of atomic representations (HMMs) and factored representations (DBNs and Kalman filters). This makes sense for worlds with a single object—perhaps a single patient in the intensive care unit or a single bird flying through the forest. In this section, we see what happens when two or more objects generate the observations. What makes this case different from plain old state estimation is that there is now the possibility of *uncertainty* about which object generated which observation. This is the **identity uncertainty** problem of Section 18.2 (page 648), now viewed in a temporal context. In the control theory literature, this is the **data association** problem—that is, the problem of associating observation data with the objects that generated them. Although we could view this as yet another example of open-universe probabilistic modeling, it is important enough in practice to deserve its own section.

18.3.1 Example: Multitarget tracking

The data association problem was studied originally in the context of radar tracking of multiple targets, where reflected pulses are detected at fixed time intervals by a rotating radar antenna. At each time step, multiple blips may appear on the screen, but there is no direct observation of which blips at time t correspond to which blips at time $t - 1$. Figure 18.8(a) shows a simple example with two blips per time step for five steps. Each blip is labeled with its time step but lacks any identifying information.

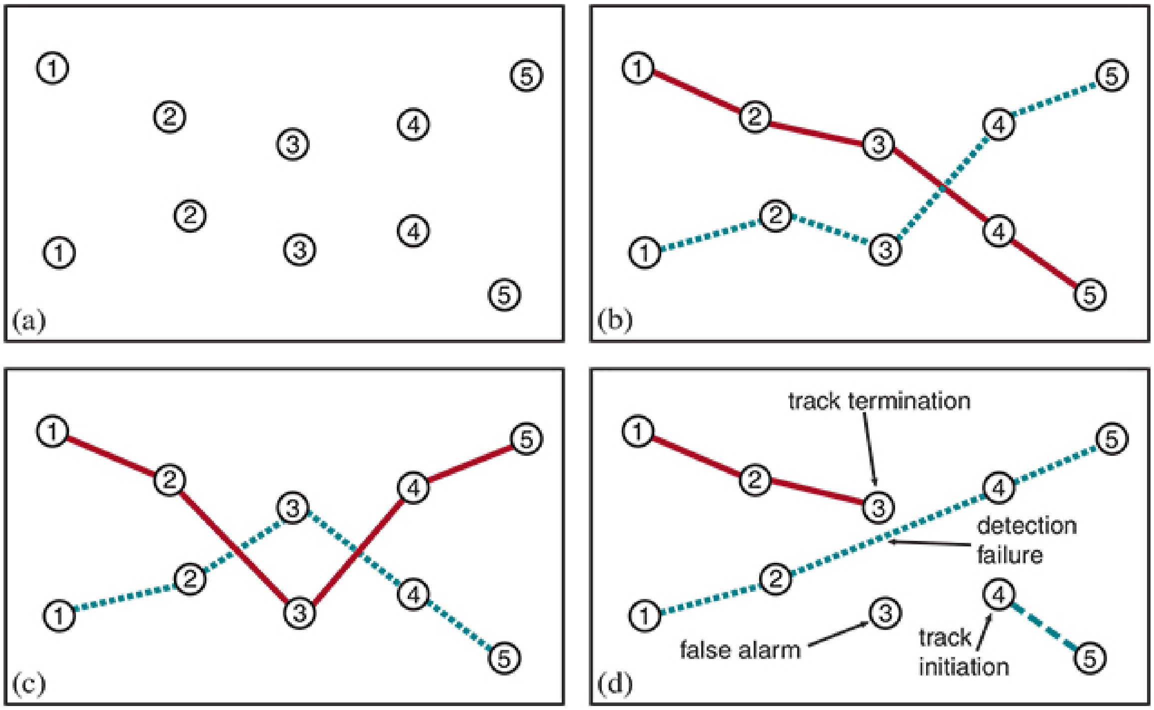


Figure 18.8 (a) Observations made of object locations in 2D space over five time steps. Each observation blip is labeled with the time step but does not identify the object that produced it. (b-c) Possible hypotheses about the underlying object tracks. (d) A hypothesis for the case in which false alarms, detection failures, and track initiation/termination are possible.

Let us assume, for the time being, that we know there are exactly two aircraft, A_1 and A_2, generating the blips. In the terminology of OUPMs, A_1 and A_2 are **guaranteed objects**, meaning that they are guaranteed to exist and to be distinct; moreover, in this case, there are no other objects. (In other words, as far as aircraft are concerned, this scenario matches the database semantics that is assumed in RPMs.) Let their true positions be

$X(A_1, t)$ and $X(A_2, t)$, where t is a nonnegative integer that indexes the sensor update times. We assume the first observation arrives at $t = 1$, and at time ϕ the prior distribution for every aircraft's location is $InitX()$. Just to keep things simple, we'll also assume that each aircraft moves independently according to a known transition model—e.g., a linear-Gaussian model as used in the Kalman filter (Section 14.4).

The final piece is the sensor model: again, we assume a linear-Gaussian model where an aircraft at position $\mathbf{x}$ produces a blip b whose observed blip position $Z(b)$ is a linear function of x with added Gaussian noise. Each aircraft generates exactly one blip at each time step, so the blip has as its origins an aircraft and a time step. So, omitting the prior for now, the model looks like this:

$$
\begin{aligned}
&\textbf{guaranteed } Aircraft\ A_1,\ A_2\\
&X(a,\ t)\ \sim\ \textbf{if } t\ =\ 0 \textbf{ then } InitX() \textbf{ else } N\ (\mathbf{F}\ X(a,\ t\ -1), \varSigma_x)\\
&\#Blip(Source = a,\ Time = t)\ =\ 1\\
&Z(b)\ \sim N(\mathbf{H}\, X\, Source(b), Time(b)), \varSigma_z)
\end{aligned}
$$

where $\mathbf{F}$ and Σ_x are matrices describing the linear transition model and transition noise covariance, and $\mathbf{H}$ and Σ_z are the corresponding matrices for the sensor model. (See page 501.)

The key difference between this model and a standard Kalman filter is that there are *two* objects producing sensor readings (blips). This means there is *uncertainty* at any given time step about which object produced which sensor reading. Each possible world in this model includes an association—defined by values of all the $Source(b)$ variables for all the time steps—between aircraft and blips. Two possible association hypotheses are shown in Figure 18.8(b-c). In general, for n objects and T time steps, there are $(n!)^T$ ways of assigning blips to aircraft—an awfully large number.

The scenario described so far involved n known objects generating n observations at each time step. Real applications of data association are typically much more complicated. Often, the reported observations include **false alarms** (also known as **clutter**), which are not caused by real objects. **Detection failures** can occur, meaning that no observation is reported for a real object. Finally, new objects arrive and old ones disappear. These phenomena, which create even more possible worlds to worry about, are illustrated in Figure 18.8(d). The corresponding OUPM is given in Figure 18.9.

$$
\begin{aligned}
&\#Aircraft(EntryTime=t) \sim Poisson(\lambda_a)\\
&Exits(a,t) \sim \textbf{if } InFlight(a,t) \textbf{ then } Boolean(\alpha_e)\\
&InFlight(a,t) = (t{=}EntryTime(a)) \lor (InFlight(a,t-1) \land \neg Exits(a,t-1))\\
&X(a,t) \sim \textbf{if } t = EntryTime(a) \textbf{ then } InitX()\\
&\quad \textbf{else if } InFlight(a,t) \textbf{ then } \mathcal{N}(\mathbf{F}X(a,t-1),\Sigma_x)\\
&\#Blip(Source{=}a,\ Time{=}t) \sim \textbf{if } InFlight(a,t) \textbf{ then } Bernoulli(DetectionProb(X(a,t)))\\
&\#Blip(Time{=}t) \sim Poisson(\lambda_f)\\
&Z(b) \sim \textbf{if } Source(b){=}null \textbf{ then } UniformZ(R) \textbf{ else } \mathcal{N}(\mathbf{H}X(Source(b),Time(b)),\Sigma_z)
\end{aligned}
$$

Figure 18.9 An OUPM for radar tracking of multiple targets with false alarms, detection failure, and entry and exit of aircraft. The rate at which new aircraft enter the scene is λ_a, while the probability per time step that an aircraft exits the scene is α_e. False alarm blips (i.e., ones not produced by an aircraft) appear uniformly in space at a rate of λ_f per time step. The probability that an aircraft is detected (i.e., produces a blip) depends on its current position.

Because of its practical importance for both civilian and military applications, tens of thousands of papers have been written on the problem of multitarget tracking and data association. Many of them simply try to work out the complex mathematical details of the probability calculations for the model in Figure 18.9, or for simpler versions of it. In one sense, this is unnecessary once the model is expressed in a probabilistic programming language, because the general-purpose inference engine does all of the mathematics correctly for any model—including this one. Furthermore, elaborations of the scenario (formation flying, objects heading for unknown destinations, objects taking off or landing, etc.) can be handled by small changes to the model without resorting to new mathematical derivations and complex programming.

From a practical point of view, the challenge with this kind of model is the complexity of inference. As for all probability models, inference means summing out the variables other than the query and the evidence. For filtering in HMMs and DBNs, we were able to sum out the state variables from 1 to $t-1$ by a simple dynamic programming trick; for Kalman filters, we also took advantage of special properties of Gaussians. For data association, we are less fortunate. There is no (known) efficient exact algorithm, for the same reason that there is none for the switching Kalman filter (page 502): the filtering distribution, which describes the joint distribution over numbers and locations of aircraft at each time step, ends up as a mixture of exponentially many distributions, one for each way of picking a sequence of observations to assign to each aircraft.

As a response to the complexity of exact inference, several approximate methods have been used. The simplest approach is to choose a single “best” assignment at each time step, given the predicted positions of the objects at the current time. This assignment associates observations with objects and enables the track of each object to be updated and a prediction made for the next time step. For choosing the “best” assignment, it is common to use the so-called **nearest-neighbor filter**, which repeatedly chooses the closest pairing of predicted position and observation and adds that pairing to the assignment. The nearest-neighbor filter works well when the objects are well separated in state space and the prediction uncertainty and observation error are small—in other words, when there is no possibility of confusion.

When there is more uncertainty as to the correct assignment, a better approach is to choose the assignment that maximizes the joint probability of the current observations given the predicted positions. This can be done efficiently using the **Hungarian algorithm** (Kuhn, 1955), even though there are $n!$ assignments to choose from as each new time step arrives.

Any method that commits to a single best assignment at each time step fails miserably under more difficult conditions. In particular, if the algorithm commits to an incorrect assignment, the prediction at the next time step may be significantly wrong, leading to more incorrect assignments, and so on. Sampling approaches can be much more effective. A **particle filtering** algorithm (see page 510) for data association works by maintaining a large collection of possible current assignments. An **MCMC** algorithm explores the space of assignment histories—for example, Figure 18.8(b-c) might be states in the MCMC state space—and can change its mind about previous assignment decisions.

One obvious way to speed up sampling-based inference for multitarget tracking is to use the **Rao-Blackwellization** trick from Chapter 14 (page 514): given a specific association hypothesis for all the objects, the filtering calculation for each object can typically be done exactly and efficiently, instead of sampling many possible state sequences for the objects. For example, with the model in Figure 18.9, the filtering calculation just means running a Kalman filter for the sequence of observations assigned to a given hypothesized object. Furthermore, when changing from one association hypothesis to another, the calculations have to be redone only for objects whose associated observations have changed. Current MCMC data association methods can handle many hundreds of objects in real time while giving a good approximation to the true posterior distributions.

18.3.2 Example: Traffic monitoring

Figure 18.10 shows two images from widely separated cameras on a California freeway. In this application, we are interested in two goals: estimating the time it takes, under current traffic conditions, to go from one place to another in the freeway system; and measuring *demand*—that is, how many vehicles travel between any two points in the system at particular times of the day and on particular days of the week. Both goals require solving the data association problem over a wide area with many cameras and tens of thousands of vehicles per hour.

(a) (b)

Figure 18.10 Images from (a) upstream and (b) downstream surveillance cameras roughly two miles apart on Highway 99 in Sacramento, California. The boxed vehicle has been identified at both cameras.

With visual surveillance, false alarms are caused by moving shadows, articulated vehicles, reflections in puddles, etc.; detection failures are caused by occlusion, fog, darkness, and lack of visual contrast; and vehicles are constantly entering and leaving the freeway system at points that may not be monitored. Furthermore, the appearance of any given vehicle can change dramatically between cameras depending on lighting conditions and vehicle pose in the image, and the transition model changes as traffic jams come and go. Finally, in dense traffic with widely separated cameras, the prediction error in the transition model for a car driving from one camera location to the next is far greater than the typical separation between vehicles. Despite these problems, modern data association algorithms have been successful in estimating traffic parameters in real-world settings.

Data association is an essential foundation for keeping track of a complex world, because without it there is no way to combine multiple observations of any given object. When objects in the world interact with each other in complex activities, understanding the world requires combining data association with the relational and open-universe probability models of Section 18.2. This is currently an active area of research.

18.4 Programs as Probability Models

Many probabilistic programming languages have been built on the insight that probability models can be defined using executable code in any programming language that incorporates a source of randomness. For such models, the possible worlds are execution traces and the probability of any such trace is the probability of the random choices required for that trace to happen. PPLs created in this way inherit all of the expressive power of the underlying language, including complex data structures, recursion, and, in some cases, higher-order functions. Many PPLs are in fact computationally universal: they can represent any probability distribution that can be sampled from by a probabilistic Turing machine that halts.

18.4.1 Example: Reading text

We illustrate this approach to probabilistic modeling and inference via the problem of writing a program that reads degraded text. These kinds of models can be built for reading text that has been smudged or blurred due to water damage, or spotted due to aging of the paper on which it is printed. They can also be built for breaking some kinds of CAPTCHAs.

Figure 18.11 shows a generative program containing two components: (i) a way to generate a sequence of letters; and (ii) a way to generate a noisy, blurry rendering of these letters using an off-the-shelf graphics library. Figure 18.12(top) shows example images generated by invoking GENERATE-IMAGE nine times.

```
function GENERATE-IMAGE() returns an image with some letters
   letters ← GENERATE-LETTERS(10)
   return RENDER-NOISY-IMAGE(letters, 32, 128)

function GENERATE-LETTERS(λ) returns a vector of letters
   n ∼ Poisson(λ)
   letters ← []
   for i = 1 to n do
       letters[i] ∼ UniformChoice({a, b, c, ···})
   return letters

function RENDER-NOISY-IMAGE(letters, width, height) returns a noisy image of the letters
   clean_image ← RENDER(letters, width, height, text_top = 10, text_left = 10)
   noisy_image ← []
   noise_variance ∼ UniformReal(0.1, 1)
   for row = 1 to width do
       for col = 1 to height do
           noisy_image[row, col] ∼ N(clean_image[row, col], noise_variance)
   return noisy_image
```

Figure 18.11 Generative program for an open-universe probability model for optical character recognition. The generative program produces degraded images containing sequences of letters by generating each sequence, rendering it into a 2D image, and incorporating additive noise at each pixel.

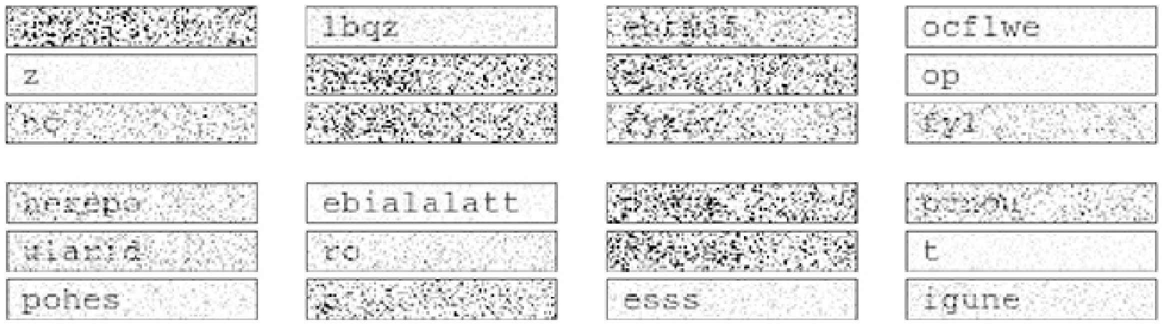


Figure 18.12 The top panel shows twelve degraded images produced by executing the generative program from Figure 18.11. The number of letters, their identities, the amount of additive noise, and the specific pixel-wise noise are all part of the domain of the probability model. The bottom panel shows twelve degraded images produced by executing the generative program from Figure 18.15. The Markov model for letters typically yields sequences of letters that are easier to pronounce.

18.4.2 Syntax and semantics

A **generative program** is an executable program in which every random choice defines a random variable in an associated probability model. Let us imagine unrolling the execution of a program that makes random choices, step by step. Let X_i be the random variable corresponding to the ith random choice made by the program; as usual, x_i denotes a possible value of X_i. Let us call $\omega = \{x_i\}$ an **execution trace** of the generative program—that is, a sequence of possible values for the random choices. Running the program once generates one such trace, hence the term "generative program."

The space of all possible execution traces Ω can be viewed as the sample space of a probability model defined by the generative program. The probability distribution over traces can be defined as the product of the probabilities of each individual random choice: $P(\omega) = \Pi_i P(x_i|x_i,...x_{i-1})$. This is analogous to the distribution over worlds in an OUPM.

It is conceptually straightforward to convert any OUPM into a corresponding generative program. This generative program makes random choices for each number statement and for the value of each basic random variable whose existence is implied by the number statements. The main extra work that the generative program needs to do is to create data structures that represent the objects, functions, and relations of the possible worlds in the OUPM. These data structures are created automatically by the OUPM inference engine because the OUPM assumes that every possible world is a first-order model structure, whereas a typical PPL makes no such assumption.

The images in Figure 18.12 can be used to get an intuitive understanding of the probability distribution $P(\Omega)$: we see varying levels of noise, and in the less noisy images, we also see sequences of letters of varying lengths. Let ω_1 be the trace corresponding to the image in the top right corner of this figure, containing the letters `ocflwe`. If we unrolled this trace ω_1 into a Bayesian network, it would have 4,104 nodes: 1 node for the variable n; 6 nodes for the variables *letters*[i]; 1 node for the *noise-variance;* and 4,096 nodes for the pixels in *noisy-image*. We thus see that this generative program defines an open-universe probability model: the number of random choices it makes is not bounded a priori, but instead depends on the value of the random variable n.

18.4.3 Inference results

Let's apply this model to interpret images of letters that have been degraded with additive noise. Figure 18.13 shows a degraded image, along with results from three independent MCMC runs. For each run, we show a rendering of the letters contained in the trace after stopping the Markov chain. In all three cases the result is the letter sequence `uncertainty`, suggesting that the posterior distribution is highly concentrated on the correct interpretation.

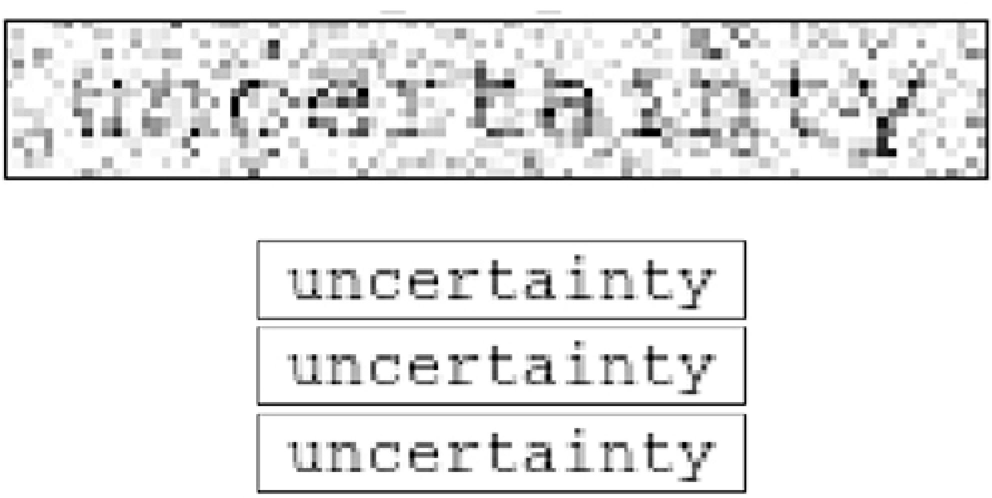


Figure 18.13 Noisy input image (top) and inference results (bottom) produced by three runs, each of 25 MCMC iterations, with the model from Figure 18.11. Note that the inference process correctly identifies the sequence of letters.

Now let's degrade the text further, blurring it enough that it is difficult for people to read. Figure 18.14 shows the inference results on this more challenging input. This time, although MCMC inference appears to have converged on (what we know to be) the correct number of letters, the first letter is misidentified as a q and there is uncertainty about five of the ten following letters.

Figure 18.14 Top: extremely noisy input image. Bottom left: with three inference results from 25 MCMC iterations with the independent-letter model from Figure 18.11. Bottom right: three inference results with the letter bigram model from Figure 18.15. Both models exhibit ambiguity in the results, but the latter model's results reflect prior knowledge of plausible letter sequences.

At this point, there are many possible ways to interpret the results. It could be that MCMC inference has mixed well and the results are a good reflection of the true posterior given the model and the image; in that case, the uncertainty in some of the letters and the error in the first letter are unavoidable. To get better results, we might need to improve the text model or reduce the noise level. It could also be that MCMC inference has not mixed properly: if we ran 300 chains for 25 thousand or 25 million iterations, we might find a quite different distribution of results, perhaps indicating that the first letter is probably u rather than q.

Running more inference could be costly in terms of dollars and waiting time. Moreover, there is no foolproof test for convergence of Monte Carlo inference methods. We could try to improve the inference algorithm, perhaps by designing a better proposal distribution for MCMC or using bottom-up clues from the image to suggest better initial hypotheses. These improvements require additional thought, implementation, and debugging. The third alternative is to improve the model. For example, we could incorporate knowledge about English words, such as the probabilities of letter pairs. We now consider this option.

18.4.4 Improving the generative program to incorporate a Markov model

Probabilistic programming languages are modular in a way that makes it easy to explore improvements to the underlying model. Figure 18.15 shows the generative program for an improved model that generates letters sequentially rather than independently. This generative program uses a Markov model that draws each letter given the previous letter, with transition probabilities estimated from a reference list of English words.

function GENERATE-MARKOV-LETTERS(λ) **returns** a vector of letters
  $n \sim Poisson(\lambda)$
  $letters \leftarrow []$
  $letter\_probs \leftarrow$ MARKOV-INITIAL()
  for $i = 1$ **to** n **do**
    $letters[i] \sim Categorical(letter\_probs)$
    $letter\_probs \leftarrow$ MARKOV-TRANSITION($letters[i]$)
  return $letters$

Figure 18.15 Generative program for an improved optical character recognition model that generates letters according to a letter bigram model whose pairwise letter frequencies are estimated from a list of English words.

Figure 18.12 shows twelve sampled images produced by this generative program. Notice that the letter sequences are significantly more English-like than those generated from the program in Figure 18.11. The right-hand panel in Figure 18.14 shows inference results from this Markov model applied to the high-noise image. The interpretations more closely match the generating trace, though there is still some uncertainty.

18.4.5 Inference in generative programs

As with OUPMs, exact inference in generative programs is usually prohibitively expensive or impossible. On the other hand, it is easy to see how to perform rejection sampling: run the program, keep just the traces that agree with the evidence, and count the different query answers found in those traces. Likelihood weighting is also straightforward: for each generated trace, keep track of the weight of the trace by multiplying all the probabilities of the values observed along the way.

Likelihood weighting works well only when the data are reasonably likely according to the model. In more difficult cases, MCMC is usually the method of choice. MCMC applied to probabilistic programs involves sampling and modifying execution traces. Many of the considerations arising with OUPMs also apply here; in addition, the algorithm has to be careful about modifications to an execution trace, such as changing the outcome of an if-statement, that may invalidate the remainder of the trace.

Further improvements in inference come from several lines of work. Some improvements can produce fundamental shifts in the class of problems that are tractable with a given PPL, even in principle; lifted inference, described earlier for RPMs, can have this effect. In many cases, generic MCMC is too slow, and special-purpose proposals are needed to enable the inference process to mix quickly.

An important focus of recent work in PPLs has been to make it easy for users to define and use such proposals so that the efficiency of PPL inference matches that of custom inference algorithms devised for specific models.

Many promising approaches are aimed at reducing the overhead of probabilistic inference. The compilation idea described for Bayes nets in Section 13.4.3 can be applied to inference in OUPMs and PPLs, and typically yields speedups of two to three orders of magnitude. There have also been proposals for *special-purpose hardware* for algorithms such as message-passing and MCMC. For example, Monte Carlo hardware exploits low-precision probability representations and massive fine-grained parallelism to deliver 100-10,000x improvements in speed and energy efficiency.

Methods based on learning can also give substantial improvements in speed. For example, **adaptive proposal distributions** can gradually learn how to generate MCMC proposals that are reasonably likely to be accepted and reasonably effective in exploring the probability landscape of the model to ensure rapid mixing. It is also possible to train deep learning models (see Chapter 22) to represent proposal distributions for importance sampling, using synthetic data that was generated from the underlying model.

In general, one expects that any formalism built on top of general programming languages will run up against the barrier of computability, and this is the case for PPLs. If we assume, however, that the underlying program halts for all inputs and all random choices, does the additional requirement of doing probabilistic inference still render the problem undecidable? It turns out that the answer is yes, but only for a computational model with infinite-precision continuous random variables. In that case, it becomes possible to write a computable probability model in which inference encodes the halting problem. On the other hand, with finite- precision numbers and with the smooth probability distributions typically used in real applications, inference remains decidable.

Summary

This chapter has explored expressive representations for probability models based on both logic and programs.

- **Relational probability models** (RPMs) define probability models on worlds derived from the **database semantics** for first-order languages; they are appropriate when all the objects and their identities are known with certainty.
- Given an RPM, the objects in each possible world correspond to the constant symbols in the RPM, and the basic random variables are all possible instantiations of the predicate symbols with objects replacing each argument. Thus, the set of possible worlds is finite.
- RPMs provide very concise models for worlds with large numbers of objects and can handle relational uncertainty.
- **Open-universe probability models** (OUPMs) build on the full semantics of first-order logic, allowing for new kinds of uncertainty such as identity and existence uncertainty.
- **Generative programs** are representations of probability models—including OUPMs— as executable programs in a **probabilistic programming language** or **PPL**. A generative program represents a distribution over **execution traces** of the program. PPLs typically provide *universal* expressive power for probability models.

Bibliographical and Historical Notes

Hailperin (1984) and Howson (2003) recount the long history of attempts to connect probability and logic, going back to Leibniz's *Nouveaux Essais* in 1704. These attempts usually involved probabilities attached directly to logical sentences. The first rigorous treatment was Gaifman's propositional **probability logic** (Gaifman, 1964b). The idea is that a probability assertion $P(\phi) \geq p$ is a constraint on the distribution over possible worlds, just as an ordinary logical sentence is a constraint on the possible worlds themselves. Any distribution P that satisfies the constraint is a model, in the standard logical sense, of the probability assertion, and one probability assertion entails another just when the models of the first are a subset of the models of the second.

Within such a logic, one can prove, for example, that $P(\alpha \wedge \beta) \leq P(\alpha \Rightarrow \beta)$ Satisfiability of sets of probability assertions can be determined in the propositional case by linear programming (Hailperin, 1984; Nilsson, 1986). Thus, we have a "probability logic" in the same sense as "temporal logic"—a logical system specialized for probabilistic reasoning.

To apply probability logic to tasks such as proving interesting theorems in probability theory, a more expressive language was needed. Gaifman (1964a) proposed a *first-order* probability logic, with possible worlds being first-order model structures and with probabilities attached to sentences of (function-free) first-order logic. Scott and Krauss (1966) extended Gaifman's results to allow infinite nesting of quantifiers and infinite sets of sentences.

Within AI, the most direct descendant of these ideas appears in **probabilistic logic programs** (Lukasiewicz, 1998), in which a probability range is attached to each first-order Horn clause and inference is performed by solving linear programs, as suggested by Hailperin. Halpern (1990) and Bacchus (1990) also built on Gaifman's approach, exploring some of the basic knowledge representation issues from the perspective of AI rather than probability theory and mathematical logic.

The subfield of **probabilistic databases** also has logical sentences labeled with probabilities (Dalvi *et al*., 2009)—but in this case probabilities are attached directly to the tuples of the database. (In AI and statistics, probability is attached to general relationships, whereas observations are viewed as incontrovertible evidence.) Although probabilistic databases can model complex dependencies, in practice one often finds such systems using global independence assumptions across tuples.

Attaching probabilities to sentences makes it very difficult to define complete and consistent probability models. Each inequality *constrains* the underlying probability model to lie in a half-space in the high-dimensional space of probability models. Conjoining assertions corresponds to intersecting the constraints. Ensuring that the intersection yields a single point is not easy. In fact, the principal result in Gaifman (1964a) is the construction of a single probability model requiring 1) a probability for every possible ground sentence and 2) probability constraints for infinitely many existentially quantified sentences.

One solution to this problem is to write a partial theory and then "complete" it by picking out one canonical model in the allowed set. Nilsson (1986) proposed choosing the *maximum entropy* model consistent with the specified constraints. Paskin (2002) developed a "maximum-entropy probabilistic logic" with constraints expressed as weights (relative probabilities) attached to first-order clauses. Such models are often called **Markov logic networks** or MLNs (Richardson and Domingos, 2006) and have become a popular technique for applications involving relational data. Maximum-entropy approaches, including MLNs, can produce unintuitive results in some cases (Milch, 2006; Jain *et al.,* 2007, 2010).

Beginning in the early 1990s, researchers working on complex applications noticed the expressive limitations of Bayesian networks and developed various languages for writing "templates" with logical variables, from which large networks could be constructed automatically for each problem instance (Breese, 1992; Wellman *et al.,* 1992). The most important such language was B UGS(Bayesian inference Using Gibbs Sampling) (Gilks *et al.,* 1994; Lunn *et al.,* 2013), which combined Bayesian networks with the **indexed random variable** notation common in statistics. (In BUGS, an indexed random variable looks like $X[i]$, where i has a defined integer range.)

These closed-universe languages inherited the key property of Bayesian networks: every well-formed knowledge base defines a unique, consistent probability model. Other closed- universe languages drew on the representational and inferential capabilities of logic programming (Poole, 1993; Sato and Kameya, 1997; Kersting *et al.*, 2000) and semantic networks (Koller and Pfeffer, 1998; Pfeffer, 2000).

Research on open-universe probability models has several origins. In statistics, the problem of **record linkage** arises when data records do not contain standard unique identifiers— for example, various citations of this book might name its first author "Stuart J. Russell" or "S. Russell" or even "Stewart Russel." Other authors share the name "S. Russell."

Hundreds of companies exist solely to solve record linkage problems in financial, medical, census, and other data. Probabilistic analysis goes back to work by Dunn (1946); the Fellegi-Sunter model (1969), which is essentially naive Bayes applied to matching, still dominates current practice. Identity uncertainty is also considered in multitarget tracking (Sittler, 1964), whose history is sketched in Chapter 14.

In AI, the working assumption until the 1990s was that sensors could supply logical sentences with unique identifiers for objects, as was the case with Shakey. In the area of natural language understanding, Charniak and Goldman (1992) proposed a probabilistic analysis of coreference, where two linguistic expressions (say, "Obama" and "the president") may refer to the same entity. Huang and Russell (1998) and Pasula *et al.* (1999) developed a Bayesian analysis of identity uncertainty for traffic surveillance. Pasula *et al.* (2003) developed a complex generative model for authors, papers, and citation strings, involving both relational and identity uncertainty, and demonstrated high accuracy for citation information extraction.

The first formal language for open-universe probability models was B LOG(Milch *et al.*,2005; Milch, 2006), which came with a (very slow) general-purpose MCMC inference engine. Laskey (2008) describes another open-universe modeling language called **multi-entity Bayesian networks**. The NET-VISA global seismic monitoring system described in the text is due to Arora et *al.* (2013). The Elo rating system was developed in 1959 by Arpad Elo (1978) but is essentially the same at Thurstone's Case V model (Thurstone, 1927). Microsoft's TrueSkill model (Herbrich et *al.*, 2007; Minka et *al.*, 2018) is based on Mark Glick- man's (1999) Bayesian version of Elo and now runs on the infer.NET PPL.

Data association for multitarget tracking was first described in a probabilistic setting by Sittler (1964). The first practical algorithm for large-scale problems was the "multiple hypothesis tracker" or MHT algorithm

(Reid, 1979). Important papers are collected by Bar- Shalom and Fortmann (1988) and Bar-Shalom (1992). The development of an MCMC algorithm for data association is due to Pasula *et al.* (1999), who applied it to traffic surveillance problems. oh *et al.* (2009) provide a formal analysis and experimental comparisons to other methods. Schulz *et al.* (2003) describe a data association method based on particle filtering.

Ingemar Cox analyzed the complexity of data association (Cox, 1993; Cox and Hingo- rani, 1994) and brought the topic to the attention of the vision community. He also noted the applicability of the polynomial-time Hungarian algorithm to the problem of finding most- likely assignments, which had long been considered an intractable problem in the tracking community. The algorithm itself was published by Kuhn (1955), based on translations of papers published in 1931 by two Hungarian mathematicians, Dénes König and Jenö Egervàry. The basic theorem had been derived previously, however, in an unpublished Latin manuscript by the famous mathematician Carl Gustav Jacobi (1804-1851).

The idea that probabilistic programs could also represent complex probability models is due to Koller *et al.* (1997). The first working PPL was Avi Pfeffer’s IBAL(2001, 2007), based on a simple functional language. B LOGcan be thought of as a declarative PPL. The connection between declarative and functional PPLs was explored by McAllester *et al.* (2008). C HURCH(Goodman *et al.*, 2008), a PPL built on the Scheme language, pioneered the idea of piggybacking on an existing programming language. CHURCH also introduced the first MCMC inference algorithm for models with random higher-order functions and generated interest in the cognitive science community as a way to model complex forms of human learning (Lake *et al.*, 2015). PPLs also connect in interesting ways to computability theory (Ackerman *et al.*, 2013) and programming language research.

In the 2010s, dozens of PPLs emerged based on a wide range of underlying programming languages. Figaro, based on the Scala language, has been used for a wide variety of applications (Pfeffer, 2016). Gen (Cusumano-Towner *et al.*, 2019), based on Julia and TensorFlow, has been used for real-time machine perception as well as Bayesian structure learning for time series data analysis. PPLs built on top of deep learning frameworks include Pyro (Bingham *et al.*, 2019) (built on PyTorch) and Edward (Tran *et al.*, 2017) (built on TensorFlow).

There have been efforts to make probabilistic programming accessible to more people, such as database and spreadsheet users. Tabular (Gordon *et al*., 2014) provides a spreadsheetlike relational schema language on top of infer.NET. BayesDB (Saad and Mansinghka, 2017) lets users combine and query probabilistic programs using an SQL-like language.

Inference in probabilistic programs has generally relied on approximate methods because exact algorithms do not scale to the kinds of models that PPLs can represent. Closed-universe languages such as BUGS, LIBBI (Murray, 2013), and STAN (Carpenter *et al.*, 2017) generally operate by constructing the full equivalent Bayesian network and then running inference on it—Gibbs sampling in the case of BUGS, sequential Monte Carlo in the case of LIBBI, and Hamiltonian Monte Carlo in the case of STAN. Programs in these languages can be read as instructions for building the ground Bayes net. Breese (1992) showed how to generate only the relevant fragment of the full network, given the query and the evidence.

Working with a grounded Bayes net means that the possible worlds visited by MCMC are represented by a vector of values for variables in the Bayes net. The idea of directly sampling first-order possible worlds is due to Russell (1999). In the F ACTORIElanguage (McCallum et *al.*, 2009), possible worlds in the MCMC process are represented within a standard relational database system. The same two papers propose incremental query reevaluation as a way to avoid full query evaluation on each possible world.

Inference methods based on grounding are analogous to the earliest propositionalization methods for first-order logical inference (Davis and Putnam, 1960). For logical inference, both resolution theorem provers and logic programming systems rely on **lifting** (Section 9.2) to avoid instantiating logical variables unnecessarily.

Pfeffer *et al.* (1999) introduced a variable elimination algorithm that cached each computed factor for reuse by later computations involving the same relations but different objects, thereby realizing some of the computational gains of lifting. The first truly lifted probabilistic inference algorithm was a form of variable elimination described by Poole (2003) and subsequently improved by de Salvo Braz et *al.* (2007). Further advances, including cases where certain aggregate probabilities can be computed in closed form, are described by Milch *et al.*(2008) and Kisynski and Poole (2009). There is now a fairly good understanding of when lifting is possible and of its complexity (Gribkoff et *al.*, 2014; Kazemi et *al.*, 2017).

Methods of speeding up inference come in several flavors, as noted in the chapter. Several projects have explored more sophisticated algorithms, combined with compiler techniques and/or learned proposals. LibBi (Murray, 2013) introduced the first particle Gibbs inference for probabilistic programs; one of the first inference compilers, with GPU support for massively parallel SMC; and use of the modeling language to define custom MCMC proposals. Compilation of probabilistic inference is also studied by Wingate et *al.* (2011), Paige and Wood (2014), Wu et *al.* (2016a). Claret et *al.* (2013), Hur et *al.* (2014), and Cusumano- Towner et *al.* (2019) demonstrate static analysis methods for transforming probabilistic programs into more efficient forms. Picture (Kulkarni *et al.* , 2015) is the first PPL that let users apply learning from forward executions of the generative program to train fast bottom-up proposals. Le *et al.* (2017) describe the use of deep learning techniques for efficient importance sampling in a PPL. In practice, inference algorithms for complex probability models often use a mixture of techniques for different subsets of variables in the model. Mansinghka *et al.*(2013) emphasized the idea of inference programs that apply diverse inference tactics to subsets of variables chosen during inference runtime.

The collection edited by Getoor and Taskar (2007) includes many important papers on first-order probability models and their use in machine learning. Probabilistic programming papers appear in all the major conferences on machine learning and probabilistic reasoning, including NeurlPS, ICML, UAI, and AISTATS. Regular PPL workshops have been attached to the NeurlPS and POPL (Principles of Programming Languages) conferences, and the first International Conference on Probabilistic Programming was held in 2018.

[1] The name *relational probability model* was given by Pfeffer (2000) to a slightly different representation, but the underlying ideas are the same.

[2] A game theorist would advise a dishonest customer to avoid detection by occasionally recommending a good book from a competitor. See Chapter 17.

[3] Some technical conditions are required for an RPM to define a proper distribution. First, the dependencies must be *acyclic;* otherwise the resulting Bayesian network will have cycles. Second, the dependencies must (usually) be *well-founded:* there can be no infinite ancestor chains, such as might arise from recursive dependencies. See Exercise 18.HAMD for an exception to this rule.

[4] Several statistical packages would view this code as *defining* the RPM, rather than just constructing a Bayes net to perform inference in the RPM. This view, however, misses an important

role for RPM syntax: without a syntax with clear semantics, there is no way the model structure can be learned from data.

[5] The name “Sybil” comes from a famous case of multiple personality disorder.

[6] A self-supporting instantiation of a set of variables is one in which the parents of every variable in the set are also in the set.

[7] As with Prolog, there may be infinitely many sets of substitutions of unbounded size; designing exploratory interfaces for such answers is an interesting visualization challenge.

[8] The answer is yes, they are the same paper. The “National Conference on Articial Intelligence” (notice how the “fi” is missing, thanks to an error in scraping the ligature character) is another name for the AAAI conference; the conference took place in Seattle whereas the proceedings publisher is in Cambridge.

[9] The multi-author case has the same overall structure but is a bit more complicated. The parts of the model not shown—the *NamePrior, rTitlePrior,* and *HMMGrammar*—are traditional probability models. For example, the *NamePrior* is a mixture of a categorical distribution over actual names and a letter trigram model (see Section 24.1) to cover names not previously seen, both trained from data in the U.S. Census database.

CHAPTER 19

LEARNING FROM EXAMPLES

In which we describe agents that can improve their behavior through diligent study of past experiences and predictions about the future.

An agent is **learning** if it improves its performance after making observations about the world. Learning can range from the trivial, such as jotting down a shopping list, to the profound, as when Albert Einstein inferred a new theory of the universe. When the agent is a computer, we call it **machine learning**: a computer observes some data, builds a **model** based on the data, and uses the model as both a **hypothesis** about the world and a piece of software that can solve problems.

Why would we want a machine to learn? Why not just program it the right way to begin with? There are two main reasons. First, the designers cannot anticipate all possible future situations. For example, a robot designed to navigate mazes must learn the layout of each new maze it encounters; a program for predicting stock market prices must learn to adapt when conditions change from boom to bust. Second, sometimes the designers have no idea how to program a solution themselves. Most people are good at recognizing the faces of family members, but they do it subconsciously, so even the best programmers don’t know how to program a computer to accomplish that task, except by using machine learning algorithms.

In this chapter, we interleave a discussion of various model classes—decision trees (Section 19.3), linear models (Section 19.6), nonparametric models such as nearest neighbors (Section 19.7), ensemble models such as random forests (Section 19.8)—with practical advice on building machine learning systems (Section 19.9), and discussion of the theory of machine learning (Sections 19.1 to 19.5).

19.1 Forms of Learning

Any component of an agent program can be improved by machine learning. The improvements, and the techniques used to make them, depend on these factors:

- Which *component* is to be improved.
- What *prior knowledge* the agent has, which influences the *model* it builds.
- What *data* and *feedback* on that data is available.

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

1. A direct mapping from conditions on the current state to actions.
2. A means to infer relevant properties of the world from the percept sequence.
3. Information about the way the world evolves and about the results of possible actions the agent can take.
4. Utility information indicating the desirability of world states.
5. Action-value information indicating the desirability of actions.
6. Goals that describe the most desirable states.
7. A problem generator, critic, and learning element that enable the system to improve.

Each of these components can be learned. Consider a self-driving car agent that learns by observing a human driver. Every time the driver brakes, the agent might learn a condition-action rule for when to brake (component 1). By seeing many camera images that it is told contain buses, it can learn to recognize them (component 2). By trying actions and observing the results

—for example, braking hard on a wet road—it can learn the effects of its actions (component 3). Then, when it receives complaints from passengers who have been thoroughly shaken up during the trip, it can learn a useful component of its overall utility function (component 4).

The technology of machine learning has become a standard part of software engineering. Any time you are building a software system, even if you don't think of it as an AI agent, components of the system can potentially be improved with machine learning. For example, software to analyze images of galaxies under gravitational lensing was speeded up by a factor of 10 million with a machine-learned model (Hezaveh *et al.*, 2017), and energy use for cooling data centers was reduced by 40% with another machine-learned model (Gao, 2014). Turing Award winner David Patterson and Google AI head Jeff Dean declared the dawn of a "Golden Age" for computer architecture due to machine learning (Dean *et al.*, 2018).

We have seen several examples of models for agent components: atomic, factored, and relational models based on logic or probability, and so on. Learning algorithms have been devised for all of these.

This chapter assumes little **prior knowledge** on the part of the agent: it starts from scratch and learns from the data. In Section 22.7.2 we consider **transfer learning**, in which knowledge from one domain is transferred to a new domain, so that learning can proceed faster with less data. We do assume, however, that the designer of the system chooses a model framework that can lead to effective learning.

Going from a specific set of observations to a general rule is called **induction**; from the observations that the sun rose every day in the past, we induce that the sun will come up tomorrow. This differs from the **deduction** we studied in Chapter 7 because the inductive conclusions may be incorrect, whereas deductive conclusions are guaranteed to be correct if the premises are correct.

This chapter concentrates on problems where the input is a **factored representation**—a vector of attribute values. It is also possible for the input to be any kind of data structure, including atomic and relational.

When the output is one of a finite set of values (such as *sunny/cloudy/rainy* or *true/false),* the learning problem is called **classification**. When it is a number (such as tomorrow's temperature, measured either as an integer or a real number), the learning problem has the (admittedly obscure[1]) name **regression**.

There are three types of **feedback** that can accompany the inputs, and that determine the three main types of learning:

- In **supervised learning** the agent observes input-output pairs and learns a function that maps from input to output. For example, the inputs could be camera images, each one accompanied by an output saying "bus" or "pedestrian," etc. An output like this is called a **label**. The agent learns a function that, when given a new image, predicts the appropriate label. In the case of braking actions (component 1 above), an input is the current state (speed and direction of the car, road condition), and an output is the distance it took to stop. In this case a set of output values can be obtained by the agent from its own percepts (after the fact); the environment is the teacher, and the agent learns a function that maps states to stopping distance.
- In **unsupervised learning** the agent learns patterns in the input without any explicit feedback. The most common unsupervised learning task is **clustering**: detecting potentially useful clusters of input examples. For example, when shown millions of images taken

from the Internet, a computer vision system can identify a large cluster of similar images which an English speaker would call “cats.”

- In **reinforcement learning** the agent learns from a series of reinforcements: rewards and punishments. For example, at the end of a chess game the agent is told that it has won (a reward) or lost (a punishment). It is up to the agent to decide which of the actions prior to the reinforcement were most responsible for it, and to alter its actions to aim towards more rewards in the future.

19.2 Supervised Learning

More formally, the task of supervised learning is this:

> Given a **training set** of *N* example input-output pairs
>
> $$(x_1, y_1),\ (x_2, y_2), \ldots\ (x_N, y_N).$$
>
> where each pair was generated by an unknown function $y = f(x)$, discover a function *h* that approximates the true function *f*.

The function *h* is called a **hypothesis** about the world. It is drawn from a **hypothesis space** $\mathcal{H}$ of possible functions. For example, the hypothesis space might be the set of polynomials of degree 3; or the set of Javascript functions; or the set of 3-SAT Boolean logic formulas.

With alternative vocabulary, we can say that *h* is a **model** of the data, drawn from a **model class** $\mathcal{H}$, or we can say a **function** drawn from a **function class**. We call the output y_i the **ground truth**—the true answer we are asking our model to predict.

How do we choose a hypothesis space? We might have some prior knowledge about the process that generated the data. If not, we can perform **exploratory data analysis**: examining the data with statistical tests and visualizations—histograms, scatter plots, box plots—to get a feel for the data, and some insight into what hypothesis space might be appropriate. Or we can just try multiple hypothesis spaces and evaluate which one works best.

How do we choose a good hypothesis from within the hypothesis space? We could hope for a **consistent hypothesis**: an *h* such that each x_i, in the training set has $h(x_i) = y_i$. With continuous-valued outputs we can't expect an exact match to the ground truth; instead we look for a **best-fit**

function for which each $h(x_i)$ is close to y_i (in a way that we will formalize in Section 19.4.2).

The true measure of a hypothesis is not how it does on the training set, but rather how well it handles inputs it has not yet seen. We can evaluate that with a second sample of (x_i, y_i) pairs called a **test set**. We say that h **generalizes** well if it accurately predicts the outputs of the test set.

Figure 19.1 shows that the function h that a learning algorithm discovers depends on the hypothesis space $\mathcal{H}$ it considers and on the training set it is given. Each of the four plots in the top row have the same training set of 13 data points in the (x, y) plane. The four plots in the bottom row have a second set of 13 data points; both sets are representative of the same unknown function $f(x)$. Each column shows the best-fit hypothesis h from a different hypothesis space:

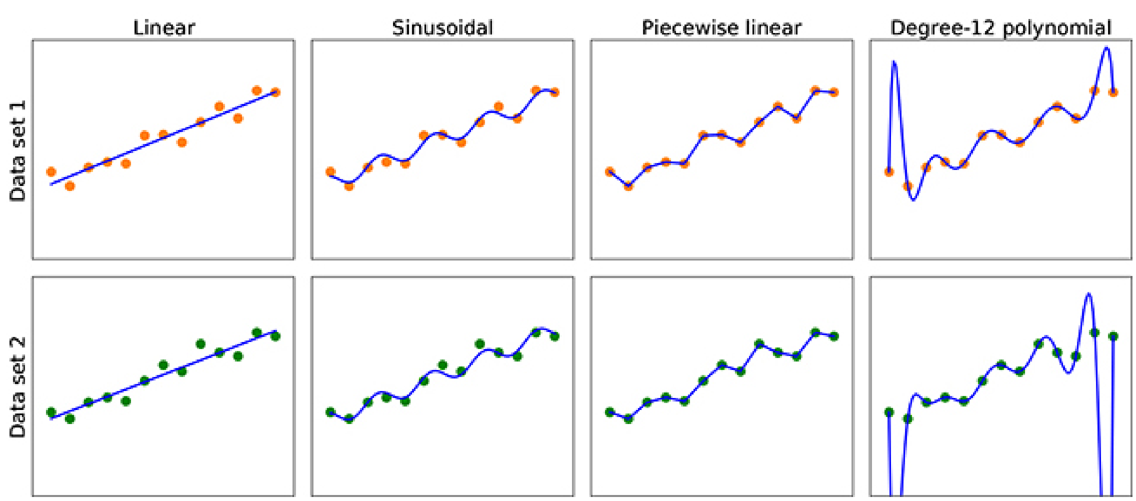


Figure 19.1 Finding hypotheses to fit data. **Top row**: four plots of best-fit functions from four different hypothesis spaces trained on data set 1. **Bottom row**: the same four functions, but trained

on a slightly different data set (sampled from the same $f(x)$ function).

- **Column 1** : Straight lines; functions of the form $h(x) = w_1x + w_0$. There is no line that would be a consistent hypothesis for the data points.
- **Column 2** : Sinusoidal functions of the form $h(x) = w_1x + \sin(w_0x)$. This choice is not quite consistent, but fits both data sets very well.
- **Column 3** : Piecewise-linear functions where each line segment connects the dots from one data point to the next. These functions are always consistent.
- **Column 4** : Degree-12 polynomials, $h(x) = \Sigma_{i=0}^{12} w_i x^i$. These are consistent: we can always get a degree-12 polynomial to perfectly fit 13 distinct points. But just because the hypothesis is consistent does not mean it is a good guess.

One way to analyze hypothesis spaces is by the bias they impose (regardless of the training data set) and the variance they produce (from one training set to another).

By **bias** we mean (loosely) the tendency of a predictive hypothesis to deviate from the expected value when averaged over different training sets. Bias often results from restrictions imposed by the hypothesis space. For example, the hypothesis space of linear functions induces a strong bias: it only allows functions consisting of straight lines. If there are any patterns in the data other than the overall slope of a line, a linear function will not be able to represent those patterns. We say that a hypothesis is **underfitting** when it fails to find a pattern in the data. On the other hand, the piecewise linear function has low bias; the shape of the function is driven by the data.

By **variance** we mean the amount of change in the hypothesis due to fluctuation in the training data. The two rows of Figure 19.1 represent data sets that were each sampled from the same $f(x)$ function. The data sets turned out to be slightly different. For the first three columns, the small difference in the data set translates into a small difference in the hypothesis. We call that low variance. But the degree-12 polynomials in the fourth column have high variance: look how different the two functions are at both ends of the x-axis. Clearly, at least one of these polynomials must be a poor approximation to the true $f(x)$. We say a function is **overfitting** the data when it pays too much attention to the particular data set it is trained on, causing it to perform poorly on unseen data.

Often there is a **bias–variance tradeoff**: a choice between more complex, low-bias hypotheses that fit the training data well and simpler, low-variance hypotheses that may generalize better. Albert Einstein said in 1933, "the supreme goal of all theory is to make the irreducible basic elements as simple and as few as possible without having to surrender the adequate representation of a single datum of experience." In other words, Einstein recommends choosing the simplest hypothesis that matches the data. This principle can be traced further back to the 14th-century English philosopher William of Ockham.[2] His principle that "plurality [of entities] should not be posited without necessity" is called **Ockham's razor** because it is used to "shave off" dubious explanations.

Defining simplicity is not easy. It seems clear that a polynomial with only two parameters is simpler than one with thirteen parameters. We will make this intuition more precise in Section 19.3.4. However, in Chapter 22 we will see that deep neural network models can often generalize quite well, even though they are very complex—some of them have billions of parameters. So the number of parameters by itself is not a good measure of a model's fitness. Perhaps we should be aiming for "appropriateness," not "simplicity" in a model class. We will consider this issue in Section 19.4.1.

Which hypothesis is best in Figure 19.1? We can't be certain. If we knew the data represented, say, the number of hits to a Web site that grows from day to day, but also cycles depending on the time of day, then we might favor the sinusoidal function. If we knew the data was definitely not cyclic but had high noise, that would favor the linear function.

In some cases, an analyst is willing to say not just that a hypothesis is possible or impossible, but rather how probable it is. Supervised learning can be done by choosing the hypothesis h* that is most probable given the data:

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

By Bayes' rule this is equivalent to

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

Then we can say that the prior probability $P(h)$ is high for a smooth degree-1 or -2 polynomial and lower for a degree-12 polynomial with large, sharp spikes. We allow unusual-looking functions when the data say we really need them, but we discourage them by giving them a low prior probability.

Why not let $\mathcal{H}$ be the class of all computer programs, or all Turing machines? The problem is that *there is a tradeoff between the expressiveness of a hypothesis space and the computational complexity of finding a good hypothesis within that space*. For example, fitting a straight line to data is an easy computation; fitting high-degree polynomials is somewhat harder; and fitting Turing machines is undecidable. A second reason to prefer simple hypothesis spaces is that presumably we will want to use h after we have learned it, and computing $h(x)$ when h is a linear function is guaranteed to be fast, while computing an arbitrary Turing machine program is not even guaranteed to terminate.

For these reasons, most work on learning has focused on simple representations. In recent years there has been great interest in deep learning (Chapter 22), where representations are not simple but where the $h(x)$ computation still takes only a *bounded number of steps* to compute with appropriate hardware.

We will see that the expressiveness–complexity tradeoff is not simple: it is often the case, as we saw with first-order logic in Chapter 8, that an expressive language makes it possible for a *simple* hypothesis to fit the data, whereas restricting the expressiveness of the language means that any consistent hypothesis must be complex.

19.2.1 Example problem: Restaurant waiting

We will describe a sample supervised learning problem in detail: the problem of deciding whether to wait for a table at a restaurant. This problem will be used throughout the chapter to demonstrate different model classes. For this problem the output, y, is a Boolean variable that we will call *WillWait;* it is true for examples where we do wait for a table. The input, x, is a vector of ten attribute values, each of which has discrete values:

1. *Alternate:* whether there is a suitable alternative restaurant nearby.
2. *Bar*: whether the restaurant has a comfortable bar area to wait in.
3. *Fri/Sat:* true on Fridays and Saturdays.
4. *Hungry:* whether we are hungry right now.
5. *Patrons:* how many people are in the restaurant (values are *None*, *Some*, and *Full)*.

6. *Price:* the restaurant's price range ($,

,

$).
7. *Raining:* whether it is raining outside.
8. *Reservation*: whether we made a reservation.
9. *Type:* the kind of restaurant (French, Italian, Thai, or burger).
10. *WaitEstimate:* host's wait estimate: 0–10, 10–30, 30–60, or > 60 minutes.

A set of 12 examples, taken from the experience of one of us (SR), is shown in Figure 19.2. Note how skimpy these data are: there are $2^6 \times 3^2 \times 4^2 =$ 9,216 possible combinations of values for the input attributes, but we are given the correct output for only 12 of them; each of the other 9,204 could be either true or false; we don't know. This is the essence of induction: we need to make our best guess at these missing 9,204 output values, given only the evidence of the 12 examples.

Example	Input Attributes										Output
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
$\mathbf{x}_1$	*Yes*	*No*	*No*	*Yes*	*Some*	*$$$*	*No*	*Yes*	*French*	*0–10*	$y_1 = Yes$
$\mathbf{x}_2$	*Yes*	*No*	*No*	*Yes*	*Full*	*$*	*No*	*No*	*Thai*	*30–60*	$y_2 = No$
$\mathbf{x}_3$	*No*	*Yes*	*No*	*No*	*Some*	*$*	*No*	*No*	*Burger*	*0–10*	$y_3 = Yes$
$\mathbf{x}_4$	*Yes*	*No*	*Yes*	*Yes*	*Full*	*$*	*Yes*	*No*	*Thai*	*10–30*	$y_4 = Yes$
$\mathbf{x}_5$	*Yes*	*No*	*Yes*	*No*	*Full*	*$$$*	*No*	*Yes*	*French*	*>60*	$y_5 = No$
$\mathbf{x}_6$	*No*	*Yes*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Italian*	*0–10*	$y_6 = Yes$
$\mathbf{x}_7$	*No*	*Yes*	*No*	*No*	*None*	*$*	*Yes*	*No*	*Burger*	*0–10*	$y_7 = No$
$\mathbf{x}_8$	*No*	*No*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Thai*	*0–10*	$y_8 = Yes$
$\mathbf{x}_9$	*No*	*Yes*	*Yes*	*No*	*Full*	*$*	*Yes*	*No*	*Burger*	*>60*	$y_9 = No$
$\mathbf{x}_{10}$	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$$$*	*No*	*Yes*	*Italian*	*10–30*	$y_{10} = No$
$\mathbf{x}_{11}$	*No*	*No*	*No*	*No*	*None*	*$*	*No*	*No*	*Thai*	*0–10*	$y_{11} = No$
$\mathbf{x}_{12}$	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$*	*No*	*No*	*Burger*	*30–60*	$y_{12} = Yes$

Figure 19.2 Examples for the restaurant domain.

19.3 Learning Decision Trees

A **decision tree** is a representation of a function that maps a vector of attribute values to a single output value—a "decision." A decision tree reaches its decision by performing a sequence of tests, starting at the root and following the appropriate branch until a leaf is reached. Each internal node in the tree corresponds to a test of the value of one of the input attributes, the branches from the node are labeled with the possible values of the attribute, and the leaf nodes specify what value is to be returned by the function.

In general, the input and output values can be discrete or continuous, but for now we will consider only inputs consisting of discrete values and outputs that are either *true* (a **positive** example) or *false* (a **negative** example). We call this **Boolean classification**. We will use j to index the examples ($\mathbf{x}_j$ is the input vector for the jth example and y_j is the output), and $x_{j,i}$ for the ith attribute of the jth example.

The tree representing the decision function that SR uses for the restaurant problem is shown in Figure 19.3. Following the branches, we see that an example with *Patrons* = *Full* and *WaitEstimate* = 0–10 will be classified as positive (i.e., yes, we will wait for a table).

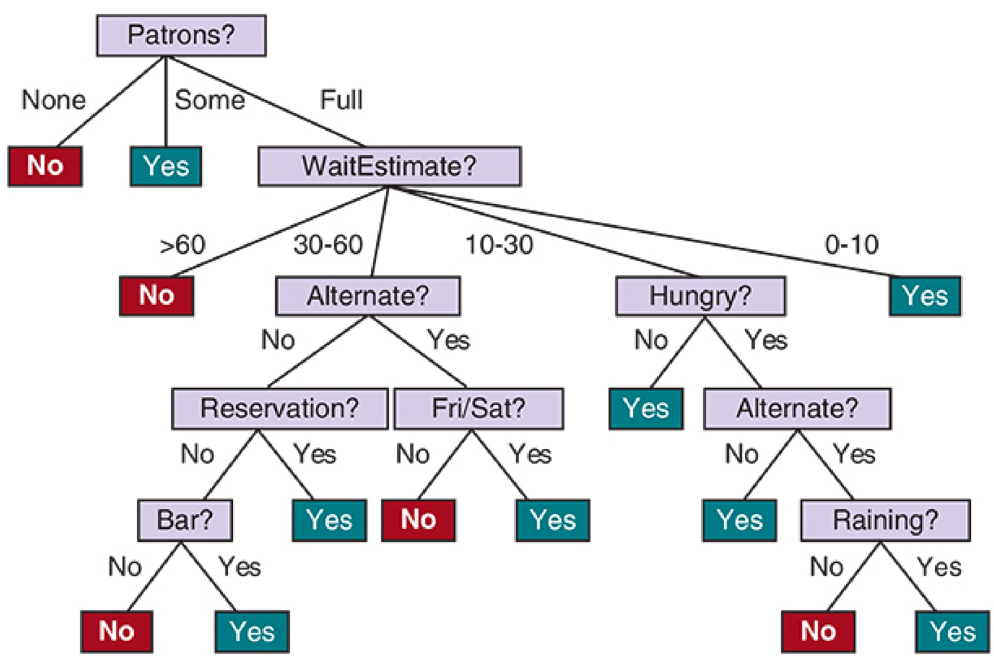


Figure 19.3 A decision tree for deciding whether to wait for a table.

19.3.1 Expressiveness of decision trees

A Boolean decision tree is equivalent to a logical statement of the form:

$$Output \Leftrightarrow (Path_1 \vee Path_2 \vee \cdots),$$

where each $Path_i$ is a conjunction of the form $(A_m = v_x \wedge A_n = v_y \wedge \cdots)$ of attribute-value tests corresponding to a path from the root to a *true* leaf. Thus, the whole expression is in disjunctive normal form, which means that any function in propositional logic can be expressed as a decision tree.

For many problems, the decision tree format yields a nice, concise, understandable result. Indeed, many "How To" manuals (e.g., for car repair) are written as decision trees. But some functions cannot be represented concisely. For example, the majority function, which returns true if and only if more than half of the inputs are true, requires an exponentially large decision tree, as does the parity function, which returns true if and only if an even number of input attributes are true. With real-valued attributes, the function $y > A_1 + A_2$ is hard to represent with a decision tree because the decision boundary is a diagonal line, and all decision tree tests divide the space up into rectangular, axis-aligned boxes. We would have to stack a lot of boxes to closely approximate the diagonal line. In other words, decision trees are good for some kinds of functions and bad for others.

Is there *any* kind of representation that is efficient for *all* kinds of functions? Unfortunately, the answer is no—there are just too many functions to be able to represent them all with a small number of bits. Even just considering Boolean functions with n Boolean attributes, the truth table will have 2^n rows, and each row can output *true* or *false*, so there are 2^{2^n} different functions. With 20 attributes there are $2^{1,048,576} \approx 10^{300,000}$ functions, so if we limit ourselves to a million-bit representation, we can't represent all these functions.

19.3.2 Learning decision trees from examples

We want to find a tree that is consistent with the examples in Figure 19.2 and is as small as possible. Unfortunately, it is intractable to find a guaranteed smallest consistent tree. But with some simple heuristics, we can efficiently find one that is close to the smallest. The LEARN-DECISION-TREE algorithm adopts a greedy divide-and-conquer strategy: always test the most important attribute first, then recursively solve the smaller subproblems that are defined by the possible results of the test. By "most important attribute," we mean the one that makes the most difference to the classification of an example. That way, we hope to get to the correct classification with a small number of tests, meaning that all paths in the tree will be short and the tree as a whole will be shallow.

Figure 19.4(a) shows that *Type* is a poor attribute, because it leaves us with four possible outcomes, each of which has the same number of positive as negative examples. On the other hand, in (b) we see that *Patrons* is a fairly important attribute, because if the value is *None* or *Some*, then we are left with example sets for which we can answer definitively (*No* and *Yes*, respectively). If the value is *Full*, we are left with a mixed set of examples. There are four cases to consider for these recursive subproblems:

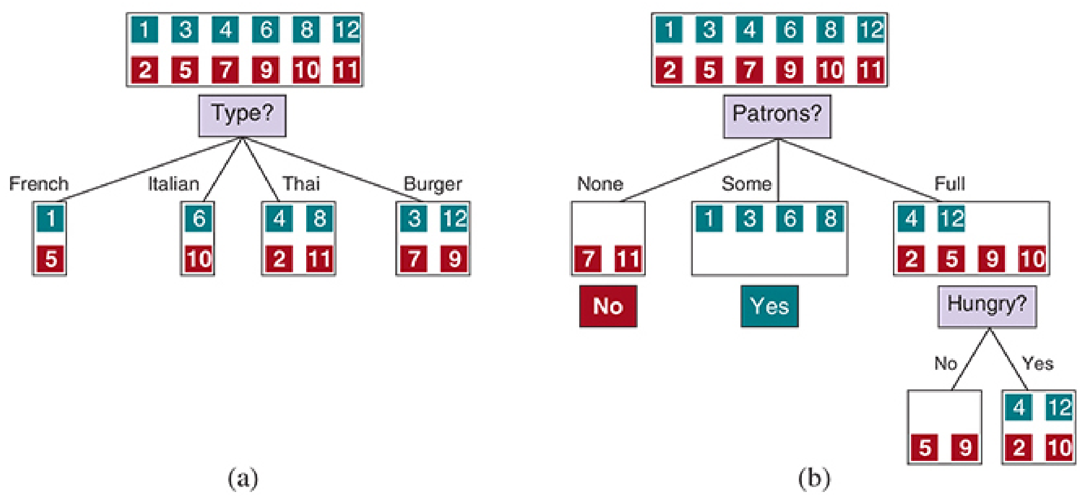


Figure 19.4 Splitting the examples by testing on attributes. At each node we show the positive (light boxes) and negative (dark boxes) examples remaining. (a) Splitting on *Type* brings us no nearer to distinguishing between positive and negative examples. (b) Splitting on *Patrons* does a good job of separating positive and negative examples. After splitting on *Patrons*, *Hungry* is a fairly good second test.

1. If the remaining examples are all positive (or all negative), then we are done: we can answer *Yes* or *No*. Figure 19.4(b) shows examples of this happening in the *None* and *Some* branches.

2. If there are some positive and some negative examples, then choose the best attribute to split them. Figure 19.4(b) shows *Hungry* being used to split the remaining examples.
3. If there are no examples left, it means that no example has been observed for this combination of attribute values, and we return the most common output value from the set of examples that were used in constructing the node's parent.
4. If there are no attributes left, but both positive and negative examples, it means that these examples have exactly the same description, but different classifications. This can happen because there is an error or **noise** in the data; because the domain is nondeterministic; or because we can't observe an attribute that would distinguish the examples. The best we can do is return the most common output value of the remaining examples.

The LEARN-DECISION-TREE algorithm is shown in Figure 19.5. Note that the set of examples is an input to the algorithm, but nowhere do the examples appear in the tree returned by the algorithm. A tree consists of tests on attributes in the interior nodes, values of attributes on the branches, and output values on the leaf nodes. The details of the IMPORTANCE function are given in Section 19.3.3. The output of the learning algorithm on our sample training set is shown in Figure 19.6. The tree is clearly different from the original tree shown in Figure 19.3. One might conclude that the learning algorithm is not doing a very good job of learning the correct function. This would be the wrong conclusion to draw, however. The learning algorithm looks at the *examples*, not at the correct function, and in fact, its hypothesis (see Figure 19.6) not only is consistent with all the examples, but is considerably simpler than the original tree! With slightly different examples the tree might be very different, but the function it represents would be similar.

```
function LEARN-DECISION-TREE(examples, attributes, parent_examples) returns a tree

  if examples is empty then return PLURALITY-VALUE(parent_examples)
  else if all examples have the same classification then return the classification
  else if attributes is empty then return PLURALITY-VALUE(examples)
  else
     A ← argmax_{a ∈ attributes} IMPORTANCE(a, examples)
     tree ← a new decision tree with root test A
     for each value v of A do
        exs ← {e : e ∈ examples and e.A = v}
        subtree ← LEARN-DECISION-TREE(exs, attributes − A, examples)
        add a branch to tree with label (A = v) and subtree subtree
     return tree
```

Figure 19.5 The decision tree learning algorithm. The function IMPORTANCE is described in Section 19.3.3. The function PLURALITY-VALUE selects the most common output value among a set of examples, breaking ties randomly.

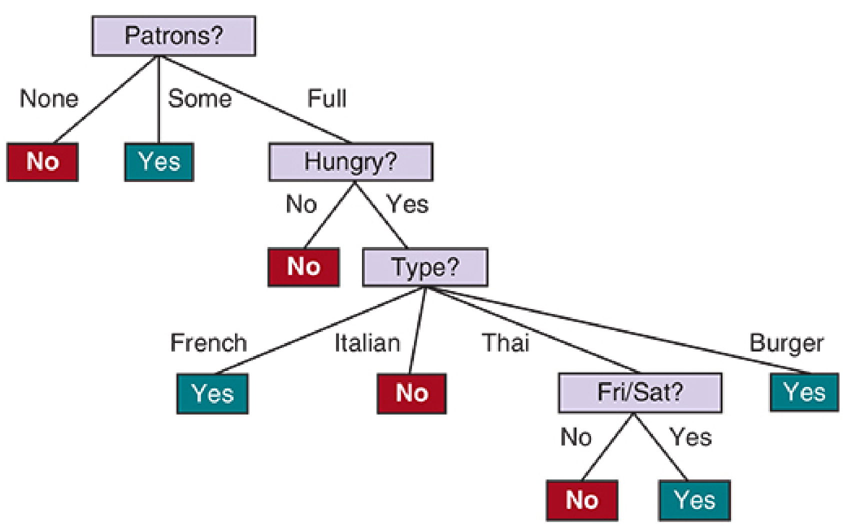


Figure 19.6 The decision tree induced from the 12-example training set.

The learning algorithm has no reason to include tests for *Raining* and *Reservation*, because it can classify all the examples without them. It has also detected an interesting and previously unsuspected pattern: SR will wait for Thai food on weekends. It is also bound to make some mistakes for cases where it has seen no examples. For example, it has never seen a case where the wait is 0–10 minutes but the restaurant is full. In that case it says not to wait when *Hungry* is false, but SR would certainly wait. With more training examples the learning program could correct this mistake.

We can evaluate the performance of a learning algorithm with a **learning curve**, as shown in Figure 19.7. For this Figure we have 100 examples at our disposal, which we split randomly into a training set and a test set. We learn a hypothesis h with the training set and measure its accuracy with the test set. We can do this starting with a training set of size 1 and increasing one at a time up to size 99. For each size, we actually repeat the process of randomly splitting into training and test sets 20 times, and average the results of the 20 trials. The curve shows that as the training set size grows, the accuracy increases. (For this reason, learning curves are also called **happy graphs**.) In this graph we reach 95% accuracy, and it looks as if the curve might continue to increase if we had more data.

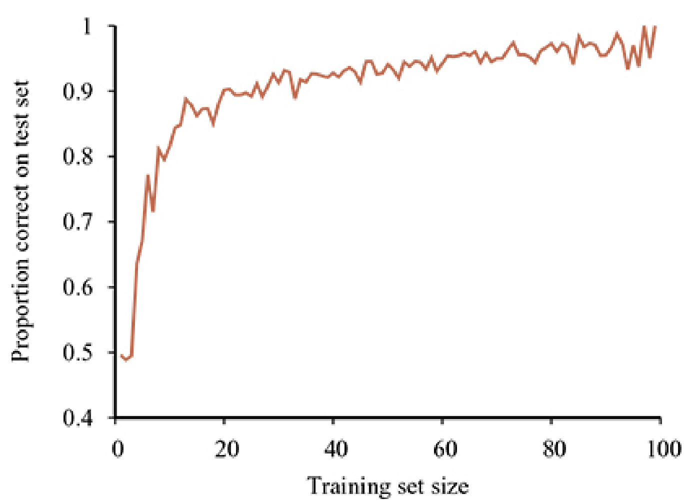


Figure 19.7 A learning curve for the decision tree learning algorithm on 100 randomly generated examples in the restaurant domain. Each data point is the average of 20 trials.

19.3.3 Choosing attribute tests

The decision tree learning algorithm chooses the attribute with the highest IMPORTANCE. We will now show how to measure importance, using the notion of information gain, which is defined in terms of **entropy**, which is the fundamental quantity in information theory (Shannon and Weaver, 1949).

Entropy is a measure of the uncertainty of a random variable; the more information, the less entropy. A random variable with only one possible value—a coin that always comes up heads—has no uncertainty and thus its entropy is defined as zero. A fair coin is equally likely to come up heads or tails when flipped, and we will soon show that this counts as "1 bit" of entropy. The roll of a fair *four*-sided die has 2 bits of entropy, because there are 2^2 equally probable choices. Now consider an unfair coin that comes up heads 99% of the time. Intuitively, this coin has less uncertainty than the fair coin—if we guess heads we'll be wrong only 1% of the time—so we would like it to have an entropy measure that is close to zero, but positive. In general, the entropy of a random variable V with values v_k having probability $P(v_k)$ is defined as

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

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

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

And of a four-sided die is 2 bits:

$$H(Die4) = -(0.25 \log_2 0.25 + 0.25 \log_2 0.25 + 0.25 \log_2 0.25 + 0.25 \log_2 0.25) = 2$$

For the loaded coin with 99% heads, we get

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

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

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

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

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

The restaurant training set in Figure 19.2 has $p = n = 6$, so the corresponding entropy is $B(0.5)$ or exactly 1 bit. The result of a test on an attribute A will give us some information, thus reducing the overall entropy by some amount. We can measure this reduction by looking at the entropy remaining after the attribute test.

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

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

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

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

In fact $Gain(A)$ is just what we need to implement the IMPORTANCE function. Returning to the attributes considered in Figure 19.4, we have

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

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

19.3.4 Generalization and overfitting

We want our learning algorithms to find a hypothesis that fits the training data, but more importantly, we want it to generalize well for previously unseen data. In Figure 19.1 we saw that a high-degree polynomial can fit all the data, but has wild swings that are not warranted by the data: it fits but can overfit. Overfitting becomes more likely as the number of attributes grows, and less likely as we increase the number of training examples. Larger hypothesis spaces (e.g., decision trees with more nodes or polynomials with high degree) have more capacity both to fit and to overfit; some model classes are more prone to overfitting than others.

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

The question is, how do we detect that a node is testing an irrelevant attribute? Suppose we are at a node consisting of p positive and n negative examples. If the attribute is irrelevant, we would expect that it would split the examples into subsets such that each subset has roughly the same proportion of positive examples as the whole set, $p/(p + n)$, and so the information gain will be close to zero.[3] Thus, a low information gain is a good clue that the attribute is irrelevant. Now the question is, how large a gain should we require in order to split on a particular attribute?

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

In this case, the null hypothesis is that the attribute is irrelevant and, hence, that the information gain for an infinitely large sample would be zero. We need to calculate the probability that, under the null hypothesis, a sample of size $v = n + p$ would exhibit the observed deviation from the expected distribution of positive and negative examples. We can measure the deviation by comparing the actual numbers of positive and negative examples in each subset, p_k and n_k, with the expected numbers, $\widehat{p}_k$ and $\widehat{n}_k$, assuming true irrelevance:

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

A convenient measure of the total deviation is given by

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

Under the null hypothesis, the value of Δ is distributed according to the χ^2 (chi-squared) distribution with $d - 1$ degrees of freedom. We can use a χ^2 statistics function to see if a particular A value confirms or rejects the null hypothesis. For example, consider the restaurant *Type* attribute, with four values and thus three degrees of freedom. A value of $\Delta = 7.82$ or more would reject the null hypothesis at the 5% level (and a value of $\Delta = 11.35$ or more would reject at the 1% level). Values below that lead to accepting the hypothesis that the attribute is irrelevant, and thus the associated branch of the tree should be pruned away. This is known as χ^2 **pruning**.

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

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

19.3.5 Broadening the applicability of decision trees

Decision trees can be made more widely useful by handling the following complications:

- **Missing data** : In many domains, not all the attribute values will be known for every example. The values might have gone unrecorded, or they might be too expensive to obtain. This gives rise to two problems: First, given a complete decision tree, how should one classify an example that is missing one of the test attributes? Second, how should one modify the information-gain formula when some examples have unknown values for the attribute? These questions are addressed in Exercise 19.MISS.
- **Continuous and multivalued input attributes** : For continuous attributes like *Height*, *Weight*, or *Time*, it may be that every example has a different attribute value. The information gain measure would give its highest score to such an attribute, giving us a shallow tree with this attribute at the root, and single-example subtrees for each possible

value below it. But that doesn't help when we get a new example to classify with an attribute value that we haven't seen before.

A better way to deal with continuous values is a **split point** test—an inequality test on the value of an attribute. For example, at a given node in the tree, it might be the case that testing on *Weight* > 160 gives the most information. Efficient methods exist for finding good split points: start by sorting the values of the attribute, and then consider only split points that are between two examples in sorted order that have different classifications, while keeping track of the running totals of positive and negative examples on each side of the split point. Splitting is the most expensive part of real-world decision tree learning applications.

For attributes that are not continuous and do not have a meaningful ordering, but have a large number of possible values (e.g., *Zipcode* or *CreditCardNumber*), a measure called the **information gain ratio** (see Exercise 19.GAIN) can be used to avoid splitting into lots of single-example subtrees. Another useful approach is to allow an **equality test** of the form $A = v_k$. For example, the test *Zipcode* = 10002 could be used to pick out a large group of people in this zip code in New York City, and to lump everyone else into the "other" subtree.

- **Continuous-valued output attribute**: If we are trying to predict a numerical output value, such as the price of an apartment, then we need a **regression tree** rather than a classification tree. A regression tree has at each leaf a linear function of some subset of numerical attributes, rather than a single output value. For example, the branch for two-bedroom apartments might end with a linear function of square footage and number of bathrooms. The learning algorithm must decide when to stop splitting and begin applying linear regression (see Section 19.6) over the attributes. The name **CART**, standing for Classification And Regression Trees, is used to cover both classes.

A decision tree learning system for real-world applications must be able to handle all of these problems. Handling continuous-valued variables is especially important, because both physical and financial processes provide numerical data. Several commercial packages have been built that meet these criteria, and they have been used to develop thousands of fielded systems. In many areas of industry and commerce, decision trees are the first method tried when a classification method is to be extracted from a data set.

Decision trees have a lot going for them: ease of understanding, scalability to large data sets, and versatility in handling discrete and continuous inputs as well as classification and regression. However, they can have suboptimal accuracy (largely due to the greedy search), and if trees are very deep, then getting a prediction for a new example can be expensive in run time. Decision trees are also **unstable** in that adding just one new example can change the test at the root, which changes the entire tree. In Section 19.8.2 we will see that the **random forest model** can fix some of these issues.

19.4 Model Selection and Optimization

Our goal in machine learning is to select a hypothesis that will optimally fit future examples. To make that precise we need to define "future example" and "optimal fit."

First we will make the assumption that the future examples will be like the past. We call this the **stationarity** assumption; without it, all bets are off. We assume that each example E_j has the same prior probability distribution:

$$\mathbf{P}(E_j) = \mathbf{P}(E_{j+1}) = \mathbf{P}(E_{j+2}) = \cdots ,$$

and is independent of the previous examples:

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

Examples that satisfy these equations are *independent and identically distributed* or **i.i.d.**.

The next step is to define "optimal fit." For now, we will say that the optimal fit is the hypothesis that minimizes the **error rate**: the proportion of times that $h(x) \neq y$ for an (x, y) example. (Later we will expand on this to allow different errors to have different costs, in effect giving partial credit for answers that are "almost" correct.) We can estimate the error rate of a hypothesis by giving it a test: measure its performance on a **test set** of examples. It would be cheating for a hypothesis (or a student) to peek at the test answers before taking the test. The simplest way to ensure this doesn't happen is to split the examples you have into two sets: a **training set** to create the hypothesis, and a **test set** to evaluate it.

If we are only going to create one hypothesis, then this approach is sufficient. But often we will end up creating multiple hypotheses: we might want to compare two completely different machine learning models, or we might want to adjust the various “knobs” within one model. For example, we could try different thresholds for χ^2 pruning of decision trees, or different degrees for polynomials. We call these “knobs” **hyperparameters**—parameters of the model class, not of the individual model.

Suppose a researcher generates a hypotheses for one setting of the χ^2 pruning hyperparameter, measures the error rates on the test set, and then tries different hyperparameters. No individual hypothesis has peeked at the test set data, but the overall *process* did, through the researcher.

The way to avoid this is to *really* hold out the test set—lock it away until you are completely done with training, experimenting, hyperparameter-tuning, re-training, etc. That means you need *three* data sets:

1. A **training set** to train candidate models.
2. A **validation set**, also known as a **development set** or **dev set**, to evaluate the candidate models and choose the best one.
3. A **test set** to do a final unbiased evaluation of the best model.

What if we don’t have enough data to make all three of these data sets? We can squeeze more out of the data using a technique called ***k*-fold cross-validation**. The idea is that each example serves double duty—as training data and validation data—but not at the same time. First we split the data into k equal subsets. We then perform k rounds of learning; on each round $1/k$ of the data are held out as a validation set and the remaining examples are used as the training set. The average test set score of the k rounds should then be a better estimate than a single score. Popular values for k are 5 and 10—enough to give an estimate that is statistically likely to be accurate, at a cost of 5 to 10 times longer computation time. The extreme is $k = n$, also known as **leave-one-out cross-validation** or **LOOCV**. Even with cross-validation, we still need a separate test set.

In Figure 19.1 (page 672) we saw a linear function underfit the data set, and a high-degree polynomial overfit the data. We can think of the task of finding a good hypothesis as two subtasks: **model selection**[4] chooses a good hypothesis space, and **optimization** (also called **training**) finds the best hypothesis within that space.

Part of model selection is qualitative and subjective: we might select polynomials rather than decision trees based on something that we know about the problem. And part is quantitative and empirical: within the class of polynomials, we might select $Degree = 2$, because that value performs best on the validation data set.

19.4.1 Model selection

Figure 19.8 describes a simple MODEL-SELECTION algorithm. It takes as argument a learning algorithm, *Learner* (for example, it could be LEARN-DECISION-TREE). *Learner* takes one hyperparameter, which is named *size* in the figure. For decision trees it could be the number of nodes in the tree; for polynomials *size* would be *Degree*. MODEL-SELECTION starts with the smallest value of *size*, yielding a simple model (which will probably underfit the data) and iterates through larger values of *size*, considering more complex models. In the end MODEL-SELECTION selects the model that has the lowest average error rate on the held-out validation data.

```
function MODEL-SELECTION(Learner, examples, k) returns a (hypothesis, error rate) pair

  err ← an array, indexed by size, storing validation-set error rates
  training_set, test_set ← a partition of examples into two sets
  for size = 1 to ∞ do
      err[size] ← CROSS-VALIDATION(Learner, size, training_set, k)
      if err is starting to increase significantly then
          best_size ← the value of size with minimum err[size]
          h ← Learner(best_size, training_set)
          return h, ERROR-RATE(h, test_set)

function CROSS-VALIDATION(Learner, size, examples, k) returns error rate

  N ← the number of examples
  errs ← 0
  for i = 1 to k do
     validation_set ← examples[(i − 1) × N/k:i × N/k]
     training_set ← examples − validation_set
     h ← Learner(size, training_set)
     errs ← errs + ERROR-RATE(h, validation_set)
  return errs / k          // average error rate on validation sets, across k-fold cross-validation
```

Figure 19.8 An algorithm to select the model that has the lowest validation error. It builds models of increasing complexity, and choosing the one with best empirical error rate, *err*, on the validation data set. *Learner*(*size*, *examples*) returns a hypothesis whose complexity is set by the parameter *size*, and which is trained on *examples*. In CROSS-VALIDATION, each iteration of the **for** loop selects a different slice of the *examples* as the validation set, and keeps the other examples as the training set. It then returns the average validation set error over all the folds. Once we have determined which value of the *size* parameter is best, MODEL-SELECTION returns the model (i.e., learner/hypothesis) of that size, trained on all the training examples, along with its error rate on the held-out test examples.

In Figure 19.9 we see two typical patterns that occur in model selection. In both (a) and (b) the training set error decreases monotonically (with slight random fluctuation) as we increase the complexity of the model. Complexity is measured by the number of decision tree nodes in (a) and by the number of neural network parameters (w_i) in (b). For many model classes, the training set error reaches zero as the complexity increases.

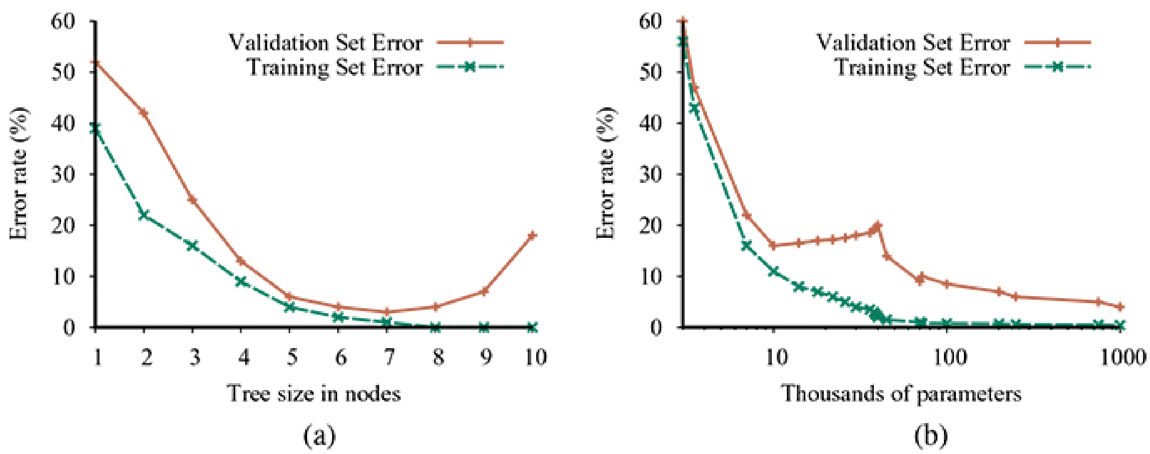


Figure 19.9 Error rates on training data (lower, green line) and validation data (upper, orange line) for models of different complexity on two different problems. MODEL-SELECTION picks the hyperparameter value with the lowest validation-set error. In (a) the model class is decision trees and the hyperparameter is the number of nodes. The data is from a version of the restaurant problem. The optimal size is 7. In (b) the model class is

convolutional neural networks (see Section 22.3) and the hyperparameter is the number of regular parameters in the network. The data is the MNIST data set of images of digits; the task is to identify each digit. The optimal number of parameters is 1,000,000 (note the log scale).

The two cases differ markedly in validation set error. In (a) we see a U-shaped validation-error curve: error decreases for a while as model complexity increases, but then we reach a point where the model begins to overfit, and validation error rises. MODEL-SELECTION picks the value at the bottom of the U-shaped validation-error curve: in this case a tree with size 7. This is the spot that best balances underfitting and overfitting. In (b) we see an initial U-shaped curve just as in (a) but then the validation error starts to decrease again; the lowest validation error rate is the final point in the plot, with 1,000,000 parameters.

Why are some validation-error curves like (a) and some like (b)? It comes down to how the different model classes make use of excess capacity, and how well that matches up with the problem at hand. As we add capacity to a model class, we often reach the point where all the training examples can be represented perfectly within the model. For example, given a training set with n distinct examples, there is always a decision tree with n leaf nodes that can represent all the examples.

We say that a model that exactly fits all the training data has **interpolated** the data.[5] Model classes typically start to overfit as the capacity approaches the point of interpolation. That seems to be because most of the model's capacity is concentrated on the training examples, and the capacity that remains is allocated rather randomly in a way that is not representative of the patterns in the validation data set. Some model classes never recover from this overfitting, as with the decision trees in (a). But for other model classes, adding capacity means that there are more candidate functions, and some of them are naturally well-suited to the patterns of data that are in the true function $f(x)$. The higher the capacity, the more of these suitable representations there are, and the more likely that the optimization mechanism will be able to land on one.

Deep neural networks (Chapter 22), kernel machines (Section 19.7.5), random forests (Section 19.8.2), and boosted ensembles (Section 19.8.4) all have the property that their validation set error tends to decrease as capacity increases, as in Figure 19.9(b).

We could extend the model selection algorithm in various ways: we could compare disparate model classes, by calling MODEL-SELECTION with DECISION-TREE-LEARNER as an argument and then with POLYNOMIAL-LEARNER, and seeing which does better. We could allow multiple hyperparameters, which means we would need a more complex optimization algorithm, such as a grid search (see Section 19.9.3) rather than a linear search.

19.4.2 From error rates to loss

So far, we have been trying to minimize error rate. This is clearly better than maximizing error rate, but it is not the full story. Consider the problem of classifying email messages as spam or non-spam. it is worse to classify non-spam as spam (and thus potentially miss an important message) than to classify spam as non-spam (and thus suffer a few seconds of annoyance). So a classifier with a 1% error rate, where almost all the errors were classifying spam as nonspam, would be better than a classifier with only a 0.5% error rate, if most of those errors were classifying non-spam as spam. We saw in Chapter 15 that decision makers should maximize expected utility, and utility is what learners should maximize as well. However, in machine learning it is traditional to express this as a negative: to minimize a **loss function** rather than maximize a utility function. The loss function $L(x, y, \hat{y})$ is defined as the amount of utility lost by predicting $h(x) = \hat{y}$ when the correct answer is $f(x) = y$:

$$\begin{aligned} L(x, y, \hat{y}) &= Utility(\text{result of using } y \text{ given an input } x) \\ &= Utility(\text{result of using } \hat{y} \text{ given an input } x) \end{aligned}$$

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

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

Note that $L(y, y)$ is always zero; by definition there is no loss when you guess exactly right. For functions with discrete outputs, we can enumerate a loss value for each possible misclassification, but we can't enumerate all the possibilities for real-valued data. If $f(x)$ is 137.035999, we would be fairly happy with $h(x) = 137.036$, but just how happy should we be? In general, small errors are better than large ones; two functions that implement that idea are the absolute value of the difference (called the L_1 loss), and the square of the difference (called the L_2 loss; think "2" for square). For discrete-valued outputs, if we are content with the idea of minimizing error rate, we can use the $L_{0/1}$ loss function, which has a loss of 1 for an incorrect answer:

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

Theoretically, the learning agent maximizes its expected utility by choosing the hypothesis that minimizes expected loss over all input-output pairs it will see. To compute this expectation we need to define a prior probability distribution **P**(*X, Y*) over examples. Let *ε* be the set of all possible input-output examples. Then the expected **generalization loss** for a hypothesis *h* (with respect to loss function *L*) is

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

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

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

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

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

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

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

There are four reasons why $\widehat{h}^*$ may differ from the true function, *f*: unrealizability, variance, noise, and computational complexity.

First, we say that a learning problem is **realizable** if the hypothesis space $\mathcal{H}$ actually contains the true function *f*. If $\mathcal{H}$ is the set of linear functions, and the true function *f* is a quadratic function, then no amount of data will recover the true *f*. Second, **variance** means that a learning algorithm will in general return different hypotheses for different sets of examples. If the problem is realizable, then variance decreases towards zero as the number of training examples increases. Third, *f* may be nondeterministic or **noisy**—it may return different values of $f(x)$ for the same x. By definition, noise cannot be predicted (it can only be characterized). And finally, when $\mathcal{H}$ is a complicated function in a large hypothesis space, it can be **computationally intractable** to systematically search all possibilities; in that case, a search can explore part of the space and return a reasonably good hypothesis, but can't always guarantee the best one.

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

In recent years there has been more emphasis on **large-scale learning**, with millions of examples. Here the generalization loss may be dominated by limits of computation: there are enough data and a rich enough model that we could find an *h* that is very close to the true *f*, but the computation to find it is complex, so we settle for an approximation.

19.4.3 Regularization

In Section 19.4.1, we saw how to do model selection with cross-validation. An alternative approach is to search for a hypothesis that directly minimizes the weighted sum of empirical loss and the complexity of the hypothesis, which we will call the total cost:

$$\begin{aligned} Cost(h) &= EmpLoss(h) + \lambda\, Complexity(h) \\ \widehat{h}^* &= \operatorname*{argmin}_{h \in \mathscr{H}} Cost(h). \end{aligned}$$

Here λ is a hyperparameter, a positive number that serves as a conversion rate between loss and hypothesis complexity. If λ is chosen well, it nicely balances the empirical loss of a simple function against a complicated function's tendency to overfit.

This process of explicitly penalizing complex hypotheses is called **regularization**: we're looking for functions that are more regular. Note that we are now making two choices: the loss function (L_1 or L_2), and the complexity measure, which is called a **regularization function**. The choice of regularization function depends on the hypothesis space. For example, for polynomials, a good regularization function is the sum of the squares of the coefficients— keeping the sum small would guide us away from the wiggly degree-12 polynomial in Figure 19.1. We will show an example of this type of regularization in Section 19.6.3.

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

It is in fact possible to have the empirical loss and the complexity measured on the same scale, without the conversion factor λ: they can both be measured in bits. First encode the hypothesis as a Turing machine program, and count the number of bits. Then count the number of bits required to encode the data, where a correctly predicted example costs zero bits and the cost of an incorrectly predicted example depends on how large the error is. The **minimum description length** or MDL hypothesis minimizes the total number of bits required. This works well in the limit, but for smaller problems the choice of encoding for the program—how best to encode a decision tree as a bit string—affects the outcome. In Chapter 21 (page 775), we describe a probabilistic interpretation of the MDL approach.

19.4.4 Hyperparameter tuning

In Section 19.4.1 we showed how to select the best value of the hyperparameter *size* by applying cross-validation to each possible value until the validation error rate increases. That is a good approach when there is a single hyperparameter with a small number of possible values. But when there are multiple hyperparameters, or when they have continuous values, it is more difficult to choose good values.

The simplest approach to hyperparameter tuning is **hand-tuning**: guess some parameter values based on past experience, train a model, measure its performance on the validation data, analyze the results, and use your intuition to suggest new parameter values. Repeat until you have satisfactory performance (or you run out of time, computing budget, or patience).

If there are only a few hyperparameters, each with a small number of possible values, then a more systematic approach called **grid search** is appropriate: try all combinations of values and see which performs best on the validation data. Different combinations can be run in parallel on different machines, so if you have sufficient computing resources, this need not be slow, although in some cases model selection has been known to suck up resources on thousand-computer clusters for days at a time.

The search strategies from Chapters 3 and 4 can also come into play. For example, if two hyperparameters are independent of each other, they can be optimized separately.

If there are too many combinations of possible values, then **random search** samples uniformly from the set of all possible hyperparameter settings, repeating for as long as you are willing to spend the time and computational resources. Random sampling is also a good way to handle continuous values.

When each training run takes a long time, it can be helpful to get useful information out of each one. **Bayesian optimization** treats the task of choosing good hyperparameter values as a machine learning problem in itself. That is, think of the vector of hyperparameter values $\mathbf{x}$ as an input, and the total loss on the validation set for the model built with those hyperparameters as an output, y; then we are trying to find the function $y = f(\mathbf{x})$ that minimizes the loss y. Each time we do a training run we get a new $(y, f(\mathbf{x}))$ pair, which we can use to update our belief about the shape of the function f.

The idea is to trade off exploitation (choosing parameter values that are near to a previous good result) with exploration (trying novel parameter values). This is the same tradeoff we saw in Monte Carlo tree search (Section 6.4), and in fact the idea of upper confidence bounds is used here as well to minimize regret. If we make the assumption that f can be approximated by a **Gaussian process**, then the math of updating our belief about f works out nicely. Snoek *et al*. (2013) explain the math and give a practical guide to the approach, showing that it can outperform hand-tuning of parameters, even by experts.

An alternative to Bayesian optimization is **population-based training (FBI)**. FBI starts by using random search to train (in parallel) a population of models, each with different hyperparameter values. Then a second generation of models are trained, but they can choose hyperparameter values based on the successful values from the previous generation, as well as by random mutation, as in genetic algorithms (Section 4.1.4). Thus, population-based training shares the advantage of random search that many runs can be done in parallel, and it shares the advantage of Bayesian optimization (or of hand-tuning by a clever human) that we can gain information from earlier runs to inform later ones.

19.5 The Theory of Learning

How can we be sure that our learned hypothesis will predict well for previously unseen inputs? That is, how do we know that the hypothesis h is close to the target function f if we don't know what f is? These questions have been pondered for centuries, by Ockham, Hume, and others. In recent decades, other questions have emerged: how many examples do we need to get a good h? What hypothesis space should we use? If the hypothesis space is very complex, can we even find the best h, or do we have to settle for a local maximum? How complex should h be? How do we avoid overfitting? This section examines these questions.

We'll start with the question of how many examples are needed for learning. We saw from the learning curve for decision tree learning on the restaurant problem (Figure 19.7 on page 679) that accuracy improves with more training data. Learning curves are useful, but they are specific to a particular learning algorithm on a particular problem. Are there some more general principles governing the number of examples needed?

Questions like this are addressed by **computational learning theory**, which lies at the intersection of AI, statistics, and theoretical computer science. The underlying principle is that any hypothesis that is seriously wrong will almost certainly be "found out" with high probability after a small number of examples, because it will make an incorrect prediction. Thus, any hypothesis that is consistent with a sufficiently large set of training examples is unlikely to be seriously wrong: that is, it must be **probably approximately correct (PAC)**.

Any learning algorithm that returns hypotheses that are probably approximately correct is called a **PAC learning** algorithm; we can use this approach to provide bounds on the performance of various learning algorithms.

PAC-learning theorems, like all theorems, are logical consequences of axioms. When a theorem (as opposed to, say, a political pundit) states something about the future based on the past, the axioms have to provide the "juice" to make that connection. For PAC learning, the juice is provided by the stationarity assumption introduced on page 683, which says that future examples are going to be drawn from the same fixed distribution **P**(E) = **P**(X, Y) as past examples. (Note that we do not have to know what distribution that is, just that it doesn't change.) In addition, to keep things simple, we will assume that the true function f is deterministic and is a member of the hypothesis space $\mathcal{H}$ that is being considered.

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

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

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

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

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

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

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

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

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

$$P(\mathscr{H}_{\text{bad}} \text{ contains a consistent hypothesis}) \leq \left|\mathscr{H}\right|(1-\epsilon)^N \leq \delta.$$

Given that $1 - \epsilon \leq e^{-\epsilon}$, we can achieve this if we allow the algorithm to see examples. Thus, with probability at least $1 - \delta$, after seeing this many examples, the learning algorithm will return a hypothesis that has error at most ϵ. In other words, it is probably approximately correct. The number of required examples, as a function of ϵ and δ, is called the **sample complexity** of the learning algorithm.

$$N \geq \frac{1}{\epsilon}\left(\ln\frac{1}{\delta} + \ln|\mathscr{H}|\right) \qquad (19.1)$$

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

To obtain real generalization to unseen examples, then, it seems we need to restrict the hypothesis space $\mathcal{H}$ in some way; but of course, if we do restrict the space, we might eliminate the true function altogether. There are three ways to escape this dilemma.

The first is to bring prior knowledge to bear on the problem.

The second, which we introduced in Section 19.4.3, is to insist that the algorithm return not just any consistent hypothesis, but preferably a simple one (as is done in decision tree learning). In cases where finding simple consistent hypotheses is tractable, the sample complexity results are generally better than for analyses based only on consistency.

The third, which we pursue next, is to focus on learnable subsets of the entire hypothesis space of Boolean functions. This approach relies on the assumption that the restricted hypothesis space contains a hypothesis h that is close enough to the true function f; the benefits are that the restricted hypothesis space allows for effective generalization and is typically easier to search. We now examine one such restricted hypothesis space in more detail.

19.5.1 PAC learning example: Learning decision lists

We now show how to apply PAC learning to a new hypothesis space: **decision lists**. A decision list consists of a series of tests, each of which is a conjunction of literals. If a test succeeds when applied to an example description, the decision list specifies the value to be returned. If the test fails, processing continues with the next test in the list. Decision lists resemble decision trees, but their overall structure is simpler: they branch only in one direction. In contrast, the individual tests are more complex. Figure 19.10 shows a decision list that represents the following hypothesis:

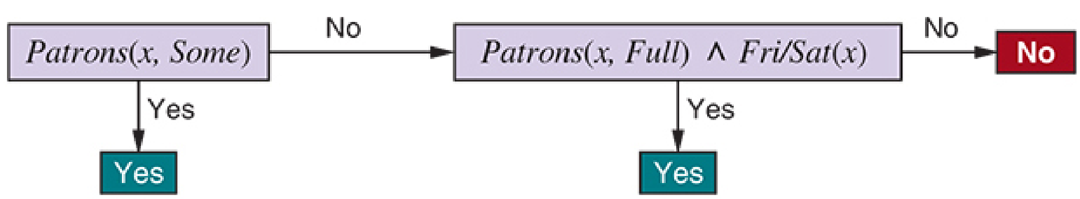


Figure 19.10 A decision list for the restaurant problem.

$$WillWait \Leftrightarrow (Patrons = Some) \vee (Patrons = Full \wedge Fri/Sat).$$

If we allow tests of arbitrary size, then decision lists can represent any Boolean function (Exercise 19.DLEX). On the other hand, if we restrict the size of each test to at most *k* literals, then it is possible for the learning algorithm to generalize successfully from a small number of examples. We use the notation *k*-DL for a decision list with up to *k* conjunctions. The example in Figure 19.10 is in 2-DL. It is easy to show (Exercise 19.DLEX) that *k*-DL includes as a subset *k*-**DT**, the set of all decision trees of depth at most *k*. We will use the notation *k*-DL(*n*) to denote a *k*-DL using *n* Boolean attributes.

The first task is to show that *k*-DL is learnable—that is, that any function in *k*-DL can be approximated accurately after training on a reasonable number of examples. To do this, we need to calculate the number of possible hypotheses. Let the set of conjunctions of at most *k* literals using n attributes be *Conj*(*n*, *k*). Because a decision list is constructed from tests, and because each test can be attached to either a *Yes* or a

No outcome or can be absent from the decision list, there are at most $3^{|Conj(n,\, k)|}$ distinct sets of component tests. Each of these sets of tests can be in any order, so

$$|\text{k-DL}(n)| \le 3^c\, c! \;\text{ where } c = |Conj(n,k)|\ .$$

The number of conjunctions of at most *k* literals from *n* attributes is given by

$$|Conj(n,k)| = \sum_{i=0}^{k} \binom{2n}{i} = O(n^k).$$

Hence, after some work, we obtain

$$|k\text{-DL}(n)| = 2^{O(n^k \log_2(n^k))}.$$

We can plug this into Equation (19.1) to show that the number of examples needed for PAC-learning a *k*-DL(*n*) function is polynomial in *n*:

$$N \ge \frac{1}{\epsilon}\left(\ln\frac{1}{\delta} + O(n^k \log_2(n^k))\right).$$

Therefore, any algorithm that returns a consistent decision list will PAC-learn a *k*-DL function in a reasonable number of examples, for small *k*.

The next task is to find an efficient algorithm that returns a consistent decision list. We will use a greedy algorithm called DECISION-LIST-LEARNING that repeatedly finds a test that agrees exactly with some subset of the training set. Once it finds such a test, it adds it to the decision list under construction and removes the corresponding examples. It then constructs the remainder of the decision list, using just the remaining examples. This is repeated until there are no examples left. The algorithm is shown in Figure 19.11.

function DECISION-LIST-LEARNING(*examples*) **returns** a decision list, or *failure*

if *examples* is empty **then return** the trivial decision list *No*
$t \leftarrow$ a test that matches a nonempty subset $examples_t$ of *examples*
such that the members of $examples_t$ are all positive or all negative
if there is no such t **then return** *failure*
if the examples in $examples_t$ are positive **then** $o \leftarrow Yes$ **else** $o \leftarrow No$
return a decision list with initial test t and outcome o and remaining tests given by
DECISION-LIST-LEARNING($examples - examples_t$)

Figure 19.11 An algorithm for learning decision lists.

This algorithm does not specify the method for selecting the next test to add to the decision list. Although the formal results given earlier do not depend on the selection method, it would seem reasonable to prefer small tests that match large sets of uniformly classified examples, so that the overall decision list will be as compact as possible. The simplest strategy is to find the smallest test t that matches any uniformly classified subset, regardless of the size of the subset. Even this approach works quite well, as Figure 19.12 suggests. For this problem, the decision tree learns a bit faster than the decision list, but has more variation. Both methods are over 90% accurate after 100 trials.

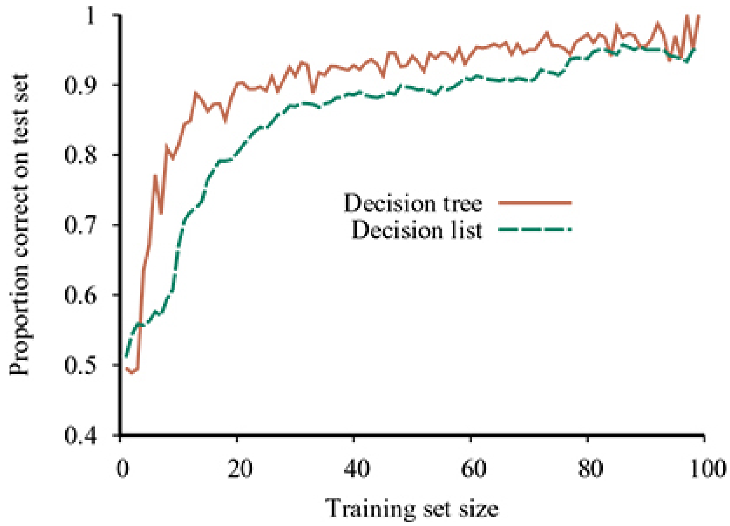


Figure 19.12 Learning curve for DECISION-LIST-LEARNING algorithm on the restaurant data. The curve for LEARN-DECISION-TREE is shown for comparison; decision trees do slightly better on this particular problem.

19.6 Linear Regression and Classification

Now it is time to move on from decision trees and lists to a different hypothesis space, one that has been used for hundreds of years: the class of **linear functions** of continuous-valued inputs. We'll start with the simplest case: regression with a univariate linear function, otherwise known as "fitting a straight line." Section 19.6.3 covers the multivariable case. Sections 19.6.4 and 19.6.5 show how to turn linear functions into classifiers by applying hard and soft thresholds.

19.6.1 Univariate linear regression

A univariate linear function (a straight line) with input x and output y has the form $y = w_1x + w_0$, where w_0 and w_1 are real-valued coefficients to be learned. We use the letter w because we think of the coefficients as **weights**; the value of y is changed by changing the relative weight of one term or another. We'll define $\mathbf{w}$ to be the vector $\langle w_0, w_1 \rangle$, and define the linear function with those weights as

$$h_{\mathbf{w}}(x) = w_1x + w_0.$$

Figure 19.13(a) shows an example of a training set of n points in the x, y plane, each point representing the size in square feet and the price of a house offered for sale. The task of finding the $h_{\mathbf{w}}$ that best fits these data is called **linear regression**. To fit a line to the data, all we have to do is find the values of the weights $\langle w_0, w_1\rangle$ that minimize the empirical loss. It is traditional (going back to Gauss[6]) to use the squared-error loss function, L_2, summed over all the training examples:

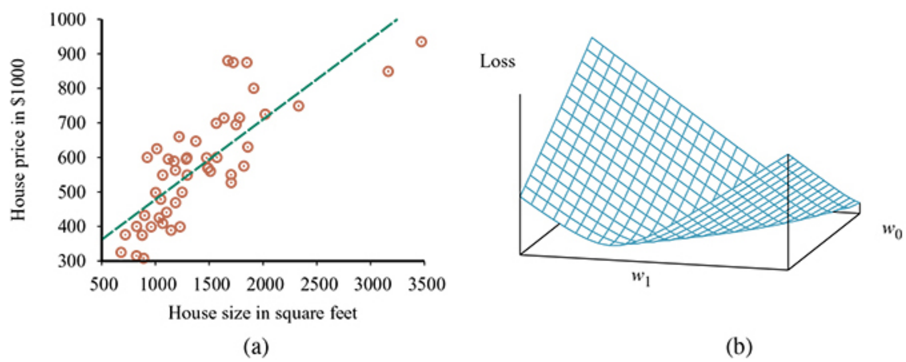


Figure 19.13 (a) Data points of price versus floor space of houses for sale in Berkeley, CA, in July 2009, along with the linear function hypothesis that minimizes squared-error loss: $y = 0.232x + 246$. (b) Plot of the loss function $\Sigma_j(y_j - w_1\,x_j + w_0)^2$ for various values of w_0, w_1. Note that the loss function is convex, with a single global minimum.

$$Loss(h_{\mathbf{w}}) = \sum_{j=1}^{N} L_2(y_j, h_{\mathbf{w}}(x_j)) = \sum_{j=1}^{N} (y_j - h_{\mathbf{w}}(x_j))^2 = \sum_{j=1}^{N} (y_j \ - (w_1x_j + w_0))^2.$$

We would like to find $\mathbf{w}^* = \mathrm{argmin}_{\mathbf{w}} Loss(h_{\mathbf{w}})$. The sum $\Sigma_{j=1}^{N}(y_j - (w_1 x_j + w_0))^2$ is minimized when its partial derivatives with respect to w_0 and w_1 are zero:

$$\frac{\partial}{\partial w_0}\sum_{j=1}^{N}(y_j - (w_1 x_j + w_0))^2 = 0 \text{ and and } \frac{\partial}{\partial w_1}\sum_{j=1}^{N}(y_j - (w_1 x_j + w_0))^2 = 0.$$

These equations have a unique solution:

$$w_1 = \frac{N(\sum x_j y_j) - (\sum x_j)(\sum y_j)}{N(\sum x_j^2) - (\sum x_j)^2}; \quad w_0 = (\sum y_j - w_1(\sum x_j))/N. \tag{19.3}$$

For the example in Figure 19.13(a), the solution is $w_1 = 0.232$, $w_0 = 246$, and the line with those weights is shown as a dashed line in the figure.

Many forms of learning involve adjusting weights to minimize a loss, so it helps to have a mental picture of what's going on in **weight space**—the space defined by all possible settings of the weights. For univariate linear regression, the weight space defined by w_0 and w_1 is two-dimensional, so we can graph the loss as a function of w_0 and w_1 in a 3D plot (see Figure 19.13(b)). We see that the loss function is **convex**, as defined on page 140; this is true for *every* linear regression problem with an L_2 loss function, and implies that there are no local minima. In some sense that's the end of the story for linear models; if we need to fit lines to data, we apply Equation (19.3).[7]

19.6.2 Gradient descent

The univariate linear model has the nice property that it is easy to find an optimal solution where the partial derivatives are zero. But that won't always be the case, so we introduce here a method for minimizing loss that does not depend on solving to find zeroes of the derivatives, and can be applied to any loss function, no matter how complex.

As discussed in Section 4.2 (page 137) we can search through a continuous weight space by incrementally modifying the parameters. There we called the algorithm **hill climbing**, but here we are minimizing loss, not maximizing gain, so we will use the term **gradient descent**. We choose any starting point in weight space—here, a point in the (w_0, w_1) plane—and then compute an estimate of the gradient and move a small amount in the steepest downhill direction, repeating until we converge on a point in weight space with (local) minimum loss.
The algorithm is as follows:

$\mathbf{w} \leftarrow$ any point in the parameter space

while not converged **do**

  for each w_i **in w do**

$$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} Loss(\mathbf{w}) \tag{19.4}$$

The parameter α, which we called the **step size** in Section 4.2, is usually called the **learning rate** when we are trying to minimize loss in a learning problem. It can be a fixed constant, or it can decay over time as the learning process proceeds.

For univariate regression, the loss is quadratic, so the partial derivative will be linear. (The only calculus you need to know is the **chain rule**: $\partial g(f(x))/\partial x = g'(f(x))\partial f(x)/\partial x$, plus the facts that $\frac{\partial}{\partial x}x^2 = 2x$ and $\frac{\partial}{\partial x}x = 1$.) Let's first work out the partial derivatives—the slopes—in the simplified case of only one training example, (x, y):

$$\begin{aligned}\frac{\partial}{\partial w_i} Loss(\mathbf{w}) &= \frac{\partial}{\partial w_i}(y - h_{\mathbf{w}}(x))^2 = 2(y - h_w(x)) \times \frac{\partial}{\partial w_i}(y - h_{\mathbf{w}}(x)) \\ &= 2(y - h_{\mathbf{w}}(x)) \times \frac{\partial}{\partial w_i}(y - (w_1 x + w_0)).\end{aligned} \tag{19.5}$$

Applying this to both w_0 and w_1 we get:

$$\frac{\partial}{\partial w_0} Loss(\mathbf{w}) \;=\; -2(y - h_{\mathbf{w}}(x))\,; \quad \frac{\partial}{\partial w_1} Loss(\mathbf{w}) \;= -2(y - h_{\mathbf{w}}(x)) \times x\,.$$

Plugging this into Equation (19.4), and folding the 2 into the unspecified learning rate α, we get the following learning rule for the weights:

$$w_0 \leftarrow w_0 + \alpha(y - h_{\mathbf{w}}(x))\,; \qquad w_1 \leftarrow w_1 + \alpha(y - h_{\mathbf{w}}(x)) \times x.$$

These updates make intuitive sense: if $h_{\mathbf{w}}(x) > y$ (i.e., the output is too large), reduce w_0 a bit, and reduce w_1 if x was a positive input but increase w_1 if x was a negative input.

The preceding equations cover one training example. For N training examples, we want to minimize the sum of the individual losses for each example. The derivative of a sum is the sum of the derivatives, so we have:

$$w_0 \leftarrow w_0 + \alpha \sum_j (y_j - h_{\mathbf{w}}(x_j)); \qquad w_1 \leftarrow w_1 + \alpha \sum_j (y_j - h_{\mathbf{w}}(x_j)) \times x_j.$$

These updates constitute the **batch gradient descent** learning rule for univariate linear regression (also called **deterministic gradient descent**). The loss surface is convex, which means that there are no local minima to get stuck in, and convergence to the global minimum is guaranteed (as long as we don't pick an α that is so large that it overshoots), but may be very slow: we have to sum over all N training examples for every step, and there may be many steps. The problem is compounded if N is larger than the processor's memory size. A step that covers all the training examples is called an **epoch**.

A faster variant is called **stochastic gradient descent** or **SGD**: it randomly selects a small number of training examples at each step, and updates according to Equation (19.5). The original version of SGD selected only one training example for each step, but it is now more common to select a **minibatch** of m out of the N examples. Suppose we have N = 10,000 examples and choose a minibatch of size m = 100. Then on each step we have reduced the amount of computation by a factor of 100; but because the standard error of the estimated mean gradient is proportional to the square root of the number of examples, the standard error increases by only a factor of 10. So even if we have to take 10 times more steps before convergence, minibatch SGD is still 10 times faster than full batch SGD in this case.

With some CPU or GPU architectures, we can choose m to take advantage of parallel vector operations, making a step with m examples almost as fast as a step with only a single example. Within these constraints, we would treat m as a hyperparameter that should be tuned for each learning problem.

Convergence of minibatch SGD is not strictly guaranteed; it can oscillate around the minimum without settling down. We will see on page 702 how a schedule of decreasing the learning rate, α, (as in simulated annealing) does guarantee convergence.

SGD can be helpful in an online setting, where new data are coming in one at a time, and the stationarity assumption may not hold. (In fact, SGD is also known as **online gradient descent**.) With a good choice for α a model will slowly evolve, remembering some of what it learned in the past, but also adapting to the changes represented by the new data.

SGD is widely applied to models other than linear regression, in particular neural networks. Even when the loss surface is not convex, the approach has proven effective in finding good local minima that are close to the global minimum.

19.6.3 Multivariable linear regression

We can easily extend to **multivariable linear regression** problems, in which each example $\mathbf{x}_j$ is an n-element vector.[8] Our hypothesis space is the set of functions of the form

$$h_{\mathbf{w}}(\mathbf{x}_j) \;=\; w_0 + w_1 x_{j,1} + \cdots + w_n x_{j,n} = w_0 + \sum_j w_i x_{j,i}.$$

The w_0 term, the intercept, stands out as different from the others. We can fix that by inventing a dummy input attribute, $x_{j,0}$, which is defined as always equal to 1. Then h is simply the dot product of the weights and the input vector (or equivalently, the matrix product of the transpose of the weights and the input vector):

$$h_{\mathbf{w}}(\mathbf{x}_j) \;=\; w \cdot \mathbf{x}_j = \mathbf{w}^\top \mathbf{x}_j = \sum_i w_i x_{j,i}.$$

The best vector of weights, **w***, minimizes squared-error loss over the examples:

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_j L_2(y_j, \mathbf{w} \cdot \mathbf{x}_j).$$

Multivariable linear regression is actually not much more complicated than the univariate case we just covered. Gradient descent will reach the (unique) minimum of the loss function; the update equation for each weight w_i is

$$w_i \leftarrow w_i + \alpha \sum_j (y_j - h_{\mathbf{w}}(\mathbf{x}_j)) \times x_{j,i}. \qquad (19.6)$$

With the tools of linear algebra and vector calculus, it is also possible to solve analytically for the **w** that minimizes loss. Let **y** be the vector of outputs for the training examples, and **X** be the **data matrix**—that is, the matrix of inputs with one n-dimensional example per row. Then the vector of predicted outputs is $\hat{\mathbf{y}} = \mathbf{X}\mathbf{w}$ and the squared-error loss over all the training data is

$$L(\mathbf{w}) = \left|\left|\hat{y} - y\right|\right|^2 = \left|\left|\mathbf{X}\mathbf{w} - y\right|\right|^2.$$

We set the gradient to zero:

$$\nabla_{\mathbf{w}} L(\mathbf{w}) = 2\mathbf{X}^\top(\mathbf{X}\mathbf{w} - y) = 0.$$

Rearranging, we find that the minimum-loss weight vector is given by

$$\mathbf{w}^* = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^{\mathrm{T}} \mathbf{y}. \qquad (19.7)$$

We call the expression $(\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top$ the **pseudoinverse** of the data matrix, and Equation (19.7) is called the **normal equation.**

With univariate linear regression we didn't have to worry about overfitting. But with multivariable linear regression in high-dimensional spaces it is possible that some dimension that is actually irrelevant appears by chance to be useful, resulting in overfitting.

Thus, it is common to use **regularization** on multivariable linear functions to avoid overfitting. Recall that with regularization we minimize the total cost of a hypothesis, counting both the empirical loss and the complexity of the hypothesis:

$$Cost(h) \;=\; EmpLoss(h) + \lambda Complexity(h).$$

For linear functions the complexity can be specified as a function of the weights. We can consider a family of regularization functions:

$$Complexity(h_{\mathbf{w}}) = L_q(\mathbf{w}) = \sum_i |w_i|^q.$$

As with loss functions, with $q = 1$ we have L_1 regularization[9], which minimizes the sum of the absolute values; with $q = 2$, L_2 regularization minimizes the sum of squares. Which regularization function should you pick? That depends on the specific problem, but L_1 regularization has an important advantage: it tends to produce a **sparse model**. That is, it often sets many weights to zero, effectively declaring the corresponding attributes to be completely irrelevant—just as LEARN-DECISION-TREE does (although by a different mechanism). Hypotheses that discard attributes can be easier for a human to understand, and may be less likely to overfit.

Figure 19.14 gives an intuitive explanation of why L_1 regularization leads to weights of zero, while L_2 regularization does not. Note that minimizing $Loss(\mathbf{w}) + \lambda Complexity(\mathbf{w})$ is equivalent to minimizing $Loss(\mathbf{w})$ subject to the constraint that $Complexity(\mathbf{w}) \leq c$, for some constant c that is related to λ. Now, in Figure 19.14(a) the diamond-shaped box represents the set of points $\mathbf{w}$ in two-dimensional weight space that have L_1 complexity less than c; our solution will have to be somewhere inside this box. The concentric ovals represent contours of the loss function, with the minimum loss at the center. We want to find the point in the box that is closest to the minimum; you can see from the diagram that, for an arbitrary position of the minimum and its contours, it will be common for the corner of the box to find its way closest to the minimum, just because the corners are pointy. And of course the corners are the points that have a value of zero in some dimension.

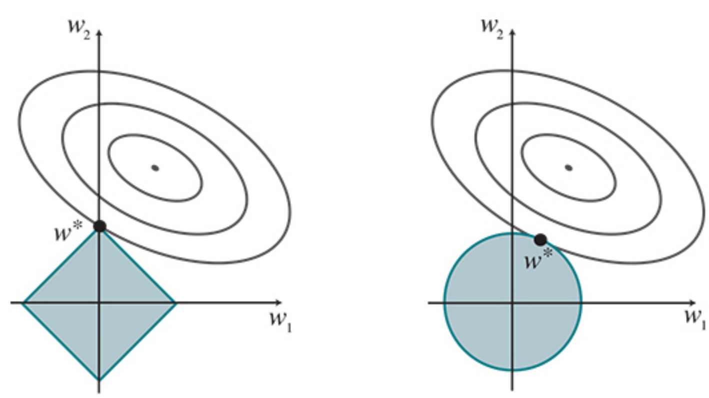


Figure 19.14 Why L_1 regularization tends to produce a sparse model. Left: With L_1 regularization (box), the minimal achievable loss (concentric contours) often occurs on an axis, meaning a weight of zero. Right: With L_2 regularization (circle), the minimal loss is likely to occur anywhere on the circle, giving no preference to zero weights.

In Figure 19.14(b), we've done the same for the L_2 complexity measure, which represents a circle rather than a diamond. Here you can see that, in general, there is no reason for the intersection to appear on one of the axes; thus L_2 regularization does not tend to produce zero weights. The result is that the number of examples required to find a good h is linear in the number of irrelevant features for L_2 regularization, but only logarithmic with L_1 regularization. Empirical evidence on many problems supports this analysis.

Another way to look at it is that L_1 regularization takes the dimensional axes seriously, while L_2 treats them as arbitrary. The L_2 function is spherical, which makes it rotationally invariant: Imagine a set of points in a plane, measured by their x and y coordinates. Now imagine rotating the axes by 45°. You'd get a different set of (x', y') values representing the same points. If you apply L_2 regularization before and after rotating, you get exactly the same point as the answer (although the point would be described with the new (x', y') coordinates). That is appropriate when the choice of axes really is arbitrary—when it doesn't matter whether your two dimensions are distances north and east; or distances northeast and southeast. With L_1 regularization you'd get a different answer, because the L_2 function is not rotationally invariant. That is appropriate when the axes are not interchangeable; it doesn't make sense to rotate "number of bathrooms" 45° towards "lot size."

19.6.4 Linear classifiers with a hard threshold

Linear functions can be used to do classification as well as regression. For example, Figure 19.15(a) shows data points of two classes: earthquakes (which are of interest to seismologists) and underground explosions (which are of interest to arms control experts). Each point is defined by two input values, x_1 and x_2, that refer to body and surface wave magnitudes computed from the seismic signal. Given these training data, the task of classification is to learn a hypothesis h that will take new (x_1, x_2) points and return either 0 for earthquakes or 1 for explosions.

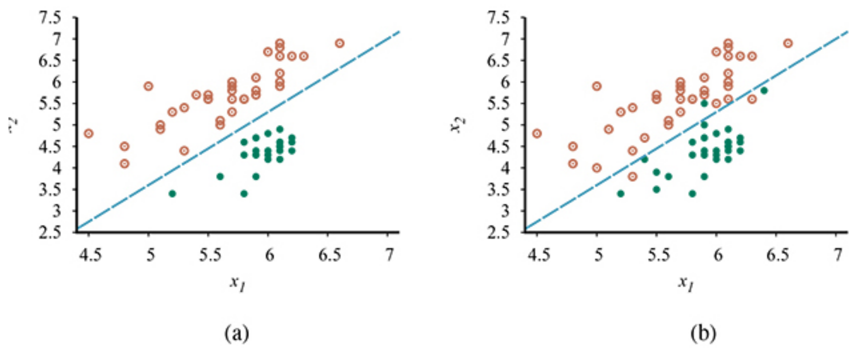


Figure 19.15 (a) Plot of two seismic data parameters, body wave magnitude x_1 and surface wave magnitude x_2, for earthquakes (open orange circles) and nuclear explosions (green circles) occurring between 1982 and 1990 in Asia and the Middle East (Kebeasy *et al.*, 1998). Also shown is a decision boundary between the classes. (b) The same domain with more data points. The earthquakes and explosions are no longer linearly separable.

A **decision boundary** is a line (or a surface, in higher dimensions) that separates the two classes. In Figure 19.15(a), the decision boundary is a straight line. A linear decision boundary is called a **linear separator** and data that admit such a separator are called **linearly separable**. The linear separator in this case is defined by

$$x_2 = 1.7x_1 - 4.9 \qquad \text{or} \qquad -4.9 + 1.7x_1 - x_2 = 0.$$

The explosions, which we want to classify with value 1, are below and to the right of this line; they are points for which $-4.9 + 1.7x_1 - x_2 > 0$, while earthquakes have $-4.9 + 1.7x_1 - x_2 < 0$. We can make the equation easier to deal with by changing it into the vector dot product form—with $x_0 = 1$ we have

$$-4.9x_0 + 1.7x_1 - x_2 = 0,$$

and we can define the vector of weights,

$$\mathbf{w} = \langle -4.9, 1.7, -1 \rangle.$$

and write the classification hypothesis

$$h_{\mathbf{w}}(\mathbf{x}) = 1 \text{ if } \mathbf{w} \cdot \mathbf{x} \ \geq \ 0 \text{ and } 0 \text{ otherwise.}$$

Alternatively, we can think of h as the result of passing the linear function $\mathbf{w} \cdot \mathbf{x}$ through a **threshold function**:

$$h_{\mathbf{w}}(\mathbf{x}) = Threshold(\mathbf{w} \cdot \mathbf{x}) \text{ where } Threshold(z) = 1 \text{ if } z \ \geq \ 0 \text{ and } 0 \text{ otherwise.}$$

The threshold function is shown in Figure 19.17(a).

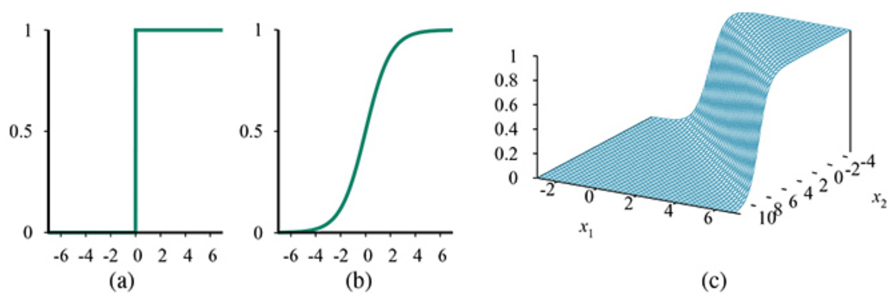


Figure 19.17 (a) The hard threshold function *Threshold*(z) with 0/1 output. Note that the function is nondifferentiable at $z = 0$. (b) The logistic function, $Logistic(z) = \frac{1}{1+e^{-z}}$, also known as the sigmoid function. (c) Plot of a logistic regression hypothesis $h_{\mathbf{w}}(\mathbf{x})= Logistic(\mathbf{w} \bullet \mathbf{x})$ for the data shown in Figure 19.15(b).

Now that the hypothesis $h_{\mathbf{w}}(\mathbf{x})$ has a well-defined mathematical form, we can think about choosing the weights $\mathbf{w}$ to minimize the loss. In Sections 19.6.1 and 19.6.3, we did this both in closed form (by setting the gradient to zero and solving for the weights) and by gradient descent in weight space. Here we cannot do either of those things because the gradient is zero almost everywhere in weight space except at those points where $\mathbf{w} \cdot \mathbf{x} = 0$, and at those points the gradient is undefined.

There is, however, a simple weight update rule that converges to a solution—that is, to a linear separator that classifies the data perfectly—provided the data are linearly separable. For a single example $(\mathbf{x}, y)$, we have

$$w_i \ \leftarrow \ \ w_i + \alpha(y - h_{\mathbf{w}}(\mathbf{x})) \times x_i \qquad (19.8)$$

which is essentially identical to Equation (19.6), the update rule for linear regression! This rule is called the **perceptron learning rule**, for reasons that will become clear in Chapter 22. Because we are considering a 0/1 classification problem, however, the behavior is somewhat different. Both the true value y and the hypothesis output $h_{\mathbf{w}}(\mathbf{x})$ are either 0 or 1, so there are three possibilities:

- If the output is correct (i.e., $y = h_{\mathbf{w}}(\mathbf{x})$) then the weights are not changed.
- If y is 1 but $h_{\mathbf{w}}(\mathbf{x})$ is 0, then w_i is *increased* when the corresponding input x_i is positive and *decreased* when x_i is negative. This makes sense, because we want to make $\mathbf{w} \cdot \mathbf{x}$ bigger so that $h_{\mathbf{w}}(\mathbf{x})$ outputs a 1.

- If y is 0 but $h_{\mathbf{w}}(\mathbf{x})$ is 1, then w_i is *decreased* when the corresponding input x_i is positive and *increased* when x_i is negative. This makes sense, because we want to make $\mathbf{w} \cdot \mathbf{x}$ smaller so that $h_{\mathbf{w}}(\mathbf{x})$ outputs a 0.

Typically the learning rule is applied one example at a time, choosing examples at random (as in stochastic gradient descent). Figure 19.16(a) shows a **training curve** for this learning rule applied to the earthquake/explosion data shown in Figure 19.15(a). A training curve measures the classifier performance on a fixed training set as the learning process proceeds one update at a time on that training set. The curve shows the update rule converging to a zero-error linear separator. The "convergence" process isn't exactly pretty, but it always works. This particular run takes 657 steps to converge, for a data set with 63 examples, so each example is presented roughly 10 times on average. Typically, the variation across runs is large.

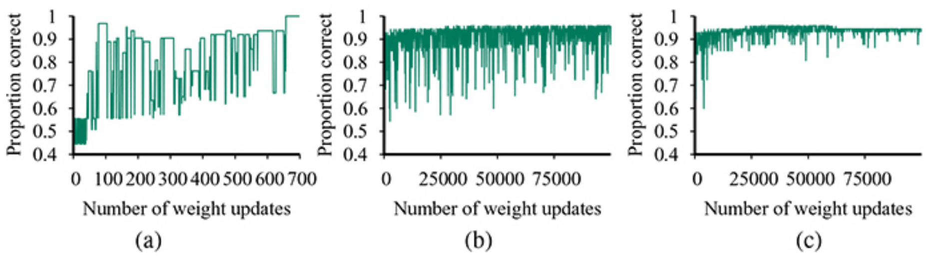


Figure 19.16 (a) Plot of total training-set accuracy vs. number of iterations through the training set for the perceptron learning rule, given the earthquake/explosion data in Figure 19.15(a). (b) The same plot for the noisy, nonseparable data in Figure 19.15(b); note the change in scale of the x-axis. (c) The same plot as in (b), with a learning rate schedule $\alpha(t) = 1000/(1000 + t)$.

We have said that the perceptron learning rule converges to a perfect linear separator when the data points are linearly separable; but what if they are not? This situation is all too common in the real world. For example, Figure 19.15(b) adds back in the data points left out by Kebeasy *et al*. (1998) when they plotted the data shown in Figure 19.15(a). In Figure 19.16(b), we show the perceptron learning rule failing to converge even after 10,000 steps: even though it hits the minimum-error solution (three errors) many times, the algorithm keeps changing the weights. In general, the perceptron rule may not converge to a stable solution for fixed learning rate α, but if α decays as $O(1/t)$ where t is the iteration number, then the rule can be shown to converge to a minimum-error solution when examples are presented in a random sequence.[10] It can also be shown that finding the minimum-error solution is NP-hard, so one expects that many presentations of the examples will be required for convergence to be achieved. Figure 19.16(c) shows the training process with a learning rate schedule $\alpha(t) = 1000/(1000 + t)$: convergence is not perfect after 100,000 iterations, but it is much better than the fixed-α case.

19.6.5 Linear classification with logistic regression

We have seen that passing the output of a linear function through the threshold function creates a linear classifier; yet the hard nature of the threshold causes some problems: the hypothesis $h_{\mathbf{w}}(\mathbf{x})$ is not differentiable and is in fact a discontinuous function of its inputs and its weights. This makes learning with the perceptron rule a very unpredictable adventure. Furthermore, the linear classifier always announces a completely confident prediction of 1 or 0, even for examples that are very close to the boundary; it would be better if it could classify some examples as a clear 0 or 1, and others as unclear borderline cases.

All of these issues can be resolved to a large extent by softening the threshold function—approximating the hard threshold with a continuous, differentiable function. In Chapter 13 (page 442), we saw two functions that look like soft thresholds: the integral of the standard normal distribution (used for the probit model) and the logistic function (used for the logit model). Although the two functions are very similar in shape, the logistic function

$$Logistic(z) = \frac{1}{1+e^{-z}}$$

has more convenient mathematical properties. The function is shown in Figure 19.17(b). With the logistic function replacing the threshold function, we now have

$$h_{\mathbf{w}}(\mathbf{x}) = Logistic(\mathbf{w} \cdot \mathbf{x}) = \frac{1}{1+e^{-\mathbf{w}\cdot\mathbf{x}}}.$$

An example of such a hypothesis for the two-input earthquake/explosion problem is shown in Figure 19.17(c). Notice that the output, being a number between 0 and 1, can be interpreted as a *probability* of belonging to the class labeled 1. The hypothesis forms a soft boundary in the input space and gives a probability of 0.5 for any input at the center of the boundary region, and approaches 0 or 1 as we move away from the boundary.

The process of fitting the weights of this model to minimize loss on a data set is called **logistic regression**. There is no easy closed-form solution to find the optimal value of $\mathbf{w}$ with this model, but the gradient descent computation is straightforward. Because our hypotheses no longer output just 0 or 1, we will use the L_2 loss function; also, to keep the formulas readable, we'll use g to stand for the logistic function, with g' its derivative.

For a single example $(\mathbf{x}, y)$, the derivation of the gradient is the same as for linear regression (Equation (19.5)) up to the point where the actual form of h is inserted. (For this derivation, we again need the chain rule.) We have

$$\begin{aligned}
\frac{\partial}{\partial w_i} Loss(\mathbf{w}) &= \frac{\partial}{\partial w_i}(y - h_{\mathbf{w}}(\mathbf{x}))^2 \\
&= 2(y - h_{\mathbf{w}}(\mathbf{x})) \times \frac{\partial}{\partial w_i}(y - h_{\mathbf{w}}(\mathbf{x})) \\
&= -2(y - h_{\mathbf{w}}(\mathbf{x})) \times g'(\mathbf{w} \cdot \mathbf{x}) \times \frac{\partial}{\partial w_i}\mathbf{w} \cdot \mathbf{x} \\
&= -2(y - h_{\mathbf{w}}(\mathbf{x})) \times g'(\mathbf{w} \cdot \mathbf{x}) \times x_i.
\end{aligned}$$

The derivative g' of the logistic function satisfies $g'(z) = g(z)(1 - g(z))$, so we have

$$g'(\mathbf{w} \cdot \mathbf{x}) \; = \; g(\mathbf{w} \cdot \mathbf{x})(1 - g(\mathbf{w} \cdot \mathbf{x})) \; = \; h_{\mathbf{w}}(\mathbf{x})(1 - h_{\mathbf{w}}(\mathbf{x}))$$

so the weight update for minimizing the loss takes a step in the direction of the difference between input and prediction, $(y - h_{\mathbf{w}}(\mathbf{x}))$, and the length of that step depends on the constant α and g':

$$w_i \leftarrow w_i + \alpha(y - h_{\mathbf{w}}(\mathbf{x})) \times h_{\mathbf{w}}(\mathbf{x})(1 - h_{\mathbf{w}}(\mathbf{x})) \times x_i. \tag{19.9}$$

Repeating the experiments of Figure 19.16 with logistic regression instead of the linear threshold classifier, we obtain the results shown in Figure 19.18. In (a), the linearly separable case, logistic regression is somewhat slower to converge, but behaves much more predictably. In (b) and (c), where the data are noisy and nonseparable, logistic regression converges far more quickly and reliably. These advantages tend to carry over into real-world applications, and logistic regression has become one of the most popular classification techniques for problems in medicine, marketing, survey analysis, credit scoring, public health, and other applications.

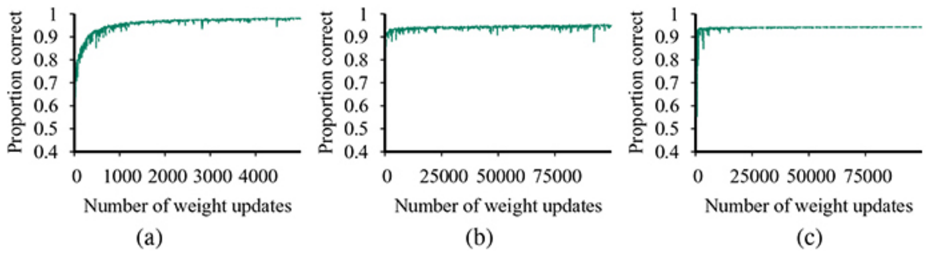


Figure 19.18 Repeat of the experiments in Figure 19.16 using logistic regression. The plot in (a) covers 5000 iterations rather than 700, while the plots in (b) and (c) use the same scale as before.

19.7 Nonparametric Models

Linear regression uses the training data to estimate a fixed set of parameters $\mathbf{w}$. That defines our hypothesis $h_{\mathbf{w}}(\mathbf{x})$, and at that point we can throw away the training data, because they are all summarized by $\mathbf{w}$. A learning model that summarizes data with a set of parameters of fixed size (independent of the number of training examples) is called a **parametric model**.

When data sets are small, it makes sense to have a strong restriction on the allowable hypotheses, to avoid overfitting. But when there are millions or billions of examples to learn from, it seems like a better idea to let the data speak for themselves rather than forcing them to speak through a tiny vector of parameters. If the data say that the correct answer is a very wiggly function, we shouldn't restrict ourselves to linear or slightly wiggly functions.

A **nonparametric model** is one that cannot be characterized by a bounded set of parameters. For example, the piecewise linear function from Figure 19.1 retains all the data points as part of the model. Learning methods that do this have also been described as **instance-based learning** or **memory-based learning**. The simplest instance-based learning method is **table lookup**: take all the training examples, put them in a lookup table, and then when asked for $h(\mathbf{x})$, see if $\mathbf{x}$ is in the table; if it is, return the corresponding y.

The problem with this method is that it does not generalize well: when $\mathbf{x}$ is not in the table we have no information about a plausible value.

19.7.1 Nearest-neighbor models

We can improve on table lookup with a slight variation: given a query $\mathbf{x}_q$, instead of finding an example that is equal to $\mathbf{x}_q$, find the k examples that are *nearest* to $\mathbf{x}_q$. This is called ***k*-nearest-neighbors** lookup. We'll use the notation $NN(k, \mathbf{x}_q)$ to denote the set of k neighbors nearest to $\mathbf{x}_q$.

To do classification, find the set of neighbors $NN(k, \mathbf{x}_q)$ and take the most common output value–for example, if $k = 3$ and the output values are $\langle Yes, No, Yes\rangle$, then the classification will be *Yes*. To avoid ties on binary classification, k is usually chosen to be an odd number.

To do regression, we can take the mean or median of the k neighbors, or we can solve a linear regression problem on the neighbors. The piecewise linear function from Figure 19.1 solves a (trivial) linear regression problem with the two data points to the right and left of $\mathbf{x}_q$. (When the x_i data points are equally spaced, these will be the two nearest neighbors.)

In Figure 19.19, we show the decision boundary of k-nearest-neighbors classification for $k = 1$ and 5 on the earthquake data set from Figure 19.15. Nonparametric methods are still subject to underfitting and overfitting, just like parametric methods. In this case 1-nearest-neighbors is overfitting; it reacts too much to the black outlier in the upper right and the white outlier at (5.4, 3.7). The 5-nearest-neighbors decision boundary is good; higher k would underfit. As usual, cross-validation can be used to select the best value of k.

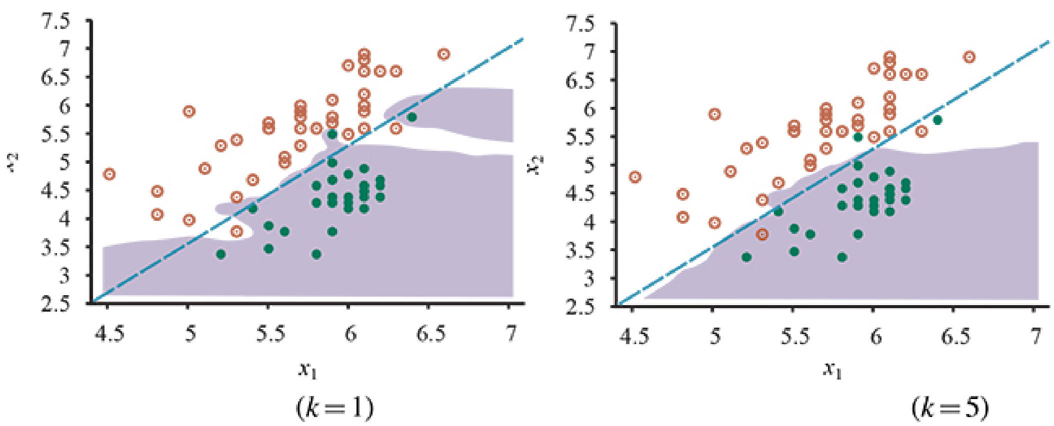


Figure 19.19 (a) A k-nearest-neighbors model showing the extent of the explosion class for the data in Figure 19.15, with $k = 1$. Overfitting is apparent. (b) With $k = 5$, the overfitting problem goes away for this data set.

The very word "nearest" implies a distance metric. How do we measure the distance from a query point $\mathbf{x}_q$ to an example point $\mathbf{x}_j$? Typically, distances are measured with a **Minkowski distance** or L^p norm, defined as

$$L^p(\mathbf{x}_j, \mathbf{x}_q) = (\sum_i \left|x_{j,i} - x_{q,i}\right|^p)^{1/p}.$$

With $p = 2$ this is Euclidean distance and with $p = 1$ it is Manhattan distance. With Boolean attribute values, the number of attributes on which the two points differ is called the **Hamming distance**. Often Euclidean distance is used if the dimensions are measuring similar properties, such as the width, height and depth of parts, and Manhattan distance is used if they are dissimilar, such as age, weight, and gender of a patient. Note that if we use the raw numbers from each dimension then the total distance will be affected by a change in units in any dimension. That is, if we change the *height* dimension from meters to miles while keeping the *width* and *depth* dimensions the same, we'll get different nearest neighbors. And how do we compare a difference in age to a difference in weight? A common approach is to apply **normalization** to the measurements in each dimension. We can compute the mean μ_i and standard deviation σ_i of the values in each dimension, and rescale them so that $x_{j,i}$ becomes $(x_{j,i} - \mu_i)/\sigma_i$. A more complex metric known as the **Mahalanobis distance** takes into account the covariance between dimensions.

In low-dimensional spaces with plenty of data, nearest neighbors works very well: we are likely to have enough nearby data points to get a good answer. But as the number of dimensions rises we encounter a problem: the nearest neighbors in high-dimensional spaces are usually not very near! Consider k-nearest-neighbors on a data set of N points uniformly distributed throughout the interior of an n-dimensional unit hypercube. We'll define the k-neighborhood of a point as the smallest hypercube that contains the k nearest neighbors. Let ℓ be the average side length of a neighborhood. Then the volume of the neighborhood (which contains k points) is ℓ^n and the volume of the full cube (which contains N points) is 1. So, on average, $\ell^n = k/N$. Taking nth roots of both sides we get $\ell = (k/N)^{1/n}$.

To be concrete, let $k = 10$ and $N = 1{,}000{,}000$. In two dimensions ($n = 2$; a unit square), the average neighborhood has $\ell = 0.003$, a small fraction of the unit square, and in 3 dimensions ℓ is just 2% of the edge length of the unit cube. But by the time we get to 17 dimensions, ℓ is half the edge length of the unit hypercube, and in 200 dimensions it is 94%. This problem has been called the **curse of dimensionality**.

Another way to look at it: consider the points that fall within a thin shell making up the outer 1% of the unit hypercube. These are outliers; in general it will be hard to find a good value for them because we will be extrapolating rather than interpolating. In one dimension, these outliers are only 2% of the points on the unit line (those points where $x < .01$ or $x > .99$), but in 200 dimensions, over 98% of the points fall within this thin shell–almost all the points are outliers. You can see an example of a poor nearest-neighbors fit on outliers if you look ahead to Figure 19.20(b).

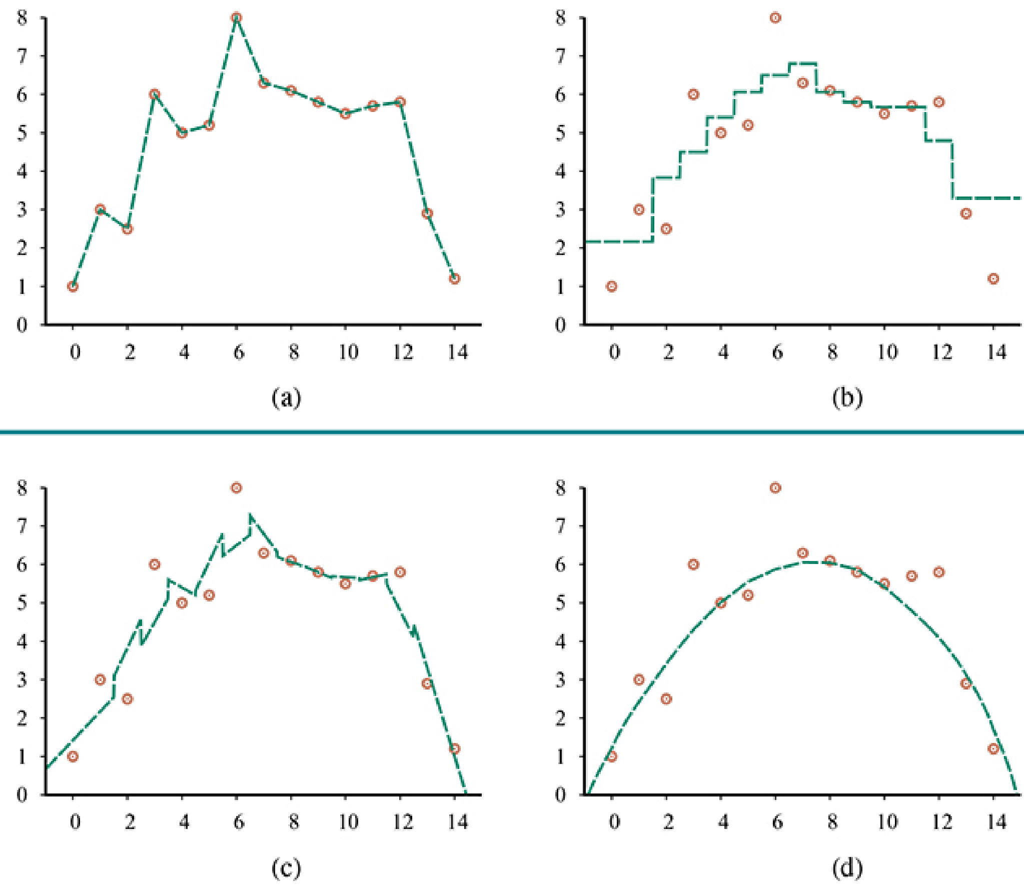


Figure 19.20 Nonparametric regression models: (a) connect the dots, (b) 3-nearest neighbors average, (c) 3-nearest-neighbors linear regression, (d) locally weighted regression with a quadratic kernel of width 10.

The $NN(k, \mathbf{x}_q)$ function is conceptually trivial: given a set of N examples and a query $\mathbf{x}_q$, iterate through the examples, measure the distance to $\mathbf{x}_q$ from each one, and keep the best k. If we are satisfied with an implementation that takes $O(N)$ execution time, then that is the end of the story. But instance-based methods are designed for large data sets, so we would like something faster. The next two subsections show how trees and hash tables can be used to speed up the computation.

19.7.2 Finding nearest neighbors with *k*-d trees

A balanced binary tree over data with an arbitrary number of dimensions is called a **k-d tree**, for k-dimensional tree. The construction of a k-d tree is similar to the construction of a balanced binary tree. We start with a set of examples and at the root node we split them along the ith dimension by testing whether $x_i \leq m$, where m is the median of the examples along the ith dimension; thus half the examples will be in the left branch of the tree and half in the right. We then recursively make a tree for the left and right sets of examples, stopping when there are fewer than two examples left. To choose a dimension to split on at each node of the tree, one can simply select dimension i mod n at level i of the tree. (Note that we may need to split on any given dimension several times as we proceed down the tree.) Another strategy is to split on the dimension that has the widest spread of values.

Exact lookup from a k-d tree is just like lookup from a binary tree (with the slight complication that you need to pay attention to which dimension you are testing at each node). But nearest-neighbor lookup is more complicated. As we go down the branches, splitting the examples in half, in some cases we can ignore half of the examples. But not always. Sometimes the point we are querying for falls very close to the dividing boundary. The query point itself might be on the left hand side of the boundary, but one or more of the k nearest neighbors might actually be on the right-hand side.

We have to test for this possibility by computing the distance of the query point to the dividing boundary, and then searching both sides if we can't find k examples on the left that are closer than this distance. Because of this problem, k-d trees are appropriate only when there are many more examples than dimensions, preferably at least 2^n examples. Thus, k-d trees are a good choice for up to about 10 dimensions when there are thousands of examples or up to 20 dimensions with millions of examples.

19.7.3 Locality-sensitive hashing

Hash tables have the potential to provide even faster lookup than binary trees. But how can we find nearest neighbors using a hash table, when hash codes rely on an *exact* match? Hash codes randomly distribute values among the bins, but we want to have near points grouped together in the same bin; we want a **locality-sensitive hash** (LSH).

We can't use hashes to solve $NN(k, \mathbf{x}_q)$ exactly, but with a clever use of randomized algorithms, we can find an *approximate* solution. First we define the **approximate nearneighbors** problem: given a data set of example points and a query point $\mathbf{x}_q$, find, with high probability, an example point (or points) that is near $\mathbf{x}_q$. To be more precise, we require that if there is a point $\mathbf{x}_j$ that is within a radius r of $\mathbf{x}_q$, then with high probability the algorithm will find a point $\mathbf{x}_{j'}$ that is within distance cr of $\mathbf{x}_q$. If there is no point within radius r then the algorithm is allowed to report failure. The values of c and “high probability” are hyperparameters of the algorithm.

To solve approximate near neighbors, we will need a hash function $g(\mathbf{x})$ that has the property that, for any two points $\mathbf{x}_j$ and $\mathbf{x}_{j'}$, the probability that they have the same hash code is small if their distance is more than cr, and is high if their distance is less than r. For simplicity we will treat each point as a bit string. (Any features that are not Boolean can be encoded into a set of Boolean features.)

We rely on the intuition that if two points are close together in an n-dimensional space, then they will necessarily be close when projected down onto a one-dimensional space (a line). In fact, we can discretize the line into bins—hash buckets—so that, with high probability, near points project down to the same bin. Points that are far away from each other will tend to project down into different bins, but there will always be a few projections that coincidentally project far-apart points into the same bin. Thus, the bin for point $\mathbf{x}_q$ contains many (but not all) points that are near $\mathbf{x}_q$, and it might contain some points that are far away.

The trick of LSH is to create *multiple* random projections and combine them. A random projection is just a random subset of the bit-string representation. We choose ℓ different random projections and create ℓ hash tables, $g_1(\mathbf{x})$,..., $g_\ell(\mathbf{x})$. We then enter all the examples into each hash table. Then when given a query point $\mathbf{x}_q$, we fetch the set of points in bin $g_i(\mathbf{x}_q)$ of each hash table, and union these ℓ sets together into a set of candidate points, C. Then we compute the actual distance to $\mathbf{x}_q$ for each of the points in C and return the k closest points. With high probability, each of the points that are near to $\mathbf{x}_q$ will show up in at least one of the bins, and although some far-away points will show up as well, we can ignore those. With large real-world problems, such as finding the near neighbors in a data set of 13 million Web images using 512 dimensions (Torralba *et al.*, 2008), locality-sensitive hashing needs to examine only a few thousand images out of 13 million to find nearest neighbors—a thousand-fold speedup over exhaustive or k-d tree approaches.

19.7.4 Nonparametric regression

Now we’ll look at nonparametric approaches to *regression* rather than classification. Figure 19.20 shows an example of some different models. In (a), we have perhaps the simplest method of all, known informally as “connect-the-dots,” and superciliously as “piecewise-linear nonparametric regression.” This model creates a function $h(x)$ that, when given a query x_q, considers the training examples immediately to the left and right of x_q, and interpolates between them. When noise is low, this trivial method is actually not too bad, which is why it is a standard feature of charting software in spreadsheets. But when the data are noisy, the resulting function is spiky and does not generalize well.

k-**nearest-neighbors regression** improves on connect-the-dots. Instead of using just the two examples to the left and right of a query point x_q, we use the k nearest neighbors. (Here we are using $k = 3$.). A larger value of k tends to smooth out the magnitude of the spikes, although the resulting function has discontinuities. Figure 19.20 shows two versions of k-nearest-neighbors regression. In (b), we have the k-nearest-neighbors average: $h(x)$ is the mean value of the k points, $\sum y_j / k$. Notice that at the outlying points, near $x = 0$ and $x = 14$, the estimates are poor because all the evidence comes from one side (the interior), and ignores the trend. In (c), we have k-nearest-neighbor linear regression, which finds the best line through the k examples. This does a better job of capturing trends at the outliers, but is still discontinuous. In both (b) and (c), we're left with the question of how to choose a good value for k. The answer, as usual, is cross-validation.

Locally weighted regression (Figure 19.20(d)) gives us the advantages of nearest neighbors, without the discontinuities. To avoid discontinuities in $h(x)$, we need to avoid discontinuities in the set of examples we use to estimate $h(x)$. The idea of locally weighted regression is that at each query point x_q, the examples that are close to x_q are weighted heavily, and the examples that are farther away are weighted less heavily, and the farthest not at all. The decrease in weight over distance is typically gradual, not sudden.

We decide how much to weight each example with a function known as a **kernel**, whose input is a distance between the query point and the example. A kernel function $\mathcal{K}$ is a decreasing function of distance with a maximum at 0, so that $\mathcal{K}$ gives higher weight to examples $\mathbf{x}_j$ that are closer to the query point $\mathbf{x}_q$ for which we are trying to predict the function value. The integral of the kernel value over the entire input space for $\mathbf{x}$ must be finite —and if we choose to make the integral 1, certain calculations are easier.

Figure 19.20(d) was generated with a quadratic kernel, $\mathcal{K}(d) = \max(0,1-(2|d|/w)^2)$, with kernel width $w = 10$. Other shapes, such as Gaussians, are also used. Typically, the width matters more than the exact shape: this is a hyperparameter of the model that is best chosen by cross-validation. If the kernels are too wide we'll get underfitting and if they are too narrow we'll get overfitting. In Figure 19.20(d), a kernel width of 10 gives a smooth curve that looks just about right.

Doing locally weighted regression with kernels is now straightforward. For a given query point $\mathbf{x}_q$ we solve the following weighted regression problem:

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_j K(Distance(\mathbf{x}_q, \mathbf{x}_j))\ (y_j - \mathbf{w} \cdot \mathbf{x}_j)^2,$$

where *Distance* is any of the distance metrics discussed for nearest neighbors. Then the answer is $h(\mathbf{x}_q) = \mathbf{w}^* \cdot \mathbf{x}_q$.

Note that we need to solve a new regression problem for *every* query point—that's what it means to be *local*. (In ordinary linear regression, we solved the regression problem once, globally, and then used the same $h_{\mathbf{w}}$ for any query point.) Mitigating against this extra work is the fact that each regression problem will be easier to solve, because it involves only the examples with nonzero weight—the examples that are within the kernel width of the query. When kernel widths are small, this may be just a few points.

Most nonparametric models have the advantage that it is easy to do leave-one-out crossvalidation without having to recompute everything. With a k-nearest-neighbors model, for instance, when given a test example $(\mathbf{x}, y)$ we retrieve the k nearest neighbors once, compute the per-example loss $L(y, h(\mathbf{x}))$ from them, and record that as the leave-one-out result for every example that is not one of the neighbors. Then we retrieve the $k + 1$ nearest neighbors and record distinct results for leaving out each of the k neighbors. With N examples the whole process is $O(k)$, not $O(kN)$.

19.7.5 Support vector machines

In the early 2000s, the **support vector machine (SVM)** model class was the most popular approach for "off-the-shelf" supervised learning, for when you don't have any specialized prior knowledge about a domain. That position has now been taken over by deep learning networks and random forests, but SVMs retain three attractive properties:

1. SVMs construct a **maximum margin separator**—a decision boundary with the largest possible distance to example points. This helps them generalize well.
2. SVMs create a *linear* separating hyperplane, but they have the ability to embed the data into a higher-dimensional space, using the so-called **kernel trick**. Often, data that are not linearly separable in the original input space are easily separable in the higherdimensional space.
3. SVMs are nonparametric—the separating hyperplane is defined by a set of example points, not by a collection of parameter values. But while nearest-neighbor models need to retain all the examples, an SVM model keeps only the examples that are closest to the separating plane—usually only a small constant times the number of dimensions. Thus SVMs combine the advantages of nonparametric and parametric models: they have the flexibility to represent complex functions, but they are resistant to overfitting.

We see in Figure 19.21(a) a binary classification problem with three candidate decision boundaries, each a linear separator. Each of them is consistent with all the examples, so from the point of view of 0/1 loss, each would be equally good. Logistic regression would find some separating line; the exact location of the line depends on *all* the example points. The key insight of SVMs is that some examples are more important than others, and that paying attention to them can lead to better generalization.

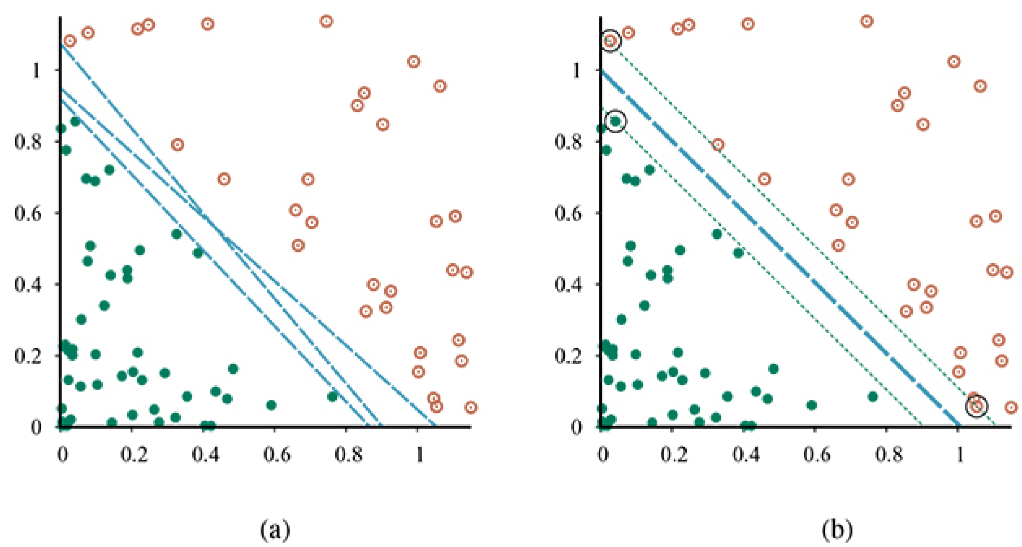


Figure 19.21 Support vector machine classification: (a) Two classes of points (orange open and green filled circles) and three candidate linear separators. (b) The maximum margin separator (heavy line), is at the midpoint of the **margin** (area between dashed lines). The **support vectors** (points with large black circles) are the examples closest to the separator; here there are three.

Consider the lowest of the three separating lines in (a). It comes very close to five of the black examples. Although it classifies all the examples correctly, and thus minimizes loss, it should make you nervous that so many examples are close to the line; it may be that other black examples will turn out to fall on the wrong side of the line.

SVMs address this issue: Instead of minimizing expected *empirical loss* on the training data, SVMs attempt to minimize expected *generalization* loss. We don't know where the as-yet-unseen points may fall, but under the probabilistic assumption that they are drawn from the same distribution as the previously seen examples, there are some arguments from computational learning theory (Section 19.5) suggesting that we minimize generalization loss by choosing the separator that is farthest away from the examples we have seen so far. We call this separator, shown in Figure 19.21(b) the **maximum margin separator**. The **margin** is the width of the area bounded by dashed lines in the figure—twice the distance from the separator to the nearest example point.

Now, how do we find this separator? Before showing the equations, some notation: Traditionally SVMs use the convention that class labels are +1 and -1, instead of the +1 and 0 we have been using so far. Also, whereas we previously put the intercept into the weight vector **w** (and a corresponding dummy 1 value into $x_{j,0}$), SVMs do not do that; they keep the intercept as a separate parameter, b.

With that in mind, the separator is defined as the set of points $\{\mathbf{x} : \mathbf{w} \cdot \mathbf{x} + b = 0\}$. We could search the space of **w** and b with gradient descent to find the parameters that maximize the margin while correctly classifying all the examples.

However, it turns out there is another approach to solving this problem. We won't show the details, but will just say that there is an alternative representation called the dual representation, in which the optimal solution is found by solving

$$\underset{\alpha}{\operatorname{argmax}} \sum_j \alpha_j - \frac{1}{2} \sum_{j,k} \alpha_j \alpha_k y_j y_k (\mathbf{x}_j \cdot \mathbf{x}_k) \tag{19.10}$$

subject to the constraints $\alpha_j \geq 0$ and $\sum_j \alpha_j y_j = 0$. This is a **quadratic programming** optimization problem, for which there are good software packages. Once we have found the vector α we can get back to **w** with the equation $\mathbf{w} = \sum_j \alpha_j y_j \mathbf{x}_j$, or we can stay in the dual representation. There are three important properties of Equation (19.10). First, the expression is convex; it has a single global maximum that can be found efficiently. Second, *the data enter the expression only in the form of dot products of pairs of points*. This second property is also true of the equation for the separator itself; once the optimal αj have been calculated, the equation is[11]

$$h(\mathbf{x}) = \operatorname{sign}\left(\sum_f \alpha_j y_j (\mathbf{x} \cdot \mathbf{x}_j) - b\right). \tag{19.11}$$

A final important property is that the weights α_j associated with each data point are *zero* except for the **support vectors**—the points closest to the separator. (They are called "support" vectors because they "hold up" the separating plane.) Because there are usually many fewer support vectors than examples, SVMs gain some of the advantages of parametric models.

What if the examples are not linearly separable? Figure 19.22(a) shows an input space defined by attributes $\mathbf{x} = (x_1, x_2)$, with positive examples ($y = +1$) inside a circular region and negative examples ($y = -1$) outside. Clearly, there is no linear separator for this problem. Now, suppose we re-express the input data—that is, we map each input vector $\mathbf{x}$ to a new vector of feature values, $F(\mathbf{x})$. In particular, let us use the three features

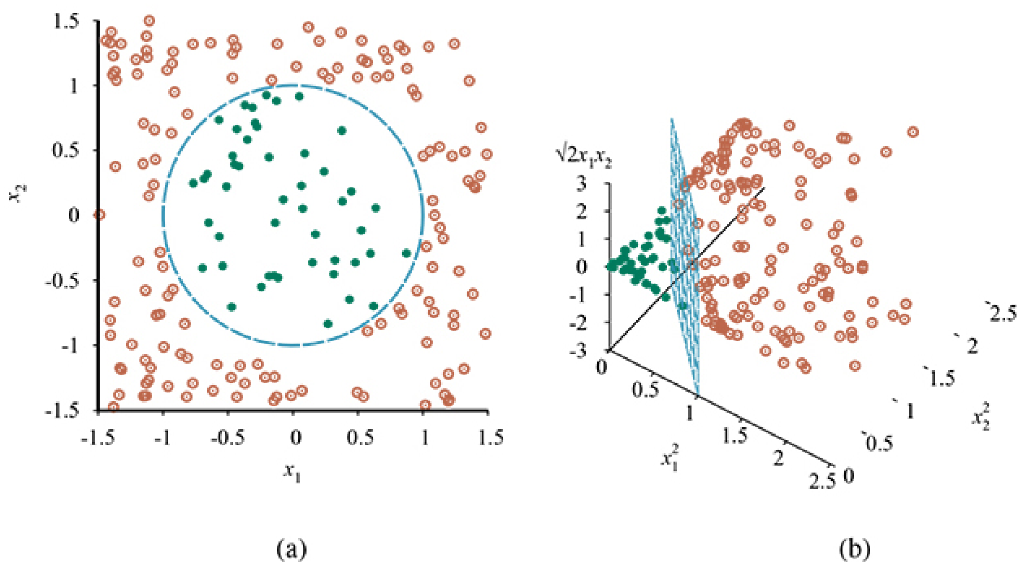


Figure 19.22 (a) A two-dimensional training set with positive examples as green filled circles and negative examples as orange open circles. The true decision boundary, $x_1^2 + x_2^2 \leq 1$ is also shown. (b) The same data after mapping into a three-dimensional input space $(x_1^2, x_2^2, \sqrt{2}x_1x_2)$. The circular decision boundary in (a) becomes a linear decision boundary in three dimensions. Figure 19.21(b) gives a closeup of the separator in (b).

$$f_1 = x_1^2, \qquad f_2 = x_2^2, \qquad f_3 = \sqrt{2}x_1x_2. \qquad (19.12)$$

We will see shortly where these came from, but for now, just look at what happens. Figure 19.22(b) shows the data in the new, three-dimensional space defined by the three features; the data are *linearly separable* in this space! This phenomenon is actually fairly general: if data are mapped into a space of sufficiently high dimension, then they will almost always be linearly separable—if you look at a set of points from enough directions, you'll find a way to make them line up. Here, we used only three dimensions;[12] Exercise 19.SVME asks you to show that four dimensions suffice for linearly separating a circle anywhere in the plane (not just at the origin), and five dimensions suffice to linearly separate any ellipse. In general (with some special cases excepted) if we have N data points then they will always be separable in spaces of $N - 1$ dimensions or more (Exercise 19.EMBE).

Now, we would not usually expect to find a linear separator in the input space $\mathbf{x}$, but we can find linear separators in the high-dimensional feature space $F(\mathbf{x})$ simply by replacing $\mathbf{x}_j \cdot \mathbf{x}_k$ in Equation (19.10) with $F(\mathbf{x}_j) \cdot F(\mathbf{x}_k)$. This by itself is not remarkable—replacing $\mathbf{x}$ by $F(\mathbf{x})$ in *any* learning algorithm has the required effect—but the dot product has some special properties. It turns out that $F(\mathbf{x}_j) \cdot F(\mathbf{x}_k)$ can often be computed without first computing F for each point. In our three-dimensional feature space defined by Equation (19.12), a little bit of algebra shows that

$$F(\mathbf{x}_j) \cdot F(\mathbf{x}_k) = (\mathbf{x}_j \cdot \mathbf{x}_k)^2 .$$

(That's why the $\sqrt{2}$ is in f_3). The expression $(\mathbf{x}_j \cdot \mathbf{x}_k)^2$ is called a **kernel function**, [13] and is usually written as $K(\mathbf{x}_j, \mathbf{x}_k)$. The kernel function can be applied to pairs of input data to evaluate dot products in some corresponding feature space. So, we can find linear separators in the higher-dimensional feature space $F(\mathbf{x})$ simply by replacing $\mathbf{x}_j \cdot x_k$ in Equation (19.10) with a kernel function $K(\mathbf{x}_j, \mathbf{x}_k)$. Thus, we can learn in the higher-dimensional space, but we compute only kernel functions rather than the full list of features for each data point.

The next step is to see that there's nothing special about the kernel $K(\mathbf{x}_j, \mathbf{x}_k) = (\mathbf{x}_j \cdot \mathbf{x}_k)^2$. It corresponds to a particular higher-dimensional feature space, but other kernel functions correspond to other feature spaces. A venerable result in mathematics, **Mercer's theorem** (1909), tells us that any "reasonable"[14] kernel function corresponds to some feature space. These feature spaces can be very large, even for innocuous-looking kernels. For example, the **polynomial kernel**, $K(\mathbf{x}_j, \mathbf{x}_k) = (1 + \mathbf{x}_j \cdot \mathbf{x}_k)^d$, corresponds to a feature space whose dimension is exponential in d. A common kernel is the Gaussian: $K(\mathbf{x}_j, \mathbf{x}_k) = e^{-\gamma|x_j - x_k|^2}$.

19.7.6 The kernel trick

This then is the clever **kernel trick**: Plugging these kernels into Equation (19.10), optimal linear separators can be found efficiently in feature spaces with billions of (or even infinitely many) dimensions. The resulting linear separators, when mapped back to the original input space, can correspond to arbitrarily wiggly, nonlinear decision boundaries between the positive and negative examples.

In the case of inherently noisy data, we may not want a linear separator in some highdimensional space. Rather, we'd like a decision surface in a lower-dimensional space that does not cleanly separate the classes, but reflects the reality of the noisy data. That is possible with the **soft margin** classifier, which allows examples to fall on the wrong side of the decision boundary, but assigns them a penalty proportional to the distance required to move them back to the correct side.

The kernel method can be applied not only with learning algorithms that find optimal linear separators, but also with any other algorithm that can be reformulated to work only with dot products of pairs of data points, as in Equations (19.10) and (19.11). Once this is done, the dot product is replaced by a kernel function and we have a **kernelized** version of the algorithm.

19.8 Ensemble Learning

So far we have looked at learning methods in which a single hypothesis is used to make predictions. The idea of **ensemble learning** is to select a collection, or **ensemble**, of hypotheses, h_1, h_2,...,h_n, and combine their predictions by averaging, voting, or by another level of machine learning. We call the individual hypotheses **base models** and their combination an **ensemble model**.

There are two reasons to do this. The first is to reduce bias. The hypothesis space of a base model may be too restrictive, imposing a strong bias (such as the bias of a linear decision boundary in logistic regression). An ensemble can be more expressive, and thus have less bias, than the base models. Figure 19.23 shows that an ensemble of three linear classifiers can represent a triangular region that could not be represented by a single linear classifier. An ensemble of n linear classifiers allows more functions to be realizable, at a cost of only n times more computation; this is often better than allowing a completely general hypothesis space that might require exponentially more computation.

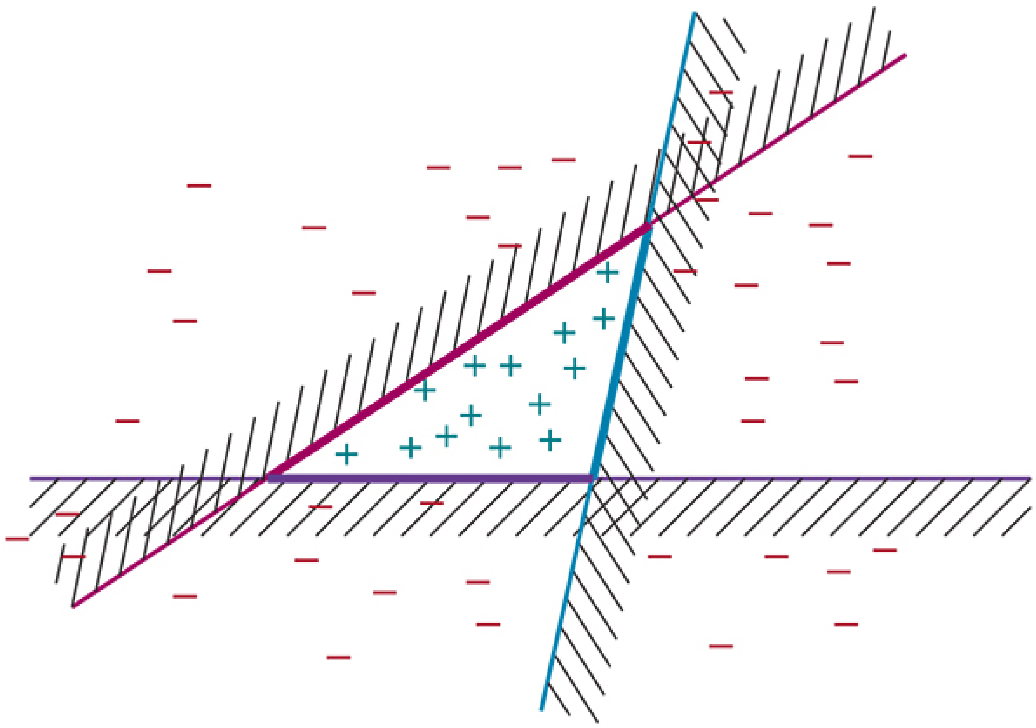


Figure 19.23 Illustration of the increased expressive power obtained by ensemble learning. We take three linear threshold hypotheses, each of which classifies positively on the unshaded side, and classify as positive any example classified positively by all three. The resulting triangular region is a hypothesis not expressible in the original hypothesis space.

The second reason is to reduce variance. Consider an ensemble of $K = 5$ binary classifiers that we combine using majority voting. For the ensemble to misclassify a new example, *at least three of the five classifiers have to misclassify it*. The hope is that this is less likely than a single misclassification by a single classifier. To quantify that, suppose you have trained a single classifier that is correct in 80% of cases. Now create an ensemble of 5 classifiers, each trained on a different subset of the data so that they are independent. Let's assume this leads to some reduction in quality, and each individual classifier is correct in only 75% of cases. But together, the majority vote of the ensemble will be correct in 89% of cases (and 99% with 17 classifiers), assuming true independence.

In practice the independence assumption is unreasonable—individual classifiers share some of the same data and assumptions, and thus are not completely independent, and will share some of the same errors. But if the component classifiers are at least somewhat uncorrelated then ensemble learning will make fewer misclassifications. We will now consider four ways of creating ensembles: bagging, random forests, stacking, and boosting.

19.8.1 Bagging

In **bagging**, [15] we generate K distinct training sets by sampling with replacement from the original training set. That is, we randomly pick N examples from the training set, but each of those picks might be an example we picked before. We then run our machine learning algorithm on the N examples to get a hypothesis. We repeat this process K times, getting K different hypotheses. Then, when asked to predict the value of a new input, we aggregate the predictions from all K hypotheses. For classification problems, that means taking the plurality vote (the majority vote for binary classification). For regression problems, the final output is the average:

$$h(\mathbf{x}) = \frac{1}{K}\sum_{i=1}^{\mathrm{K}} h_i(\mathbf{x})$$

Bagging tends to reduce variance and is a standard approach when there is limited data or when the base model is seen to be overfitting. Bagging can be applied to any class of model, but is most commonly used with decision trees. It is appropriate because decision trees are unstable: a slightly different set of examples can lead to a wildly different tree. Bagging smoothes out this variance. If you have access to multiple computers then bagging is efficient, because the hypotheses can be computed in parallel.

19.8.2 Random forests

Unfortunately, bagging decision trees often ends up giving us K trees that are highly correlated. If there is one attribute with a very high information gain, it is likely to be the root of most of the trees. The **random forest** model is a form of decision tree bagging in which we take extra steps to make the ensemble of K trees more diverse, to reduce variance. Random forests can be used for classification or regression.

The key idea is to randomly vary the *attribute choices* (rather than the training examples). At each split point in constructing the tree, we select a random sampling of attributes, and then compute which of those gives the highest information gain. If there are n attributes, a common default choice is that each split randomly picks $\sqrt{n}$ attributes to consider for classification problems, or $n/3$ for regression problems.

A further improvement is to use randomness in selecting the split point *value*: for each selected attribute, we randomly sample several candidate values from a uniform distribution over the attribute's range. Then we select the value that has the highest information gain. That makes it more likely that every tree in the forest will be different. Trees constructed in this fashion are called **extremely randomized trees (ExtraTrees)**.

Random forests are efficient to create. You might think that it would take K times longer to create an ensemble of K trees, but it is not that bad, for three reasons: (a) each split point runs faster because we are considering fewer attributes, (b) we can skip the pruning step for each individual tree, because the ensemble as a whole decreases overfitting, and (c) if we happen to have K computers available, we can build all the trees in parallel. For example, Adele Cutler reports that for a 100-attribute problem, if we have just three CPUs we can grow a forest of $K = 100$ trees in about the same time as it takes to create a single decision tree on a single CPU.

All the hyperparameters of random forests can be trained by cross-validation: the number of trees K, the number of examples used by each tree N (often expressed as a percentage of the complete data set), the number of attributes used at each split point (often expressed as a function of the total number of attributes, such as $\sqrt{n}$), and the number of random split points tried if we are using ExtraTrees. In place of the regular cross-validation strategy, we could measure the **out-of-bag error**: the mean error on each example, using only the trees whose example set didn't include that particular example.

We have been warned that more complex models can be prone to overfitting, and observed that to be true for decision trees, where we found that **pruning** was an answer to prevent overfitting. Random forests are complex, unpruned models. Yet they are resistant to overfitting. As you increase capacity by adding more trees to the forest they tend to improve on validation-set error rate. The curve typically looks like Figure 19.9(b), not (a).

Breiman (2001) gives a mathematical proof that (in almost all cases) as you add more trees to the forest, the error converges; it does not grow. One way to think of it is that the random selection of attributes yields a variety of trees, thus reducing variance, but because we don't need to prune the trees, they can cover the full input space at higher resolution. Some number of trees can cover unique cases that appear only a few times in the data, and their votes can prove decisive, but they can be outvoted when they do not apply. That said, random forests are not totally immune to overfitting. Although the error can't increase in the limit, that does not mean that the error will go to zero.

Random forests have been very successful across a wide variety of application problems. In Kaggle data science competitions they were the most popular approach of winning teams from 2011 through 2014, and remain a common approach to this day (although **deep learning** and **gradient boosting** have become even more common among recent winners). The randomForest package in R has been a particular favorite. In finance, random forests have been used for credit card default prediction, household income prediction, and option pricing. Mechanical applications include machine fault diagnosis and remote sensing. Bioin-formatic and medical applications include diabetic retinopathy, microarray gene expression, mass spectrum protein expression analysis, biomarker discovery, and protein-protein interaction prediction.

19.8.3 Stacking

Whereas bagging combines multiple base models of the same model class trained on different data, the technique of **stacked generalization** (or **stacking** for short) combines multiple base models from different model classes trained on the same data. For example, suppose we are given the restaurant data set, the first row of which is shown here:

$$\mathbf{x}_1 = \text{Yes, No, No, Yes, Some, \$\$\$, No, Yes, French, } 0-10;\ y_1 = \text{Yes}$$

We separate the data into training, validation, and test sets and use the training set to train, say, three separate base models—an SVM model, a logistic regression model, and a decision tree model.

In the next step we take the validation data set and augment each row with the predictions made from the three base models, giving us rows that look like this (where the predictions are shown in bold):

$$\mathbf{x}_2 = \text{Yes, No, No, Yes, Full, \$, No, No, Thai, } 30-60, \ \mathbf{Yes, No, No}; \ y_2 = \text{No}$$

We use this validation set to train a new ensemble model, let's say a logistic regression model (but it need not be one of the base model classes). The ensemble model can use the predictions and the original data as it sees fit. It might learn a weighted average of the base models, for example that the predictions should be weighted in a ratio of 50%:30%:20%. Or it might learn nonlinear interactions between the data and the predictions, perhaps trusting the SVM model more when the wait time is long, for example. We used the same training data to train each of the base models, and then used the held-out validation data (plus predictions) to train the ensemble model. It is also possible to use cross-validation if desired.

The method is called "stacking" because it can be thought of as a layer of base models with an ensemble model stacked above it, operating on the output of the base models. In fact, it is possible to stack multiple layers, each one operating on the output of the previous layer. Stacking reduces bias, and usually leads to performance that is better than any of the individual base models. Stacking is frequently used by winning teams in data science competitions (such as Kaggle and the KDD Cup), because individuals can work independently, each refining their own base model, and then come together to build the final stacked ensemble model.

19.8.4 Boosting

The most popular ensemble method is called **boosting**. To understand how it works, we need first to introduce the idea of a **weighted training set**, in which each example has an associated weight $w_j \geq 0$ that describes how much the example should count during training. For example, if one example had a weight of 3 and the other examples all had a weight of 1, that would be equivalent to having 3 copies of the one example in the training set.

Boosting starts with equal weights $w_j = 1$ for all the examples. From this training set, it generates the first hypothesis, h_1. In general, h_1 will classify some of the training examples correctly and some incorrectly. We would like the next hypothesis to do better on the misclas-sified examples, so we increase their weights while decreasing the weights of the correctly classified examples.

From this new weighted training set, we generate hypothesis h_2. The process continues in this way until we have generated K hypotheses, where K is an input to the boosting algorithm. Examples that are difficult to classify will get increasingly larger weights until the algorithm is forced to create a hypothesis that classifies them correctly. Note that this is a greedy algorithm in the sense that it does not backtrack; once it has chosen a hypothesis h_i it will never undo that choice; rather it will add new hypotheses. It is also a sequential algorithm, so we can't compute all the hypotheses in parallel as we could with bagging.

The final ensemble lets each hypothesis vote, as in bagging, except that each hypothesis gets a weighted number of votes—the hypotheses that did better on their respective weighted training sets are given more voting weight. For regression or binary classification we have

$$h(\mathbf{x}) = \sum_{i=1}^{K} z_i h_i(\mathbf{x})$$

where z_i is the weight of the ith hypothesis. (This weighting of hypotheses is distinct from the weighting of examples.)

Figure 19.24 shows how the algorithm works conceptually. There are many variants of the basic boosting idea, with different ways of adjusting the example weights and combining the hypotheses. The variants all share the general idea that difficult examples get more weight as we move from one hypothesis to the next. Like the Bayesian learning methods we will see in Chapter 21, they also give more weight to more accurate hypotheses.

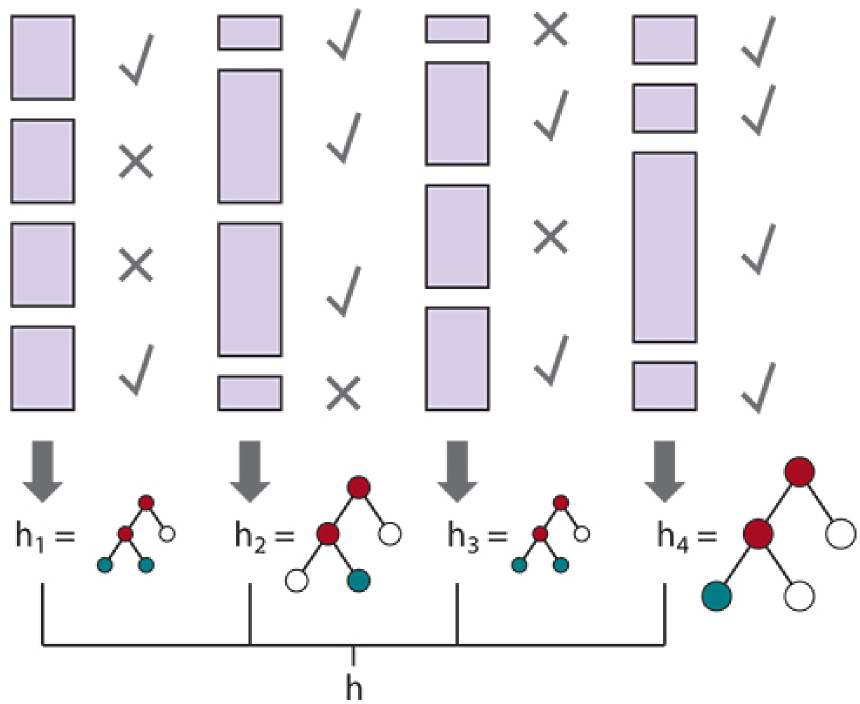


Figure 19.24 How the boosting algorithm works. Each shaded rectangle corresponds to an example; the height of the rectangle corresponds to the weight. The checks and crosses indicate whether the example was classified correctly by the current hypothesis. The size of the decision tree indicates the weight of that hypothesis in the final ensemble.

One specific algorithm, called ADABOOST, is shown in Figure 19.25. It is usually applied with decision trees as the component hypotheses; often the trees are limited in size. ADABOOST has a very important property: if the input learning algorithm L is a **weak learning** algorithm—which means that L always returns a hypothesis with accuracy on the training set that is slightly better than random guessing (that is, $50\% + \epsilon$ for Boolean classification)—then ADABOOST will return a hypothesis that *classifies the training data perfectly* for large enough K. Thus, the algorithm *boosts* the accuracy of the original learning algorithm on the training data.

```
function ADABOOST(examples, L, K) returns a hypothesis
  inputs: examples, set of N labeled examples (x_1, y_1), ..., (x_N, y_N)
          L, a learning algorithm
          K, the number of hypotheses in the ensemble
  local variables: w, a vector of N example weights, initially all 1/N
                   h, a vector of K hypotheses
                   z, a vector of K hypothesis weights

  ϵ ← a small positive number, used to avoid division by zero
  for k = 1 to K do
     h[k] ← L(examples, w)
     error ← 0
     for j = 1 to N do        // Compute the total error for h[k]
        if h[k](x_j) ≠ y_j then error ← error + w[j]
     if error > 1/2 then break from loop
     error ← min(error, 1 − ϵ)
     for j = 1 to N do        // Give more weight to the examples h[k] got wrong
        if h[k](x_j) = y_j then w[j] ← w[j] · error/(1 − error)
     w ← NORMALIZE(w)
     z[k] ← 1/2 log((1 − error)/error)        // Give more weight to accurate h[k]
  return Function(x) : ∑ z_i h_i(x)
```

Figure 19.25 The ADABOOST variant of the boosting method for ensemble learning. The algorithm generates hypotheses by successively reweighting the training examples. The function WEIGHTED-MAJORITY generates a hypothesis that returns the output value with the highest vote from the hypotheses in **h**, with votes weighted by **z**. For regression problems, or for binary classification with two classes -1 and 1, this is $\sum_k \mathbf{h}[k]\mathbf{z}[k]$.

In other words, boosting can overcome any amount of bias in the base model, as long as the base model is ϵ better than random guessing. (In our pseudocode we stop generating hypotheses if we get one that is worse than random.) This result holds no matter how inexpressive the original hypothesis space and no matter how complex the function being learned. The exact formulas for weights in Figure 19.25 (with *error*/ (1 –

error, etc.) are chosen to make the proof of this property easy (see Freund and Schapire, 1996). Of course, this property does not guarantee accuracy on previously unseen examples.

Let us see how well boosting does on the restaurant data. We will choose as our original hypothesis space the class of **decision stumps**, which are decision trees with just one test, at the root. The lower curve in Figure 19.26(a) shows that unboosted decision stumps are not very effective for this data set, reaching a prediction performance of only 81% on 100 training examples. When boosting is applied (with $K = 5$), the performance is better, reaching 93% after 100 examples.

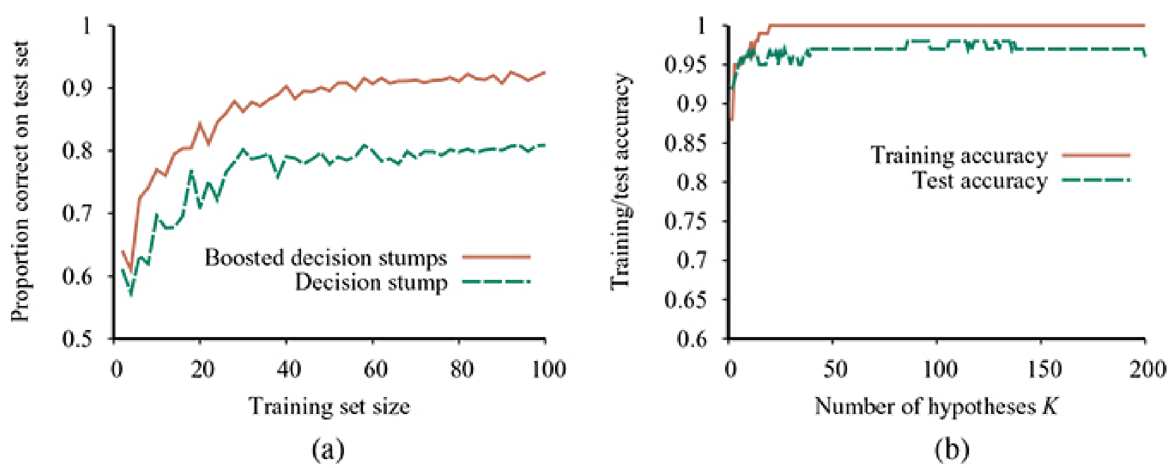


Figure 19.26 (a) Graph showing the performance of boosted decision stumps with $K = 5$ versus unboosted decision stumps on the restaurant data. (b) The proportion correct on the training set and the test set as a function of K, the number of hypotheses in the ensemble. Notice that the test set accuracy improves slightly even after the training accuracy reaches 1, i.e., after the ensemble fits the data exactly.

An interesting thing happens as the ensemble size K increases. Figure 19.26(b) shows the training set performance (on 100 examples) as a function of K. Notice that the error reaches zero when K is 20; that is, a weighted-majority combination of 20 decision stumps suffices to fit the 100 examples exactly—this is the interpolation pont. As more stumps are added to the ensemble, the error remains at zero. The graph also shows that *the test set performance continues to increase long after the training set error has reached zero*. At $K = 20$, the test performance is 0.95 (or 0.05 error), and the performance increases to 0.98 as late as $K = 137$, before gradually dropping to 0.95.

This finding, which is quite robust across data sets and hypothesis spaces, came as quite a surprise when it was first noticed. Ockham's razor tells us not to make hypotheses more complex than necessary, but the graph tells us that the predictions *improve* as the ensemble hypothesis gets more complex! Various explanations have been proposed for this. One view is that boosting approximates **Bayesian learning** (see Chapter 21), which can be shown to be an optimal learning algorithm, and the approximation improves as more hypotheses are added. Another possible explanation is that the addition of further hypotheses enables the ensemble to be more confident in its distinction between positive and negative examples, which helps it when it comes to classifying new examples.

19.8.5 Gradient boosting

For regression and classification of factored tabular data, **gradient boosting**, sometimes called gradient boosting machines (GBM) or gradient boosted regression trees (GBRT), has become a very popular method. As the name implies, gradient boosting is a form of boosting using gradient descent. Recall that in ADABOOST, we start with one hypothesis $\mathbf{h}_1$, and boost it with a sequence of hypotheses that pay special attention to the examples that the previous ones got wrong. In gradient boosting we also add new boosting hypotheses, which pay attention not to specific examples, but to the **gradient** between the right answers and the answers given by the previous hypotheses.

As in the other algorithms that used gradient descent, we start with a differentiable loss function; we might use squared error for regression, or logarithmic loss for classification. As in ADABOOST, we then build a decision tree. In Section 19.6.2, we used gradient descent to minimize the parameters of a model—we calculate the loss, and update the parameters in the direction of less loss. With gradient boosting, we are not updating parameters of the existing model, we are updating the parameters of the next tree—but we must do that in a way that reduces the loss by moving in the right direction along the gradient.

As in the models we saw in Section 19.4.3, **regularization** can help prevent overfitting. That can come in the form of limiting the number of trees or their size (in terms of their depth or number of nodes). It can come from the learning rate, α, which says how far to move along the direction of the gradient; values in the range 0.1 to 0.3 are common, and the smaller the learning rate, the more trees we will need in the ensemble.

Gradient boosting is implemented in the popular XGBOOST (eXtreme Gradient Boosting) package, which is routinely used for both large-scale applications in industry (for problems with billions of examples), and by the winners of data science competitions (in 2015, it was used by every team in the top 10 of the KDDCup). XGBOOST does gradient boosting with pruning and regularization, and takes care to be efficient, carefully organizing memory to avoid cache misses, and allowing for parallel computation on multiple machines.

19.8.6 Online learning

So far, everything we have done in this chapter has relied on the assumption that the data are i.i.d. (independent and identically distributed). On the one hand, that is a sensible assumption: if the future bears no resemblance to the past, then how can we predict anything? On the other hand, it is too strong an assumption: we know that there are correlations between the past and the future, and in complex scenarios it is unlikely that we will capture all the data that would make the future independent of the past given the data.

In this section we examine what to do when the data are not i.i.d.—when they can change over time. In this case, it matters *when* we make a prediction, so we will adopt the perspective called **online learning**: an agent receives an input x_j from nature, predicts the corresponding y_j, and then is told the correct answer. Then the process repeats with x_{j+1}, and so on. One might think this task is hopeless–if nature is adversarial, all the predictions may be wrong. It turns out that there are some guarantees we can make.

Let us consider the situation where our input consists of predictions from a panel of experts. For example, each day K pundits predict whether the stock market will go up or down, and our task is to pool those predictions and make our own. One way to do this is to keep track of how well each expert performs, and choose to believe them in proportion to their past performance. This is called the **randomized weighted majority algorithm**. We can describe it more formally:

Initialize a set of weights $\{w_1,...,w_K\}$ all to 1.

for each problem to be solved **do**

1. Receive the predictions $\{\hat{y}_1,....,\hat{y}_K\}$ from the experts.
2. Randomly choose an expert $k*$ in proportion to its weight: $P(k) = w_k$.
3. **yield** $\hat{y}_{k*}$ as the answer to this problem.
4. Receive the correct answer y.
5. For each expert k such that $\hat{y}_k \neq y$, update $w_k \leftarrow \beta w_k$
6. Normalize the weights so that $\sum_k w_k = 1$.

Here β is a number, $0 < \beta < 1$, that tells how much to penalize an expert for each mistake.

We measure the success of this algorithm in terms of **regret**, which is defined as the number of additional mistakes we make compared to the expert who, in hindsight, had the best prediction record. Let $M*$ be the number of mistakes made by the best expert. Then the number of mistakes, M, made by the random weighted majority algorithm, is bounded by[16]

$$M < \frac{M^* \ln(1/\beta) + \ln K}{1-\beta}.$$

This bound holds for *any* sequence of examples, even ones chosen by adversaries trying to do their worst. To be specific, when there are $K = 10$ experts, if we choose $\beta = 1/2$ then our number of mistakes is bounded by $1.39M* + 4.6$, and if $\beta = 3/4$ by $1.15M* + 9.2$. In general, if β is close to 1 then we are responsive to change over the long run; if the best expert changes, we will pick up on it before too long. However, we pay a penalty at the beginning, when we start with all experts trusted equally; we may accept the advice of the bad experts for too long. When β is closer to 0, these two factors are reversed. Note that we can choose β so that M gets asymptotically close to $M*$ in the long run; this is called **no-regret learning** (because the average amount of regret per trial tends to 0 as the number of trials increases).

Online learning is helpful when the data may be changing rapidly over time. It is also useful for applications that involve a large collection of data that is constantly growing, even if changes are gradual. For example, with a data set of millions of Web images, you wouldn't want to retrain from scratch every time a single new image is added. It would be more practical to have an online algorithm that allows images to be added incrementally. For most learning algorithms based on minimizing loss, there is an online version based on minimizing regret. Many of these online algorithms come with guaranteed bounds on regret.

It may seem surprising that there are such tight bounds on how well we can do compared to a panel of experts. What is even more surprising is that when such panels convene to prognosticate about political contests or sporting events, the viewing public is so willing to listen to their predictions and so uninterested in knowing their error rates.

19.9 Developing Machine Learning Systems

In this chapter we have concentrated on explaining the *theory* of machine learning. The *practice* of using machine learning to solve practical problems is a separate discipline. Over the last 50 years, the software industry has evolved a software development methodology that makes it more likely that a (traditional) software project will be a success. But we are still in the early stages of defining a methodology for machine learning projects; the tools and techniques are not as well-developed. Here is a breakdown of typical steps in the process.

19.9.1 Problem formulation

The first step is to Figure out what problem you want to solve. There are two parts to this. First ask, “what problem do I want to solve for my users?” An answer such as “make it easier for users to organize and access their photos” is too vague; “help a user find all photos that match a specific term, such as *Paris*” is better. Then ask, “what part(s) of the problem can be solved by machine learning?” perhaps settling on “learn a function that maps a photo to a set of labels; then, when given a label as a query, retrieve all photos with that label.”

To make this concrete, you need to specify a loss function for your machine learning component, perhaps measuring the system’s accuracy at predicting a correct label. This objective should be correlated with your true goals, but usually will be distinct—the true goal might be to maximize the number of users you gain and keep on your system, and the revenue that they produce. Those are metrics you should track, but not necessarily ones that you can directly build a machine learning model for.

When you have decomposed your problem into parts, you may find that there are multiple components that can be handled by old-fashioned software engineering, not machine learning. For example, for a user who asks for “best photos,” you could implement a simple procedure that sorts photos by the number of likes and views. Once you have developed your overall system to the point where it is viable, you can then go back and optimize, replacing the simple components with more sophisticated machine learning models.

Part of problem formulation is deciding whether you are dealing with supervised, unsupervised, or reinforcement learning. The distinctions are not always so crisp. In **semisupervised learning** we are given a few labeled examples and use them to mine more information from a large collection of unlabeled examples. This has become a common approach, with companies emerging whose missions are to quickly label some examples, in order to help machine learning systems make better use of the remaining unlabeled examples.

Sometimes you have a choice of which approach to use. Consider a system to recommend songs or movies to customers. We could approach this as a supervised learning problem, where the inputs include a representation of the customer and the labeled output is whether or not they liked the recommendation, or we could approach it as a reinforcement learning problem, where the system makes a series of recommendation actions, and occasionally gets a reward from the customer for making a good suggestion.

The labels themselves may not be the oracular truths that we hope for. Imagine that you are trying to build a system to guess a person’s age from a photo. You gather some labeled examples by having people upload photos and state their age. That’s supervised learning. But in reality some of the people lied about their age. It’s not just that there is random noise in the data; rather the inaccuracies are systematic, and to uncover them is an unsupervised learning problem involving images, self-reported ages, and true (unknown) ages. Thus, both noise and lack of labels create a continuum between supervised and unsupervised learning. The field of **weakly supervised learning** focuses on using labels that are noisy, imprecise, or supplied by non-experts.

19.9.2 Data collection, assessment, and management

Every machine learning project needs data; in the case of our photo identification project there are freely available image data sets, such as **ImageNet**, which has over 14 million photos with about 20,000 different labels. Sometimes we may have to manufacture our own data, which can be done by our own labor, or by **crowdsourcing** to paid workers or unpaid volunteers operating over an Internet service. Sometimes data come from your users. For example, the Waze navigation service encourages users to upload data about traffic jams, and uses that to provide up-to-date navigation directions for all users. Transfer learning (see Section 22.7.2) can be used when you don’t have enough of your own data: start with a publicly available general-purpose data set (or a model that has been pretrained on this data), and then add specific data from your users and retrain.

If you deploy a system to users, your users will provide feedback—perhaps by clicking on one item and ignoring the others. You will need a strategy for dealing with this data. That involves a review with privacy experts (see Section 28.3.2) to make sure that you get the proper permission for the data you collect, and that you have processes for insuring the integrity of the user’s data, and that they understand what you will do with it. You also need to ensure that your processes are fair and unbiased (see Section 28.3.3). If there is data that you feel is too sensitive to collect but that would be useful for a machine learning model, consider a federated learning approach where the data stays on the user's device, but model parameters are shared in a way that does not reveal private data.

It is good practice to maintain **data provenance** for all your data. For each column in your data set, you should know the exact definition, where the data come from, what the possible values are, and who has worked on it. Were there periods of time in which a data feed was interrupted? Did the definition of some data source evolve over time? You'll need to know this if you want to compare results across time periods.

This is particularly true if you are relying on data that are produced by someone else— their needs and yours might diverge, and they might end up changing the way the data are produced, or might stop updating it all together. You need to monitor your data feeds to catch this. Having a reliable, flexible, secure, data-handling pipeline is more critical to success than the exact details of the machine learning algorithm. Provenance is also important for legal reasons, such as compliance with privacy law.

For any task there will be questions about the data: Is this the right data for my task? Does it capture enough of the right inputs to give us a chance of learning a model? Does it contain the outputs I want to predict? If not, can I build an unsupervised model? Or can I label a portion of the data and then do semisupervised learning? Is it relevant data? It is great to have 14 million photos, but if all your users are specialists interested in a specific topic, then a general database won't help—you'll need to collect photos on the specific topic. How much training data is enough? (Do I need to collect more data? Can I discard some data to make computation faster?) The best way to answer this is to reason by analogy to a similar project with known training set size.

Once you get started you can draw a learning curve (see Figure 19.7) to see if more data will help, or if learning has already plateaued. There are endless ad hoc, unjustified rules of thumb for the number of training examples you'll need: millions for hard problems; thousands for average problems; hundreds or thousands for each class in a classification problem; 10 times more examples than parameters of the model; 10 times more examples than input features; $O(d \log d)$ examples for d input features; more examples for nonlinear models than for linear models; more examples if greater accuracy is required; fewer examples if you use regularization; enough examples to achieve the statistical power necessary to reject the null hypothesis in classification. All these rules come with caveats—as does the sensible rule that suggests trying what has worked in the past for similar problems.

You should think defensively about your data. Could there be data entry errors? What can be done with missing data fields? If you collect data from your customers (or other people) could some of the people be adversaries out to game the system? Are there spelling errors or inconsistent terminology in text data? (For example, do "Apple," "AAPL," and "Apple Inc." all refer to the same company?) You will need a process to catch and correct all these potential sources of data error.

When data are limited, **data augmentation** can help. For example, with a data set of images, you can create multiple versions of each image by rotating, translating, cropping, or scaling each image, or by changing the brightness or color balance or adding noise. As long as these are small changes, the image label should remain the same, and a model trained on such augmented data will be more robust.

Sometimes data are plentiful but are classified into **unbalanced classes**. For example, a training set of credit card transactions might consist of 10,000,000 valid transactions and 1,000 fraudulent ones. A classifier that says “valid” regardless of the input will achieve 99.99% accuracy on this data set. To go beyond that, a classifier will have to pay more attention to the fraudulent examples. To help it do that, you can **undersample** the majority class (i.e., ignore some of the “valid” class examples) or **oversample** the minority class (i.e., duplicate some of the “fraudulent” class examples). You can use a weighted loss function that gives a larger penalty to missing a fraudulent case.

Boosting can also help you focus on the minority class. If you are using an ensemble method, you can change the rules by which the ensemble votes and give “fraudulent” as the response even if only a minority of the ensemble votes for “fraudulent.” You can help balance unbalanced classes by generating synthetic data with techniques such as SMOTE (Chawla *et al.*, 2002) or ADASYN (He *et al.*, 2008).

You should carefully consider **outliers** in your data. An outlier is a data point that is far from other points. For example, in the restaurant problem, if price were a numeric value rather than a categorical one, and if one example had a price of $316 while all the others were $30 or less, that example would be an outlier. Methods such as linear regression are susceptible to outliers because they must form a single global linear model that takes all inputs into account—they can’t treat the outlier differently from other example points, and thus a single outlier can have a large effect on all the parameters of the model.

With attributes like price that are positive numbers, we can diminish the effect of outliers by transforming the data, taking the logarithm of each value, so $20, $25, and $316 become 1.3, 1.4, and 2.5. This makes sense from a practical point of view because the high value now has less influence on the model, and from a theoretical point of view because, as we saw in Section 15.3.2, the utility of money is logarithmic.

Methods such as decision trees that are built from multiple local models can treat outliers individually: it doesn't matter if the biggest value is \$300 or \$31; either way it can be treated in its own local node after a test of the form $cost \leq 30$. That makes decision trees (and thus random forests and gradient boosting) more robust to outliers.

Feature engineering

After correcting overt errors, you may also want to preprocess your data to make it easier to digest. We have already seen the process of quantization: forcing a continuous valued input, such as the wait time, into fixed bins (0–10 minutes, 10–30, 30–60, or > 60). Domain knowledge can tell you what thresholds are important, such as comparing $age \geq 18$ when studying voting patterns. We also saw (page 706) that nearest-neighbor algorithms perform better when data are normalized to have a standard deviation of 1. With categorical attributes such as sunny/cloudy/rainy, it is often helpful to transform the data into three separate Boolean attributes, exactly one of which is true (we call this a **one-hot encoding**). This is particularly useful when the machine learning model is a neural network.

You can also introduce new attributes based on your domain knowledge. For example, given a data set of customer purchases where each entry has a date attribute, you might want to augment the data with new attributes saying whether the date is a weekend or holiday.

As another example, consider the task of estimating the true value of houses that are for sale. In Figure 19.13 we showed a toy version of this problem, doing linear regression of house size to asking price. But we really want to estimate the selling price of a house, not the asking price. To solve this task we'll need data on actual sales. But that doesn't mean we should throw away the data about asking price—we can use it as one of the input features. Besides the size of the house, we'll need more information: the number of rooms, bedrooms, and bathrooms; whether the kitchen and bathrooms have been recently remodeled; the age of the house and perhaps its state of repair; whether it has central heating and air conditioning; the size of the yard and the state of the landscaping.

We'll also need information about the lot and the neighborhood. But how do we define neighborhood? By zip code? What if a zip code straddles a desirable and an undesirable neighborhood? What about the school district? Should the *name* of the school district be a feature, or the *average test scores*? The ability to do a good job of feature engineering is critical to success. As Pedro Domingos (2012) says, "At the end of the day, some machine learning projects succeed and some fail. What makes the difference? Easily the most important factor is the features used."

Exploratory data analysis and visualization

John Tukey (1977) coined the term **exploratory data analysis** (EDA) for the process of exploring data in order to gain an understanding of it, not to make predictions or test hypotheses. This is done mostly with visualizations, but also with summary statistics. Looking at a few histograms or scatter plots can often help determine if data are missing or erroneous; whether your data are normally distributed or heavy-tailed; and what learning model might be appropriate.

It can be helpful to cluster your data and then visualize a prototype data point at the center of each cluster. For example, in the data set of images, I can identify that here is a cluster of cat faces; nearby is a cluster of sleeping cats; other clusters depict other objects. Expect to iterate several times between visualizing and modeling—to create clusters you need a distance function to tell you which items are near each other, but to choose a good distance function you need some feel for the data.

It is also helpful to detect outliers that are far from the prototypes; these can be considered **critics** of the prototype model, and can give you a feel for what type of errors your system might make. An example would be a cat wearing a lion costume.

Our computer display devices (screens or paper) are two-dimensional, which means that it is easy to visualize two-dimensional data. And our eyes are experienced at understanding three-dimensional data that has been projected down to two dimensions. But many data sets have dozens or even millions of dimensions. In order to visualize them we can do dimensionality reduction, projecting the data down to a **map** in two dimensions (or sometimes to three dimensions, which can then be explored interactively).[17]

The map can’t maintain all relationships between data points, but should have the property that similar points in the original data set are close together in the map. A technique called **t-distributed stochastic neighbor embedding (t-SNE)** does just that. Figure 19.27 shows a t-SNE map of the MNIST digit recognition data set. Data analysis and visualization packages such as Pandas, Bokeh, and Tableau can make it easier to work with your data.

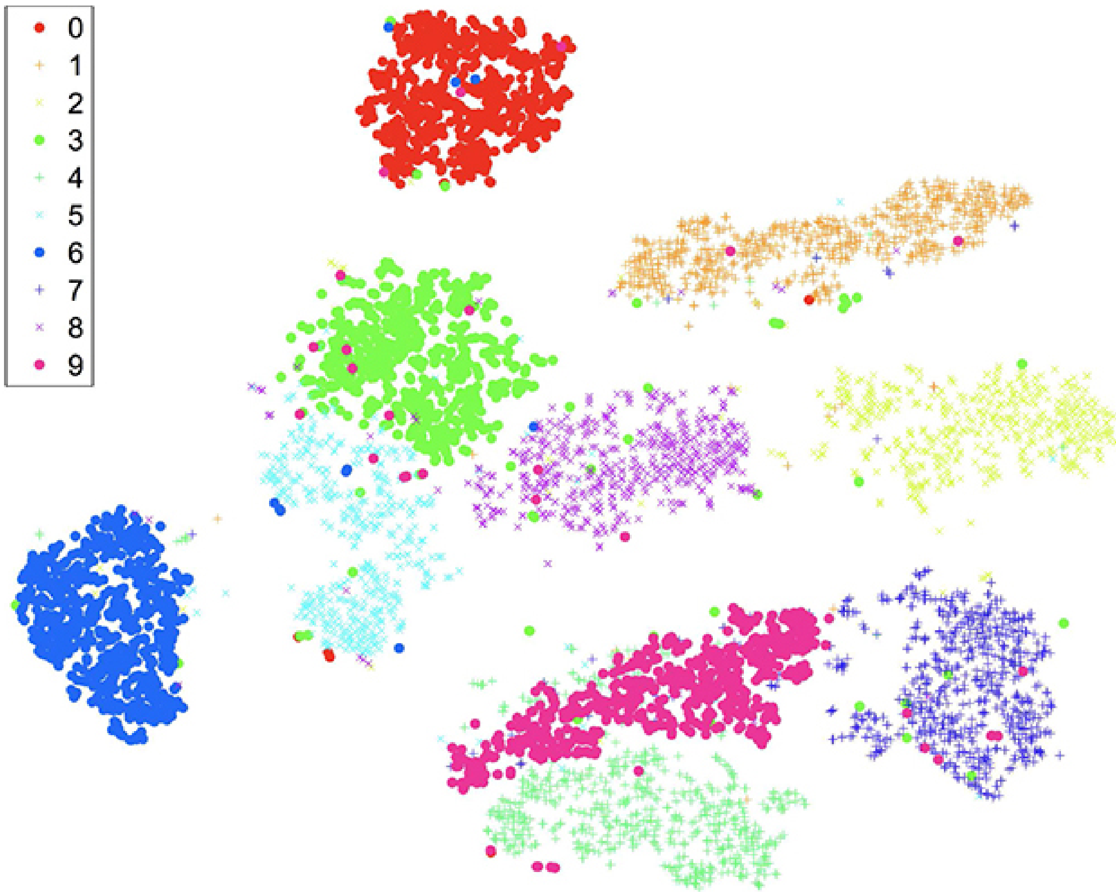


Figure 19.27 A two-dimensional t-SNE map of the MNIST data set, a collection of 60,000 images of handwritten digits, each 28 × 28 pixels and thus 784 dimensions. You can clearly see clusters for the ten digits, with a few confusions in each cluster; for example the top cluster is for the digit 0, but within the bounds of the cluster are a few data points representing the digits 3 and 6. The t-SNE algorithm finds a representation that accentuates the differences between clusters.

19.9.3 Model selection and training

With cleaned data in hand and an intuitive feel for it, it is time to build a model. That means choosing a model class (random forests? deep neural networks? an ensemble?), training your model with the training data, tuning any hyperparameters of the class (number of trees? number of layers?) with the validation data, debugging the process, and finally evaluating the model on the test data.

There is no guaranteed way to pick the best model class, but there are some rough guidelines. Random forests are good when there are a lot of categorical features and you believe that many of them may be irrelevant. Nonparametric methods are good when you have a lot of data and no prior knowledge, and when you don't want to worry too much about choosing just the right features (as long as there are fewer than 20 or so). However, nonparametric methods usually give you a function h that is more expensive to run.

Logistic regression does well when the data are linearly separable, or can be converted to be so with clever feature engineering. Support vector machines are a good method to try when the data set is not too large; they perform similarly to logistic regression on separable data and can be better for high-dimensional data. Problems dealing with pattern recognition, such as image or speech processing, are most often approached with deep neural networks (see Chapter 22).

Choosing hyperparameters can be done with a combination of experience—do what worked well in similar past problems—and search: run experiments with multiple possible values for hyperparameters. As you run more experiments you will get ideas for different models to try. However, if you measure performance on the validation data, get a new idea, and run more experiments, then you run the risk of overfitting on the validation data. If you have enough data, you may want to have several separate validation data sets to avoid this problem. This is especially true if you inspect the validation data by eye, rather than just run evaluations on it.

Suppose you are building a classifier—for example a system to classify spam email. Labeling a legitimate piece of mail as spam is called a **false positive**. There will be a tradeoff between false positives and false negatives (labeling a piece of spam as legitimate); if you want to keep more legitimate mail out of the spam folder, you will necessarily end up sending more spam to the inbox. But what is the best way to make the tradeoff? You can try different values of hyperparameters and get different rates for the two types of errors—different points on this tradeoff. A chart called the **receiver operating characteristic (ROC) curve** plots false positives versus true positives for each value of the hyperparameter, helping you visualize values that would be good choices for the tradeoff. A metric called the “area under the ROC curve” or **AUC** provides a single-number summary of the ROC curve, which is useful if you want to deploy a system and let each user choose their tradeoff point.

Another helpful visualization tool for classification problems is a **confusion matrix**: a two-dimensional table of counts of how often each category is classified or misclassified as each other category.

There can be tradeoffs in factors other than the loss function. If you can train a stock market prediction model that makes you $10 on every trade, that’s great—but not if it costs you $20 in computation cost for each prediction. A machine translation program that runs on your phone and allows you to read signs in a foreign city is helpful—but not if it runs down the battery after an hour of use. Keep track of all the factors that lead to acceptance or rejection of your system, and design a process where you can quickly iterate the process of getting a new idea, running an experiment, and evaluating the results of the experiment to see if you have made progress. Making this iteration process fast is one of the most important factors for success in machine learning.

19.9.4 Trust, interpretability, and explainability

We have described a machine learning methodology where you develop your model with training data, choose hyperparameters with validation data, and get a final metric with test data. Doing well on that metric is a necessary but not sufficient condition for you to **trust** your model. And it is not just you—other stakeholders including regulators, lawmakers, the press, and your users are also interested in the trustworthiness of your system (as well as in related attributes such as reliability, accountability, and safety).

A machine learning system is still a piece of software, and you can build trust with all the typical tools for verifying and validating any software system:

- **Source control** : Systems for version control, build, and bug/issue tracking.
- **Testing** : Unit tests for all the components covering simple canonical cases as well as tricky adversarial cases, fuzz tests (where random inputs are generated), regression tests, load tests, and system integration tests: these are all important for any software system. For machine learning, we also have tests on the training, validation, and test data sets.
- **Review** : Code walk-throughs and reviews, privacy reviews, fairness reviews (see Section 28.3.3), and other legal compliance reviews.
- **Monitoring** : Dashboards and alerts to make sure that the system is up and running and is continuing to performing at a high level of accuracy.

- **Accountability** : What happens when the system is wrong? What is the process for complaining about or appealing the system's decision? How can we track who was responsible for the error? Society expects (but doesn't always get) accountability for important decisions made by banks, politicians, and the law, and they should expect accountability from software systems including machine learning systems.

In addition, there are some factors that are especially important for machine learning systems, as we shall detail below.

Interpretability: We say that a machine learning model is **interpretable** if you can inspect the actual model and understand why it got a particular answer for a given input, and how the answer would change when the input changes.[18] Decision tree models are considered to be highly interpretable; we can understand that following the path *Patrons* = *Full* and *WaitEstimate* = 0–10 in a decision tree leads to a decision to *wait*. A decision tree is interpretable for two reasons. First, we humans have experience in understanding IF/THEN rules. (In contrast, it is very difficult for humans to get an intuitive understanding of the result of a matrix multiply followed by an activation function, as is done in some neural network models.) Second, the decision tree was in a sense constructed to be interpretable—the root of the tree was chosen to be the attribute with the highest information gain.

Linear regression models are also considered to be interpretable; we can examine a model for predicting the rent on an apartment and see that for each bedroom added, the rent increases by $500, according to the model. This idea of "If I change x, how will the output change?" is at the core of interpretability. Of course, correlation is not causation, so interpretable models are answering *what* is the case, but not necessarily *why* it is the case.

Explainability: An explainable model is one that can help you understand "*why* was this output produced for this input?" In our terminology, interpretability derives from inspecting the actual model, whereas explainability can be provided by a separate process. That is, the model itself can be a hard-to-understand black box, but an explanation module can summarize what the model does. For a neural network image-recognition system that classifies a picture as *dog*, if we tried to interpret the model directly, the best we could come away with would be something like "after processing the convolutional layers, the activation for the *dog* output in the softmax layer was higher than any other class." That's not a very compelling argument.

But a separate explanation module might be able to examine the neural network model and come up with the explanation "it has four legs, fur, a tail, floppy ears, and a long snout; it is smaller than a wolf, and it is lying on a dog bed, so I think it is a dog." Explanations are one way to build trust, and some regulations such as the European GDPR (General Data Protection Regulation) require systems to provide explanations.

As an example of a separate explanation module, the local interpretable model-agnostic explanations (LIME) system works like this: no matter what model class you use, LIME builds an interpretable model—often a decision tree or linear model—that is an approximation of your model, and then interprets the linear model to create explanations that say how important each feature is. LIME accomplishes this by treating the machine-learned model as a black box, and probing it with different random input values to create a data set from which the interpretable model can be built. This approach is appropriate for structured data, but not for things like images, where each pixel is a feature, and no one pixel is "important" by itself.

Sometimes we choose a model class because of its explainability—we might choose decision trees over neural networks not because they have higher accuracy but because the explainability gives us more trust in them.

However, a simple explanation can lead to a false sense of security. After all, we typically choose to use a machine learning model (rather than a hand-written traditional program) because the problem we are trying to solve is inherently complex, and we don't know how to write a traditional program. In that case, we shouldn't expect that there will necessarily be a simple explanation for every prediction.

If you are building a machine learning model primarily for the purpose of understanding the domain, then interpretability and explainability will help you arrive at that understanding. But if you just want the best-performing piece of software then testing may give you more confidence and trust than explanations. Which would you trust: an experimental aircraft that has never flown before but has a detailed explanation of why it is safe, or an aircraft that safely completed 100 previous flights and has been carefully maintained, but comes with no guaranted explanation?

19.9.5 Operation, monitoring, and maintenance

Once you are happy with your model's performance, you can deploy it to your users. You'll face additional challenges. First, there is the problem of the **long tail** of user inputs. You may have tested your system on a large test set, but if your system is popular, you will soon see inputs that were never tested before. You need to know whether your model generalizes well for them, which means you need to **monitor** your performance on live data—tracking statistics, displaying a dashboard, and sending alerts when key metrics fall below a threshold. In addition to automatically updating statistics on user interactions, you may need to hire and train human raters to look at your system and grade how well it is doing.

Second, there is the problem of **nonstationarity**—the world changes over time. Suppose your system classifies email as spam or non-spam. As soon as you successfully classify a batch of spam messages, the spammers will see what you have done and change their tactics, sending a new type of message you haven't seen before. Non-spam also evolves, as users change the mix of email versus messaging or desktop versus mobile services that they use.

You will continually face the question of what is better: a model that has been well tested but was built from older data, versus a model that is built from the latest data but has not been tested in actual use. Different systems have different requirements for freshness: some problems benefit from a new model every day, or even every hour, while other problems can keep the same model for months. If you are deploying a new model every hour, it will be impractical to run a heavy test suite and a manual review process for each update. You will need to automate the testing and release process so that small changes can be automatically approved, but larger changes trigger appropriate review. You can consider the tradeoff between an online model where new data incrementally modifies the existing model, versus an offline model where each new release requires building a new model from scratch.

It it is not just that the data will be changing—for example, new words will be used in spam email messages. It is also that the entire data schema may change—you might start out classifying spam email, and need to adapt to classify spam text messages, spam voice messages, spam videos, etc.

Figure 19.28 gives a general rubric to guide the practitioner in choosing the appropriate level of testing and monitoring.

Tests for Features and Data

(1) Feature expectations are captured in a schema. (2) All features are beneficial. (3) No feature's cost is too much. (4) Features adhere to meta-level requirements. (5) The data pipeline has appropriate privacy controls. (6) New features can be added quickly. (7) All input feature code is tested.

Tests for Model Development

(1) Every model specification undergoes a code review. (2) Every model is checked in to a repository. (3) Offline proxy metrics correlate with actual metrics (4) All hyperparameters have been tuned. (5) The impact of model staleness is known. (6) A simpler model is not better. (7) Model quality is sufficient on all important data slices. The model has been tested for considerations of inclusion.

Tests for Machine Learning Infrastructure

(1) Training is reproducible. (2) Model specification code is unit tested. (3) The full ML pipeline is integration tested. (4) Model quality is validated before attempting to serve it. (5) The model allows debugging by observing the step-by-step computation of training or inference on a single example. (6) Models are tested via a canary process before they enter production serving environments. (7) Models can be quickly and safely rolled back to a previous serving version.

Monitoring Tests for Machine Learning

(1) Dependency changes result in notification. (2) Data invariants hold in training and serving inputs. (3) Training and serving features compute the same values. (4) Models are not too stale. (5) The model is numerically stable. (6) The model has not experienced regressions in training speed, serving latency, throughput, or RAM usage. (7) The model has not experienced a regression in prediction quality on served data.

Figure 19.28 A set of criteria to see how well you are doing at deploying your machine learning model with sufficient tests. Abridged from Breck *et al*. (2016), who also provide a scoring metric.

Summary

This chapter introduced machine learning, and focused on supervised learning from examples. The main points were:

- Learning takes many forms, depending on the nature of the agent, the component to be improved, and the available feedback.
- If the available feedback provides the correct answer for example inputs, then the learning problem is called **supervised learning**. The task is to learn a function $y = h(x)$. Learning a function whose output is a continuous or ordered value (like *weight*) is called **regression**; learning a function with a small number of possible output categories is called **classification**;
- We want to learn a function that not only agrees with the data but also is likely to agree with future data. We need to balance agreement with the data against simplicity of the hypothesis.
- **Decision trees** can represent all Boolean functions. The **information-gain** heuristic provides an efficient method for finding a simple, consistent decision tree.
- The performance of a learning algorithm can be visualized by a **learning curve**, which shows the prediction accuracy on the **test set** as a function of the **training set** size.
- When there are multiple models to choose from, **model selection** can pick good values of hyperparameters, as confirmed by **cross-validation** on validation data. Once the hyperparameter values are chosen, we build our best model using all the training data.
- Sometimes not all errors are equal. A **loss function** tells us how bad each error is; the goal is then to minimize loss over a validation set.

- **Computational learning theory** analyzes the sample complexity and computational complexity of inductive learning. There is a tradeoff between the expressiveness of the hypothesis space and the ease of learning.
- **Linear regression** is a widely used model. The optimal parameters of a linear regression model can be calculated exactly, or can be found by gradient descent search, which is a technique that can be applied to models that do not have a closed-form solution.
- A linear classifier with a hard threshold—also known as a **perceptron** —can be trained by a simple weight update rule to fit data that are **linearly separable**. In other cases, the rule fails to converge.
- **Logistic regression** replaces the perceptron's hard threshold with a soft threshold defined by a logistic function. Gradient descent works well even for noisy data that are not linearly separable.
- **Nonparametric models** use all the data to make each prediction, rather than trying to summarize the data with a few parameters. Examples include **nearest neighbors** and **locally weighted regression**.
- **Support vector machines** find linear separators with **maximum margin** to improve the generalization performance of the classifier. **Kernel methods** implicitly transform the input data into a high-dimensional space where a linear separator may exist, even if the original data are nonseparable.
- Ensemble methods such as **bagging** and **boosting** often perform better than individual methods. In **online learning** we can aggregate the opinions of experts to come arbitrarily close to the best expert's performance, even when the distribution of the data are constantly shifting.

- Building a good machine learning model requires experience in the complete development process, from managing data to model selection and optimization, to continued maintenance.

Bibliographical and Historical Notes

Chapter 1 covered the history of philosophical investigations into the topic of inductive learning. William of Ockham (1280–1349), the most influential philosopher of his century and a major contributor to medieval epistemology, logic, and metaphysics, is credited with a statement called "Ockham's Razor"—in Latin, *Entia non sunt multiplicanda praeter necessitatem*, and in English, "Entities are not to be multiplied beyond necessity." Unfortunately, this laudable piece of advice is nowhere to be found in his writings in precisely these words (although he did say "Pluralitas non est ponenda sine necessitate," or "Plurality shouldn't be posited without necessity"). A similar sentiment was expressed by Aristotle in 350 BCE in *Physics* book I, chapter VI: "For the more limited, if adequate, is always preferable."

David Hume (1711–1776) formulated the *problem of induction*, recognizing that generalizing from examples admits the possibility of errors, in a way that logical deduction does not. He saw that there was no way to have a guaranteed correct solution to the problem, but proposed the principle of *uniformity of nature*, which we have called *stationarity*. What Ockham and Hume were getting at is that when we do induction, we are choosing from the multitude of consistent models one that is more likely—because it is simpler and matches our expectations. In modern day, the *no free lunch* theorem (Wolpert and Macready, 1997; Wolpert, 2013) says that if a learning algorithm performs well on a certain set of problems, it is only because it will perform poorly on a different set: if our decision tree correctly predicts SR's restaurant waiting behavior, it must perform poorly for some other hypothetical person who has the opposite waiting behavior on the unobserved inputs.

Machine learning was one of the key ideas at the birth of computer science. Alan Turing (1947) anticipated it, saying “Let us suppose we have set up a machine with certain initial instruction tables, so constructed that these tables might on occasion, if good reason arose, modify those tables.” Arthur Samuel (1959) defined machine learning as the “field of study that gives computers the ability to learn without being explicitly programmed” while creating his learning checkers program.

The first notable use of **decision trees** was in EPAM, the “Elementary Perceiver And Memorizer” (Feigenbaum, 1961), which was a simulation of human concept learning. ID3 (Quinlan, 1979) added the crucial idea of choosing the attribute with maximum entropy. The concepts of entropy and information theory were developed by Claude Shannon to aid in the study of communication (Shannon and Weaver, 1949). (Shannon also contributed one of the earliest examples of machine learning, a mechanical mouse named Theseus that learned to navigate through a maze by trial and error.) The χ^2 method of tree pruning was described by Quinlan (1986). A description of C4.5, an industrial-strength decision tree package, can be found in Quinlan (1993). An alternative industrial-strength software package, CART (for Classification and Regression Trees) was developed by the statistician Leo Breiman and his colleagues (Breiman *et al.*, 1984).

Hyafil and Rivest (1976) proved that finding an *optimal* decision tree (rather than finding a good tree through locally greedy selections) is NP-complete. But Bertsimas and Dunn (2017) point out that in the last 25 years, advances in hardware design and in algorithms for mixed-integer programming have resulted in an 800 billion-fold speedup, which means that it is now feasible to solve this NP-hard problem at least for problems with not more than a few thousand examples and a few dozen features.

Cross-validation was first introduced by Larson (1931), and in a form close to what we show by Stone (1974) and Golub *et al*. (1979). The regularization procedure is due to Tikhonov (1963).

On the question of overfitting, John von Neumann was quoted (Dyson, 2004) as boasting, “With four parameters I can fit an elephant, and with five I can make him wiggle his trunk,” meaning that a high-degree polynomial can be made to fit almost any data, but at the cost of potentially overfitting. Mayer *et al*. (2010) proved him right by demonstrating a four-parameter elephant and five-parameter wiggle, and Boué (2019) went even further, demonstrating an elephant and other animals with a one-parameter chaotic function.

Zhang *et al*. (2016) analyze under what conditions a model can memorize the training data. They perform experiments using random data—surely an algorithm that gets zero error on a training set with random labels must be memorizing the data set. However, they conclude that the field has yet to discover a precise measure of what it means for a model to be “simple” in the sense of Ockham’s razor. Arpit *et al*. (2017) show that the conditions under which memorization can occur depend on details of both the model and the data set.

Belkin *et al*. (2019) discuss the bias–variance tradeoff in machine learning and why some model classes continue to improve after reaching the interpolation point, while other model classes exhibit the U-shaped curve. Berrada *et al*. (2019) develop a new learning algorithm based on gradient descent that exploits the ability of models to memorize to set good values for the learning rate hyperparameter.

Theoretical analysis of learning algorithms began with the work of Gold (1967) on **identification in the limit**. This approach was motivated in part by models of scientific discovery from the philosophy of science (Popper, 1962), but has been applied mainly to the problem of learning grammars from example sentences (Osherson *et al.*, 1986).

Whereas the identification-in-the-limit approach concentrates on eventual convergence, the study of **Kolmogorov complexity** or **algorithmic complexity**, developed independently by Solomonoff (1964, 2009) and Kolmogorov (1965), attempts to provide a formal definition for the notion of simplicity used in Ockham's razor. To escape the problem that simplicity depends on the way in which information is represented, it is proposed that simplicity be measured by the length of the shortest program for a universal Turing machine that correctly reproduces the observed data. Although there are many possible universal Turing machines, and hence many possible "shortest" programs, these programs differ in length by at most a constant that is independent of the amount of data. This beautiful insight, which essentially shows that *any* initial representation bias will eventually be overcome by the data, is marred only by the undecidability of computing the length of the shortest program. Approximate measures such as the **minimum description length**, or MDL (Rissanen, 1984, 2007) can be used instead and have produced excellent results in practice. The text by Li and Vitanyi (2008) is the best source for Kolmogorov complexity.

The theory of **PAC learning** was inaugurated by Leslie Valiant (1984), stressing the importance of computational and sample complexity. With Michael Kearns (1990), Valiant showed that several concept classes cannot be PAC-learned tractably, even though sufficient information is available in the examples. Some positive results were obtained for classes such as decision lists (Rivest, 1987).

An independent tradition of sample-complexity analysis has existed in statistics, beginning with the work on **uniform convergence theory** (Vapnik and Chervonenkis, 1971). The so-called **VC dimension** provides a measure roughly analogous to, but more general than, the $\ln |\mathcal{H}|$ measure obtained from PAC analysis. The VC dimension can be applied to continuous function classes, to which standard PAC analysis does not apply. PAC-learning theory and VC theory were first connected by the “four Germans” (none of whom actually is German): Blumer, Ehrenfeucht, Haussler, and Warmuth (1989).

Linear regression with squared error loss goes back to Legendre (1805) and Gauss (1809), who were both working on predicting orbits around the sun. (Gauss claimed to be using the technique since 1795, but delayed in publishing it.) The modern use of multivariable regression for machine learning is covered in texts such as Bishop (2007). The differences between L_1 and L_2 regularization are analyzed by Ng (2004) and Moore and DeNero (2011).

The term **logistic function** comes from Pierre-Francois Verhulst (1804–1849), a statistician who used the curve to model population growth with limited resources, a more realistic model than the unconstrained geometric growth proposed by Thomas Malthus. Verhulst called it the *courbe logistique*, because of its relation to the logarithmic curve. The term **curse of dimensionality** comes from Richard Bellman (1961).

Logistic regression can be solved with gradient descent or with the Newton–Raphson method (Newton, 1671; Raphson, 1690). A variant of the Newton method called L-BFGS is often used for large-dimensional problems; the L stands for “limited memory,” meaning that it avoids creating the full matrices all at once, and instead creates parts of them on the fly. BFGS are the authors’ initials (Byrd *et al.*, 1995). The idea of gradient descent goes back to Cauchy (1847); stochastic gradient descent (SGD) was introduced in the statistical optimization community by Robbins and Monro (1951), rediscovered for neural networks by Rosenblatt (1960), and popularized for large-scale machine learning by Bottou and Bousquet (2008). Bottou *et al*. (2018) reconsider the topic of large-scale learning with a decade of additional experience.

Nearest-neighbors models date back at least to Fix and Hodges (1951) and have been a standard tool in statistics and pattern recognition ever since. Within AI, they were popularized by Stanfill and Waltz (1986), who investigated methods for adapting the distance metric to the data. Hastie and Tibshirani (1996) developed a way to localize the metric to each point in the space, depending on the distribution of data around that point. Gionis *et al*. (1999) introduced locality-sensitive hashing (LSH), which revolutionized the retrieval of similar objects in highdimensional spaces. Andoni and Indyk (2006) provide a survey of LSH and related methods, and Samet (2006) covers properties of high-dimensional spaces. The technique is particularly useful for genomic data, where each record has millions of attributes (Berlin *et al.,* 2015).

The ideas behind **kernel machines** come from Aizerman *et al*. (1964) (who also introduced the kernel trick), but the full development of the theory is due to Vapnik and his colleagues (Boser *et al.,* 1992). SVMs were made practical with the introduction of the soft-margin classifier for handling noisy data in a paper that won the 2008 ACM Theory and Practice Award (Cortes and Vapnik, 1995), and of the Sequential Minimal Optimization (SMO) algorithm for efficiently solving SVM problems using quadratic programming (Platt, 1999). SVMs have proven to be very effective for tasks such as text categorization (Joachims, 2001), computational genomics (Cristianini and Hahn, 2007), and handwritten digit recognition of DeCoste and Schölkopf (2002).

As part of this process, many new kernels have been designed that work with strings, trees, and other nonnumerical data types. A related technique that also uses the kernel trick to implicitly represent an exponential feature space is the voted perceptron (Freund and Schapire, 1999; Collins and Duffy, 2002). Textbooks on SVMs include Cristianini and Shawe-Taylor (2000) and Scholkopf and Smola (2002). A friendlier exposition appears in the *AI Magazine* article by Cristianini and Scholkopf (2002). Bengio and LeCun (2007) show some of the limitations of SVMs and other local, nonparametric methods for learning functions that have a global structure but do not have local smoothness.

The first mathematical proof of the value of an ensemble was Condorcet's jury theorem (1785), which proved that if jurors are independent and an individual juror has at least a 50% chance of deciding a case correctly, then the more jurors you add, the better the chance of deciding the case correctly. More recently, **ensemble learning** has become an increasingly popular technique for improving the performance of learning algorithms.

The first **random forest** algorithm, using random attribution selection, is by Ho (1995); an independent version was introduced by Amit and Geman (1997). Breiman (2001) added the ideas of **bagging** and "out-of-bag error." Friedman (2001) introduced the terminology Gradient Boosting Machine (GBM), expanding the approach to allow for multiclass classification, regression, and ranking problems.

Michel Kearns (1988) defined the Hypothesis Boosting Problem: given a learner that predicts only slightly better than random guessing, is it possible to derive a learner that performs arbitrarily well? The problem was answered in the affirmative in a theoretical paper by Schapire (1990) that led to the ADABOOST algorithm Freund and Schapire (1996) and to further theoretical work Schapire (2003). Friedman *et al*. (2000) explain boosting from a statistician's viewpoint. Chen and Guestrin (2016) describe the XGBOOST system, which has been used with great success in many large-scale applications.

Online learning is covered in a survey by Blum (1996) and a book by Cesa-Bianchi and Lugosi (2006). Dredze *et al*. (2008) introduce the idea of confidence-weighted online learning for classification: in addition to keeping a weight for each parameter, they also maintain a measure of confidence, so that a new example can have a large effect on features that were rarely seen before (and thus had low confidence) and a small effect on common features that have already been well estimated. Yu *et al*. (2011) describe how a team of students work together to build an ensemble classifier in the KDD competition. One exciting possibility is to create an "outrageously large" mixture-of-experts ensemble that uses a sparse subset of experts for each incoming example (Shazeer *et al.*, 2017). Seni and Elder (2010) survey ensemble methods.

In terms of practical advice for building machine learning systems, Pedro Domingos describes a few things to know (2012). Andrew Ng gives hints for developing and debugging a product using machine learning (Ng, 2019). O'Neil and Schutt (2013) describe the process of doing data science. Tukey (1977) introduced **exploratory data analysis**, and Gelman (2004) gives an updated view of the process. Bien *et al*. (2011) describe the process of choosing prototypes for interpretability, and Kim *et al*. (2017) show how to find critics that are maximally distant from the prototypes using a metric called maximum mean discrepancy. Wattenberg *et al*. (2016) describe how to use t-SNE. To get a comprehensive view of how well your deployed machine learning system is doing, Breck *et al*. (2016) offer a checklist of 28 tests that you can apply to get an overall ML test score. Riley (2019) describes three common pitfalls of ML development.

Banko and Brill (2001), Halevy *et al*. (2009), and Gandomi and Haider (2015) discuss the advantages of using the large amounts of data that are now available. Lyman and Varian (2003) estimated that about 5 exabytes (5×10^{18} bytes) of data was produced in 2002, and that the rate of production is doubling every 3 years; Hilbert and Lopez (2011) estimated 2×10^{21} bytes for 2007, indicating an acceleration. Guyon and Elisseeff (2003) discuss the problem of feature selection with large data sets.

Doshi-Velez and Kim (2017) propose a framework for **interpretable machine learning** or **explainable AI (XAI)**. Miller *et al*. (2017) point out that there are two kinds of explanations, one for the designers of an AI system and one for the users, and we need to be clear what we are aiming for. The LIME system (Ribeiro *et al.*, 2016) builds interpretable linear models that approximate whatever machine learning system you have. A similar system, SHAP (Lundberg and Lee, 2018) (Shapley Additive explanations), uses the notion of a Shapley value (page 618) to determine the contribution of each feature.

The idea that we could apply machine learning to the task of solving machine learning problems is a tantalizing one. Thrun and Pratt (2012) give an early overview of the field in an edited collection titled *Learning to Learn*. Recently the field has adopted the name **automated machine learning (AutoML)**; Hutter *et al*. (2019) give an overview.

Kanter and Veeramachaneni (2015) describe a system for doing automated feature selection. Bergstra and Bengio (2012) describe a system for searching the space of hyperparameters, as do Thornton *et al*. (2013) and Bermúdez-Chacón *et al*. (2015). Wong *et al*. (2019) show how transfer learning can speed up AutoML for deep learning models. Competitions have been organized to see which systems are best at AutoML tasks (Guyon *et al.,* 2015). (Steinruecken *et al.,* 2019) describe a system called the Automatic Statistician: you give it some data and it writes a report, mixing text, charts, and calculations. The major cloud computing providers have included AutoML as part of their offerings. Some researchers prefer the term **metalearning**: for example, the MAML (Model-Agnostic Meta-Learning) system (Finn *et al.,* 2017) works with any model that can be trained by gradient descent; it trains a core model so that it will be easy to fine-tune the model with new data on new tasks.

Despite all this work, we still don't have a complete system for automatically solving machine learning problems. To do that with supervised machine learning we would need to start with a data set of ($\mathbf{x}_j$, y_j) examples. Here the input $\mathbf{x}_j$ is a specification of the problem, in the form that a problem is initially encountered: a vague description of the goals, and some data to work with, perhaps with a vague plan for how to acquire more data. The output y_i would be a complete running machine learning program, along with a methodology for maintaining the program: gathering more data, cleaning it, testing and monitoring the system, etc. One would expect we would need a data set of thousands of such examples. But no such data set exists, so existing AutoML systems are limited in what they can accomplish.

There is a dizzying array of books that introduce data science and machine learning in conjunction with software packages such as Python (Segaran, 2007; Raschka, 2015; Nielsen, 2015), Scikit-Learn (Pedregosa *et al.,* 2011), R (Conway and White, 2012), Pandas (McKinney, 2012), NumPy (Marsland, 2014), PyTorch (Howard and Gugger, 2020), TensorFlow (Ramsundar and Zadeh, 2018), and Keras (Chollet, 2017; Géron, 2019).

There are a number of valuable textbooks in machine learning (Bishop, 2007; Murphy, 2012) and in the closely allied and overlapping fields of pattern recognition (Ripley, 1996; Duda *et al.*, 2001), statistics (Wasserman, 2004; Hastie *et al.*, 2009; James *et al.*, 2013), data science (Blum *et al.*, 2020), data mining (Han *et al.*, 2011; Witten and Frank, 2016; Tan *et al.*, 2019), computational learning theory (Kearns and Vazirani, 1994; Vapnik, 1998), and information theory (Shannon and Weaver, 1949; MacKay, 2002; Cover and Thomas, 2006). Burkov (2019) attempts the shortest possible introduction to machine learning, and Domingos (2015) offers a nontechnical overview of the field. Current research in machine learning is published in the annual proceedings of the International Conference on Machine Learning (ICML), the International Conference on Learning Representations (ICLR), and the conference on Neural Information Processing Systems (NeurIPS); and in *Machine Learning* and the *Journal of Machine Learning Research*.

[1] A better name would have been *function approximation* or *numeric prediction*. But in 1886 Francis Galton wrote an influential article on the concept of *regression to the mean* (e.g., the children of tall parents are likely to be taller than average, but not as tall as the parents). Galton showed plots with what he called “regression lines,” and readers came to associate the word “regression” with the statistical technique of function approximation rather than with the topic of regression to the mean.

[2] The name is often misspelled as “Occam.”

[3] The gain will be strictly positive except for the unlikely case where all the proportions are *exactly* the same. (See Exercise 19.NNGA.)

[4] Although the name "model selection" is in common use, a better name would have been "model *class* selection" or "hypothesis space selection." The word "model" has been used in the literature to refer to three different levels of specificity: a broad hypothesis space (like "polynomials"), a hypothesis space with hyperparameters filled in (like "degree-2 polynomials"), and a specific hypothesis with all parameters filled in (like $5x^2+3x-2$).

[5] Some authors say the model has "memorized" the data.

[6] Gauss showed that if the y_j values have normally distributed noise, then the most likely values of w_1 and w_0 are obtained by using L_2 loss, minimizing the sum of the squares of the errors. (If the values have noise that follows a Laplace (double exponential) distribution, then L_1 loss is appropriate.)

[7] With some caveats: the L_2 loss function is appropriate when there is normally distributed noise that is independent of x; all results rely on the stationarity assumption; etc.

[8] The reader may wish to consult Appendix A for a brief summary of linear algebra. Also, note that we use the term “multivariable regression” to mean that the input is a vector of multiple values, but the output is a single variable. We will use the term “multivariate regression” for the case where the output is also a vector of multiple variables. However, other authors use the two terms interchangeably.

[9] It is perhaps confusing that the notation L_1 and L_2 is used for both loss functions and regularization functions. They need not be used in pairs: you could use L_2 loss with L_1 regularization, or vice versa.

[10] Technically, we require that $\Delta_{t=1}^{\infty}\alpha(t)=\infty$ and $\sum_{t=1}^{\infty}\alpha^2(t)<\infty$. The learning rate $\propto(t) = O(1/t)$ satisfies these conditions. Often we use $c/(c + t)$ for some fairly large constant c.

[11] The function sign(x) returns +1 for a positive x, – 1 for a negative x.

[12] The reader may notice that we could have used just f_1 and f_2, but the 3D mapping illustrates the idea better.

[13] This usage of “kernel function” is slightly different from the kernels in locally weighted regression. Some SVM kernels are distance metrics, but not all are.

[14] Here, "reasonable" means that the matrix $\mathbf{K}_{jk} = K(\mathbf{x}_j, \mathbf{x}_k)$ is positive definite.

[15] Note on terminology: In statistics, a sample with replacement is called a **bootstrap**, and “bagging” is short for “bootstrap aggregating.”

[16] Blum (1996) gives an elegant proof.

[17] Geoffrey Hinton provides the helpful advice “To deal with a 14-dimensional space, visualize a 3D space and say ‘fourteen’ to yourself very loudly.”

[18] This terminology is not universally accepted; some authors use “interpretable” and “explainable” as synonyms, both referring to reaching some kind of understanding of a model.

CHAPTER 20

KNOWLEDGE IN LEARNING

In which we examine the problem of learning when you know something already.

In all of the approaches to learning described in the previous chapter, the idea is to construct a function that has the input–output behavior observed in the data. In each case, the learning methods can be understood as searching a hypothesis space to find a suitable function, starting from only a very basic assumption about the form of the function, such as “second-degree polynomial” or “decision tree” and perhaps a preference for simpler hypotheses. Doing this amounts to saying that before you can learn something new, you must first forget (almost) everything you know. In this chapter, we study learning methods that can take advantage of **prior knowledge** about the world. In most cases, the prior knowledge is represented as general first-order logical theories; thus for the first time we bring together the work on knowledge representation and learning.

20.1 A Logical Formulation of Learning

Chapter 19 defined pure inductive learning as a process of finding a hypothesis that agrees with the observed examples. Here, we specialize this definition to the case where the hypothesis is represented by a set of logical sentences. Example descriptions and classifications will also be logical sentences, and a new example can be classified by inferring a classification sentence from the hypothesis and the example description. This approach allows for incremental construction of hypotheses, one sentence at a time. It also allows for prior knowledge, because sentences that are already known can assist in the classification of new examples. The logical formulation of learning may seem like a lot of extra work at first, but it turns out to clarify many of the issues in learning. It enables us to go well beyond the simple learning methods of Chapter 19 by using the full power of logical inference in the service of learning.

20.1.1 Examples and hypotheses

Recall from Chapter 19 the restaurant learning problem: learning a rule for deciding whether to wait for a table. Examples were described by **attributes** such as *Alternate*, *Bar*, *Fri/Sat*, and so on. In a logical setting, an example is described by a logical sentence; the attributes become unary predicates. Let us generically call the ith example X_i. For instance, the first example from Figure 19.3 (page 676) is described by the sentences

$$Alternate(X_1) \land \neg Bar(X_1) \land \neg Fri/Sat(X_1) \land Hungry(X_1) \land \ldots$$

We will use the notation $D_i(X_i)$ to refer to the description of X_i, where D_i can be any logical expression taking a single argument. The classification of the example is given by a literal using the goal predicate, in this case

$$WillWait(X_1) \qquad \text{or} \qquad \neg WillWait(X_1).$$

The complete training set can thus be expressed as the conjunction of all the example descriptions and goal literals.

The aim of inductive learning in general is to find a hypothesis that classifies the examples well and generalizes well to new examples. Here we are concerned with hypotheses expressed in logic; each hypothesis h_j will have the form

$$\forall x \;\; Goal(x) \;\Leftrightarrow\; C_j(x),$$

where $C_j(x)$ is a candidate definition—some expression involving the attribute predicates. For example, a decision tree can be interpreted as a logical expression of this form. Thus, the tree in Figure 19.6 (page 678) expresses the following logical definition (which we will call h_r for future reference):

$$\begin{aligned}\forall r\, WillWait(r) \Leftrightarrow\; & Patrons(r, Some)\\ \lor\; & Patrons(r, Full) \land Hungry(r) \land Type(r, French)\\ \lor\; & Patrons(r, Full) \land Hungry(r) \land Type(r, Thai)\\ & \qquad \land Fri/Sat(r)\\ \lor\; & Patrons(r, Full) \land Hungry(r) \land Type(r, Burger).\end{aligned} \tag{20.1}$$

Each hypothesis predicts that a certain set of examples—namely, those that satisfy its candidate definition—will be examples of the goal predicate. This set is called the **extension** of the predicate. Two hypotheses with different extensions are therefore logically inconsistent with each other, because they disagree on their predictions for at least one example. If they have the same extension, they are logically equivalent.

The hypothesis space $\mathcal{H}$ is the set of all hypotheses $\{h_1,...,h_n\}$ that the learning algorithm is designed to entertain. For example, the DECISION-TREE-LEARNING algorithm can entertain any decision tree hypothesis defined in terms of the attributes provided; its hypothesis space therefore consists of all these decision trees. Presumably, the learning algorithm believes that one of the hypotheses is correct; that is, it believes the sentence

$$h_1 \vee h_2 \vee h_3 \vee \cdots \vee h_n. \qquad (20.2)$$

As the examples arrive, hypotheses that are not **consistent** with the examples can be ruled out. Let us examine this notion of consistency more carefully. Obviously, if hypothesis h_j is consistent with the entire training set, it has to be consistent with each example in the training set. What would it mean for it to be inconsistent with an example? There are two possible ways that this can happen:

- An example can be a **false negative** for the hypothesis, if the hypothesis says it should be negative but in fact it is positive. For instance, the new example X_{13} described by $(X_{13}, Full) \wedge \neg Hungry(X_{13}) \wedge \ldots \wedge WillWait(X_{13})$ would be a false negative for the hypothesis h_r given earlier. From h_r and the example description, we can deduce both *WillWait*(X_{13}), which is what the example says, and ¬*WillWait*(X_{13}), which is what the hypothesis predicts. The hypothesis and the example are therefore logically inconsistent.
- An example can be a **false positive** for the hypothesis, if the hypothesis says it should be positive but in fact it is negative.[1]

If an example is a false positive or false negative for a hypothesis, then the example and the hypothesis are logically inconsistent with each other. Assuming that the example is a correct observation of fact, then the hypothesis can be ruled out. Logically, this is exactly analogous to the resolution rule of inference (see Chapter 9), where the disjunction of hypotheses corresponds to a clause and the example corresponds to a literal that resolves against one of the literals in the clause. An ordinary logical inference system therefore could, in principle, learn from the example by eliminating one or more hypotheses. Suppose, for example, that the example is denoted by the sentence I_1, and the hypothesis space is $h_1 \vee h_2 \vee h_3 \vee h_4$. Then if I_1 is inconsistent with h_2 and h_3, the logical inference system can deduce the new hypothesis space $h_1 \vee h_4$.

We therefore can characterize inductive learning in a logical setting as a process of gradually eliminating hypotheses that are inconsistent with the examples, narrowing down the possibilities. Because the hypothesis space is usually vast (or even infinite in the case of first-order logic), we do not recommend trying to build a learning system using resolution-based theorem proving and a complete enumeration of the hypothesis space. Instead, we will describe two approaches that find logically consistent hypotheses with much less effort.

20.1.2 Current-best-hypothesis search

The idea behind **current-best-hypothesis** search is to maintain a single hypothesis, and to adjust it as new examples arrive in order to maintain consistency. The basic algorithm was described by John Stuart Mill (1843), and may well have appeared even earlier.

Suppose we have some hypothesis such as h_r, of which we have grown quite fond. As long as each new example is consistent, we need do nothing. Then along comes a false negative example, X_{13}. What do we do? Figure 20.1(a) shows h_r schematically as a region: everything inside the rectangle is part of the extension of h_r. The examples that have actually been seen so far are shown as “+” or “–”, and we see that h_r correctly categorizes all the examples as positive or negative examples of *WillWait*. In Figure 20.1(b), a new example (circled) is a false negative: the hypothesis says it should be negative but it is actually positive. The extension of the hypothesis must be increased to include it. This is called **generalization**; one possible generalization is shown in Figure 20.1(c). Then in Figure 20.1(d), we see a false positive: the hypothesis says the new example (circled) should be positive, but it actually is negative. The extension of the hypothesis must be decreased to exclude the example. This is called **specialization**; in Figure 20.1(e) we see one possible specialization of the hypothesis. The "more general than" and "more specific than" relations between hypotheses provide the logical structure on the hypothesis space that makes efficient search possible.

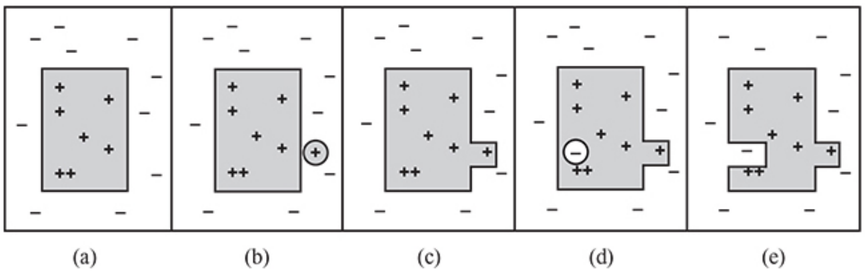


Figure 20.1 (a) A consistent hypothesis. (b) A false negative. (c) The hypothesis is generalized. (d) A false positive. (e) The hypothesis is specialized.

We can now specify the CURRENT-BEST-LEARNING algorithm, shown in Figure 20.2. Notice that each time we consider generalizing or specializing the hypothesis, we must check for consistency with the other examples, because an arbitrary increase/decrease in the extension might include/exclude previously seen negative/positive examples.

```
function CURRENT-BEST-LEARNING(examples, h) returns a hypothesis or fail
  if examples is empty then
     return h
  e ← FIRST(examples)
  if e is consistent with h then
     return CURRENT-BEST-LEARNING(REST(examples), h)
  else if e is a false positive for h then
     for each h′ in  specializations of h consistent with examples seen so far do
        h″ ← CURRENT-BEST-LEARNING(REST(examples), h′)
        if h″ ≠ fail then return h″
  else if e is a false negative for h then
     for each h′ in generalizations of h consistent with examples seen so far do
        h″ ← CURRENT-BEST-LEARNING(REST(examples), h′)
        if h″ ≠ fail then return h″
  return fail
```

Figure 20.2 The current-best-hypothesis learning algorithm. It searches for a consistent hypothesis that fits all the examples and backtracks when no consistent specialization/generalization can be found. To start the algorithm, any hypothesis can be passed in; it will be specialized or gneralized as needed.

We have defined generalization and specialization as operations that change the *extension* of a hypothesis. Now we need to determine exactly how they can be implemented as syntactic operations that change the candidate definition associated with the hypothesis, so that a program can carry them out. This is done by first noting that generalization and specialization are also *logical* relationships between hypotheses. If hypothesis h_1, with definition C_1, is a generalization of hypothesis h_2 with definition C_2, then we must have

$$\forall x \; C_2(x) \;\Rightarrow\; C_1(x).$$

Therefore in order to construct a generalization of h_2, we simply need to find a definition C_1 that is logically implied by C_2. This is easily done. For example, if $C_2(x)$ is $Alternate(x) \wedge Patrons(x, Some)$, then one possible generalization is given by $C_1(x) \equiv Patrons(x, Some)$. This is called **dropping conditions**. Intuitively, it generates a weaker definition and therefore allows a larger set of positive examples. There are a number of other generalization operations, depending on the language being operated on. Similarly, we can specialize a hypothesis by adding extra conditions to its candidate definition or by removing disjuncts from a disjunctive definition. Let us see how this works on the restaurant example, using the data in Figure 19.3.

- The first example, X_1, is positive. The attribute $Alternate(X_1)$ is true, so let the initial hypothesis be

$$h_1: \quad \forall x \; WillWait(x) \;\Leftrightarrow\; Alternate(x)\,.$$

- The second example, X_2, is negative. h_1 predicts it to be positive, so it is a false positive. Therefore, we need to specialize h_1. This can be done by adding an extra condition that will rule out X_2, while continuing to classify X_1 as positive. One possibility is

$$h_2: \quad \forall x \; WillWait(x) \;\Leftrightarrow\; Alternate(x) \wedge Pa$$

- The third example, X_3, is positive. h_2 predicts it to be negative, so it is a false negative. Therefore, we need to generalize h_2. We drop the *Alternate* condition, yielding

$$h_3: \quad \forall x \; WillWait(x) \;\Leftrightarrow\; Patrons(x, Some).$$

- The fourth example, X_4, is positive. h_3 predicts it to be negative, so it is a false negative. We therefore need to generalize h_3. We cannot drop the *Patrons* condition, because that would yield an all-inclusive hypothesis that would be inconsistent with X_2. One possibility is to add a disjunct:

$$\begin{aligned} h_4: \;\; \forall x \; WillWait(x) \;\Leftrightarrow\; & Patrons(x, Some). \\ & \vee\, (Patrons\,(x, Full) \wedge Fri/Sat(x)). \end{aligned}$$

Already, the hypothesis is starting to look reasonable. Obviously, there are other possibilities consistent with the first four examples; here are two of them:

$$h\prime_4: \quad \forall x \; WillWait(x) \;\Leftrightarrow\; \neg WaitEstimate(x, 30-6C$$

$$h\prime\prime_4: \quad \forall x \; WillWait(x) \;\Leftrightarrow\; Patrons(x, Some)$$

$$\vee\, (Patrons\,(x, Full) \wedge WaitEstimate(x, 10-30)).$$

The CURRENT-BEST-LEARNING algorithm is described nondeterministically, because at any point, there may be several possible specializations or generalizations that can be applied. The choices that are made will not necessarily lead to the simplest hypothesis, and may lead to an unrecoverable situation where no simple modification of the hypothesis is consistent with all of the data. In such cases, the program must backtrack to a previous choice point.

The CURRENT-BEST-LEARNING algorithm and its variants have been used in many machine learning systems, starting with Patrick Winston's (1970) "arch-learning" program. With a large number of examples and a large space, however, some difficulties arise:

1. Checking all the previous examples over again for each modification is very expensive.
2. The search process may involve a great deal of backtracking. As we saw in Chapter 19, hypothesis space can be a doubly exponentially large place.

20.1.3 Least-commitment search

Backtracking arises because the current-best-hypothesis approach has to *choose* a particular hypothesis as its best guess even though it does not have enough data yet to be sure of the choice. What we can do instead is to keep around all and only those hypotheses that are consistent with all the data so far. Each new example will either have no effect or will get rid of some of the hypotheses. Recall that the original hypothesis space can be viewed as a disjunctive sentence

$$h_1 \vee h_2 \vee h_3 \ldots \vee h_n.$$

As various hypotheses are found to be inconsistent with the examples, this disjunction shrinks, retaining only those hypotheses not ruled out. Assuming that the original hypothesis space does in fact contain the right answer, the reduced disjunction must still contain the right answer because only incorrect hypotheses have been removed. The set of hypotheses remaining is called the **version space**, and the learning algorithm (sketched in Figure 20.3) is called the version space learning algorithm (also the **candidate elimination** algorithm).

```
function VERSION-SPACE-LEARNING(examples) returns a version space
  local variables: V, the version space: the set of all hypotheses

  V ← the set of all hypotheses
  for each example e in examples do
     if V is not empty then V ← VERSION-SPACE-UPDATE(V, e)
  return V
```

```
function VERSION-SPACE-UPDATE(V, e) returns an updated version space
  V ← {h ∈ V : h is consistent with e}
```

Figure 20.3 The version space learning algorithm. It finds a subset of V that is consistent with all the *examples*.

One important property of this approach is that it is *incremental*: one never has to go back and reexamine the old examples. All remaining hypotheses are guaranteed to be consistent with them already. But there is an obvious problem. We already said that the hypothesis space is enormous, so how can we possibly write down this enormous disjunction?

The following simple analogy is very helpful. How do you represent all the real numbers between 1 and 2? After all, there are an infinite number of them! The answer is to use an interval representation that just specifies the boundaries of the set: [1,2]. It works because we have an *ordering* on the real numbers.

We also have an ordering on the hypothesis space, namely, generalization/specialization. This is a partial ordering, which means that each boundary will not be a point but rather a set of hypotheses called a **boundary set**. The great thing is that we can represent the entire version space using just two boundary sets: a most general boundary (the **G-set**) and a most specific boundary (the **S-set**). *Everything in between is guaranteed to be consistent with the examples*. Before we prove this, let us recap:

- The current version space is the set of hypotheses consistent with all the examples so far. It is represented by the S-set and G-set, each of which is a set of hypotheses.
- Every member of the S-set is consistent with all observations so far, and there are no consistent hypotheses that are more specific.
- Every member of the G-set is consistent with all observations so far, and there are no consistent hypotheses that are more general.

We want the initial version space (before any examples have been seen) to represent all possible hypotheses. We do this by setting the G-set to contain *True* (the hypothesis that contains everything), and the S-set to contain *False* (the hypothesis whose extension is empty).

Figure 20.4 shows the general structure of the boundary-set representation of the version space. To show that the representation is sufficient, we need the following two properties:

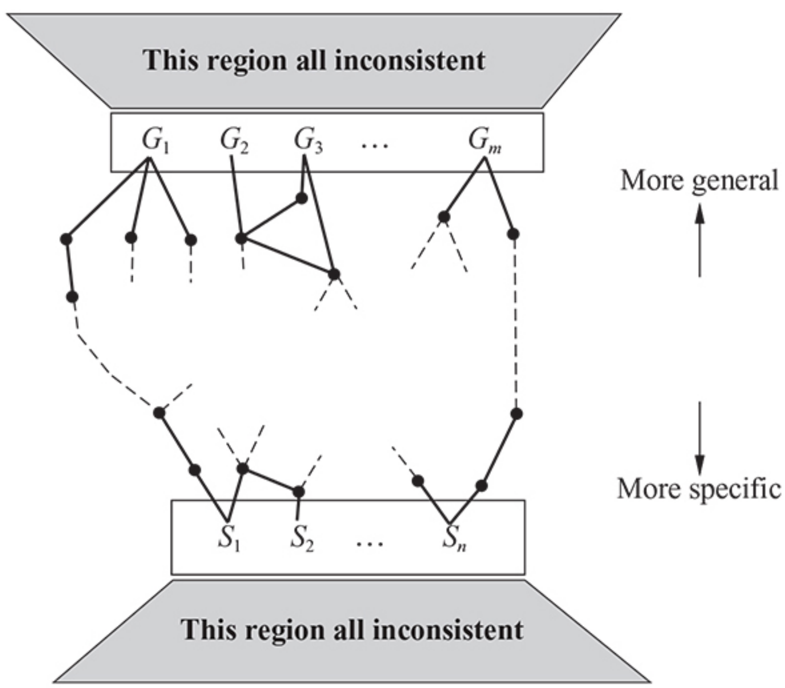


Figure 20.4 The version space contains all hypotheses consistent with the examples.

1. Every consistent hypothesis (other than those in the boundary sets) is more specific than some member of the G-set, and more general than some member of the S-set. (That is, there are no "stragglers" left outside.) This follows directly from the definitions of S and G. If there were a straggler h, then it would have to be no more specific than any member of G, in which case it belongs in G; or no more general than any member of S, in which case it belongs in S.
2. Every hypothesis more specific than some member of the G-set and more general than some member of the S-set is a consistent hypothesis. (That is, there are no "holes" between the boundaries.) Any h between S and G must reject all the negative examples rejected by each member of G (because it is more specific), and must accept all the positive examples accepted by any member of S (because it is more general). Thus, h must agree with all the examples, and therefore cannot be inconsistent. Figure 20.5 shows the situation: there are no known examples outside S but inside G, so any hypothesis in the gap must be consistent.

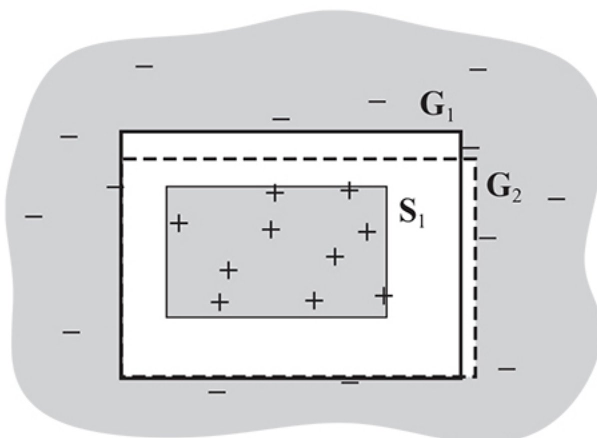


Figure 20.5 The extensions of the members of G and S. No known examples lie in between the two sets of boundaries.

We have therefore shown that *if* S and G are maintained according to their definitions, then they provide a satisfactory representation of the version space. The only remaining problem is how to *update* S and G for a new example (the job of the VERSION-SPACE-UPDATE function). This may appear rather complicated at first, but from the definitions and with the help of Figure 20.4, it is not too hard to reconstruct the algorithm.

We need to worry about the members S_i and G_i of the S- and G-sets. For each one, the new example may be a false positive or a false negative.

1. False positive for S_i: This means S_i is too general, but there are no consistent specializations of S_i (by definition), so we throw it out of the S-set.
2. False negative for S_i: This means S_i is too specific, so we replace it by all its immediate generalizations, provided they are more specific than some member of G.
3. False positive for G_i: This means G_i is too general, so we replace it by all its immediate specializations, provided they are more general than some member of S.

4. False negative for G_i: This means G_i is too specific, but there are no consistent generalizations of G_i (by definition) so we throw it out of the G-set.

We continue these operations for each new example until one of three things happens:

1. We have exactly one hypothesis left in the version space, in which case we return it as the unique hypothesis.
2. The version space *collapses*—either S or G becomes empty, indicating that there are no consistent hypotheses for the training set. This is the same case as the failure of the simple version of the decision tree algorithm.
3. We run out of examples and have several hypotheses remaining in the version space. This means the version space represents a disjunction of hypotheses. For any new example, if all the disjuncts agree, then we can return their classification of the example. If they disagree, one possibility is to take the majority vote.

We leave as an exercise the application of the VERSION-SPACE-LEARNING algorithm to the restaurant data.

There are two principal drawbacks to the version-space approach:

- If the domain contains noise or insufficient attributes for exact classification, the version space will always collapse.
- If we allow unlimited disjunction in the hypothesis space, the S-set will always contain a single most-specific hypothesis, namely, the disjunction of the descriptions of the positive examples seen to date. Similarly, the G-set will contain just the negation of the disjunction of the descriptions of the negative examples.
- For some hypothesis spaces, the number of elements in the S-set or G-set may grow exponentially in the number of attributes, even though efficient learning algorithms exist for those hypothesis spaces.

To date, no completely successful solution has been found for the problem of noise. The problem of disjunction can be addressed by allowing only limited forms of disjunction or by including a **generalization hierarchy** of more general predicates. For example, instead of using the disjunction *WaitEstimate*(*x*, *30-60*) ∨ *WaitEstimate*(*x*, *>60*), we might use the single literal *LongWait*(*x*). The set of generalization and specialization operations can be easily extended to handle this.

The pure version space algorithm was first applied in the Meta-DENDRAL system, which was designed to learn rules for predicting how molecules would break into pieces in a mass spectrometer (Buchanan and Mitchell, 1978). Meta-DENDRAL was able to generate rules that were sufficiently novel to warrant publication in a journal of analytical chemistry—the first real scientific knowledge generated by a computer program. It was also used in the elegant LEX system (Mitchell *et al.*, 1983), which was able to learn to solve symbolic integration problems by studying its own successes and failures. Although version space methods are probably not practical in most real-world learning problems, mainly because of noise, they provide a good deal of insight into the logical structure of hypothesis space.

20.2 Knowledge in Learning

The preceding section described the simplest setting for inductive learning. To understand the role of prior knowledge, we need to talk about the logical relationships among hypotheses, example descriptions, and classifications. Let *Descriptions* denote the conjunction of all the example descriptions in the training set, and let *Classifications* denote the conjunction of all the example classifications. Then a *Hypothesis* that "explains the observations" must satisfy the following property (recall that ⊨ means "logically entails"):

$$Hypothesis \wedge Descriptions \models Classifications\ . \qquad (20.3)$$

We call this kind of relationship an **entailment constraint**, in which *Hypothesis* is the "unknown." Pure inductive learning means solving this constraint, where *Hypothesis* is drawn from some predefined hypothesis space. For example, if we consider a decision tree as a logical formula (see Equation (20.1) on page 740), then a decision tree that is consistent with all the examples will satisfy Equation (20.3). If we place *no* restrictions on the logical form of the hypothesis, of course, then *Hypothesis* = *Classifications* also satisfies the constraint. Ockham's razor tells us to prefer *small*, consistent hypotheses, so we try to do better than simply memorizing the examples.

This simple knowledge-free picture of inductive learning persisted until the early 1980s. The modern approach is to design agents that *already know something* and are trying to learn some more. This may not sound like a terrifically deep insight, but it makes quite a difference to the way we design agents. It might also have some relevance to our theories about how science itself works. The general idea is shown schematically in Figure 20.6.

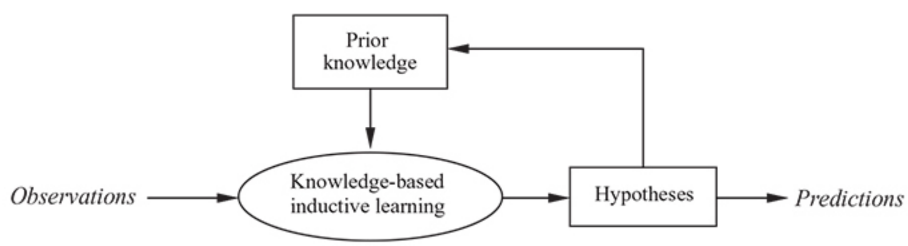


Figure 20.6 A cumulative learning process uses, and adds to, its stock of background knowledge over time.

An autonomous learning agent that uses background knowledge must somehow obtain the background knowledge in the first place, in order for it to be used in the new learning episodes. This method must itself be a learning process. The agent's life history will therefore be characterized by *cumulative*, or *incremental*, development. Presumably, the agent could start out with nothing, performing inductions *in vacuo* like a good little pure induction program. But once it has eaten from the Tree of Knowledge, it can no longer pursue such naive speculations and should use its background knowledge to learn more and more effectively. The question is then how to actually do this.

20.2.1 Some simple examples

Let us consider some commonsense examples of learning with background knowledge. Many apparently rational cases of inferential behavior in the face of observations clearly do not follow the simple principles of pure induction.

- Sometimes one leaps to general conclusions after only one observation. Gary Larson once drew a cartoon in which a bespectacled caveman, Zog, is roasting his lizard on the end of a pointed stick. He is watched by an amazed crowd of his less intellectual contemporaries, who have been using their bare hands to hold their victuals over the fire. This enlightening experience is enough to convince the watchers of a general principle of painless cooking.
- Or consider the case of the traveler to Brazil meeting her first Brazilian. On hearing him speak Portuguese, she immediately concludes that Brazilians speak Portuguese, yet on discovering that his name is Fernando, she does not conclude that all Brazilians are called Fernando. Similar examples appear in science. For example, when a freshman physics student measures the density and conductance of a sample of copper at a particular temperature, she is quite confident in generalizing those values to all pieces of copper. Yet when she measures its mass, she does not even consider the hypothesis that all pieces of copper have that mass. On the other hand, it would be quite reasonable to make such a generalization over all pennies.
- Finally, consider the case of a pharmacologically ignorant but diagnostically sophisticated medical student observing a consulting session between a patient and an expert internist. After a series of questions and answers, the expert tells the patient to take a course of a particular antibiotic. The medical student infers the general rule that that particular antibiotic is effective for a particular type of infection.

These are all cases in which *the use of background knowledge allows much faster learning than one might expect from a pure induction program*.

20.2.2 Some general schemes

In each of the preceding examples, one can appeal to prior knowledge to try to justify the generalizations chosen. We will now look at what kinds of entailment constraints are operating in each case. The constraints will involve the *Background* knowledge, in addition to the *Hypothesis* and the observed *Descriptions* and *Classifications*.

In the case of lizard toasting, the cavemen generalize by *explaining* the success of the pointed stick: it supports the lizard while keeping the hand away from the fire. From this explanation, they can infer a general rule: that any long, rigid, sharp object can be used to toast small, soft-bodied edibles. This kind of generalization process has been called **explanation-based learning**, or **EBL**. Notice that the general rule *follows logically* from the background knowledge possessed by the cavemen. Hence, the entailment constraints satisfied by EBL are the following:

$$Hypothesis \land Descriptions \models Classifications$$
$$Background \models Hypothesis.$$

Because EBL uses Equation (20.3), it was initially thought to be a way to learn from examples. But because it requires that the background knowledge be sufficient to explain the *Hypothesis*, which in turn explains the observations, *the agent does not actually learn anything factually new from the example*. The agent *could have* derived the example from what it already knew, although that might have required an unreasonable amount of computation. EBL is now viewed as a method for converting first-principles theories into useful, special-purpose knowledge. We describe algorithms for EBL in Section 20.3.

The situation of our traveler in Brazil is quite different, for she cannot necessarily explain why Fernando speaks the way he does, unless she knows her papal bulls. Moreover, the same generalization would be forthcoming from a traveler entirely ignorant of colonial history. The relevant prior knowledge in this case is that, within any given country, most people tend to speak the same language; on the other hand, Fernando is not assumed to be the name of all Brazilians because this kind of regularity does not hold for names. Similarly, the freshman physics student also would be hard put to explain the particular values that she discovers for the conductance and density of copper. She does know, however, that the material of which an object is composed and its temperature together determine its conductance. In each case, the prior knowledge *Background* concerns the **relevance** of a set of features to the goal predicate. This knowledge, *together with the observations*, allows the agent to infer a new, general rule that explains the observations:

$$Hypothesis \land Descriptions \models Classifications$$
$$Background \land Descriptions \land Classifications \models Hypothesis. \qquad (20.4)$$

We call this kind of generalization **relevance-based learning**, or **RBL** (although the name is not standard). Notice that whereas RBL does make use of the content of the observations, it does not produce hypotheses that go beyond the logical content of the background knowledge and the observations. It is a *deductive* form of learning and cannot by itself account for the creation of new knowledge starting from scratch.

In the case of the medical student watching the expert, we assume that the student's prior knowledge is sufficient to infer the patient’s disease *D* from the symptoms. This is not, however, enough to explain the fact that the doctor prescribes a particular medicine *M*. The student needs to propose another rule, namely, that *M* generally is effective against *D*. Given this rule and the student’s prior knowledge, the student can now explain why the expert prescribes *M* in this particular case. We can generalize this example to come up with the entailment constraint

$$Background \land Hypothesis \land Descriptions \models Classifications \qquad (20.5)$$

That is, *the background knowledge and the new hypothesis combine to explain the examples*. As with pure inductive learning, the learning algorithm should propose hypotheses that are as simple as possible, consistent with this constraint. Algorithms that satisfy constraint (20.5) are called **knowledge-based inductive learning**, or **KBIL**, algorithms.

KBIL algorithms, which are described in detail in Section 20.5, have been studied mainly in the field of **inductive logic programming**, or **ILP**. In ILP systems, prior knowledge plays two key roles in reducing the complexity of learning:

1. Because any hypothesis generated must be consistent with the prior knowledge as well as with the new observations, the effective hypothesis space size is reduced to include only those theories that are consistent with what is already known.
2. For any given set of observations, the size of the hypothesis required to construct an explanation for the observations can be much reduced, because the prior knowledge will be available to help out the new rules in explaining the observations. The smaller the hypothesis, the easier it is to find.

In addition to allowing the use of prior knowledge in induction, ILP systems can formulate hypotheses in general first-order logic, rather than in the restricted attribute-based language of Chapter 19. This means that they can learn in environments that cannot be understood by simpler systems.

20.3 Explanation-Based Learning

Explanation-based learning is a method for extracting general rules from individual observations. As an example, consider the problem of differentiating and simplifying algebraic expressions (Exercise 9.17). If we differentiate an expression such as X^2 with respect to X, we obtain $2X$. (We use a capital letter for the arithmetic unknown X, to distinguish it from the logical variable x.) In a logical reasoning system, the goal might be expressed as $\text{ASK}(\mathit{Derivative}(X^2, X) = d, \mathit{KB})$, with solution $d = 2X$.

Anyone who knows differential calculus can see this solution "by inspection" as a result of practice in solving such problems. A student encountering such problems for the first time, or a program with no experience, will have a much more difficult job. Application of the standard rules of differentiation eventually yields the expression $1 \times (2 \times (X^{(2-1)}))$, and eventually this simplifies to $2X$. In the authors' logic programming implementation, this takes 136 proof steps, of which 99 are on dead-end branches in the proof. After such an experience, we would like the program to solve the same problem much more quickly the next time it arises.

The technique of **memoization** has long been used in computer science to speed up programs by saving the results of computation. The basic idea of memo functions is to accumulate a database of input–output pairs; when the function is called, it first checks the database to see whether it can avoid solving the problem from scratch. Explanation-based learning takes this a good deal further, by creating *general* rules that cover an entire class of cases. In the case of differentiation, memoization would remember that the derivative of X^2 with respect to X is $2X$, but would leave the agent to calculate the derivative of Z^2 with respect to Z from scratch. We would like to be able to extract the general rule that for any arithmetic unknown u, the derivative of u^2 with respect to u is $2u$. (An even more general rule for u^n can also be produced, but the current example suffices to make the point.) In logical terms, this is expressed by the rule

$$ArithmeticUnknown\,(u) \Rightarrow Derivative(u^2, u) = 2u.$$

If the knowledge base contains such a rule, then any new case that is an instance of this rule can be solved immediately.

This is, of course, merely a trivial example of a very general phenomenon. Once something is understood, it can be generalized and reused in other circumstances. It becomes an “obvious” step and can then be used as a building block in solving problems still more complex. Alfred North Whitehead (1911), co-author with Bertrand Russell of *Principia Mathematica,* wrote *“Civilization advances by extending the number of important operations that we can do without thinking about them,”* perhaps himself applying EBL to his understanding of events such as Zog’s discovery. If you have understood the basic idea of the differentiation example, then your brain is already busily trying to extract the general principles of explanation-based learning from it. Notice that you hadn’t *already* invented EBL before you saw the example. Like the cavemen watching Zog, you (and we) needed an example before we could generate the basic principles. This is because *explaining why* something is a good idea is much easier than coming up with the idea in the first place.

20.3.1 Extracting general rules from examples

The basic idea behind EBL is first to construct an explanation of the observation using prior knowledge, and then to establish a definition of the class of cases for which the same explanation structure can be used. This definition provides the basis for a rule covering all of the cases in the class. The “explanation” can be a logical proof, but more generally it can be any reasoning or problem-solving process whose steps are well defined. The key is to be able to identify the necessary conditions for those same steps to apply to another case.

We will use for our reasoning system the simple backward-chaining theorem prover described in Chapter 9. The proof tree for $Derivative(X^2, X) = 2X$ is too large to use as an example, so we will use a simpler problem to illustrate the generalization method. Suppose our problem is to simplify $1 \times (0 + X)$. The knowledge base includes the following rules:

$$Rewrite(u, v) \land Simplify(v, w) \Rightarrow Simplify(u, w).$$
$$Primitive(u) \ \Rightarrow \ Simplify(u, u) \ .$$
$$ArithmeticUnknown(u) \ \Rightarrow \ Primitive(u).$$
$$Number(u) \ \Rightarrow \ Primitive(u) \ .$$
$$Rewrite(1 \times u, u).$$
$$Rewrite(0 + u, u).$$
$$\vdots$$

The proof that the answer is X is shown in the top half of Figure 20.7. The EBL method actually constructs two proof trees simultaneously. The second proof tree uses a *variabilized* goal in which the constants from the original goal are replaced by variables. As the original proof proceeds, the variabilized proof proceeds in step, using *exactly the same rule applications*. This could cause some of the variables to become instantiated. For example, in order to use the rule $Rewrite(1 \times u, u)$, the variable x in the subgoal $Rewrite(x \times (y + z), v)$ must be bound to 1. Similarly, y must be bound to 0 in the subgoal $Rewrite(y + z, v')$ in order to use the rule $Rewrite(0 + u, u)$. Once we have the generalized proof tree, we take the leaves (with the necessary bindings) and form a general rule for the goal predicate:

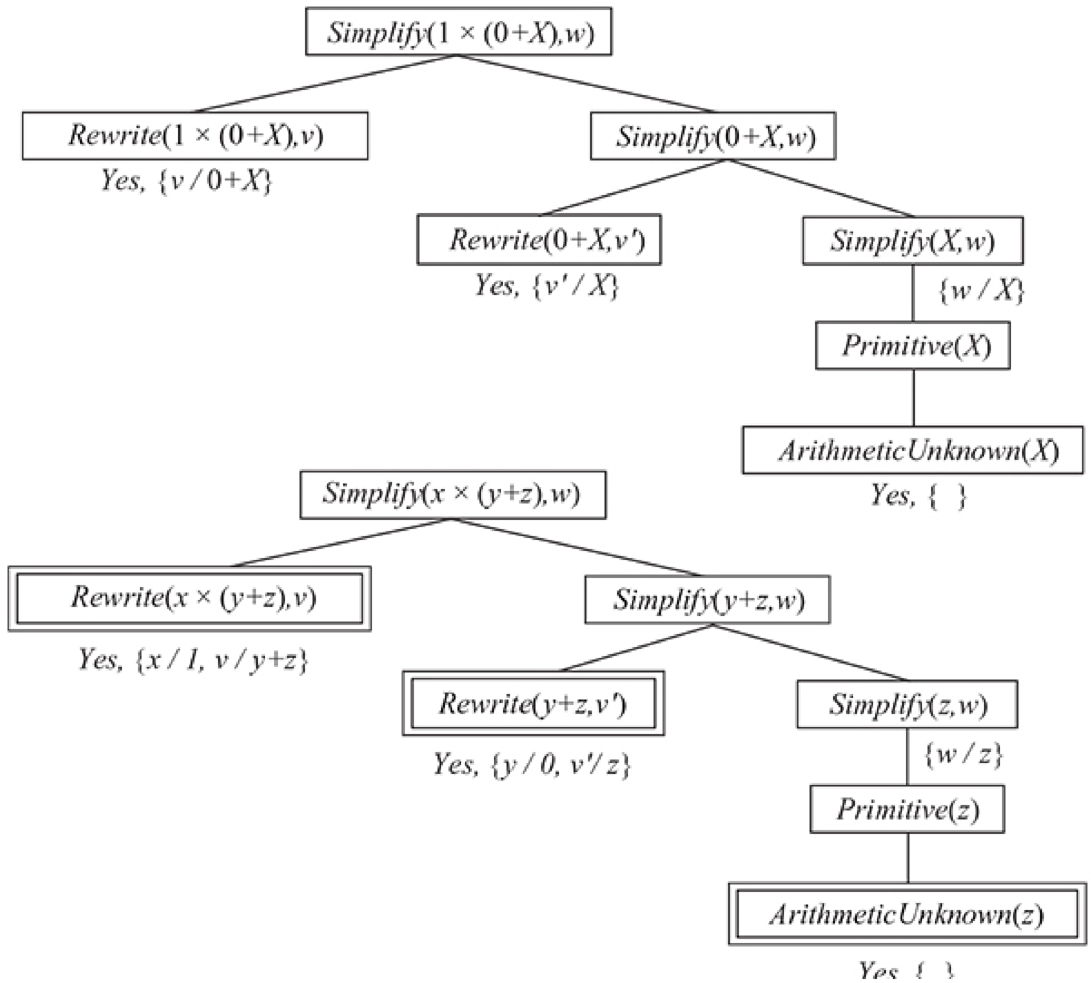


Figure 20.7 Proof trees for the simplification problem. The first tree shows the proof for the original problem instance, from which we can derive

$$ArithmeticUnknown(z) \Rightarrow Simplify(1 \times (0 + z), z) .$$

The second tree shows the proof for a problem instance with all constants replaced by variables, from which we can derive a variety of other rules.

$$Rewrite(1 \times (0 + z); 0 + z) \wedge Rewrite(0 + z, z) \wedge ArithmeticUnknown(z)$$
$$\Rightarrow \quad Simplify(1 \times (0 + z), z).$$

Notice that the first two conditions on the left-hand side are true *regardless of the value of* z. We can therefore drop them from the rule, yielding

$$ArithmeticUnknown\,(z) \;\Rightarrow\; \text{Simplify}\,(1 \times (0 + z), z)\;.$$

In general, conditions can be dropped from the final rule if they impose no constraints on the variables on the right-hand side of the rule, because the resulting rule will still be true and will be more efficient. Notice that we cannot drop the condition *ArithmeticUnknown*(*z*), because not all possible values of *z* are arithmetic unknowns. Values other than arithmetic unknowns might require different forms of simplification: for example, if *z* were 2 × 3, then the correct simplification of 1 × (0 + (2 × 3)) would be 6 and not 2 × 3.

To recap, the basic EBL process works as follows:

1. Given an example, construct a proof that the goal predicate applies to the example using the available background knowledge.
2. In parallel, construct a generalized proof tree for the variabilized goal using the same inference steps as in the original proof.
3. Construct a new rule whose left-hand side consists of the leaves of the proof tree and whose right-hand side is the variabilized goal (after applying the necessary bindings from the generalized proof).
4. Drop any conditions from the left-hand side that are true regardless of the values of the variables in the goal.

20.3.2 Improving efficiency

The generalized proof tree in Figure 20.7 actually yields more than one generalized rule. For example, if we terminate, or **prune**, the growth of the right-hand branch in the proof tree when it reaches the *Primitive* step, we get the rule

$$Primitive(z) \;\Rightarrow\; Simplify(1 \times (0 + z),\; z).$$

This rule is as valid as, but *more general* than, the rule using *ArithmeticUnknown*, because it covers cases where *z* is a number. We can extract a still more general rule by pruning after the step *Simplify* ($y + z, w$), yielding the rule

$$Simplify(y + z, w) \;\Rightarrow\; Simplify(1 \times (y + z), w).$$

In general, a rule can be extracted from *any partial subtree* of the generalized proof tree. Now we have a problem: which of these rules do we choose?

The choice of which rule to generate comes down to the question of efficiency. There are three factors involved in the analysis of efficiency gains from EBL:

1. Adding large numbers of rules can slow down the reasoning process, because the inference mechanism must still check those rules even in cases where they do not yield a solution. In other words, it increases the **branching factor** in the search space.
2. To compensate for the slowdown in reasoning, the derived rules must offer significant increases in speed for the cases that they do cover. These increases come about mainly because the derived rules avoid dead ends that would otherwise be taken, but also because they shorten the proof itself.
3. Derived rules should be as general as possible, so that they apply to the largest possible set of cases.

A common approach to ensuring that derived rules are efficient is to insist on the **operationality** of each subgoal in the rule. A subgoal is operational if it is “easy” to solve. For example, the subgoal *Primitive*(z) is easy to solve, requiring at most two steps, whereas the subgoal *Simplify* ($y + z$, w) could lead to an arbitrary amount of inference, depending on the values of y and z. If a test for operationality is carried out at each step in the construction of the generalized proof, then we can prune the rest of a branch as soon as an operational subgoal is found, keeping just the operational subgoal as a conjunct of the new rule.

Unfortunately, there is usually a tradeoff between operationality and generality. More specific subgoals are generally easier to solve but cover fewer cases. Also, operationality is a matter of degree: one or two steps is definitely operational, but what about 10 or 100? Finally, the cost of solving a given subgoal depends on what other rules are available in the knowledge base. It can go up or down as more rules are added. Thus, EBL systems really face a very complex optimization problem in trying to maximize the efficiency of a given initial knowledge base. It is sometimes possible to derive a mathematical model of the effect on overall efficiency of adding a given rule and to use this model to select the best rule to add. The analysis can become very complicated, however, especially when recursive rules are involved. One promising approach is to address the problem of efficiency empirically, simply by adding several rules and seeing which ones are useful and actually speed things up.

Empirical analysis of efficiency is actually at the heart of EBL. What we have been calling loosely the "efficiency of a given knowledge base" is actually the average-case complexity on a distribution of problems. *By generalizing from past example problems, EBL makes the knowledge base more efficient for the kind of problems that it is reasonable to expect*. This works as long as the distribution of past examples is roughly the same as for future examples—the same assumption used for PAC-learning in Section 19.5. If the EBL system is carefully engineered, it is possible to obtain significant speedups. For example, a very large Prolog-based natural language system designed for speech-to-speech translation between Swedish and English was able to achieve real-time performance only by the application of EBL to the parsing process (Samuelsson and Rayner, 1991).

20.4 Learning Using Relevance Information

Our traveler in Brazil seems to be able to make a confident generalization concerning the language spoken by other Brazilians. The inference is sanctioned by her background knowledge, namely, that people in a given country (usually) speak the same language. We can express this in first-order logic as follows:[2]

$$Nationality(x,n) \wedge Nationality(y,n) \wedge Language(x,l) \Rightarrow Language(y,l).$$

(Literal translation: "If *x* and *y* have the same nationality *n* and *x* speaks language *l*, then *y* also speaks it.") It is not difficult to show that, from this sentence and the observation that

$$Nationality(Fernando, Brazil) \wedge Language(Fernando, Portuguese),$$

the following conclusion is entailed (see Exercise 20.1):

$$Nationality(x, Brazil) \;\Rightarrow\; Language\,(x, Portuguese).$$

Sentences such as (20.6) express a strict form of relevance: given nationality, language is fully determined. (Put another way: language is a function of nationality.) These sentences are called **functional dependencies** or **determinations**. They occur so commonly in certain kinds of applications (e.g., defining database designs) that a special syntax is used to write them. We adopt the notation of Davies (1985):

$$Nationality(x,n) \;\succ\; Language(x,\ l).$$

As usual, this is simply a syntactic sugaring, but it makes it clear that the determination is really a relationship between the predicates: nationality determines language. The relevant properties determining conductance and density can be expressed similarly:

$$Material(x,m) \wedge Temperature(x,t) \;\succ Conductance(x,\rho);$$
$$Material(x,m) \wedge Temperature(x,\ t) \;\succ Density(x,d).$$

The corresponding generalizations follow logically from the determinations and observations.

20.4.1 Determining the hypothesis space

Although the determinations sanction general conclusions concerning all Brazilians, or all pieces of copper at a given temperature, they cannot, of course, yield a general predictive theory for *all* nationalities, or for *all* temperatures and materials, from a single example. Their main effect can be seen as limiting the space of hypotheses that the learning agent need consider. In predicting conductance, for example, one need consider only material and temperature and can ignore mass, ownership, day of the week, the current president, and so on. Hypotheses can certainly include terms that are in turn determined by material and temperature, such as molecular structure, thermal energy, or free-electron density. *Determinations specify a sufficient basis vocabulary from which to construct hypotheses concerning the target predicate*. This statement can be proven by showing that a given determination is logically equivalent to a statement that the correct definition of the target predicate is one of the set of all definitions expressible using the predicates on the left-hand side of the determination.

Intuitively, it is clear that a reduction in the hypothesis space size should make it easier to learn the target predicate. Using the basic results of computational learning theory (Section 19.5), we can quantify the possible gains. First, recall that for Boolean functions, $\log(|\mathcal{H}|)$ examples are required to converge to a reasonable hypothesis, where $|\mathcal{H}|$ is the size of the hypothesis space. If the learner has *n* Boolean features with which to construct hypotheses, then, in the absence of further restrictions, $|\mathcal{H}| = O(2^{2^n})$, so the number of examples is $O(2^n)$. If the determination contains d predicates in the left-hand side, the learner will require only $O(2^d)$ examples, a reduction of $O(2^{n-d})$.

20.4.2 Learning and using relevance information

As we stated in the introduction to this chapter, prior knowledge is useful in learning; but it too has to be learned. In order to provide a complete story of relevance-based learning, we must therefore provide a learning algorithm for determinations. The learning algorithm we now present is based on a straightforward attempt to find the simplest determination consistent with the observations. A determination $P \prec Q$ says that if any examples match on P, then they must also match on Q. A determination is therefore consistent with a set of examples if every pair that matches on the predicates on the left-hand side also matches on the goal predicate. For example, suppose we have the following examples of conductance measurements on material samples:
The minimal consistent determination is $Material \wedge Temperature \succ Conductance$. There is a nonminimal but consistent determination, namely, $Mass \wedge Size \wedge Temperature \succ Conductance$. This is consistent with the examples because mass and size determine density and, in our data set, we do not have two different materials with the same density. As usual, we would need a larger sample set in order to eliminate a nearly correct hypothesis.

There are several possible algorithms for finding minimal consistent determinations. The most obvious approach is to conduct a search through the space of determinations, checking all determinations with one predicate, two predicates, and so on, until a consistent determination is found. We will assume a simple attribute-based representation, like that used for decision tree learning in Chapter 19. A determination d will be represented by the set of attributes on the left-hand side, because the target predicate is assumed to be fixed. The basic algorithm is outlined in Figure 20.8.

---

function MINIMAL-CONSISTENT-DET(E, A) **returns** a set of attributes
  inputs: E, a set of examples
    A, a set of attributes, of size n

  for $i = 0$ **to** n **do**
    for each subset A_i of A of size i **do**
      if CONSISTENT-DET?(A_i, E) **then return** A_i

---

function CONSISTENT-DET?(A, E) **returns** a truth value
  inputs: A, a set of attributes
    E, a set of examples
  local variables: H, a hash table

  for each example e **in** E **do**
    if some example in H has the same values as e for the attributes A
      but a different classification **then return** *false*
    store the class of e in H, indexed by the values for attributes A of the example e
  return *true*

Figure 20.8 An algorithm for finding a minimal consistent determination.

The time complexity of this algorithm depends on the size of the smallest consistent determination. Suppose this determination has p attributes out of the n total attributes. Then the algorithm will not find it until searching the subsets of A of size p. There are $\binom{n}{p} = O(n^p)$ such subsets; hence the algorithm is exponential in the size of the minimal determination. It turns out that the problem is NP-complete, so we cannot expect to do better in the general case. In most domains, however, there will be sufficient local structure (see chapter 13 for a definition of locally structured domains) that p will be small.

Given an algorithm for learning determinations, a learning agent has a way to construct a minimal hypothesis within which to learn the target predicate. For example, we can combine MINIMAL-CONSISTENT-DET with the DECISION-TREE-LEARNING algorithm. This yields a relevance-based decision-tree learning algorithm RBDTL that first identifies a minimal set of relevant attributes and then passes this set to the decision tree algorithm for learning. Unlike DECISION-TREE-LEARNING, RBDTL simultaneously learns and uses relevance information in order to minimize its hypothesis space. We expect that RBDTL will learn faster than DECISION-TREE-LEARNING, and this is in fact the case. Figure 20.9 shows the learning performance for the two algorithms on randomly generated data for a function that depends on only 5 of 16 attributes. obviously, in cases where all the available attributes are relevant, RBDTL will show no advantage.

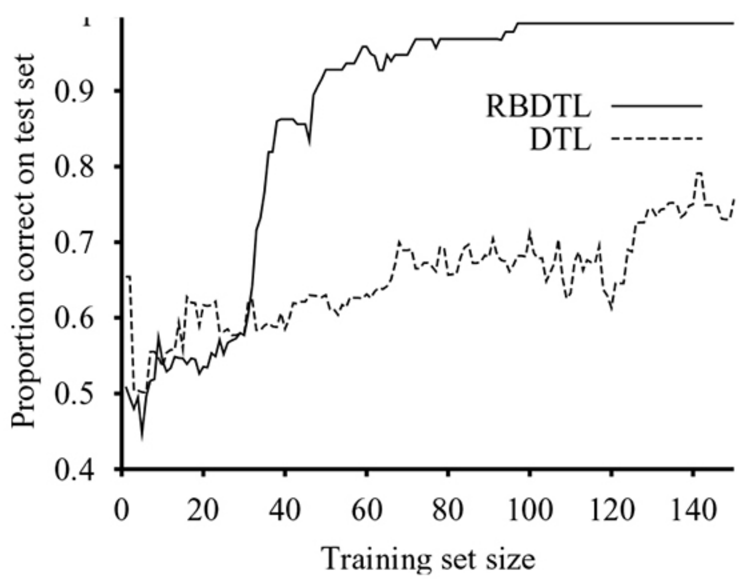


Figure 20.9 A performance comparison between DECISION-TREE-LEARNING and RBDTL on randomly generated data for a target function that depends on only 5 of 16 attributes.

This section has only scratched the surface of the field of **declarative bias**, which aims to understand how prior knowledge can be used to identify the appropriate hypothesis space within which to search for the correct target definition. There are many unanswered questions:

- How can the algorithms be extended to handle noise?
- Can we handle continuous-valued variables?

- How can other kinds of prior knowledge be used, besides determinations?
- How can the algorithms be generalized to cover any first-order theory, rather than just an attribute-based representation?

Some of these questions are addressed in the next section.

20.5 Inductive Logic Programming

Inductive logic programming (ILP) combines inductive methods with the power of first-order representations, concentrating in particular on the representation of hypotheses as logic programs.[3] It has gained popularity for three reasons. First, ILP offers a rigorous approach to the general knowledge-based inductive learning problem. Second, it offers complete algorithms for inducing general, first-order theories from examples, which can therefore learn successfully in domains where attribute-based algorithms are hard to apply. An example is in learning how protein structures fold (Figure 20.10). The three-dimensional configuration of a protein molecule cannot be represented reasonably by a set of attributes, because the configuration inherently refers to *relationships* between objects, not to attributes of a single object. First-order logic is an appropriate language for describing the relationships. Third, inductive logic programming produces hypotheses that are (relatively) easy for humans to read. For example, the English translation in Figure 20.10 can be scrutinized and criticized by working biologists. This means that inductive logic programming systems can participate in the scientific cycle of experimentation, hypothesis generation, debate, and refutation. Such participation would not be possible for systems that generate "black-box" classifiers, such as neural networks.

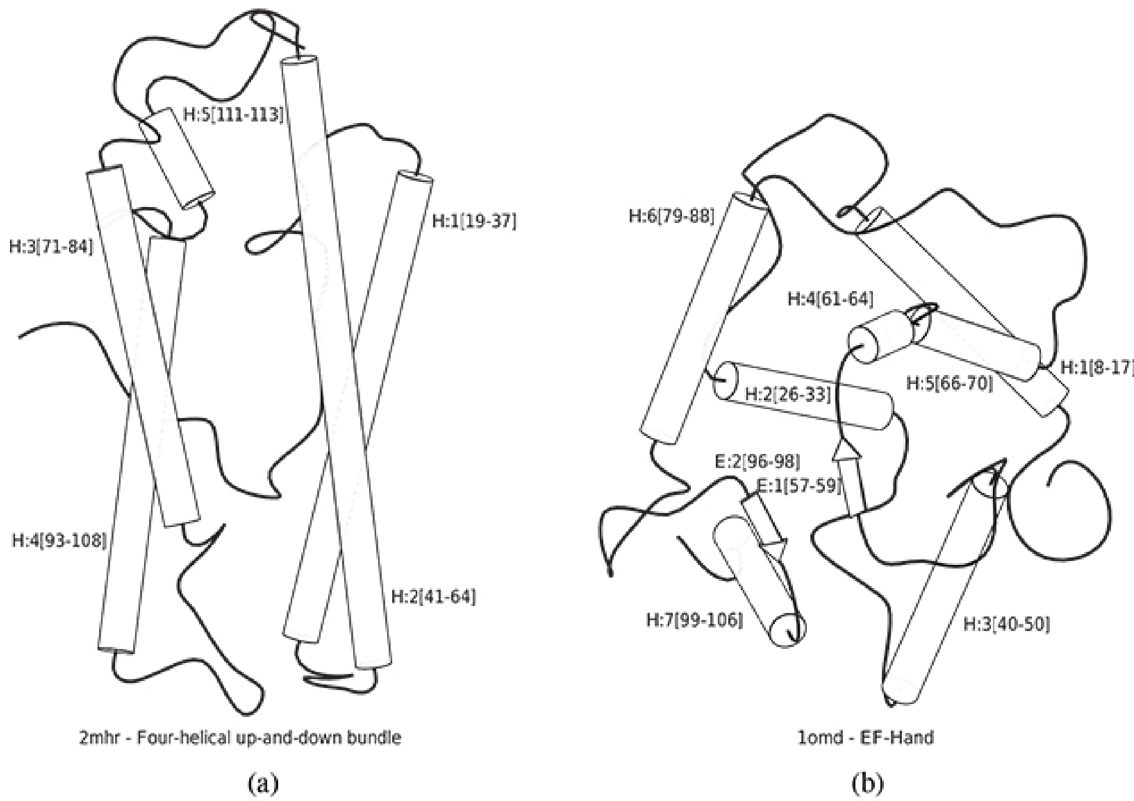


Figure 20.10 (a) and (b) show positive and negative examples, respectively, of the "four-helical up-and-down bundle" concept in the domain of protein folding. Each example structure is coded into a logical expression of about 100 conjuncts such as *TotalLength*(*D2mhr*;118) ∧ *NumberHelices*(*D2mhr*;6) ∧ .... From these descriptions and from classifications such as *Fold* (FOUR-HELICAL-UP-AND-DOWN-BUNDLE,*D2mhr*), the ILP system PROGOL (Muggleton, 1995) learned the following rule:

$$\begin{aligned}&Fold\,(\text{FOUR-HELICAL-UP-AND-DOWN-BUNDLE}, p) \Leftarrow\\&\quad Helix(p, h_1) \wedge Length(h_1, HIGH) \wedge Position(p, h_1, n)\\&\qquad \wedge(1 \leq n \leq 3) \wedge Adjacent(p, h_1, h_2) \wedge Helix(p, h_2).\end{aligned}$$

This kind of rule could not be learned, or even represented, by an attribute-based mechanism such as we saw in previous chapters. The rule can be translated into English as " Protein p has fold class "Four-helical up-and-down-bundle" if it contains a long helix h_1 at a secondary structure position between 1 and 3 and h_1 is next to a second helix."

20.5.1 An example

Recall from Equation (20.5) that the general knowledge-based induction problem is to "solve" the entailment constraint

$$Background \wedge Hypothesis \wedge Descriptions \models Classifications$$

for the unknown *Hypothesis*, given the *Background* knowledge and examples described by *Descriptions* and *Classifications*. To illustrate this, we will use the problem of learning family relationships from examples. The descriptions will consist of an extended family tree, described in terms of *Mother*, *Father*, and *Married* relations and *Male* and *Female* properties. As an example, we will use the family tree from Exercise 8.15, shown here in Figure 20.11. The corresponding descriptions are as follows:

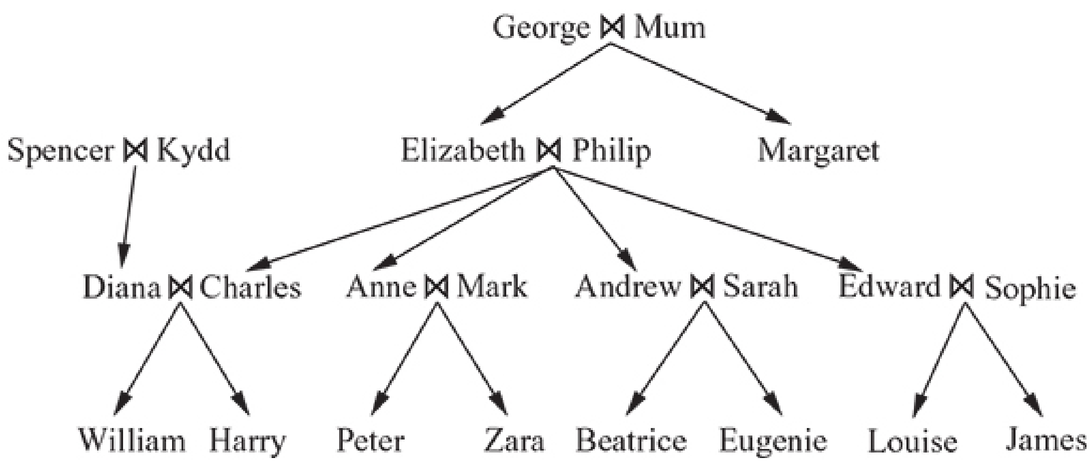


Figure 20.11 A typical family tree.

$$\begin{array}{lll}
Father(Philip, Charles) & Father(Philip, Anne) & \ldots \\
Mother(Mum, Margaret) & Mother(Mum, Elizabeth) & \ldots \\
Married(Diana, Charles) & Married(Elizabeth, Philip) & \ldots \\
Male(Philip) & Male(Charles) & \ldots \\
Female(Beatrice) & Female(Margaret) & \ldots
\end{array}$$

The sentences in *Classifications* depend on the target concept being learned. We might want to learn *Grandparent, BrotherInLaw,* or *Ancestor,* for example. For *Grandparent,* the complete set of *Classifications* contains 20 × 20 = 400 conjuncts of the form

$$\begin{array}{lll}
Grandparent(Mum, Charles) & Grandparent(Elizabeth, Beatrice) & \ldots \\
\neg Grandparent(Mum, Harry) & \neg Grandparent(Spencer, Peter) & \ldots
\end{array}$$

We could of course learn from a subset of this complete set.

The object of an inductive learning program is to come up with a set of sentences for the *Hypothesis* such that the entailment constraint is satisfied. Suppose, for the moment, that the agent has no background knowledge:

Background is empty. Then one possible solution for *Hypothesis* is the following:

$$\begin{aligned} Grandparent(x,y) \;\Leftrightarrow\; & [\exists z\; Mother(x,z) \wedge Mother(z,y)] \\ \vee\; & [\exists z\; Mother(x,z) \wedge Father(z,y)] \\ \vee\; & [\exists z\; Father(x,z) \wedge Mother(z,y)] \\ \vee\; & [\exists z\; Father(x,z) \wedge Father(z,y)] \;: \end{aligned}$$

Notice that an attribute-based learning algorithm, such as DECISION-TREE-LEARNING, will get nowhere in solving this problem. In order to express *Grandparent* as an attribute (i.e., a unary predicate), we would need to make *pairs* of people into objects:

$$Grandparent(\langle Mum, Charles\rangle)\; \ldots$$

Then we get stuck in trying to represent the example descriptions. The only possible attributes are horrible things such as

$$FirstElementIsMotherOfElizabeth(\langle Mum, Charles\rangle)\;.$$

The definition of *Grandparent* in terms of these attributes simply becomes a large disjunction of specific cases that does not generalize to new examples at all. *Attribute-based learning algorithms are incapable of learning relational predicates*. Thus, one of the principal advantages of ILP algorithms is their applicability to a much wider range of problems, including relational problems.

The reader will certainly have noticed that a little bit of background knowledge would help in the representation of the *Grandparent* definition. For example, if *Background* included the sentence

$$Parent(x,y) \;\Leftrightarrow\; [Mother(x,y) \vee Father(x,y)],$$

then the definition of *Grandparent* would be reduced to

$$Grandparent(x,y) \Leftrightarrow [\exists z\, Parent(x,z) \wedge Parent(z,y)].$$

This shows how background knowledge can dramatically reduce the size of hypotheses required to explain the observations.

It is also possible for ILP algorithms to *create* new predicates in order to facilitate the expression of explanatory hypotheses. Given the example data shown earlier, it is entirely reasonable for the ILP program to propose an additional predicate, which we would call *"Parent,"* in order to simplify the definitions of the target predicates. Algorithms that can generate new predicates are called **constructive induction** algorithms. Clearly, constructive induction is a necessary part of the picture of cumulative learning. It has been one of the hardest problems in machine learning, but some ILP techniques provide effective mechanisms for achieving it.

In the rest of this chapter, we will study the two principal approaches to ILP. The first uses a generalization of decision tree methods, and the second uses techniques based on inverting a resolution proof.

20.5.2 Top-down inductive learning methods

The first approach to ILP works by starting with a very general rule and gradually specializing it so that it fits the data. This is essentially what happens in decision-tree learning, where a decision tree is gradually grown until it is consistent with the observations. To do ILP we use first-order literals instead of attributes, and the hypothesis is a set of clauses instead of a decision tree. This section describes FOIL (Quinlan, 1990), one of the first ILP programs.

Suppose we are trying to learn a definition of the *Grandfather*(x, y) predicate, using the same family data as before. As with decision-tree learning, we can divide the examples into positive and negative examples. Positive examples are

$$\langle George, Anne\rangle,\ \langle Philip, Peter\rangle,\ \langle Spencer, Harry\rangle, \ldots$$

and negative examples are

$$\langle George,\ Elizabeth\rangle,\ \langle Harry, Zara\rangle,\ \langle Charles,\ Philip\rangle, \ldots$$

Notice that each example is a *pair* of objects, because *Grandfather* is a binary predicate. In all, there are 12 positive examples in the family tree and 388 negative examples (all the other pairs of people).

FOIL constructs a set of clauses, each with *Grandfather*(x, y) as the head. The clauses must classify the 12 positive examples as instances of the *Grandfather*(x, y) relationship, while ruling out the 388 negative examples. The clauses are Horn clauses, with the extension that negated literals are allowed in the body of a clause and are interpreted using negation as failure, as in Prolog. The initial clause has an empty body:

$$\Rightarrow \ Grandfather(x, y).$$

This clause classifies every example as positive, so it needs to be specialized. We do this by adding literals one at a time to the left-hand side. Here are three potential additions:

$$Father(x, y) \ \Rightarrow \ Grandfather(x, y).$$
$$Parent(x,\ z) \ \Rightarrow \ Grandfather(x, y).$$
$$Father(x,\ z) \ \Rightarrow \ Grandfather(x, y).$$

(Notice that we are assuming that a clause defining *Parent* is already part of the background knowledge.) The first of these three clauses incorrectly classifies all of the 12 positive examples as negative and can thus be ignored. The second and third agree with all of the positive examples, but the second is incorrect on a larger fraction of the negative examples—twice as many, because it allows mothers as well as fathers. Hence, we prefer the third clause.

Now we need to specialize this clause further, to rule out the cases in which x is the father of some z, but z is not a parent of y. Adding the single literal *Parent*(z, y) gives

$$Father(x, z) \land Parent(z, y) \Rightarrow Grandfather(x, y),$$

which correctly classifies all the examples. FOIL will find and choose this literal, thereby solving the learning task. In general, the solution is a set of Horn clauses, each of which implies the target predicate. For example, if we didn't have the *Parent* predicate in our vocabulary, then the solution might be

$$Father(x,z) Father(z,y) \Rightarrow Grandfather(x,y)$$
$$Father(x,z) Mother(z,y) \Rightarrow Grandfather(x,y).$$

Note that each of these clauses covers some of the positive examples, that together they cover all the positive examples, and that NEW-CLAUSE is designed in such a way that no clause will incorrectly cover a negative example. In general FOIL will have to search through many unsuccessful clauses before finding a correct solution.

This example is a very simple illustration of how FOIL operates. A sketch of the complete algorithm is shown in Figure 20.12. Essentially, the algorithm repeatedly constructs a clause, literal by literal, until it agrees with some subset of the positive examples and none of the negative examples. Then the positive examples covered by the clause are removed from the training set, and the process continues until no positive examples remain. The two main subroutines to be explained are NEW-LITERALS, which constructs all possible new literals to add to the clause, and CHOOSE-LITERAL, which selects a literal to add.

```
function FOIL(examples, target) returns a set of Horn clauses
   inputs: examples, set of examples
           target, a literal for the goal predicate
   local variables: clauses, set of clauses, initially empty

   while examples contains positive examples do
      clause ← NEW-CLAUSE(examples, target)
      remove positive examples covered by clause from examples
      add clause to clauses
   return clauses
```

```
function NEW-CLAUSE(examples, target) returns a Horn clause
   local variables: clause, a clause with target as head and an empty body
                    l, a literal to be added to the clause
                    extended_examples, a set of examples with values for new variables

   extended_examples ← examples
   while extended_examples contains negative examples do
      l ← CHOOSE-LITERAL(NEW-LITERALS(clause), extended_examples)
      append l to the body of clause
      extended_examples ← set of examples created by applying EXTEND-EXAMPLE
         to each example in extended_examples
   return clause
```

```
function EXTEND-EXAMPLE(example, literal) returns a set of examples
   if example satisfies literal
      then return the set of examples created by extending example with
         each possible constant value for each new variable in literal
   else return the empty set
```

Figure 20.12 Sketch of the FOIL algorithm for learning sets of first-order Horn clauses from examples. NEW-LITERALS and CHOOSE-LITERAL are explained in the text.

NEW-LITERALS takes a clause and constructs all possible “useful” literals that could be added to the clause. Let us use as an example the clause

$$Father(x,\ z)\ \Rightarrow\ Grandfather(x, y).$$

There are three kinds of literals that can be added:

1. *Literals using predicates:* the literal can be negated or unnegated, any existing predicate (including the goal predicate) can be used, and the arguments must all be variables. Any variable can be used for any argument of the predicate, with one restriction: each literal must include *at least one* variable from an earlier literal or from the head of the clause. Literals such as *Mother*(z, u), *Married*(z, z), $\neg$*Male*(y), and *Grandfather*(v, x) are allowed, whereas *Married*(u, v) is not. Notice that the use of the predicate from the head of the clause allows FOIL to learn *recursive* definitions.
2. *Equality and inequality literals:* these relate variables already appearing in the clause. For example, we might add $z \neq x$. These literals can also include user-specified constants. For learning arithmetic we might use 0 and 1, and for learning list functions we might use the empty list [].
3. *Arithmetic comparisons:* when dealing with functions of continuous variables, literals such as $x > y$ and $y \leq z$ can be added. As in decision-tree learning, a constant threshold value can be chosen to maximize the discriminatory power of the test.

The resulting branching factor in this search space is very large (see Exercise 20.6), but FOIL can also use type information to reduce it. For example, if the domain included numbers as well as people, type restrictions would prevent NEW-LITERALS from generating literals such as *Parent*(x, n), where x is a person and n is a number.

CHOOSE-LITERAL uses a heuristic somewhat similar to information gain (see page 680) to decide which literal to add. The exact details are not important here, and a number of different variations have been tried. One interesting additional feature of FOIL is the use of Ockham's razor to eliminate some hypotheses. If a clause becomes longer (according to some metric) than the total length of the positive examples that the clause explains, that clause is not considered as a potential hypothesis. This technique provides a way to avoid overcomplex clauses that fit noise in the data.

FOIL and its relatives have been used to learn a wide variety of definitions. One of the most impressive demonstrations (Quinlan and Cameron-Jones, 1993) involved solving along sequence of exercises on list-processing functions from Bratko's (1986) Prolog textbook. In each case, the program was able to learn a correct definition of the function from a small set of examples, using the previously learned functions as background knowledge.

20.5.3 Inductive learning with inverse deduction

The second major approach to ILP involves inverting the normal deductive proof process. **Inverse resolution** is based on the observation that if the example *Classifications* follow from *Background* ∧ *Hypothesis* ∧ *Descriptions*, then one must be able to prove this fact by resolution (because resolution is complete). If we can "run the proof backward," then we can find a *Hypothesis* such that the proof goes through. The key, then, is to find a way to invert the resolution process.

We will show a backward proof process for inverse resolution that consists of individual backward steps. Recall that an ordinary resolution step takes two clauses C_1 and C_2 and resolves them to produce the **resolvent** C. An inverse resolution step takes a resolvent C and produces two clauses C_1 and C_2, such that C is the result of resolving C_1 and C_2. Alternatively, it may take a resolvent C and clause C_1 and produce a clause C_2 such that C is the result of resolving C_1 and C_2.

The early steps in an inverse resolution process are shown in Figure 20.13, where we focus on the positive example *Grandparent(George, Anne*). The process begins at the end of the proof (shown at the bottom of the figure). We take the resolvent C to be empty clause (i.e. a contradiction) and C_2 to be ¬*Grandparent(George;Anne)*, which is the negation of the goal example. The first inverse step takes C and C_2 and generates the clause *Grandparent*(*George*, *Anne*) for C_1. The next step takes this clause as C and the clause *Parent (Elizabeth, Anne*) as C_2, and generates the clause

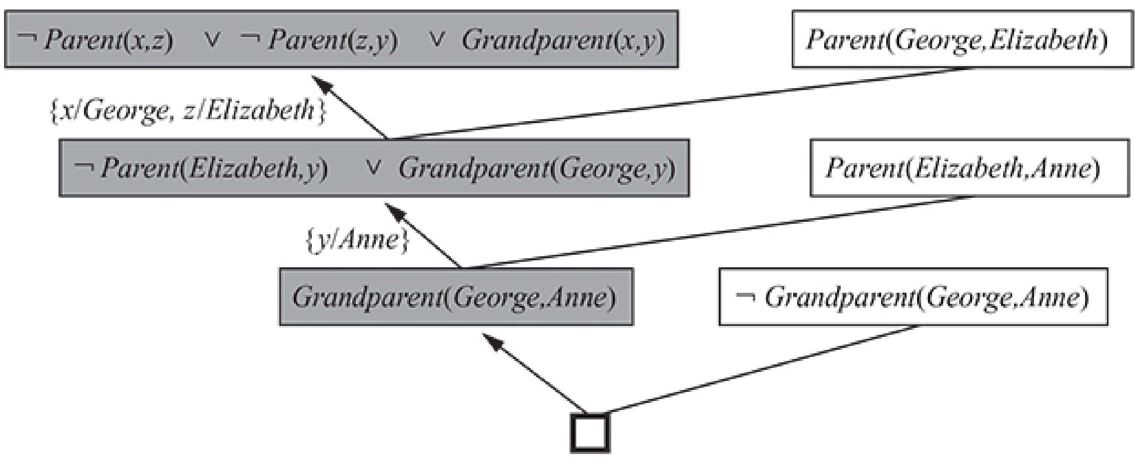


Figure 20.13 Early steps in an inverse resolution process. The shaded clauses are generated by inverse resolution steps from the clause to the right and the clause below. The unshaded clauses are from the *Descriptions* and *Classifications* (including negated *Classifications*).

$$\neg Parent(Elizabeth, y) \vee Grandparent(George, y)$$

as C_1. The final step treats this clause as the resolvent. With *Parent (George, Elizabeth*) as C_2, one possible clause C_1 is the hypothesis

$$Parent(x, z) \wedge Parent(z, y) \Rightarrow Grandparent(x, y).$$

Now we have a resolution proof that the hypothesis, descriptions, and background knowledge entail the classification *Grandparent* (*George, Anne*).

Clearly, inverse resolution involves a search. Each inverse resolution step is nondeterministic, because for any C, there can be many or even an infinite number of clauses C_1 and C_2 that resolve to C. For example, instead of choosing

$\neg Parent(Elizabeth,y) \vee Grandparent(George,y)$ for C_1 in the last step of Figure 20.13, the inverse resolution step might have chosen any of the following sentences:

$$\neg Parent(Elizabeth, Anne) \vee Grandparent(George, Anne).$$
$$\neg Parent(z, Anne) \vee Grandparent(George, Anne).$$
$$\neg Parent(z, y) \vee Grandparent(George, y).$$

(See Exercises 20.4 and 20.5.) Furthermore, the clauses that participate in each step can be chosen from the *Background* knowledge, from the example *Descriptions*, from the negated *Classifications*, or from hypothesized clauses that have already been generated in the inverse resolution tree. The large number of possibilities means a large branching factor (and therefore an inefficient search) without additional controls. A number of approaches to taming the search have been tried in implemented ILP systems:

1. Redundant choices can be eliminated—for example, by generating only the most specific hypotheses possible and by requiring that all the hypothesized clauses be consistent with each other, and with the observations. This last criterion would rule out the clause $\neg Parent(z,y) \vee Grandparent(George,y)$, listed before.
2. The proof strategy can be restricted. For example, we saw in Chapter 9 that **linear resolution** is a complete, restricted strategy. Linear resolution produces proof trees that have a linear branching structure—the whole tree follows one line, with only single clauses branching off that line (as in Figure 20.13).
3. The representation language can be restricted, for example by eliminating function symbols or by allowing only Horn clauses. For instance, PROGOL operates with Horn clauses using **inverse entailment**. The idea is to change the entailment constraint

$$Background \wedge Hypothesis \wedge Descriptions \models Classifications$$

to the logically equivalent form

$$Background \wedge Descriptions \wedge \neg Classifications \models \neg Hypothesis.$$

From this, one can use a process similar to the normal Prolog Horn-clause deduction, with negation-as-failure to derive *Hypothesis*. Because it is restricted to Horn clauses, this is an incomplete method, but it can be more efficient than full resolution. It is also possible to apply complete inference with inverse entailment (Inoue, 2001).

4. Inference can be done with model checking rather than theorem proving. The PROGOL system (Muggleton, 1995) uses a form of model checking to limit the search. That is, like answer set programming, it generates possible values for logical variables, and checks for consistency.
5. Inference can be done with ground propositional clauses rather than in first-order logic. The LINUS system (Lavra**u**c and D**u**zeroski, 1994) works by translating first-order theories into propositional logic, solving them with a propositional learning system, and then translating back. Working with propositional formulas can be more efficient on some problems, as we saw with SATPLAN in Chapter 10.

20.5.4 Making discoveries with inductive logic programming

An inverse resolution procedure that inverts a complete resolution strategy is, in principle, a complete algorithm for learning first-order theories. That is, if some unknown *Hypothesis* generates a set of examples, then an inverse resolution procedure can generate *Hypothesis* from the examples. This observation suggests an interesting possibility: Suppose that the available examples include a variety of trajectories of falling bodies. Would an inverse resolution program be theoretically capable of inferring the law of gravity? The answer is clearly yes, because the law of gravity allows one to explain the examples, given suitable background mathematics. Similarly, one can imagine that electromagnetism, quantum mechanics, and the theory of relativity are also within the scope of ILP programs. Of course, they are also within the scope of a monkey with a typewriter; we still need better heuristics and new ways to structure the search space.

One thing that inverse resolution systems *will* do for you is invent new predicates. This ability is often seen as somewhat magical, because computers are often thought of as “merely working with what they are given.” In fact, new predicates fall directly out of the inverse resolution step. The simplest case arises in hypothesizing two new clauses C_1 and C_2, given a clause C. The resolution of C_1 and C_2 eliminates a literal that the two clauses share; hence, it is quite possible that the eliminated literal contained a predicate that does not appear in C. Thus, when working backward, one possibility is to generate a new predicate from which to reconstruct the missing literal.

Figure 20.14 shows an example in which the new predicate P is generated in the process of learning a definition for *Ancestor*. Once generated, P can be used in later inverse resolution steps. For example, a later step might hypothesize that $Mother(x, y) \Rightarrow P(x, y)$. Thus, the new predicate P has its meaning constrained by the generation of hypotheses that involve it. Another example might lead to the constraint $Father(x, y) \Rightarrow P(x, y)$. In other words, the predicate P is what we usually think of as the *Parent* relationship. As we mentioned earlier, the invention of new predicates can significantly reduce the size of the definition of the goal predicate. Hence, by including the ability to invent new predicates, inverse resolution systems can often solve learning problems that are infeasible with other techniques.

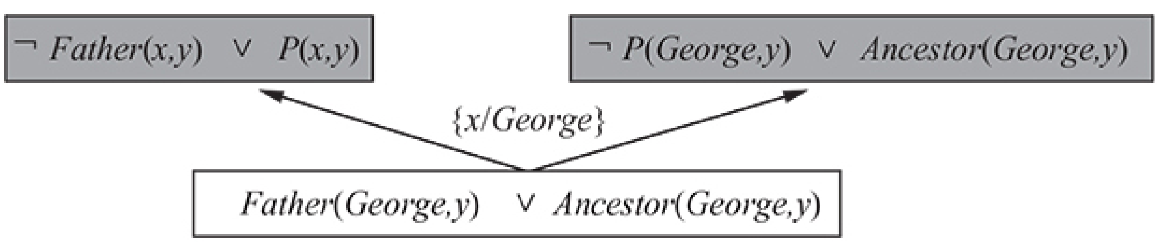


Figure 20.14 An inverse resolution step that generates a new predicate *P*.

Some of the deepest revolutions in science come from the invention of new predicates and functions—for example, Galileo's invention of acceleration or Joule's invention of thermal energy. Once these terms are available, the discovery of new laws becomes (relatively) easy. The difficult part lies in realizing that some new entity, with a specific relationship to existing entities, will allow an entire body of observations to be explained with a much simpler and more elegant theory than previously existed.

As yet, ILP systems have not made discoveries on the level of Galileo or Joule, but their discoveries have been deemed publishable in the scientific literature. For example, in the *Journal of Molecular Biology*, Turcotte *et al*. (2001) describe the automated discovery of rules for protein folding by the ILP program PROGOL. Many of the rules discovered by PROGOL could have been derived from known principles, but most had not been previously published as part of a standard biological database. (See Figure 20.10 for an example.). In related work, Srinivasan *et al*. (1994) dealt with the problem of discovering molecular-structure-based rules for the mutagenicity of nitroaromatic compounds. These compounds are found in automobile exhaust fumes. For 80% of the compounds in a standard database, it is possible to identify four important features, and linear regression on these features outperforms ILP. For the remaining 20%, the features alone are not predictive, and ILP identifies relationships that allow it to outperform linear regression, neural nets, and decision trees. Most impressively, King *et al*. (2009) endowed a robot with the ability to perform molecular biology experiments and extended ILP techniques to include experiment design, thereby creating an autonomous scientist that actually discovered new knowledge about the functional genomics of yeast. For all these examples it appears that the ability both to represent relations and to use background knowledge contribute to ILP's high performance. The fact that the rules found by ILP can be interpreted by humans contributes to the acceptance of these techniques in biology journals rather than just computer science journals.

ILP has made contributions to other sciences besides biology. One of the most important is natural language processing, where ILP has been used to extract complex relational information from text.

Summary

This chapter has investigated various ways in which prior knowledge can help an agent to learn from new experiences. Because much prior knowledge is expressed in terms of relational models rather than attribute-based models, we have also covered systems that allow learning of relational models. The important points are:

- The use of prior knowledge in learning leads to a picture of **cumulative learning**, in which learning agents improve their learning ability as they acquire more knowledge.
- Prior knowledge helps learning by eliminating otherwise consistent hypotheses and by “filling in” the explanation of examples, thereby allowing for shorter hypotheses. These contributions often result in faster learning from fewer examples.
- Understanding the different logical roles played by prior knowledge, as expressed by **entailment constraints**, helps to define a variety of learning techniques.
- **Explanation-based learning** (EBL) extracts general rules from single examples by *explaining* the examples and generalizing the explanation. It provides a deductive method for turning first-principles knowledge into useful, efficient, special-purpose expertise.
- **Relevance-based learning** (RBL) uses prior knowledge in the form of determinations to identify the relevant attributes, thereby generating a reduced hypothesis space and speeding up learning. RBL also allows deductive generalizations from single examples.
- **Knowledge-based inductive learning** (KBIL) finds inductive hypotheses that explain sets of observations with the help of

background knowledge.

- **Inductive logic programming** (ILP) techniques perform KBIL on knowledge that is expressed in first-order logic. ILP methods can learn relational knowledge that is not expressible in attribute-based systems.
- ILP can be done with a top-down approach of refining a very general rule or through a bottom-up approach of inverting the deductive process.
- ILP methods naturally generate new predicates with which concise new theories can be expressed and show promise as general-purpose scientific theory formation systems.

Bibliographical and Historical Notes

Although the use of prior knowledge in learning would seem to be a natural topic for philosophers of science, little formal work was done until quite recently. *Fact, Fiction, and Forecast*, by the philosopher Nelson Goodman (1954), refuted the earlier supposition that induction was simply a matter of seeing enough examples of some universally quantified proposition and then adopting it as a hypothesis. Consider, for example, the hypothesis "All emeralds are grue," where *grue* means "green if observed before time t, but blue if observed thereafter." At any time up to t, we might have observed millions of instances confirming the rule that emeralds are grue, and no disconfirming instances, and yet we are unwilling to adopt the rule. This can be explained only by appeal to the role of relevant prior knowledge in the induction process. Goodman proposes a variety of different kinds of prior knowledge that might be useful, including a version of determinations called **overhypotheses**. Unfortunately, Goodman's ideas were never pursued in machine learning.

The **current-best-hypothesis** approach is an old idea in philosophy (Mill, 1843). Early work in cognitive psychology also suggested that it is a natural form of concept learning in humans (Bruner *et al.*, 1957). In AI, the approach is most closely associated with the work of Patrick Winston, whose Ph.D. thesis (Winston, 1970) addressed the problem of learning descriptions of complex objects. The **version space** method (Mitchell, 1977, 1982) takes a different approach, maintaining the set of *all* consistent hypotheses and eliminating those found to be inconsistent with new examples. The approach was used in the Meta-DENDRAL expert system for chemistry (Buchanan and Mitchell, 1978), and later in Mitchell's (1983)

LEX system, which learns to solve calculus problems. A third influential thread was formed by the work of Michalski and colleagues on the AQ series of algorithms, which learned sets of logical rules (Michalski, 1969; Michalski *et al.*, 1986).

EBL had its roots in the techniques used by the STRIPS planner (Fikes *et al.*, 1972). When a plan was constructed, a generalized version of it was saved in a plan library and used in later planning as a **macro-operator**. Similar ideas appeared in Anderson's ACT* architecture, under the heading of **knowledge compilation** (Anderson, 1983), and in the SOAR architecture, as **chunking** (Laird *et al.*, 1986). **Schema acquisition** (DeJong, 1981), **analytical generalization** (Mitchell, 1982), and **constraint-based generalization** (Minton, 1984) were immediate precursors of the rapid growth of interest in EBL stimulated by the papers of Mitchell *et al*. (1986) and DeJong and Mooney (1986). Hirsh (1987) introduced the EBL algorithm described in the text, showing how it could be incorporated directly into a logic programming system. Van Harmelen and Bundy (1988) explain EBL as a variant of the **partial evaluation** method used in program analysis systems (Jones *et al.*, 1993).

Initial enthusiasm for EBL was tempered by Minton's finding (1988) that, without extensive extra work, EBL could easily slow down a program significantly. Formal probabilistic analysis of the expected payoff of EBL can be found in Greiner (1989) and Subramanian and Feldman (1990). An excellent survey of early work on EBL appears in Dietterich (1990).

Instead of using examples as foci for generalization, one can use them directly to solve new problems, in a process known as **analogical reasoning**. This form of reasoning ranges from a form of plausible reasoning based on degree of similarity (Gentner, 1983), through a form of deductive inference based on determinations but requiring the participation of the example (Davies and Russell, 1987), to a form of "lazy" EBL that tailors the direction of generalization of the old example to fit the needs of the new problem. This latter form of analogical reasoning is found most commonly in **case-based reasoning** (Kolodner, 1993) and **derivational analogy** (Veloso and Carbonell, 1993).

Relevance information in the form of functional dependencies was first developed in the database community, where it is used to structure large sets of attributes into manageable subsets. Functional dependencies were used for analogical reasoning by Carbonell and Collins (1973) and rediscovered and given a full logical analysis by Davies and Russell (Davies, 1985; Davies and Russell, 1987). Their role as prior knowledge in inductive learning was explored by Russell and Grosof (1987). The equivalence of determinations to a restricted-vocabulary hypothesis space was proved in Russell (1988). Learning algorithms for determinations and the improved performance obtained by RBDTL were first shown in the FOCUS algorithm, due to Almuallim and Dietterich (1991). Tadepalli (1993) describes a very ingenious algorithm for learning with determinations that shows large improvements in learning speed.

The idea that inductive learning can be performed by inverse deduction can be traced to W. S. Jevons (1874), who wrote, "The study both of Formal Logic and of the Theory of Probabilities has led me to adopt the opinion that there is no such thing as a distinct method of induction as contrasted with deduction, but that induction is simply an inverse employment of deduction." Computational investigations began with the remarkable Ph.D. thesis by Gordon Plotkin (1971) at Edinburgh. Although Plotkin developed many of the theorems and methods that are in current use in ILP, he was discouraged by some undecidability results for certain subproblems in induction. MIS (Shapiro, 1981) reintroduced the problem of learning logic programs, but was seen mainly as a contribution to the theory of automated debugging. Work on rule induction, such as the ID3 (Quinlan, 1986) and CN2 (Clark and Niblett, 1989) systems, led to FOIL (Quinlan, 1990), which for the first time allowed practical induction of relational rules. The field of relational learning was reinvigorated by Muggleton and Buntine (1988), whose CIGOL program incorporated a slightly incomplete version of inverse resolution and was capable of generating new predicates. The inverse resolution method also appears in (Russell, 1986), with a simple algorithm given in a footnote. The next major system was GOLEM (Muggleton and Feng, 1990), which uses a covering algorithm based on Plotkin's concept of relative least general generalization. ITOU (Rouveirol and Puget, 1989) and CLINT (De Raedt, 1992) were other systems of that era. More recently, PROGOL (Muggleton, 1995) has taken a hybrid (top-down and bottom-up) approach to inverse entailment and has been applied to a number of practical problems, particularly in biology and natural language processing. Muggleton (2000) describes an extension of PROGOL to handle uncertainty in the form of stochastic logic programs.

A formal analysis of ILP methods appears in Muggleton (1991), a large collection of papers in Muggleton (1992), and a collection of techniques and applications in the book by Lavra**u**c and D**u**zeroski (1994). Page and Srinivasan (2002) give a more recent overview of the field's history and challenges for the future. Early complexity results by Haussler (1989) suggested that learning first-order sentences was intractible. However, with better understanding of the importance of syntactic restrictions on clauses, positive results have been obtained even for clauses with recursion (D**u**zeroski *et al.,* 1992). Learnability results for ILP are surveyed by Kietz and D**u**zeroski (1994) and Cohen and Page (1995).

Although ILP now seems to be the dominant approach to constructive induction, it has not been the only approach taken. So-called **discovery systems** aim to model the process of scientific discovery of new concepts, usually by a direct search in the space of concept definitions. Doug Lenat's Automated Mathematician, or AM (Davis and Lenat, 1982), used discovery heuristics expressed as expert system rules to guide its search for concepts and conjectures in elementary number theory. Unlike most systems designed for mathematical reasoning, AM lacked a concept of proof and could only make conjectures. It rediscovered Goldbach's conjecture and the Unique Prime Factorization theorem. AM's architecture was generalized in the EURISKO system (Lenat, 1983) by adding a mechanism capable of rewriting the system's own discovery heuristics. EURISKO was applied in a number of areas other than mathematical discovery, although with less success than AM. The methodology of AM and EURISKO has been controversial (Ritchie and Hanna, 1984; Lenat and Brown, 1984).

Another class of discovery systems aims to operate with real scientific data to find new laws. The systems DALTON, GLAUBER, and STAHL (Langley *et al.*, 1987) are rule-based systems that look for quantitative relationships in experimental data from physical systems; in each case, the system has been able to recapitulate a well-known discovery from the history of science. Discovery systems based on probabilistic techniques—especially clustering algorithms that discover new categories—are discussed in Chapter 21.

[1] The terms "false positive" and "false negative" are used in medicine to describe erroneous results from lab tests. A result is a false positive if it indicates that the patient has the disease when in fact no disease is present.

[2] We assume for the sake of simplicity that a person speaks only one language. Clearly, the rule would have to be amended for countries such as Switzerland and India.

[3] It might be appropriate at this point for the reader to refer to Chapter 7 for some of the underlying concepts, including Horn clauses, conjunctive normal form, unification, and resolution.

CHAPTER 21

LEARNING PROBABILISTIC MODELS

In which we view learning as a form of uncertain reasoning from observations, and devise models to represent the uncertain world.

Chapter 12 pointed out the prevalence of uncertainty in real environments. Agents can handle uncertainty by using the methods of probability and decision theory, but first they must learn their probabilistic theories of the world from experience. This chapter explains how they can do that, by formulating the learning task itself as a process of probabilistic inference (Section 21.1). We will see that a Bayesian view of learning is extremely powerful, providing general solutions to the problems of noise, overfitting, and optimal prediction. It also takes into account the fact that a less-than-omniscient agent can never be certain about which theory of the world is correct, yet must still make decisions by using some theory of the world.

We describe methods for learning probability models—primarily Bayesian networks—in Sections 21.2 and 21.3. Some of the material in this chapter is fairly mathematical, although the general lessons can be understood without plunging into the details. It may benefit the reader to review Chapters 12 and 13 and peek at Appendix A.

21.1 Statistical Learning

The key concepts in this chapter, just as in Chapter 19, are **data** and **hypotheses**. Here, the data are **evidence**—that is, instantiations of some or all of the random variables describing the domain. The hypotheses in this chapter are probabilistic theories of how the domain works, including logical theories as a special case.

Consider a simple example. Our favorite surprise candy comes in two flavors: cherry (yum) and lime (ugh). The manufacturer has a peculiar sense of humor and wraps each piece of candy in the same opaque wrapper, regardless of flavor. The candy is sold in very large bags, of which there are known to be five kinds—again, indistinguishable from the outside:

h_1: 100% cherry,
h_2: 75% cherry + 25% lime,
h_3: 50% cherry + 50% lime,
h_4: 25% cherry + 75% lime,
h_5: 100% lime.

Given a new bag of candy, the random variable H (for *hypothesis*) denotes the type of the bag, with possible values h_1 through h_5. H is not directly observable, of course. As the pieces of candy are opened and inspected, data are revealed—D_1, D_2, ..., D_N, where each D_i is a random variable with possible values *cherry* and *lime*. The basic task faced by the agent is to predict the flavor of the next piece of candy.[1] Despite its apparent triviality, this scenario serves to introduce many of the major issues. The agent really does need to infer a theory of its world, albeit a very simple one.

Bayesian learning simply calculates the probability of each hypothesis, given the data, and makes predictions on that basis. That is, the predictions are made by using *all* the hypotheses, weighted by their probabilities, rather than by using just a single "best" hypothesis. In this way, learning is reduced to probabilistic inference.

Let **D** represent all the data, with observed value **d**. The key quantities in the Bayesian approach are the **hypothesis prior**, $P(h_i)$, and the **likelihood** of the data under each hypothesis, $P(\mathbf{d} \mid h_i)$. The probability of each hypothesis is obtained by Bayes' rule:

$$P(h_i|\mathbf{d}) = \alpha\, P(\mathbf{d}|h_i)P(h_i). \qquad (21.1)$$

Now, suppose we want to make a prediction about an unknown quantity X. Then we have

$$P(X|\mathbf{d}) = \sum_i \mathbf{P}(X|h_i)P(h_i|\mathbf{d}). \qquad (21.2)$$

where each hypothesis determines a probability distribution over X. This equation shows that predictions are weighted averages over the predictions of the individual hypotheses, where the weight $P(h_i \mid \mathbf{d})$ is proportional to the prior probability of h_i and its degree of fit, according to Equation (21.1). The hypotheses themselves are essentially "intermediaries" between the raw data and the predictions.

For our candy example, we will assume for the time being that the prior distribution over $h_1,...,h_5$ is given by $\langle 0.1,0.2,0.4,0.2,0.1 \rangle$, as advertised by the manufacturer. The likelihood of the data is calculated under the assumption that the observations are **i.i.d.** (see page 683), so that

$$P(\mathbf{d}|h_i) = \prod_j P(d_j|h_i). \qquad (21.3)$$

For example, suppose the bag is really an all-lime bag (h_5) and the first 10 candies are all lime; then $P(\mathbf{d} \mid h_3)$ is 0.5^{10}, because half the candies in an h_3 bag are lime.[2] Figure 21.1(a) shows how the posterior probabilities of the five hypotheses change as the sequence of 10 lime candies is observed. Notice that the probabilities start out at their prior values, so h_3 is initially the most likely choice and remains so after 1 lime candy is unwrapped. After 2 lime candies are unwrapped, h_4 is most likely; after 3 or more, h_5 (the dreaded all-lime bag) is the most likely. After 10 in a row, we are fairly certain of our fate. Figure 21.1(b) shows the predicted probability that the next candy is lime, based on Equation (21.2). As we would expect, it increases monotonically toward 1.

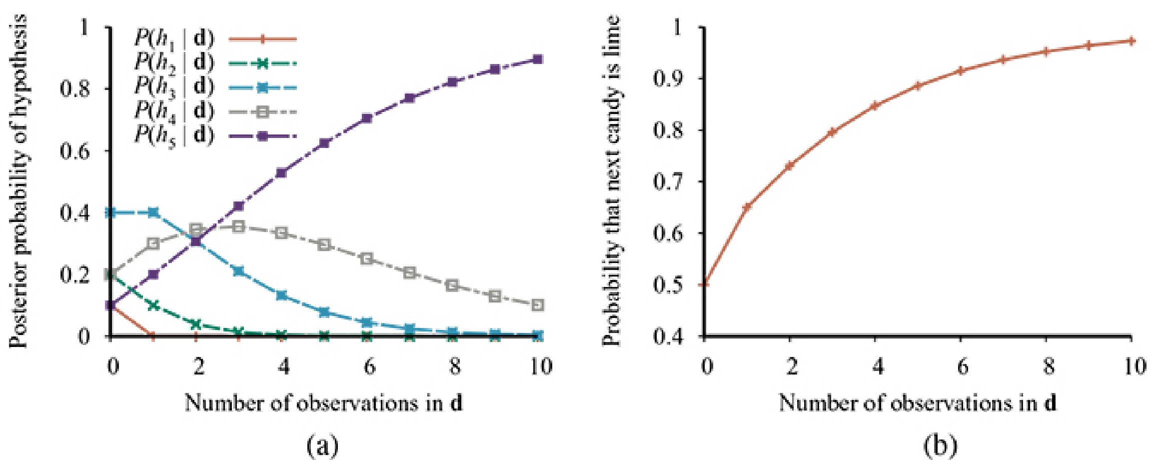


Figure 21.1 (a) Posterior probabilities $P(h_i \mid d_1,\ldots,d_N)$ from Equation (21.1). The number of observations N ranges from 1 to 10, and each observation is of a lime candy. (b) Bayesian prediction $P(D_{N+1} = lime \mid d_1,\ldots,d_N)$ from Equation (21.2).

The example shows that *the Bayesian prediction eventually agrees with the true hypothesis*. This is characteristic of Bayesian learning. For any fixed prior that does not rule out the true hypothesis, the posterior probability of any false hypothesis will, under certain technical conditions, eventually vanish. This happens simply because the probability of generating "uncharacteristic" data indefinitely is vanishingly small. (This point is analogous to one made in the discussion of PAC learning in Chapter 19.) More important, the Bayesian prediction is *optimal*, whether the data set is small or large. Given the hypothesis prior, any other prediction is expected to be correct less often.

The optimality of Bayesian learning comes at a price, of course. For real learning problems, the hypothesis space is usually very large or infinite, as we saw in Chapter 19. In some cases, the summation in Equation (21.2) (or integration, in the continuous case) can be carried out tractably, but in most cases we must resort to approximate or simplified methods.

A very common approximation—one that is usually adopted in science—is to make predictions based on a single *most probable* hypothesis—that is, an h_i that maximizes $P(h_i \mid \mathbf{d})$. This is often called a **maximum a posteriori** or MAP (pronounced "em-ay-pee") hypothesis. Predictions made according to an MAP hypothesis h_{MAP} are approximately Bayesian to the extent that $\mathbf{P}(X \mid \mathbf{d}) \approx \mathbf{P}(X \mid h_{\mathrm{MAP}})$. In our candy example, $h_{\mathrm{MAP}} = h_5$ after three lime candies in a row, so the MAP learner then predicts that the fourth candy is lime with probability 1.0—a much more dangerous prediction than the Bayesian prediction of 0.8 shown in Figure 21.1(b). As more data arrive, the MAP and Bayesian predictions become closer, because the competitors to the MAP hypothesis become less and less probable.

Although this example doesn't show it, finding MAP hypotheses is often much easier than Bayesian learning, because it requires solving an optimization problem instead of a large summation (or integration) problem.

In both Bayesian learning and MAP learning, the hypothesis prior $P(h_i)$ plays an important role. We saw in Chapter 19 that **overfitting** can occur when the hypothesis space is too expressive, that is, when it contains many hypotheses that fit the data set well. Bayesian and MAP learning methods use the prior to *penalize complexity*. Typically, more complex hypotheses have a lower prior probability—in part because there so many of them. On the other hand, more complex hypotheses have a greater capacity to fit the data. (In the extreme case, a lookup table can reproduce the data exactly.) Hence, the hypothesis prior embodies a tradeoff between the complexity of a hypothesis and its degree of fit to the data.

We can see the effect of this tradeoff most clearly in the logical case, where H contains only *deterministic* hypotheses (such as h_1, which says that every candy is cherry). In that case, $P(\mathbf{d} \mid h_i)$ is 1 if h_i is consistent and 0 otherwise. Looking at Equation (21.1), we see that h_{MAP} will then be the *simplest logical theory that is consistent with the data*. Therefore, maximum a posteriori learning provides a natural embodiment of Ockham's razor.

Another insight into the tradeoff between complexity and degree of fit is obtained by taking the logarithm of Equation (21.1). Choosing h_{MAP} to maximize $P(\mathbf{d} \mid h_i)P(h_i)$ is equivalent to minimizing

$$-\log_2 P(\mathbf{d}|h_i) - \log_2 P(h_i).$$

Using the connection between information encoding and probability that we introduced in Section 19.3.3, we see that the $-\log_2 P(h_i)$ term equals the number of bits required to specify the hypothesis h_i. Furthermore, $-\log_2 P(\mathbf{d} \mid h_i)$ is the additional number of bits required to specify the data, given the hypothesis. (To see this, consider that no bits are required if the hypothesis predicts the data exactly—as with h_5 and the string of lime candies—and $\log_2 1 = 0$.) Hence, MAP learning is choosing the hypothesis that provides maximum *compression* of the data. The same task is addressed more directly by the **minimum description length**, or MDL, learning method. Whereas MAP learning expresses simplicity by assigning higher probabilities to simpler hypotheses, MDL expresses it directly by counting the bits in a binary encoding of the hypotheses and data.

A final simplification is provided by assuming a **uniform** prior over the space of hypotheses. In that case, MAP learning reduces to choosing an h_i that maximizes $P(\mathbf{d} \mid h_i)$. This is called a **maximum-likelihood** hypothesis, h_{ML}. Maximum-likelihood learning is very common in statistics, a discipline in which many researchers distrust the subjective nature of hypothesis priors. It is a reasonable approach when there is no reason to prefer one hypothesis over another *a priori*—for example, when all hypotheses are equally complex.

When the data set is large, the prior distribution over hypotheses is less important—the evidence from the data is strong enough to swamp the prior distribution over hypotheses. That means maximum likelihood learning is a good approximation to Bayesian and MAP learning with large data sets, but it has problems (as we shall see) with small data sets.

21.2 Learning with Complete Data

The general task of learning a probability model, given data that are assumed to be generated from that model, is called **density estimation**. (The term applied originally to probability density functions for continuous variables, but it is used now for discrete distributions too.) Density estimation is a form of unsupervised learning. This section covers the simplest case, where we have **complete data**. Data are complete when each data point contains values for every variable in the probability model being learned. We focus on **parameter learning**—finding the numerical parameters for a probability model whose structure is fixed. For example, we might be interested in learning the conditional probabilities in a Bayesian network with a given structure. We will also look briefly at the problem of learning structure and at nonparametric density estimation.

21.2.1 Maximum-likelihood parameter learning: Discrete models

Suppose we buy a bag of lime and cherry candy from a new manufacturer whose flavor proportions are completely unknown; the fraction of cherry could be anywhere between 0 and 1. In that case, we have a continuum of hypotheses. The **parameter** in this case, which we call θ, is the proportion of cherry candies, and the hypothesis is h_θ. (The proportion of lime candies is just $1 - \theta$.) If we assume that all proportions are equally likely *a priori*, then a maximum-likelihood approach is reasonable. If we model the situation with a Bayesian network, we need just one random variable, *Flavor* (the flavor of a randomly chosen candy from the bag). It has values *cherry* and *lime*, where the probability of *cherry* is θ (see Figure 21.2(a)). Now suppose we unwrap N candies, of which c are cherry and $\ell = N - c$ are lime. According to Equation (21.3), the likelihood of this particular data set is

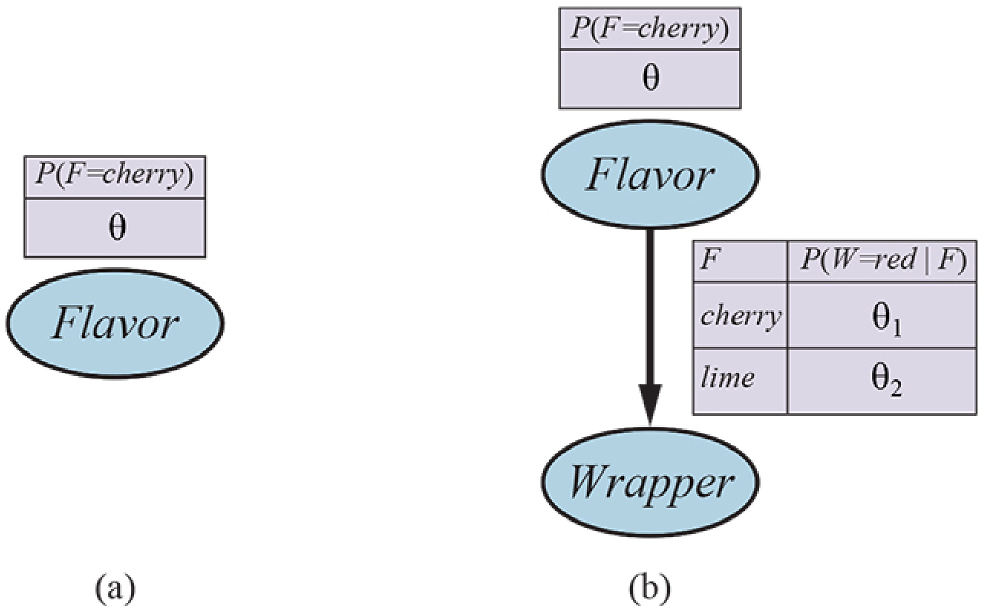


Figure 21.2 (a) Bayesian network model for the case of candies with an unknown proportion of cherry and lime. (b) Model for the case where the wrapper color depends (probabilistically) on the candy flavor.

$$P(\mathbf{d}\,\big|\,h_\theta) = \prod_{j=1}^{N} P(d_j|h_\theta) = \theta^c.\,(1-\theta)^\ell.$$

The maximum-likelihood hypothesis is given by the value of θ that maximizes this expression. Because the log function is monotonic, the same value is obtained by maximizing the **log likelihood** instead:

$$L(\mathbf{d}\,\big|\,h_\theta) = \log P(\mathbf{d}\,\big|\,h_\theta) = \sum_{j=1}^{N} \log P(d_j|h_\theta) = c\,\log\theta + \ell\log(1-\theta).$$

(By taking logarithms, we reduce the product to a sum over the data, which is usually easier to maximize.) To find the maximum-likelihood value of θ, we differentiate L with respect to θ and set the resulting expression to zero:

$$\frac{dL(\mathbf{d}|h_\theta)}{d_\theta} = \frac{c}{\theta} - \frac{\ell}{1-\theta} = 0 \qquad \Rightarrow \qquad \theta = \frac{c}{c+\ell} \qquad = \frac{c}{N}.$$

In English, then, the maximum-likelihood hypothesis h_{ML} asserts that the actual proportion of cherry candies in the bag is equal to the observed proportion in the candies unwrapped so far!

It appears that we have done a lot of work to discover the obvious. In fact, though, we have laid out one standard method for maximum-likelihood parameter learning, a method with broad applicability:

1. Write down an expression for the likelihood of the data as a function of the parameter(s).
2. Write down the derivative of the log likelihood with respect to each parameter.
3. Find the parameter values such that the derivatives are zero.

The trickiest step is usually the last. In our example, it was trivial, but we will see that in many cases we need to resort to iterative solution algorithms or other numerical optimization techniques, as described in Section 4.2. (We will need to verify that the Hessian matrix is negative-definite.) The example also illustrates a significant problem with maximum-likelihood learning in general: *when the data set is small enough that some events have not yet been observed—for instance, no cherry candies—the maximum-likelihood hypothesis assigns zero probability to those events.* Various tricks are used to avoid this problem, such as initializing the counts for each event to 1 instead of 0.

Let us look at another example. Suppose this new candy manufacturer wants to give a little hint to the consumer and uses candy wrappers colored red and green. The *Wrapper* for each candy is selected *probabilistically*, according to some unknown conditional distribution, depending on the flavor. The corresponding probability model is shown in Figure 21.2(b). Notice that it has three parameters: θ, θ_1, and θ_2. With these parameters, the likelihood of seeing, say, a cherry candy in a green wrapper can be obtained from the standard semantics for Bayesian networks (page 433):

$$\begin{aligned} &P(Flavor = cherry, Wrapper = green|h_{\theta,\theta_1,\theta_2}) \\ &\quad = P(Flavor = cherry|h_{\theta,\theta_1,\theta_2})P(Wrapper = green|Flavor = cherry, h_{\theta,\theta_1,\theta_2}) \\ &\quad = \theta \cdot (1 - \theta_1). \end{aligned}$$

Now we unwrap N candies, of which c are cherry and ℓ are lime. The wrapper counts are as follows: r_c of the cherry candies have red wrappers and g_c have green, while r_ℓ of the lime candies have red and g_ℓ have green. The likelihood of the data is given by

$$P(\mathbf{d}\big|h_{\theta,\theta_1,\theta_2}) = \theta^c(1-\theta)^\ell \cdot \theta_1^{r_c}(1-\theta_1)^{g_c} \cdot \theta_2^{r_\ell}(1-\theta_2)^{g_\ell}.$$

This looks pretty horrible, but taking logarithms helps:

$$L = [c \log \theta + \ell \log\ (1-\theta)] + [r_c \log \theta_1 + g_c \log(1-\theta_1)] + [r_\ell \log \theta_2 + g_\ell \log(1-\theta_2)].$$

The benefit of taking logs is clear: the log likelihood is the sum of three terms, each of which contains a single parameter. When we take derivatives with respect to each parameter and set them to zero, we get three independent equations, each containing just one parameter:

$$\frac{\partial L}{\partial \theta} = \frac{c}{\theta} - \frac{\ell}{1-\theta} = 0 \qquad \Rightarrow \qquad \theta = \frac{c}{c+\ell}$$
$$\frac{\partial L}{\partial \theta_1} = \frac{r_c}{\theta_1} - \frac{g_c}{1-\theta_1} = 0 \qquad \Rightarrow \qquad \theta_1 = \frac{r_c}{r_c+g_c}$$
$$\frac{\partial L}{\partial \theta_2} = \frac{r_\ell}{\theta_2} - \frac{g_\ell}{1-\theta_2} = 0 \qquad \Rightarrow \qquad \theta_2 = \frac{r_\ell}{r_\ell+g_\ell}.$$

The solution for θ is the same as before. The solution for θ_1, the probability that a cherry candy has a red wrapper, is the observed fraction of cherry candies with red wrappers, and similarly for θ_2.

These results are very comforting, and it is easy to see that they can be extended to any Bayesian network whose conditional probabilities are represented as tables. The most important point is that *with complete data, the maximum-likelihood parameter learning problem for a Bayesian network decomposes into separate learning problems, one for each parameter.* (See Exercise 21.NORX for the nontabulated case, where each parameter affects several conditional probabilities.) The second point is that the parameter values for a variable, given its parents, are just the observed frequencies of the variable values for each setting of the parent values. As before, we must be careful to avoid zeroes when the data set is small.

21.2.2 Naive Bayes models

Probably the most common Bayesian network model used in machine learning is the **naive Bayes** model first introduced on page 420. In this model, the "class" variable C (which is to be predicted) is the root and the "attribute" variables X_i are the leaves. The model is "naive" because it assumes that the attributes are conditionally independent of each other, given the class. (The model in Figure 21.2(b) is a naive Bayes model with class *Flavor* and just one attribute, *Wrapper.)* In the case of Boolean variables, the parameters are

$$\theta = P(C = true), \theta_{i1} = P(X_i = true|C = true), \theta_{i2} = P(X_i = true|C = false).$$

The maximum-likelihood parameter values are found in exactly the same way as in Figure 21.2(b). Once the model has been trained in this way, it can be used to classify new examples for which the class variable C is unobserved. With observed attribute values $x_1,...,x_n$, the probability of each class is given by

$$\mathbf{P}(C \,\Big|\, x_1, \ldots, x_n) = \alpha \mathbf{P}(C) \prod_i \mathbf{P}(x_i|C).$$

A deterministic prediction can be obtained by choosing the most likely class. Figure 21.3 shows the learning curve for this method when it is applied to the restaurant problem from Chapter 19. The method learns fairly well but not as well as decision tree learning; this is presumably because the true hypothesis—which is a decision tree—is not representable exactly using a naive Bayes model. Naive Bayes learning turns out to do surprisingly well in a wide range of applications; the boosted version (Exercise 21.BNBX) is one of the most effective general-purpose learning algorithms. Naive Bayes learning scales well to very large problems: with n Boolean attributes, there are just $2n + 1$ parameters, and *no search is required to find* h_{ML}, *the maximum-likelihood naive Bayes hypothesis.* Finally, naive Bayes learning systems deal well with noisy or missing data and can give probabilistic predictions when appropriate. Their primary drawback is the fact that the conditional independence assumption is seldom accurate; as noted on page 421, the assumption leads to overconfident probabilities that are often very close to 0 or 1, especially with large numbers of attributes.

Figure 21.3 The learning curve for naive Bayes learning applied to the restaurant problem from Chapter 19; the learning curve for decision tree learning is shown for comparison.

21.2.3 Generative and discriminative models

We can distinguish two kinds of machine learning models used for classifiers: generative and discriminative. A **generative model** models the probability distribution of each class. For example, the naive Bayes text classifier from Section 12.6.1 creates a separate model for each possible category of text—one for sports, one for weather, and so on. Each model includes the prior probability of the category—for example $P(Category = weather)$—as well as the conditional probability $\mathbf{P}(Inputs \mid Category = weather)$. From these we can compute the joint probability $\mathbf{P}(Inputs, Category = weather)$ and we can generate a random selection of words that is representative of texts in the *weather* category.

A **discriminative model** directly learns the decision boundary between classes. That is, it learns $\mathbf{P}(\mathit{Category} \mid \mathit{Inputs})$. Given example inputs, a discriminative model will come up with an output category, but you cannot use a discriminative model to, say, generate random words that are representative of a category. Logistic regression, decision trees, and support vector machines are all discriminative models.

Since discriminative models put all their emphasis on defining the decision boundary—that is, actually doing the classification task they were asked to do—they tend to perform better in the limit, with an arbitrary amount of training data. However, with limited data, in some cases a generative model performs better. (Ng and Jordan, 2002) compare the generative naive Bayes classifier to the discriminative logistic regression classifier on 15 (small) data sets, and find that with the maximum amount of data, the discriminative model does better on 9 out of 15 data sets, but with only a small amount of data, the generative model does better on 14 out of 15 data sets.

21.2.4 Maximum-likelihood parameter learning: Continuous models

Continuous probability models such as the **linear–Gaussian** model were shown on page 440. Because continuous variables are ubiquitous in real-world applications, it is important to know how to learn the parameters of continuous models from data. The principles for maximum-likelihood learning are identical in the continuous and discrete cases.

Let us begin with a very simple case: learning the parameters of a Gaussian density function on a single variable. That is, we assume the data are generated as follows:

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

The parameters of this model are the mean μ and the standard deviation σ. (Notice that the normalizing "constant" depends on σ, so we cannot ignore it.) Let the observed values be $x_1,..., x_N$. Then the log likelihood is

$$L = \sum_{j=1}^{N} \log \frac{1}{\alpha\sqrt{2\pi}} e^{-\frac{(x_j-\mu)^2}{2\sigma^2}} = N(-\log\sqrt{2\pi} - \log\alpha) - \sum_{j=1}^{N} \frac{(x_j-\mu)^2}{2\sigma^2}.$$

Setting the derivatives to zero as usual, we obtain

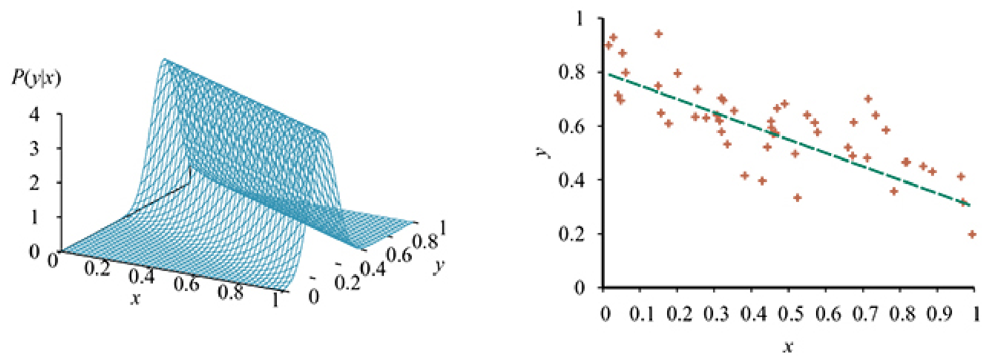
$$\begin{aligned} \frac{\partial L}{\partial \mu} &= \frac{1}{\sigma^2}\textstyle\sum_{j=1}^{N}(x_j-\mu) = 0 & \Rightarrow \quad & \mu = \frac{\sum_j x_j}{N} \\ \frac{\partial L}{\partial \sigma} &= -\frac{N}{\sigma} + \frac{1}{\sigma^3}\textstyle\sum_{j=1}^{N}(x_j-\mu)^2 = 0 & \Rightarrow \quad & \sigma = \sqrt{\frac{\sum_j (x_j-\mu)^2}{N}}. \end{aligned} \qquad (21.4)$$

That is, the maximum-likelihood value of the mean is the sample average and the maximum-likelihood value of the standard deviation is the square root of the sample variance. Again, these are comforting results that confirm "commonsense" practice.

Now consider a linear–Gaussian model with one continuous parent X and a continuous child Y. As explained on page 440, Y has a Gaussian distribution whose mean depends linearly on the value of X and whose standard deviation is fixed. To learn the conditional distribution $P(Y \mid X)$, we can maximize the conditional likelihood

$$P(y \mid x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(y-(\theta_1 x+\theta_2))^2}{2\sigma_2}}. \qquad (12.5)$$

Here, the parameters are θ_1, θ_2, and σ. The data are a collection of (x_j, y_j) pairs, as illustrated in Figure 21.4. Using the usual methods (Exercise 21.LINR), we can find the maximum-likelihood values of the parameters. The point here is different. If we consider just the parameters θ_1 and θ_2 that define the linear relationship between x and y, it becomes clear that maximizing the log likelihood with respect to these parameters is the same as *minimizing* the numerator $(y-(\theta_1 x+\theta_2))^2$ in the exponent of Equation (21.5). This is the L_2 loss, the squared error between the actual value y and the prediction $\theta_1 x + \theta_2$.



(a) (b)

Figure 21.4 (a) A linear–Gaussian model described as $y = \theta_1 x + \theta_2$ plus Gaussian noise with fixed variance. (b) A set of 50 data points generated from this model and the best-fit line.

This is the quantity minimized by the standard **linear regression** procedure described in Section 19.6. Now we can understand why: minimizing the sum of squared errors gives the maximum-likelihood straight-line model, *provided that the data are generated with Gaussian noise of fixed variance.*

21.2.5 Bayesian parameter learning

Maximum-likelihood learning gives rise to simple procedures, but it has serious deficiencies with small data sets. For example, after seeing one cherry candy, the maximum-likelihood hypothesis is that the bag is 100% cherry (i.e., $\theta = 1.0$). Unless one's hypothesis prior is that bags must be either all cherry or all lime, this is not a reasonable conclusion. It is more likely that the bag is a mixture of lime and cherry. The Bayesian approach to parameter learning starts with a hypothesis prior and updates the distribution as data arrive.

The candy example in Figure 21.2(a) has one parameter, θ: the probability that a randomly selected piece of candy is cherry-flavored. In the Bayesian view, θ is the (unknown) value of a random variable Θ that defines the hypothesis space; the hypothesis prior is the prior distribution over $\mathbf{P}(\Theta)$. Thus, $P(\Theta{=}\theta)$ is the prior probability that the bag has a fraction θ of cherry candies.

If the parameter θ can be any value between 0 and 1, then $\mathbf{P}(\Theta)$ is a continuous probability density function (see Section A.3). If we don't know anything about the possible values of θ we can use the uniform density function $P(\theta) = Uniform(\theta;0,1)$, which says all values are equally likely.

A more flexible family of probability density functions is known as the **beta distributions**. Each beta distribution is defined by two **hyperparameters**[3] a and b such that

$$Beta(\theta; a, b) \ = \alpha\, \theta^{a-1}(1-\theta)^{b-1}, \qquad (21.6)$$

for θ in the range [0, 1]. The normalization constant α, which makes the distribution integrate to 1, depends on a and b. Figure 21.5 shows what the distribution looks like for various values of a and b. The mean value of the beta distribution is $a/(a+b)$, so larger values of a suggest a belief that Θ is closer to 1 than to 0. Larger values of $a + b$ make the distribution more peaked, suggesting greater certainty about the value of Θ. It turns out that the uniform density function is the same as $Beta(1, 1)$: the mean is 1/2, and the distribution is flat.

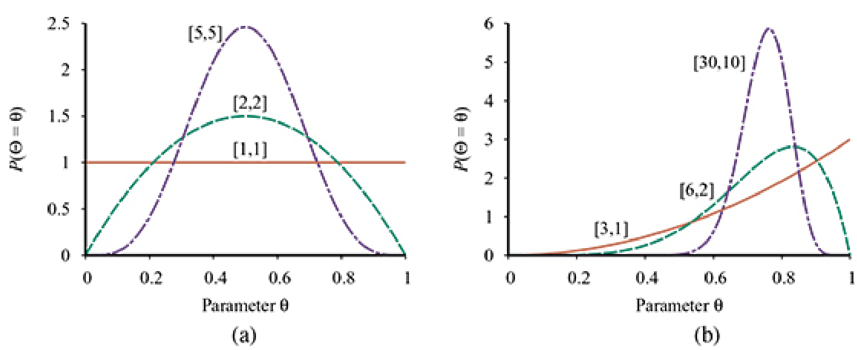


Figure 21.5 Examples of the $Beta(a, b)$ distribution for different values of (a, b).

Besides its flexibility, the beta family has another wonderful property: if Θ has a prior $Beta(a, b)$, then, after a data point is observed, the posterior distribution for Θ is also a beta distribution. In other words, *Beta* is closed under update. The beta family is called the **conjugate prior** for the family of distributions for a Boolean variable.[4] Let's see how this works. Suppose we observe a cherry candy; then we have

$$\begin{aligned} P(\theta|D_1 = cherry) &= \alpha\, P(D_1 = cherry|\theta)P(\theta) \\ &= \alpha' \theta \cdot Beta\,(\theta; a, b) = \alpha' \theta \cdot \theta^{a-1}(1-\theta)^{b-1} \\ &= \alpha' \theta^{a}(1-\theta)^{b-1} = \alpha' Beta(\theta; a+1, b). \end{aligned}$$

Thus, after seeing a cherry candy, we simply increment the *a* parameter to get the posterior; similarly, after seeing a lime candy, we increment the *b* parameter. Thus, we can view the *a* and *b* hyperparameters as **virtual counts**, in the sense that a prior *Beta*(*a*, *b*) behaves exactly as if we had started out with a uniform prior *Beta*(1, 1) and seen $a - 1$ actual cherry candies and $b - 1$ actual lime candies.

By examining a sequence of beta distributions for increasing values of *a* and *b*, keeping the proportions fixed, we can see vividly how the posterior distribution over the parameter Θ changes as data arrive. For example, suppose the actual bag of candy is 75% cherry. Figure 21.5(b) shows the sequence *Beta*(3, 1), *Beta*(6, 2), *Beta*(30, 10). Clearly, the distribution is converging to a narrow peak around the true value of Θ. For large data sets, then, Bayesian learning (at least in this case) converges to the same answer as maximum-likelihood learning.

Now let us consider a more complicated case. The network in Figure 21.2(b) has three parameters, θ, θ_1, and θ_2, where θ_1 is the probability of a red wrapper on a cherry candy and θ_2 is the probability of a red wrapper on a lime candy. The Bayesian hypothesis prior must cover all three parameters—that is, we need to specify $\mathbf{P}(\Theta, \Theta_1, \Theta_2)$. Usually, we assume **parameter independence**:

$$\mathbf{P}(\Theta, \Theta_1, \Theta_2) = \mathbf{P}(\Theta)\mathbf{P}(\Theta_1)\mathbf{P}(\Theta_2).$$

With this assumption, each parameter can have its own beta distribution that is updated separately as data arrive. Figure 21.6 shows how we can incorporate the hypothesis prior and any data into a Bayesian network, in which we have a node for each parameter variable.

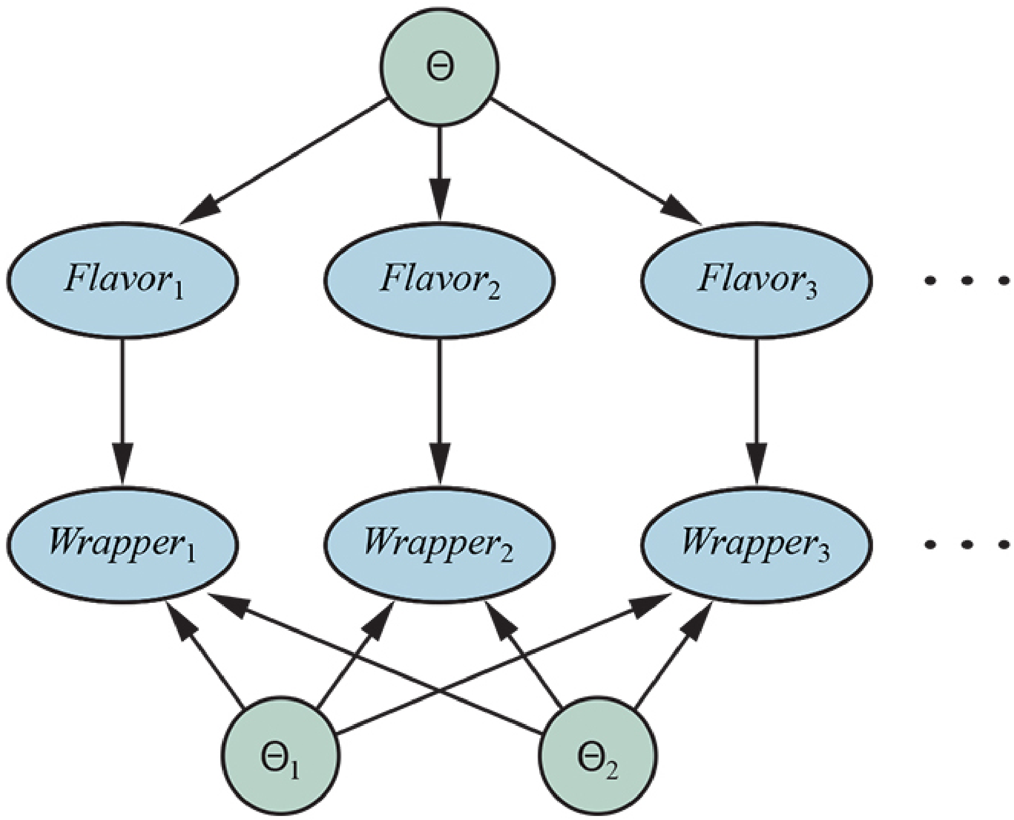


Figure 21.6 A Bayesian network that corresponds to a Bayesian learning process. Posterior distributions for the parameter variables Θ, Θ_1, and Θ_2 can be inferred from their prior distributions and the evidence in *Flavor*$_i$ and *Wrapper*$_i$

The nodes Θ, Θ_1, Θ_2 have no parents. For the *i*th observation of a wrapper and corresponding flavor of a piece of candy, we add nodes *Wrapper*$_i$ and *Flavor*$_i$. *Flavori*$_i$ is dependent on the flavor parameter Θ:

$$P(Flavor_i = cherry|\Theta = \theta) = \theta.$$

Wrapper$_i$ is dependent on Θ_1 and Θ_2:

$$P(Wrapper_i = red|(Flavor_i = cherry|\Theta_1 = \theta_1) = \theta_1$$
$$P(Wrapper_i = red|(Flavor_i = lime, \Theta_2 = \theta_2) = \theta_2.$$

Now, the entire Bayesian learning process for the original Bayes net in Figure 21.2(b) can be formulated as an *inference* problem in the derived Bayes net shown in Figure 21.6, where the data and parameters become nodes. Once we have added all the new evidence nodes, we can then query the parameter variables (in this case, Θ, Θ_1, Θ_2). Under this formulation *there is just one learning algorithm*—the inference algorithm for Bayesian networks.

Of course, the nature of these networks is somewhat different from those of Chapter 13 because of the potentially huge number of evidence variables representing the training set and the prevalence of continuous-valued parameter variables. Exact inference may be impossible except in very simple cases such as the naive Bayes model. Practitioners typically use approximate inference methods such as MCMC (Section 13.4.2); many statistical software packages incorporate efficient implementations of MCMC for this purpose.

21.2.6 Bayesian linear regression

Here we illustrate how to apply a Bayesian approach to a standard statistical task: linear regression. The conventional approach was described in Section 19.6 as minimizing the sum of squared errors and reinterpreted in Section 21.2.4 as maximizing likelihood assuming a Gaussian error model. These produce a single best hypothesis: a straight line with specific values for the slope and intercept and a fixed variance for the prediction error at any given point. There is no measure of how confident one should be in the slope and intercept values.

Furthermore, if one is predicting a value for an unseen data point far from the observed data points, it seems to make no sense to assume a prediction error that is the same as the prediction error for a data point right next to an observed data point. It would seem more sensible for the prediction error to be larger, the farther the data point is from the observed data, because a small change in the slope will cause a large change in the predicted value for a distant point.

The Bayesian approach fixes both of these problems. The general idea, as in the preceding section, is to place a prior on the model parameters—here, the coefficients of the linear model and the noise variance—and then to compute the parameter posterior given the data. For multivariate data and unknown noise model, this leads to rather a lot of linear algebra, so we focus on a simple case: univariable data, a model that is constrained to go through the origin, and known noise: a normal distribution with variance σ^2. Then we have just one parameter θ and the model is

$$P(y, \middle| x, \theta) = N(y; \theta x, \sigma_y^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{(y-x)^2}{\sigma^2}\right)}. \qquad (21.7)$$

As the log likelihood is quadratic in θ, the appropriate form for a conjugate prior on θ is also a Gaussian. This ensures that the posterior for θ will also be Gaussian. We'll assume a mean θ_0 and variance σ_0^2 for the prior, so that

$$P(\theta) = N(\theta; \theta_0, \sigma_0^2) = \frac{1}{\sigma_0\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{(\theta-\theta_0)^2}{\sigma_0^2}\right)}. \qquad (21.8)$$

Depending on the data being modeled, one might have some idea of what sort of slope θ to expect, or one might be completely agnostic. In the latter case, it makes sense to choose θ_0 to be 0 and σ_0^2

to be large—a so-called **uninformative prior**. Finally, we can assume a prior $P(x)$ for the x-value of each data point, but this is completely immaterial to the analysis because it doesn't depend on θ.

Now the setup is complete, so we can compute the posterior for θ using Equation (21.1): $P(\theta \mid \mathbf{d}) \propto P(\mathbf{d} \mid \theta)P(\theta)$. The observed data points are $\mathbf{d} = (x_1, y_1),\ldots, (x_N, y_N)$, so the likelihood for the data is obtained from Equation (21.7) as follows:

$$P(\mathbf{d}|\theta) = \left(\prod_i P(x_i)\right)\prod_i P(y_i|x_i,\theta)) = \alpha \prod_i e^{-\frac{1}{2}\left(\frac{(y_i-\theta x_i)^2}{\sigma^2}\right)}$$

$$= \alpha e^{-\frac{1}{2}\sum_i\left(\frac{(y_i-\theta x_i)^2}{\sigma^2}\right)}.$$

where we have absorbed the x-value priors and the normalizing constants for the N Gaussians into a constant α that is independent of θ. Now we combine this and the parameter prior from Equation (21.8) to obtain the posterior:

$$P(\theta|\mathbf{d}) = \alpha^n{}_e{}^{-\frac{1}{2}\left(\frac{(\theta-\theta_0)^2}{\sigma_0^2}\right)} e^{-\frac{1}{2}\sum_i\left(\frac{(y_i-\theta x_i)^2}{\sigma^2}\right)}.$$

Although this looks complicated, each exponent is a quadratic function of θ, so the sum of the two exponents is as well. Hence, the whole expression represents a Gaussian distribution for θ. Using algebraic manipulations very similar to those in Section 14.4, we find

$$P(\theta|\mathbf{d}) = \alpha^m{}_e{}^{-\frac{1}{2}\left(\frac{(\theta-\theta_N)^2}{\sigma_N^2}\right)}$$

with "updated" mean and variance given by

$$\theta_N = \frac{\sigma^2\theta_0+\sigma_0^2\sum_i x_i y_i}{\sigma^2+\sigma_0^2\sum_i x_i^2} \qquad \text{and} \qquad \sigma_N^2 = \frac{\sigma^2\sigma_0^2}{\sigma^2+\sigma_0^2\sum_i x_i^2}.$$

Let's look at these formulas to see what they mean. When the data are narrowly concentrated on a small region of the x-axis near the origin, $\Sigma_i x_i^2$ will be small and the posterior variance σ_N^2 will be large, roughly equal to the prior variance σ_0^2. This is as one would expect: the data do little to constrain the rotation of the line around the origin. Conversely, when the data are widely spread along the axis, $\Sigma_i x_i^2$ will be large and the posterior variance σ_N^2 will be small, roughly equal to $\sigma^2/(\Sigma_i x_i^2)$, so the slope will be very tightly constrained.

To make a prediction at a specific data point, we have to integrate over the possible values of θ, as suggested by Equation (21.2):

$$P(y|x,\mathbf{d}) = \int_{-\infty}^{\infty} P(y|x,\mathbf{d},\theta)P(\theta|x,\mathbf{d})d\theta = \int_{-\infty}^{\infty} P(y|x,\theta)P(\theta|\mathbf{d})d\theta$$

$$= \alpha \int_{-\infty}^{\infty} e^{-\frac{1}{2}\left(\frac{(y-\theta x)^2}{\sigma^2}\right)} e^{-\frac{1}{2}\left(\frac{(\theta-\theta_N)^2}{\sigma_N^2}\right)} d\theta$$

Again, the sum of the two exponents is a quadratic function of θ, so we have a Gaussian over θ whose integral is 1. The remaining terms in y form another Gaussian:

$$P(y|x,\mathbf{d}) \propto e^{-\frac{1}{2}\left(\frac{(y-\theta_N x)^2}{\sigma^2+\sigma_N^2 x^2}\right)}.$$

Looking at this expression, we see that the mean prediction for y is $\theta_N x$, that is, it is based on the posterior mean for θ. The variance of the prediction is given by the model noise σ^2 plus a term proportional to x^2, which means that the standard deviation of the prediction increases asymptotically linearly with the distance from the origin. Figure 21.7 illustrates this phenomenon. As noted at the beginning of this section, having greater uncertainty for predictions that are further from the observed data points makes perfect sense.

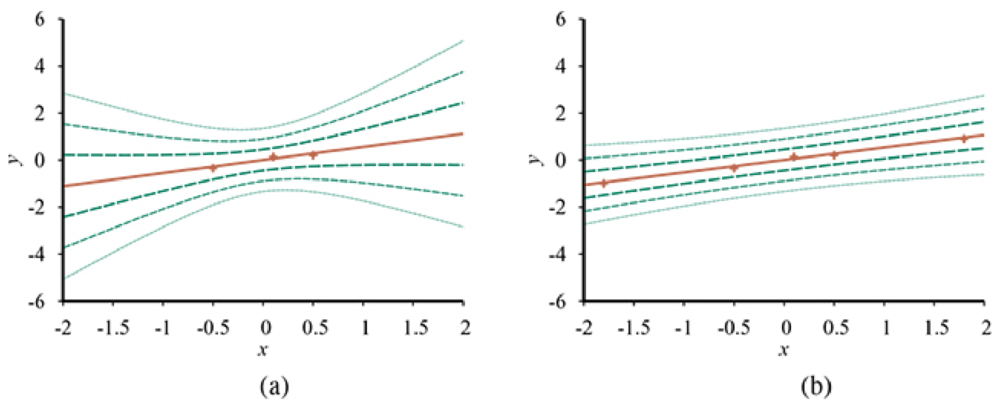


Figure 21.7 Bayesian linear regression with a model constrained to pass through the origin and fixed noise variance $\sigma^2 = 0.2$. Contours at ±1, ±2, and ±3 standard deviations are shown for the predictive density. (a) With three data points near the origin, the slope is quite uncertain, with $\sigma_N^2 \approx 0.3861$. Notice how the uncertainty increases with distance from the observed data points. (b) With two additional data points further away, the slope θ is very tightly constrained, with $\sigma_N^2 \approx 0.0286$. The remaining variance in the predictive density is almost entirely due to the fixed noise σ^2.

21.2.7 Learning Bayes net structures

So far, we have assumed that the structure of the Bayes net is given and we are just trying to learn the parameters. The structure of the network represents basic causal knowledge about the domain that is often easy for an expert, or even a naive user, to supply. In some cases, however, the causal model may be unavailable or subject to dispute—for example, certain corporations have long claimed that smoking does not cause cancer and other corporations assert that CO_2 concentrations have no effect on climate—so it is important to understand how the structure of a Bayes net can be learned from data. This section gives a brief sketch of the main ideas.

The most obvious approach is to *search* for a good model. We can start with a model containing no links and begin adding parents for each node, fitting the parameters with the methods we have just covered and measuring the accuracy of the resulting model. Alternatively, we can start with an initial guess at the structure and use hill climbing or simulated annealing search to make modifications, retuning the parameters after each change in the structure. Modifications can include reversing, adding, or deleting links. We must not introduce cycles in the process, so many algorithms assume that an ordering is given for the variables, and that a node can have parents only among those nodes that come earlier in the ordering (just as in the construction process in Chapter 13). For full generality, we also need to search over possible orderings.

There are two alternative methods for deciding when a good structure has been found. The first is to test whether the conditional independence assertions implicit in the structure are actually satisfied in the data. For example, the use of a naive Bayes model for the restaurant problem assumes that

$$\mathbf{P}(Hungry, Bar|WillWait) \;=\; \mathbf{P}(Hungry|WillWait)\mathbf{P}(Bar|WillWait)$$

and we can check in the data whether the same equation holds between the corresponding conditional frequencies. But even if the structure describes the true causal nature of the domain, statistical fluctuations in the data set mean that the equation will never be satisfied *exactly,* so we need to perform a suitable statistical test to see if there is sufficient evidence that the independence hypothesis is violated. The complexity of the resulting network will depend on the threshold used for this test—the stricter the independence test, the more links will be added and the greater the danger of overfitting.

An approach more consistent with the ideas in this chapter is to assess the degree to which the proposed model explains the data (in a probabilistic sense). We must be careful how we measure this, however. If we just try to find the maximum-likelihood hypothesis, we will end up with a fully connected network, because adding more parents to a node cannot decrease the likelihood (Exercise 21.MLPA). We are forced to penalize model complexity in some way. The MAP (or MDL) approach simply subtracts a penalty from the likelihood of each structure (after parameter tuning) before comparing different structures. The Bayesian approach places a joint prior over structures and parameters. There are usually far too many structures to sum over (superexponential in the number of variables), so most practitioners use MCMC to sample over structures.

Penalizing complexity (whether by MAP or Bayesian methods) introduces an important connection between the optimal structure and the nature of the representation for the conditional distributions in the network. With tabular distributions, the complexity penalty for a node's distribution grows exponentially with the number of parents, but with, say, noisy-OR distributions, it grows only linearly. This means that learning with noisy-OR (or other compactly parameterized) models tends to produce learned structures with more parents than does learning with tabular distributions.

21.2.8 Density estimation with nonparametric models

It is possible to learn a probability model without making any assumptions about its structure and parameterization by adopting the nonparametric methods of Section 19.7. The task of **nonparametric density estimation** is typically done in continuous domains, such as that shown in Figure 21.8(a). The figure shows a probability density function on a space defined by two continuous variables. In Figure 21.8(b) we see a sample of data points from this density function. The question is, can we recover the model from the samples?

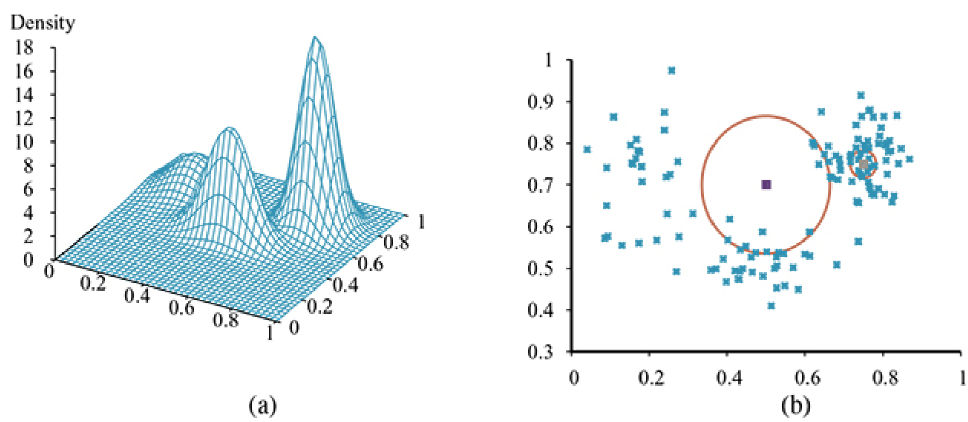


Figure 21.8 (a) A 3D plot of the mixture of Gaussians from Figure 21.12(a). (b) A 128-point sample of points from the mixture, together with two query points (small orange squares) and their 10-nearest-neighborhoods (large circle and smaller circle to the right).

First we will consider *k*-**nearest-neighbors** models. (In Chapter 19 we saw nearest-neighbor models for classification and regression; here we see them for density estimation.) Given a sample of data points, to estimate the unknown probability density at a query point**x** we can simply measure the density of the data points in the neighborhood of **x**. Figure 21.8(b) shows two query points (small squares). For each query point we have drawn the smallest circle that encloses 10 neighbors—the 10-nearest-neighborhood. We can see that the central circle is large, meaning there is a low density there, and the circle on the right is small, meaning there is a high density there. In Figure 21.9 we show three plots of density estimation using k-nearest-neighbors, for different values of k. It seems clear that (b) is about right, while (a) is too spiky (k is too small) and (c) is too smooth (k is too big).

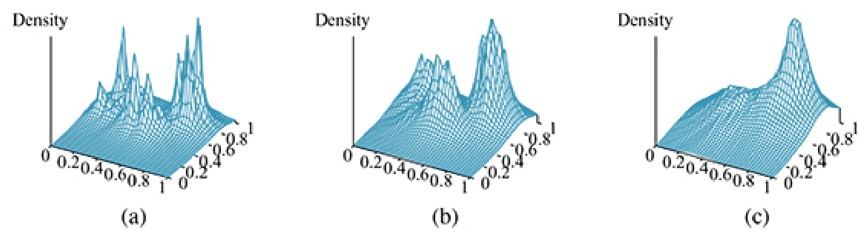


Figure 21.9 Density estimation using k-nearest-neighbors, applied to the data in Figure 21.8(b), for $k = 3$, 10, and 40 respectively. $k = 3$ is too spiky, 40 is too smooth, and 10 is just about right. The best value for k can be chosen by cross-validation.

Another possibility is to use **kernel functions**, as we did for locally weighted regression. To apply a kernel model to density estimation, assume that each data point generates its own little density function. For example, we might use spherical Gaussians with standard deviation w along each axis. Then estimated density at a query point $\mathbf{x}$ is the average of the data kernels:

$$P(\mathbf{x}) = \frac{1}{N}\sum_{j=1}^{N} K(\mathbf{x}, \mathbf{x}_j) \qquad \text{where} \qquad K(\mathbf{x}, \mathbf{x}_j) = \frac{1}{(w^2\sqrt{2\pi})^d} e^{-\frac{D(\mathbf{x}, \mathbf{x}_j)^2}{2w^2}},$$

where d is the number of dimensions in $\mathbf{x}$ and D is the Euclidean distance function. We still have the problem of choosing a suitable value for kernel width w; Figure 21.10 shows values that are too small, just right, and too large. A good value of w can be chosen by using cross-validation.

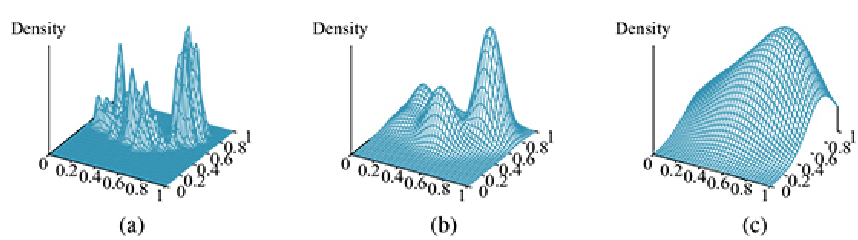


Figure 21.10 Density estimation using kernels for the data in Figure 21.8(b), using Gaussian kernels with $w = 0.02$, 0.07, and 0.20 respectively. $w = 0.07$ is about right.

21.3 Learning with Hidden Variables: The EM Algorithm

The preceding section dealt with the fully observable case. Many real-world problems have **hidden variables** (sometimes called **latent variables**), which are not observable in the data. For example, medical records often include the observed symptoms, the physician's diagnosis, the treatment applied, and perhaps the outcome of the treatment, but they seldom contain a direct observation of the disease itself! (Note that the *diagnosis* is not the *disease;* it is a causal consequence of the observed symptoms, which are in turn caused by the disease.) One might ask, "If the disease is not observed, could we construct a model based only on the observed variables?" The answer appears in Figure 21.11, which shows a small, fictitious diagnostic model for heart disease. There are three observable predisposing factors and three observable symptoms (which are too depressing to name). Assume that each variable has three possible values (e.g., *none*, *moderate,* and severe). Removing the hidden variable from the network in (a) yields the network in (b); the total number of parameters increases from 78 to 708. Thus, *latent variables can dramatically reduce the number of parameters required to specify a Bayesian network.* This, in turn, can dramatically reduce the amount of data needed to learn the parameters.

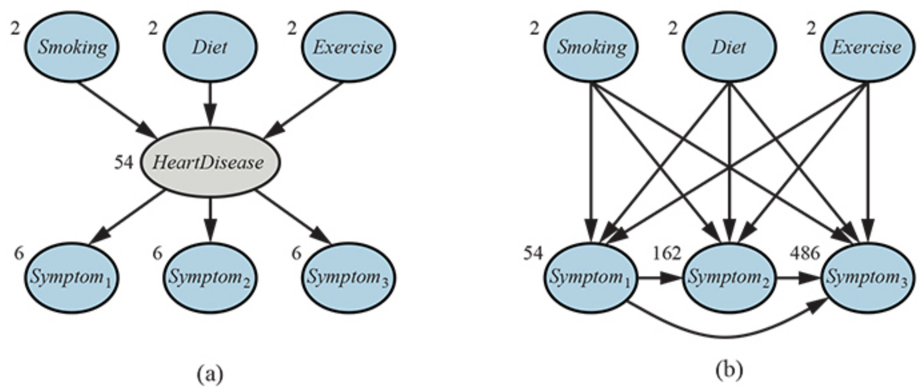


Figure 21.11 (a) A simple diagnostic network for heart disease, which is assumed to be a hidden variable. Each variable has three possible values and is labeled with the number of independent parameters in its conditional distribution; the total number is 78. (b) The equivalent network with *HeartDisease* removed. Note that the symptom variables are no longer conditionally independent given their parents. This network requires 708 parameters.

Hidden variables are important, but they do complicate the learning problem. In Figure 21.11(a), for example, it is not obvious how to learn the conditional distribution for *HeartDisease*, given its parents, because we do not know the value of *HeartDisease* in each case; the same problem arises in learning the distributions for the symptoms. This section describes an algorithm called **expectation–maximization**, or EM, that solves this problem in a very general way. We will show three examples and then provide a general description. The algorithm seems like magic at first, but once the intuition has been developed, one can find applications for EM in a huge range of learning problems.

21.3.1 Unsupervised clustering: Learning mixtures of Gaussians

Unsupervised clustering is the problem of discerning multiple categories in a collection of objects. The problem is unsupervised because the category labels are not given. For example, suppose we record the spectra of a hundred thousand stars; are there different *types* of stars revealed by the spectra, and, if so, how many types and what are their characteristics? We are all familiar with terms such as “red giant” and “white dwarf,” but the stars do not carry these labels on their hats—astronomers had to perform unsupervised clustering to identify these categories. Other examples include the identification of species, genera, orders, phylum, and so on in the Linnaean taxonomy and the creation of natural kinds for ordinary objects (see Chapter 10).

Unsupervised clustering begins with data. Figure 21.12(b) shows 500 data points, each of which specifies the values of two continuous attributes. The data points might correspond to stars, and the attributes might correspond to spectral intensities at two particular frequencies.

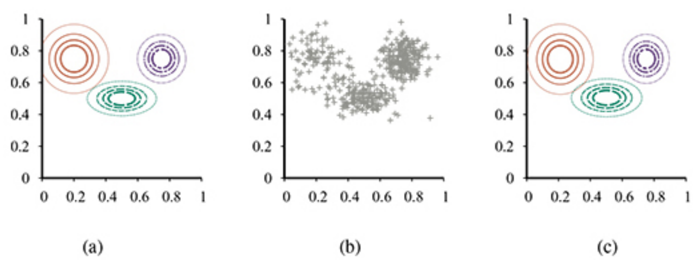


Figure 21.12 (a) A Gaussian mixture model with three components; the weights (left-to-right) are 0.2, 0.3, and 0.5. (b) 500 data points sampled from the model in (a). (c) The model reconstructed by EM from the data in (b).

Next, we need to understand what kind of probability distribution might have generated the data. Clustering presumes that the data are generated from a **mixture distribution**, P. Such a distribution has k **components**, each of which is a distribution in its own right. A data point is generated by first choosing a component and then generating a sample from that component. Let the random variable C denote the component, with values $1,...,k$; then the mixture distribution is given by

$$P(\mathbf{x}) = \sum_{i=1}^{k} P(C=i)P(\mathbf{x}|C=i),$$

where $\mathbf{x}$ refers to the values of the attributes for a data point. For continuous data, a natural choice for the component distributions is the multivariate Gaussian, which gives the so-called **mixture of Gaussians** family of distributions. The parameters of a mixture of Gaussians are $w_i = P(C = i)$ (the weight of each component), μ_i (the mean of each component), and $\sum_i$ (the covariance of each component). Figure 21.12(a) shows a mixture of three Gaussians; this mixture is in fact the source of the data in (b) as well as being the model shown in Figure 21.8(a) on page 787.

The unsupervised clustering problem, then, is to recover a Gaussian mixture model like the one in Figure 21.12(a) from raw data like that in Figure 21.12(b). Clearly, if we *knew* which component generated each data point, then it would be easy to recover the component Gaussians: we could just select all the data points from a given component and then apply (a multivariate version of) Equation (21.4) (page 780) for fitting the parameters of a Gaussian to a set of data. On the other hand, if we *knew* the parameters of each component, then we could, at least in a probabilistic sense, assign each data point to a component.

The problem is that we know neither the assignments nor the parameters. The basic idea of EM in this context is to *pretend* that we know the parameters of the model and then to infer the probability that each data point belongs to each component. After that, we refit the components to the data, where each component is fitted to the entire data set with each point weighted by the probability that it belongs to that component. The process iterates until convergence. Essentially, we are "completing" the data by inferring probability distributions over the hidden variables—which component each data point belongs to—based on the current model. For the mixture of Gaussians, we initialize the mixture-model parameters arbitrarily and then iterate the following two steps:

1. **E-step**: Compute the probabilities $p_{ij} = P(C = i \mid \mathbf{x}_j)$, the probability that datum $\mathbf{x}_j$ was generated by component i. By Bayes' rule, we have $p_{ij} = \alpha P(\mathbf{x}_j \mid C = i)P(C = i)$. The term $P(\mathbf{x}_j \mid C = i)$ is just the probability at $\mathbf{x}_j$ of the ith Gaussian, and the term $P(C = i)$ is just the weight parameter for the ith Gaussian. Define $n_i = \sum_j p_{ij}$, the effective number of data points currently assigned to component i.
2. **M-step:** Compute the new mean, covariance, and component weights using the following steps in sequence:

$$\mu_i \leftarrow \sum_j p_{ij}\mathbf{x}_j / n_i$$

$$\Sigma_i \leftarrow \sum_j p_{ij}(\mathbf{x}_j - \mu_i)(\mathbf{x}_j - \mu_i)^{\mathrm{T}} / n_i$$

$$w_i \leftarrow n_i / N$$

where N is the total number of data points. The E-step, or *expectation* step, can be viewed as computing the expected values p_{ij} of the hidden **indicator variables** Z_{ij}, where Z_{ij} is 1 if datum $\mathbf{x}_j$ was generated by the ith component and 0 otherwise. The M-step, or *maximization* step, finds the new values of the parameters that maximize the log likelihood of the data, given the expected values of the hidden indicator variables.

The final model that EM learns when it is applied to the data in Figure 21.12(a) is shown in Figure 21.12(c); it is virtually indistinguishable from the original model from which the data were generated (horizontal line). Figure 21.13(a) plots the log likelihood of the data according to the current model as EM progresses.

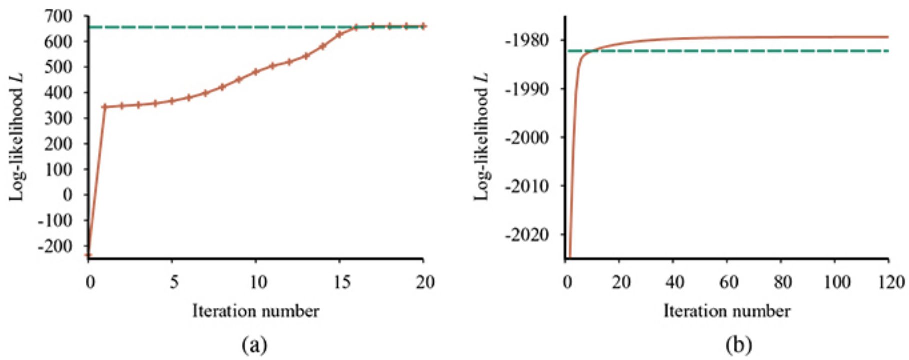


Figure 21.13 Graphs showing the log likelihood of the data, L, as a function of the EM iteration. The horizontal line shows the log likelihood according to the true model. (a) Graph for the Gaussian mixture model in Figure 21.12. (b) Graph for the Bayesian network in Figure 21.14(a).

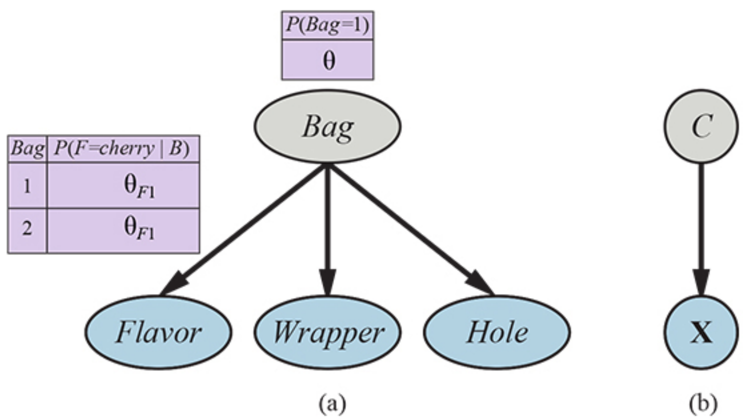


Figure 21.14 (a) A mixture model for candy. The proportions of different flavors, wrappers, and presence of holes depend on the bag, which is not observed. (b) Bayesian network for a Gaussian mixture. The mean and covariance of the observable variables $\mathbf{X}$ depend on the component C.

There are two points to notice. First, the log likelihood for the final learned model slightly *exceeds* that of the original model, from which the data were generated. This might seem surprising, but it simply reflects the fact that the data were generated randomly and might not provide an exact reflection of the underlying model. The second point is that *EM increases the log likelihood of the data at every iteration.* This fact can be proved in general. Furthermore, under certain conditions (that hold in most cases), EM can be proven to reach a local maximum in likelihood. (In rare cases, it could reach a saddle point or even a local minimum.) In this sense, EM resembles a gradient-based hill-climbing algorithm, but notice that it has no “step size” parameter.

Things do not always go as well as Figure 21.13(a) might suggest. It can happen, for example, that one Gaussian component shrinks so that it covers just a single data point. Then its variance will go to zero and its likelihood will go to infinity! If we don't know how many components are in the mixture we have to try different values of k and see which is best; that can be a source of error. Another problem is that two components can "merge," acquiring identical means and variances and sharing their data points. These kinds of degenerate local maxima are serious problems, especially in high dimensions. One solution is to place priors on the model parameters and to apply the MAP version of EM. Another is to restart a component with new random parameters if it gets too small or too close to another component. Sensible initialization also helps.

21.3.2 Learning Bayes net parameter values for hidden variables

To learn a Bayesian network with hidden variables, we apply the same insights that worked for mixtures of Gaussians. Figure 21.14(a) represents a situation in which there are two bags of candy that have been mixed together. Candies are described by three features: in addition to the *Flavor* and the *Wrapper*, some candies have a *Hole* in the middle and some do not. The distribution of candies in each bag is described by a **naive Bayes** model: the features are independent, given the bag, but the conditional probability distribution for each feature depends on the bag. The parameters are as follows: θ is the prior probability that a candy comes from Bag 1; θ_{F1} and θ_{F2} are the probabilities that the flavor is cherry, given that the candy comes from Bag 1 or Bag 2 respectively; θ_{W1} and θ_{W2} give the probabilities that the wrapper is red; and θ_{W1} and θ_{W2} give the probabilities that the candy has a hole.

The overall model is a mixture model: a weighted sum of two different distributions, each of which is a product of independent, univariate distributions. (In fact, we can also model the mixture of Gaussians as a Bayesian network, as shown in Figure 21.14(b).) In the figure, the bag is a hidden variable because, once the candies have been mixed together, we no longer know which bag each candy came from. In such a case, can we recover the descriptions of the two bags by observing candies from the mixture? Let us work through an iteration of EM for this problem. First, let's look at the data. We generated 1000 samples from a model whose true parameters are as follows:

$$\theta = 0.5,\ \theta_{F1} = \theta_{W1} = \theta_{H1} = 0.8,\ \theta_{F2} = \theta_{W2} = \theta_{H2} = 0.3. \qquad (21.9)$$

That is, the candies are equally likely to come from either bag; the first is mostly cherry with red wrappers and holes; the second is mostly lime with green wrappers and no holes. The counts for the eight possible kinds of candy are as follows:

We start by initializing the parameters. For numerical simplicity, we arbitrarily choose[5]

$$\theta^{(0)} = 0.6,\ \theta^{(0)}_{F1} = \theta^{(0)}_{W1} = \theta^{(0)}_{H1} = 0.6,\ \theta^{(0)}_{F2} = \theta^{(0)}_{W2} = \theta^{(0)}_{H2} = 0.4. \qquad (21.10)$$

First, let us work on the θ parameter. In the fully observable case, we would estimate this directly from the *observed* counts of candies from bags 1 and 2. Because the bag is a hidden variable, we calculate the *expected* counts instead. The expected count $\widehat{N}(Bag = 1)$ is the sum, over all candies, of the probability that the candy came from bag 1:

$$\theta^{(1)} = \hat{N}(Bag = 1)/N = \sum_{j=1}^{N} P(Bag = 1|flavor_j, wrapper_j,\ holes_j)/N.$$

These probabilities can be computed by any inference algorithm for Bayesian networks. For a naive Bayes model such as the one in our example, we can do the inference "by hand," using Bayes' rule and applying conditional independence:

$$\theta^{(1)} = \frac{1}{N}\sum_{j=1}^{N} \frac{P(flavor_j|Bag=1)P(wrapper_j|Bag=1)P(holes_j|Bag=1)P(Bag=1)}{\sum_i P(flavor_j|Bag=i)P(wrapper_j|Bag=i)P(holes_j|Bag=i)P(Bag=i)}.$$

Applying this formula to, say, the 273 red-wrapped cherry candies with holes, we get a contribution of

$$\frac{273}{1000} \cdot \frac{\theta_{F1}^{(0)}\theta_{W1}^{(0)}\theta_{H1}^{(0)}\theta^{(0)}}{\theta_{F1}^{(0)}\theta_{W1}^{(0)}\theta_{H1}^{(0)}\theta^{(0)}+\theta_{F2}^{(0)}\theta_{W2}^{(0)}\theta_{H2}^{(0)}(1-\theta^{(0)})} \approx 0.22797.$$

Continuing with the other seven kinds of candy in the table of counts, we obtain $\theta^{(1)} = 0.6124$.

Now let us consider the other parameters, such as θ_{F1}. In the fully observable case, we would estimate this directly from the *observed* counts of cherry and lime candies from bag 1. The *expected* count of cherry candies from bag 1 is given by

$$\sum_{j:Flavor\ j=cherry} P(Bag=1|Flavor_j = cherry, wrapper_j,\ holes_j).$$

Again, these probabilities can be calculated by any Bayes net algorithm. Completing this process, we obtain the new values of all the parameters:

$$\begin{aligned}&\theta^{(1)} = 0.6124,\ \theta_{F1}^{(1)} = 0.6684,\ \theta_{W1}^{(1)} = 0.6483,\ \theta_{H1}^{(1)} = 0.6558\\ &\theta_{F2}^{(1)} = 0.3887,\ \theta_{W2}^{(1)} = 0.3817, \theta_{H2}^{(1)} = 0.3827.\end{aligned} \tag{21.11}$$

The log likelihood of the data increases from about –2044 initially to about –2021 after the first iteration, as shown in Figure 21.13(b). That is, the update improves the likelihood itself by a factor of about $e^{23} \approx 10^{10}$. By the tenth iteration, the learned model is a better fit than the original model (L = –1982.214). Thereafter, progress becomes very slow. This is not uncommon with EM, and many practical systems combine EM with a gradient-based algorithm such as Newton–Raphson (see Chapter 4) for the last phase of learning.

The general lesson from this example is that *the parameter updates for Bayesian network learning with hidden variables are directly available from the results of inference on each example. Moreover, only local posterior probabilities are needed for each parameter.* Here, "local" means that the conditional probability table (CPT) for each variable X_i can be learned from posterior probabilities involving just X_i and its parents $\mathbf{U}_i$. Defining θ_{ijk} to be the CPT parameter $P(X_i = x_{ij} \mid \mathbf{U}_i = \mathbf{u}_{ik})$, the update is given by the normalized expected counts as follows:

$$\theta_{ijk} \leftarrow \widehat{N}(X_i = x_{ij}, \mathbf{U}_i = \mathbf{u}_{ik})/\widehat{N}(\mathbf{U}_i = \mathbf{u}_{ik}).$$

The expected counts are obtained by summing over the examples, computing the probabilities $P(X_i = x_{ij}, \mathbf{U}_i = \mathbf{u}_{ik})$ for each by using any Bayes net inference algorithm. For the exact algorithms—including variable elimination—all these probabilities are obtainable directly as a by-product of standard inference, with no need for extra computations specific to learning. Moreover, the information needed for learning is available *locally* for each parameter.

Standing back a little, we can think about what the EM algorithm is doing in this example as recovering seven parameters (θ, θ_{F1}, θ_{W1}, θ_{H1}, θ_{F2}, θ_{W2}, θ_{H2}) from seven ($2^3 - 1$) observed counts in the data. (The eighth count is fixed by the fact that the counts sum to 1000.) If each candy were described by two attributes rather than three (say, omitting the holes), we would have had five parameters (θ, θ_{F1}, θ_{W1}, θ_{F2}, θ_{W2}) but only three ($2^2 - 1$) observed counts. In such a case it is not possible to recover the mixture weight d or the characteristics of the two bags that were mixed together. We say that the two-attribute model is not **identifiable**.

Identifiability in Bayesian networks is a tricky issue. Note that even with three attributes and seven counts, we cannot uniquely recover the model, because there are two observationally equivalent models with the *Bag* variable flipped. Depending on how the parameters are initialized, EM will converge either to a model where bag 1 has mostly cherry and bag 2 mostly lime, or vice versa. This kind if non-identifiability is unavoidable with variables that are never observed.

21.3.3 Learning hidden Markov models

Our final application of EM involves learning the transition probabilities in hidden Markov models (HMMs). Recall from Section 14.3 that a hidden Markov model can be represented by a dynamic Bayes net with a single discrete state variable, as illustrated in Figure 21.15. Each data point consists of an observation *sequence* of finite length, so the problem is to learn the transition probabilities from a set of observation sequences (or from just one long sequence).

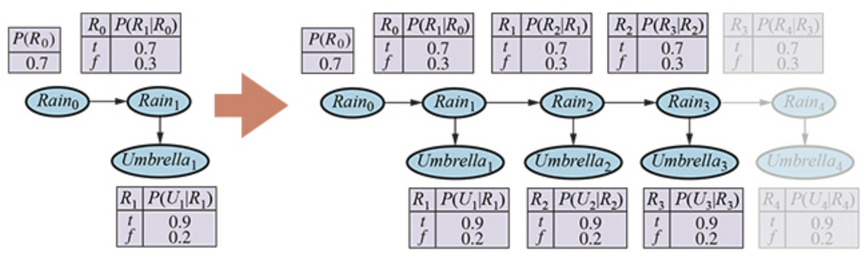


Figure 21.15 An unrolled dynamic Bayesian network that represents a hidden Markov model (repeat of Figure 14.16).

We have already seen how to learn Bayes nets, but there is a complication: in Bayes nets, each parameter is distinct; in a hidden Markov model, on the other hand, the individual transition probabilities from state i to state j at time t, $\theta_{ijt} = P(X_{t+1} = j \mid X_t = i)$, are *repeated* across time—that is, $\theta_{ijt} = \theta_{ij}$ for all t. To estimate the transition probability from state i to state j, we simply calculate the expected proportion of times that the system undergoes a transition to state j when in state i:

$$\theta_{ij} \leftarrow \sum_t \widehat{N}(X_{t+1} = j, X_t = i) / \sum_t \widehat{N}(X_t = i).$$

The expected counts are computed by an HMM inference algorithm. The **forward–backward** algorithm shown in Figure 14.4 can be modified very easily to compute the necessary probabilities. One important point is that the probabilities required are obtained by **smoothing** rather than **filtering**. Filtering gives the probability distribution of the current state given the past, but smoothing gives the distribution given all evidence, including what happens after a particular transition occurred. The evidence in a murder case is usually obtained *after* the crime (i.e., the transition from state i to state j) has taken place.

21.3.4 The general form of the EM algorithm

We have seen several instances of the EM algorithm. Each involves computing expected values of hidden variables for each example and then recomputing the parameters, using the expected values as if they were observed values. Let **x** be all the observed values in all the examples, let **Z** denote all the hidden variables for all the examples, and let θ be all the parameters for the probability model. Then the EM algorithm is

$$\theta^{(i+1)} = \underset{\theta}{\operatorname{argmax}} \sum_{\mathbf{z}} P(\mathbf{Z} = \mathbf{z} \big| \mathbf{x}, \theta^{(i)}) L(\mathbf{x}, \mathbf{Z} = \mathbf{z} \big| \theta).$$

This equation is the EM algorithm in a nutshell. The E-step is the computation of the summation, which is the expectation of the log likelihood of the "completed" data with respect to the distribution $P(\mathbf{Z} = \mathbf{z} \mid \mathbf{x}, \theta^{(i)})$, which is the posterior over the hidden variables, given the data. The M-step is the maximization of this expected log likelihood with respect to the parameters. For mixtures of Gaussians, the hidden variables are the where Z_{ij}s, where Z_{ij} is 1 if example j was generated by component i. For Bayes nets, Z_{ij} is the value of unobserved variable X_i in example j. For HMMs, Z_{jt} is the state of the sequence in example j at time t. Starting from the general form, it is possible to derive an EM algorithm for a specific application once the appropriate hidden variables have been identified.

As soon as we understand the general idea of EM, it becomes easy to derive all sorts of variants and improvements. For example, in many cases the E-step—the computation of posteriors over the hidden variables—is intractable, as in large Bayes nets. It turns out that one can use an *approximate* E-step and still obtain an effective learning algorithm. With a sampling algorithm such as MCMC (see Section 13.4), the learning process is very intuitive: each state (configuration of hidden and observed variables) visited by MCMC is treated exactly as if it were a complete observation. Thus, the parameters can be updated directly after each MCMC transition. Other forms of approximate inference, such as variational methods and loopy belief propagation, have also proved effective for learning very large networks.

21.3.5 Learning Bayes net structures with hidden variables

In Section 21.2.7, we discussed the problem of learning Bayes net structures with complete data. When unobserved variables influence observed data, things get more difficult. In the simplest case, a human expert might tell the learning algorithm that certain hidden variables exist, leaving it to the algorithm to find a place for them in the network structure. For example, an algorithm might try to learn the structure shown in Figure 21.11(a) on page 789, given the information that *HeartDisease* (a three-valued variable) should be included in the model. As in the complete-data case, the overall algorithm has an outer loop that searches over structures and an inner loop that fits the network parameters given the structure.

If the learning algorithm is not told which hidden variables exist, then there are two choices: either pretend that the data are really complete—which may force the algorithm to learn a parameter-intensive model such as the one in Figure 21.11(b)—or *invent* new hidden variables in order to simplify the model. The latter approach can be implemented by including new modification choices in the structure search: in addition to modifying links, the algorithm can add or delete a hidden variable or change its arity. Of course, the algorithm will not know that the new variable it has invented is called *HeartDisease*; nor will it have meaningful names for the values. Fortunately, newly invented hidden variables will usually be connected to preexisting variables, so a human expert can often inspect the local conditional distributions involving the new variable and ascertain its meaning.

As in the complete-data case, pure maximum-likelihood structure learning will result in a completely connected network (moreover, one with no hidden variables), so some form of complexity penalty is required. We can also apply MCMC to sample many possible network structures, thereby approximating Bayesian learning. For example, we can learn mixtures of Gaussians with an unknown number of components by sampling over the number; the approximate posterior distribution for the number of Gaussians is given by the sampling frequencies of the MCMC process.

For the complete-data case, the inner loop to learn the parameters is very fast—just a matter of extracting conditional frequencies from the data set. When there are hidden variables, the inner loop may involve many iterations of EM or a gradient-based algorithm, and each iteration involves the calculation of posteriors in a Bayes net, which is itself an NP-hard problem. To date, this approach has proved impractical for learning complex models.

One possible improvement is the so-called **structural EM** algorithm, which operates in much the same way as ordinary (parametric) EM except that the algorithm can update the structure as well as the parameters. Just as ordinary EM uses the current parameters to compute the expected counts in the E-step and then applies those counts in the M-step to choose new parameters, structural EM uses the current structure to compute expected counts and then applies those counts in the M-step to evaluate the likelihood for potential new structures. (This contrasts with the outer-loop/inner-loop method, which computes new expected counts for each potential structure.) In this way, structural EM may make several structural alterations to the network without once recomputing the expected counts, and is capable of learning nontrivial Bayes net structures. Structural EM has a search space over the space of structures rather than the space of structures and parameters. Nonetheless, much work remains to be done before we can say that the structure-learning problem is solved.

Summary

Statistical learning methods range from simple calculation of averages to the construction of complex models such as Bayesian networks. They have applications throughout computer science, engineering, computational biology, neuroscience, psychology, and physics. This chapter has presented some of the basic ideas and given a flavor of the mathematical underpinnings. The main points are as follows:

- **Bayesian learning** methods formulate learning as a form of probabilistic inference, using the observations to update a prior distribution over hypotheses. This approach provides a good way to implement Ockham's razor, but quickly becomes intractable for complex hypothesis spaces.
- **Maximum a posteriori** (MAP) learning selects a single most likely hypothesis given the data. The hypothesis prior is still used and the method is often more tractable than full Bayesian learning.
- **Maximum-likelihood** learning simply selects the hypothesis that maximizes the likelihood of the data; it is equivalent to MAP learning with a uniform prior. In simple cases such as linear regression and fully observable Bayesian networks, maximum-likelihood solutions can be found easily in closed form.**Naive Bayes** learning is a particularly effective technique that scales well.
- When some variables are hidden, local maximum likelihood solutions can be found using the **expectation maximization** (EM) algorithm. Applications include unsupervised clustering using mixtures of Gaussians, learning Bayesian networks, and learning hidden Markov models.

- Learning the structure of Bayesian networks is an example of **model selection**. This usually involves a discrete search in the space of structures. Some method is required for trading off model complexity against degree of fit.
- **Nonparametric models** represent a distribution using the collection of data points. Thus, the number of parameters grows with the training set. Nearest-neighbors methods look at the examples nearest to the point in question, whereas **kernel** methods form a distance-weighted combination of all the examples.

Statistical learning continues to be a very active area of research. Enormous strides have been made in both theory and practice, to the point where it is possible to learn almost any model for which exact or approximate inference is feasible.

Bibliographical and Historical Notes

The application of statistical learning techniques in AI was an active area of research in the early years (see Duda and Hart, 1973) but became separated from mainstream AI as the latter field concentrated on symbolic methods. A resurgence of interest occurred shortly after the introduction of Bayesian network models in the late 1980s; at roughly the same time, a statistical view of neural network learning began to emerge. In the late 1990s, there was a noticeable convergence of interests in machine learning, statistics, and neural networks, centered on methods for creating large probabilistic models from data.

The naive Bayes model is one of the oldest and simplest forms of Bayesian network, dating back to the 1950s. Its origins were mentioned in Chapter 12. Its surprising success is partially explained by Domingos and Pazzani (1997). A boosted form of naive Bayes learning won the first KDD Cup data mining competition (Elkan, 1997). Heckerman (1998) gives an excellent introduction to the general problem of Bayes net learning. Bayesian parameter learning with Dirichlet priors forBayesian networks was discussed by Spiegelhalter *et al.* (1993). The beta distribution as a conjugate prior for a Bernoulli variable was first derived by Thomas (Bayes, 1763) and later reintroduced by Karl Pearson (1895) as a model for skewed data; for many years it was known as a "Pearson Type I distribution." Bayesian linear regression is discussed in the text by Box and Tiao (1973); Minka (2010) provides a concise summary of the derivations for the general multivariate case.

Several software packages incorporate mechanisms for statistical learning with Bayes net models. These include BUGS (Bayesian inference

Using Gibbs Sampling) (Gilks *et al.,* 1994; Lunn *et al.,* 2000, 2013), JAGS (Just Another Gibbs Sampler) (Plummer, 2003), and STAN (Carpenter *et al.,* 2017).

The first algorithms for learning Bayes net structures used conditional independence tests (Pearl, 1988; Pearl and Verma, 1991). Spirtes *et al.* (1993) implemented a comprehensive approach in the TETRAD package for Bayes net learning. Algorithmic improvements since then led to a clear victory in the 2001 KDD Cup data mining competition for a Bayes net learning method (Cheng *et al.,* 2002). (The specific task here was a bioinformatics problem with 139,351 features!) A structure-learning approach based on maximizing likelihood was developed by Cooper and Herskovits (1992) and improved by Heckerman *et al.* (1994).

More recent algorithms have achieved quite respectable performance in the complete-data case (Moore and Wong, 2003; Teyssier and Koller, 2005). One important component is an efficient data structure, the AD-tree, for caching counts over all possible combinations of variables and values (Moore and Lee, 1997). Friedman and Goldszmidt (1996) pointed out the influence of the representation of local conditional distributions on the learned structure.

The general problem of learning probability models with hidden variables and missing data was addressed by Hartley (1958), who described the general idea of what was later called EM and gave several examples. Further impetus came from the Baum–Welch algorithm for HMM learning (Baum and Petrie, 1966), which is a special case of EM. The paper by Dempster, Laird, and Rubin (1977), which presented the EM algorithm in general form and analyzed its convergence, is one of the most cited papers in both computer science and statistics. (Dempster himself views EM as a schema rather than an algorithm, since a good deal of mathematical work may be required before it can be applied to a new family of distributions.) McLachlan and Krishnan (1997) devote an entire book to the algorithm and its properties. The specific problem of learning mixture models, including mixtures of Gaussians, is covered by Titterington *et al.* (1985).

Within AI, AUTOCLASS(Cheeseman *et al.,* 1988; Cheeseman and Stutz, 1996) was the first successful system that used EM for mixture modeling. AUTOCLASS was applied to a number of real-world scientific classification tasks, including the discovery of new types of stars from spectral data (Goebel *et al.*, 1989) and new classes of proteins and introns in DNA/protein sequence databases (Hunter and States, 1992).

For maximum-likelihood parameter learning in Bayes nets with hidden variables, EM and gradient-based methods were introduced around the same time by Lauritzen (1995) and Russell *et al.* (1995). The structural EM algorithm was developed by Friedman (1998) and applied to maximum-likelihood learning of Bayes net structures with latent variables. Friedman and Koller (2003) describe Bayesian structure learning. Daly *et al.* (2011) review the field of Bayes net learning, providing extensive citations to the literature.

The ability to learn the structure of Bayesian networks is closely connected to the issue of recovering *causal* information from data. That is, is it possible to learn Bayes nets in such a way that the recovered network structure indicates real causal influences? For many years, statisticians avoided this question, believing that observational data (as opposed to data generated from experimental trials) could yield only correlational information—after all, any two variables that appear related might in fact be influenced by a third, unknown causal factor rather than influencing each other directly. Pearl (2000) has presented convincing arguments to the contrary, showing that there are in fact many cases where causality can be ascertained and developing the **causal network** formalism to express causes and the effects of intervention as well as ordinary conditional probabilities.

Nonparametric density estimation, also called **Parzen window** density estimation, was investigated initially by Rosenblatt (1956) and Parzen (1962). Since that time, a huge literature has developed investigating the properties of various estimators. Devroye (1987) gives a thorough introduction. There is also a rapidly growing literature on nonparametric Bayesian methods, originating with the seminal work of Ferguson (1973) on the **Dirichlet process**, which can be thought of as a distribution over Dirichlet distributions. These methods are particularly useful for mixtures with unknown numbers of components. Ghahramani (2005) and Jordan (2005) provide useful tutorials on the many applications of these ideas to statistical learning. The text by Rasmussen and Williams (2006) covers the **Gaussian process**, which gives a way of defining prior distributions over the space of continuous functions.

The material in this chapter brings together work from the fields of statistics and pattern recognition, so the story has been told many times in many ways. Good texts on Bayesian statistics include those by DeGroot (1970), Berger (1985), and Gelman *et al.* (1995). Bishop (2007), Hastie *et al.* (2009), Barber (2012), and Murphy (2012) provide excellent introductions to statistical machine learning. For pattern classification, the classic text for many years has been Duda and Hart (1973), now updated (Duda *et al.,* 2001). The annual NeurIPS (Neural Information Processing Systems, formerly NIPS) conference, whose proceedings are published as the series *Advances in Neural Information Processing Systems,* includes many Bayesian learning papers, as does the annual conference on Artificial Intelligence and Statistics. Specifically Bayesian venues include the

Valencia International Meetings on Bayesian Statistics and the journal *Bayesian Analysis*.

[1] Statistically sophisticated readers will recognize this scenario as a variant of the **urn-and-ball** setup. We find urns and balls less compelling than candy.

[2] We stated earlier that the bags of candy are very large; otherwise, the i.i.d. assumption fails to hold. Technically, it is more correct (but less hygienic) to rewrap each candy after inspection and return it to the bag.

[3] They are called hyperparameters because they parameterize a distribution over θ, which is itself a parameter.

[4] Other conjugate priors include the **Dirichlet** family for the parameters of a discrete multivalued distribution and the **Normal–Wishart** family for the parameters of a Gaussian distribution. See Bernardo and Smith (1994).

[5] It is better in practice to choose them randomly, to avoid local maxima due to symmetry.

CHAPTER 22

DEEP LEARNING

In which gradient descent learns multistep programs, with significant implications for the major subfields of artificial intelligence.

Deep learning is a broad family of techniques for machine learning in which hypotheses take the form of complex algebraic circuits with tunable connection strengths. The word “deep” refers to the fact that the circuits are typically organized into many **layers**, which means that computation paths from inputs to outputs have many steps. Deep learning is currently the most widely used approach for applications such as visual object recognition, machine translation, speech recognition, speech synthesis, and image synthesis; it also plays a significant role in reinforcement learning applications (see Chapter 23).

Deep learning has its origins in early work that tried to model networks of neurons in the brain (McCulloch and Pitts, 1943) with computational circuits. For this reason, the networks trained by deep learning methods are often called **neural networks**, even though the resemblance to real neural cells and structures is superficial.

While the true reasons for the success of deep learning have yet to be fully elucidated, it has self-evident advantages over some of the methods covered in Chapter 19—particularly for high-dimensional data such as images. For example, although methods such as linear and logistic regression can handle a large number of input variables, the computation path from each input to the output is very short: multiplication by a single weight, then adding into the aggregate output. Moreover, the different input variables contribute independently to the output, without interacting with each other (Figure 22.1(a)). This significantly limits the expressive power of such models. They can represent only linear functions and boundaries in the input space, whereas most real-world concepts are far more complex.

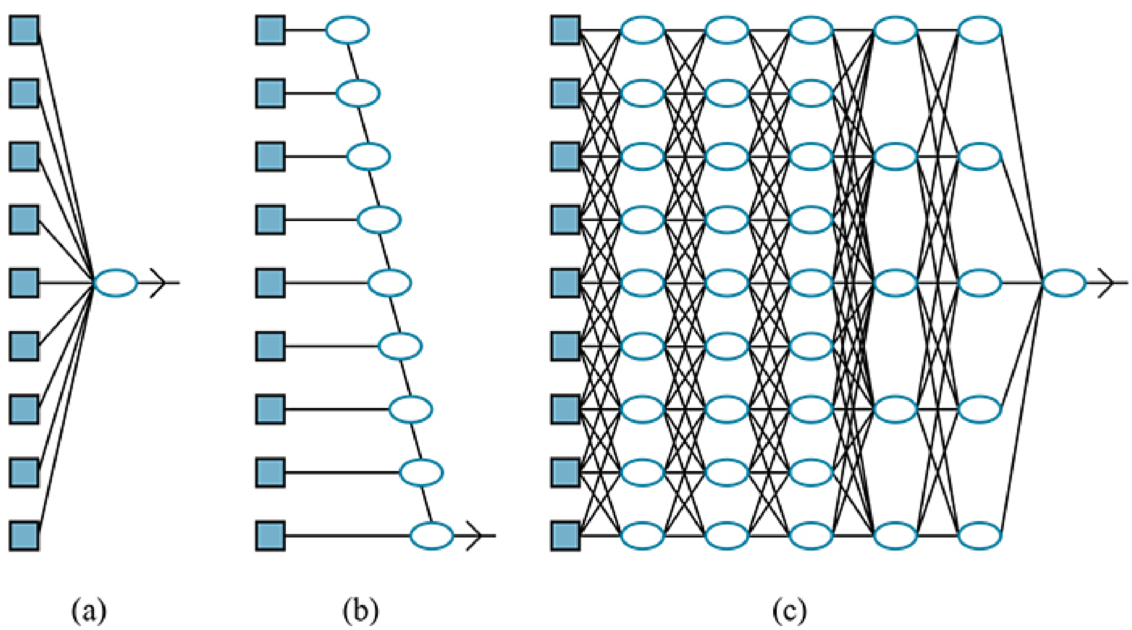


Figure 22.1 (a) A shallow model, such as linear regression, has short computation paths between inputs and output. (b) A decision list network (page 692) has some long paths for some possible input values, but most paths are short. (c) A deep learning

network has longer computation paths, allowing each variable to interact with all the others.

Decision lists and decision trees, on the other hand, allow for long computation paths that can depend on many input variables—but only for a relatively small fraction of the possible input vectors (Figure 22.1(b)). If a decision tree has long computation paths for a significant fraction of the possible inputs, it must be exponentially large in the number of input variables. The basic idea of deep learning is to train circuits such that the computation paths are long, allowing all the input variables to interact in complex ways (Figure 22.1(c)). These circuit models turn out to be sufficiently expressive to capture the complexity of real-world data for many important kinds of learning problems.

Section 22.1 describes simple feedforward networks, their components, and the essentials of learning in such networks. Section 22.2 goes into more detail on how deep networks are put together, and Section 22.3 covers a class of networks called convolutional neural networks that are especially important in vision applications. Sections 22.4 and 22.5 go into more detail on algorithms for training networks from data and methods for improving generalization. Section 22.6 covers networks with recurrent structure, which are well suited for sequential data. Section 22.7 describes ways to use deep learning for tasks other than supervised learning. Finally, Section 22.8 surveys the range of applications of deep learning.

22.1 Simple Feedforward Networks

A **feedforward network**, as the name suggests, has connections only in one direction—that is, it forms a directed acyclic graph with designated input and output nodes. Each node computes a function of its inputs and passes the result to its successors in the network. Information flows through the network from the input nodes to the output nodes, and there are no loops. A **recurrent network**, on the other hand, feeds its intermediate or final outputs back into its own inputs. This means that the signal values within the network form a dynamical system that has internal state or memory. We will consider recurrent networks in Section 22.6.

Boolean circuits, which implement Boolean functions, are an example of feedforward networks. In a Boolean circuit, the inputs are limited to 0 and 1, and each node implements a simple Boolean function of its inputs, producing a 0 or a 1. In neural networks, input values are typically continuous, and nodes take continuous inputs and produce continuous outputs. Some of the inputs to nodes are **parameters** of the network; the network learns by adjusting the values of these parameters so that the network as a whole fits the training data.

22.1.1 Networks as complex functions

Each node within a network is called a **unit**. Traditionally, following the design proposed by McCulloch and Pitts, a unit calculates the weighted sum of the inputs from predecessor nodes and then applies a nonlinear function to produce its output. Let a_j denote the output of unit j and let $w_{i,j}$ be the weight attached to the link from unit i to unit j; then we have

$$a_j = g_j\left(\sum_i w_{i,j}\, a_i\right) \equiv g_j(in_j) \ ,$$

where g_j is a nonlinear **activation function** associated with unit j and in_j is the weighted sum of the inputs to unit j.

As in Section 19.6.3 (page 697), we stipulate that each unit has an extra input from a dummy unit 0 that is fixed to +1 and a weight $w_{0,j}$ for that input. This allows the total weighted input in_j to unit j to be nonzero even when the outputs of the preceding layer are all zero. With this convention, we can write the preceding equation in vector form:

$$a_j = g_j\left(\mathbf{w}^\top \mathbf{x}\right) \qquad (22.1)$$

where $\mathbf{w}$ is the vector of weights leading into unit j (including $w_{0,j}$) and $\mathbf{x}$ is the vector of inputs to unit j (including the +1).

The fact that the activation function is nonlinear is important because if it were not, any composition of units would still represent a linear function. The nonlinearity is what allows sufficiently large networks of units to represent arbitrary functions. The **universal approximation** theorem states that a network with just two layers of computational units, the first nonlinear and the second linear, can approximate any continuous function to an arbitrary degree of accuracy. The proof works by showing that an exponentially large network can represent exponentially many "bumps" of different heights at different locations in the input space, thereby approximating the desired function. In other words, sufficiently large networks can implement a lookup table for continuous functions, just as sufficiently large decision trees implement a lookup table for Boolean functions.

A variety of different activation functions are used. The most common are the following:

- The logistic or **sigmoid** function, which is also used in logistic regression (see page 703):

  $\sigma(x) = 1/(1 + e^{-x})$.

- The **ReLU** function, whose name is an abbreviation for **rectified linear unit**:

  $\mathrm{ReLU}(x) = \max(0, x)$.

- The **softplus** function, a smooth version of the ReLU function:

  $\mathrm{softplus}(x) = \log(1+e^x)$.

  The derivative of the softplus function is the sigmoid function.

- The **tanh** function:

  $\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$.

  Note that the range of tanh is (–1, +1). Tanh is a scaled and shifted version of the sigmoid, as $\tanh(x) = 2\sigma(2x) - 1$.

These functions are shown in Figure 22.2. Notice that all of them are monotonically nondecreasing, which means that their derivatives g' are nonnegative. We will have more to say about the choice of activation function in later sections.

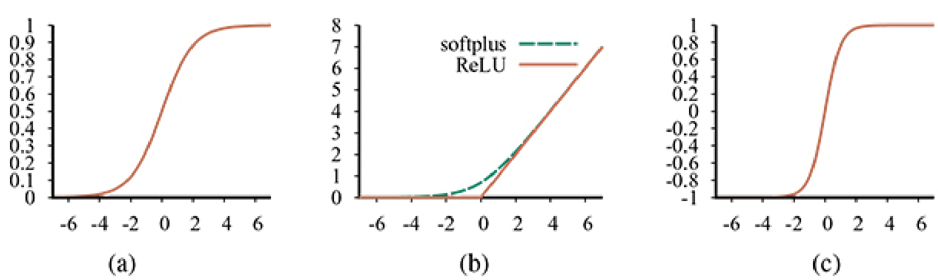


Figure 22.2 Activation functions commonly used in deep learning systems: (a) the logistic or sigmoid function; (b) the ReLU function and the softplus function; (c) the tanh function.

Coupling multiple units together into a network creates a complex function that is a composition of the algebraic expressions represented by the individual units. For example, the network shown in Figure 22.3(a) represents a function $h_{\mathbf{w}}(\mathbf{x})$, parameterized by weights $\mathbf{w}$, that maps a two-element input vector $\mathbf{x}$ to a scalar output value $\hat{y}$ The internal structure of the function mirrors the structure of the network. For example, we can write an expression for the output $\hat{y}$ as follows:

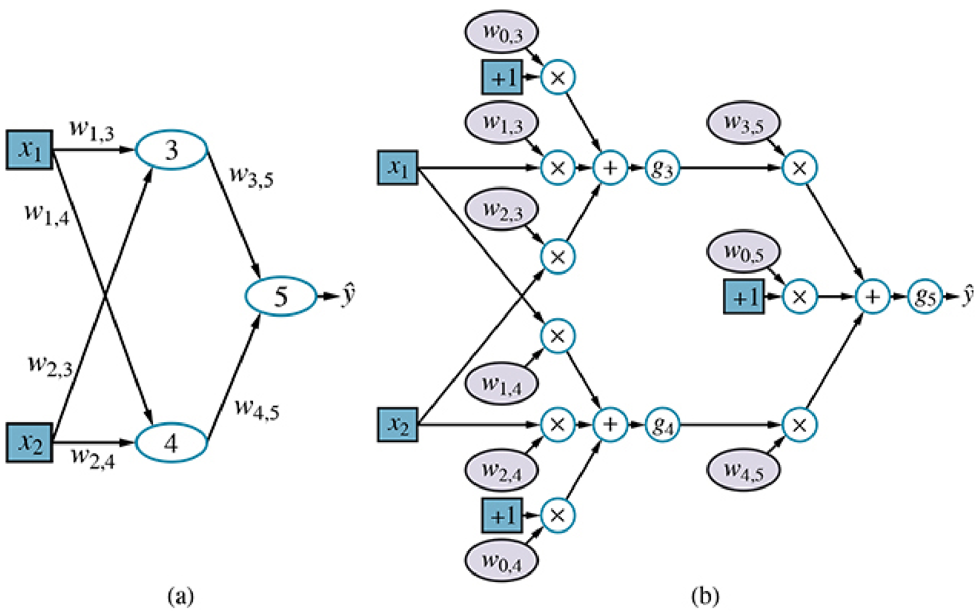


Figure 22.3 (a) A neural network with two inputs, one hidden layer of two units, and one output unit. Not shown are the dummy inputs and their associated weights. (b) The network in (a) unpacked into its full computation graph.

$$
\begin{aligned}
\hat{y} = g_5(in_5) \;&=\; g_5(w_{0,5} + w_{3,5}\, a_3 + w_{4,5}\, a_4) \\
&= g_5(w_{0,5} + w_{3,5}\, g_3(in_3) + w_4, 5\, g_4(in_4)) \\
&= g_5(w_{0,5} + w_{3,5}\, g_3(w_{0,3} + w_{1,3}\, x_1 + w_{2,3}\, x_2) \\
&\qquad + w_{4,5}\, g_4(w_{0,4} + w_{1,4}\, x_1 + w_{2,4}\, x_2)). \qquad (22.2)
\end{aligned}
$$

Thus, we have the output $\hat{y}$ expressed as a function $h_{\mathbf{w}}(\mathbf{x})$ of the inputs and the weights.

Figure 22.3(a) shows the traditional way a network might be depicted in a book on neural networks. A more general way to think about the network is as a **computation graph** or **dataflow graph**—essentially a circuit in which each node represents an elementary computation. Figure 22.3(b) shows the computation graph corresponding to the network in Figure 22.3(a); the graph makes each element of the overall computation explicit. It also distinguishes between the inputs (in blue) and the weights (in light mauve): the weights can be adjusted to make the output $\hat{y}$ agree more closely with the true value y in the training data. Each weight is like a volume control knob that determines how much the next node in the graph hears from that particular predecessor in the graph.

Just as Equation (22.1) described the operation of a unit in vector form, we can do something similar for the network as a whole. We will generally use $\mathbf{W}$ to denote a weight matrix; for this network, $\mathbf{W}^{(1)}$ denotes the weights in the first layer ($w_{1,3}$, $w_{1,4}$, etc.) and $\mathbf{W}^{(2)}$ denotes the weights in the second layer ($w_{3,5}$ etc.). Finally, let $\mathbf{g}^{(1)}$ and $\mathbf{g}^{(2)}$ denote the activation functions in the first and second layers. Then the entire network can be written as follows:

$$
h_{\mathbf{w}}(\mathbf{x}) \;=\; \mathbf{g}^{(2)}(\mathbf{W}^{(2)}\mathbf{g}^{(1)}(\mathbf{W}^{(1)}\mathbf{x})). \qquad (22.3)
$$

Like Equation (22.2), this expression corresponds to a computation graph, albeit a much simpler one than the graph in Figure 22.3(b): here, the graph is simply a chain with weight matrices feeding into each layer.

The computation graph in Figure 22.3(b) is relatively small and shallow, but the same idea applies to all forms of deep learning: we construct computation graphs and adjust their weights to fit the data. The graph in Figure 22.3(b) is also **fully connected**, meaning that every node in each layer is connected to every node in the next layer. This is in some sense the default, but we will see in Section 22.3 that choosing the connectivity of the network is also important in achieving effective learning.

22.1.2 Gradients and learning

In Section 19.6, we introduced an approach to supervised learning based on **gradient descent**: calculate the gradient of the loss function with respect to the weights, and adjust the weights along the gradient direction to reduce the loss. (If you have not already read Section 19.6, we recommend strongly that you do so before continuing.) We can apply exactly the same approach to learning the weights in computation graphs. For the weights leading into units in the **output layer**—the ones that produce the output of the network, the gradient calculation is essentially identical to the process in Section 19.6. For weights leading into units in the **hidden layers**, which are not directly connected to the outputs, the process is only slightly more complicated.

For now, we will use the squared loss function, L_2, and we will calculate the gradient for the network in Figure 22.3 with respect to a single training example $(\mathbf{x}, y)$. (For multiple examples, the gradient is just the sum of the gradients for the individual examples.) The network outputs a prediction $\hat{y} = h_{\mathbf{w}}(\mathbf{x})$ and the true value is y, so we have

$$Loss(h_{\mathbf{w}}) \;=\; L_2(y, h_{\mathbf{w}}(\mathbf{x})) \;=\; \| \, y - h_{\mathbf{w}}(\mathbf{x}) \, \|^2 \;=\; (y - \hat{y})^2 .$$

To compute the gradient of the loss with respect to the weights, we need the same tools of calculus we used in Chapter 19—principally the **chain rule**, $\partial g(f(x))/\partial x = g'(f(x))\partial f(x)/\partial x$. We'll start with the easy case: a weight such as $w_{3,5}$ that is connected to the output unit. We operate directly on the network-defining expressions from Equation (22.2):

$$\begin{aligned}
\frac{\partial}{\partial w_{3,5}} Loss(h_{\mathbf{w}}) &= \frac{\partial}{\partial w_{3,5}} (y - \hat{y})^2 - 2(y - \hat{y}) \frac{\partial \hat{y}}{\partial w_{3,5}} \\
&= -2(y - \hat{y}) \frac{\partial}{\partial w_{3,5}} g_5(in_5) = -2(y - \hat{y}) g_5'(in_5) \frac{\partial}{\partial w_{3,5}} in_5 \\
&= -2(y - \hat{y}) g_5'(in_5) \frac{\partial}{\partial w_{3,5}} (w_{0,5} + w_{3,5}\, a_3 + w_{4,5}\, a_4) \\
&= -2(y - \hat{y}) g_5'(in_5) a_3. \qquad (22.4)
\end{aligned}$$

The simplification in the last line follows because $w_{0,5}$ and $w_{4,5}a_4$ do not depend on $w_{3,5}$, nor does the coefficient of $w_{3,5}$, a_3.

The slightly more difficult case involves a weight such as $w_{1,3}$ that is not directly connected to the output unit. Here, we have to apply the chain rule one more time. The first few steps are identical, so we omit them:

$$\begin{aligned}
\frac{\partial}{\partial w_{1,3}} Loss(h_{\mathbf{w}}) &= -2(y - \hat{y}) g_5'(in_5) \frac{\partial}{\partial w_{1,3}} (w_{0,5} + w_{3,5}\, a_3 + w_{4,5}\, a_4) \\
&= -2(y - \hat{y}) g_5'(in_5) w_{3,5} \frac{\partial}{\partial w_{1,3}} a_3 \\
&= -2(y - \hat{y}) g_5'(in_5) w_{3,5} \frac{\partial}{\partial w_{1,3}} g_3(in_3) \\
&= -2(y - \hat{y}) g_5'(in_5) w_{3,5} g_3'(in_3) \frac{\partial}{\partial w_{1,3}} (in_3) \\
&= -2(y - \hat{y}) g_5'(in_5) w_{3,5} g_3'(in_3) \frac{\partial}{\partial w_{1,3}} (w_{0,3} + w_{1,3}\, x_1 + w_{2,3}\, x_2) \\
&= -2(y - \hat{y}) g_5'(in_5) w_{3,5} g_3'(in_3) x_1. \qquad (22.5)
\end{aligned}$$

So, we have fairly simple expressions for the gradient of the loss with respect to the weights $w_{3,5}$ and $w_{1,3}$.

If we define $\Delta_5 = 2(\hat{y} - y)g'_5(in_5)$ as a sort of "perceived error" at the point where unit 5 receives its input, then the gradient with respect to $w_{3,5}$ is just $\Delta_5 a_3$. This makes perfect sense: if Δ_5 is positive, that means $\hat{y}$ is too big (recall that g' is always nonnegative); if a_3 is also positive, then increasing $w_{3,5}$ will only make things worse, whereas if a_3 is negative, then increasing $w_{3,5}$ will reduce the error. The magnitude of a_3 also matters: if a_3 is small for this training example, then $w_{3,5}$ didn't play a major role in producing the error and doesn't need to be changed much.

If we also define $\Delta_3 = \Delta_5\, w_{3,5} g'_3(in_3)$, then the gradient for $w_{1,3}$ becomes just $\Delta_3 x_1$. Thus, the perceived error at the input to unit 3 is the perceived error at the input to unit 5, multiplied by information along the path from 5 back to 3. This phenomenon is completely general, and gives rise to the term **back-propagation** for the way that the error at the output is passed back through the network.

Another important characteristic of these gradient expressions is that they have as factors the local derivatives $g'_j\,(in_j)$. As noted earlier, these derivatives are always nonnegative, but they can be very close to zero (in the case of the sigmoid, softplus, and tanh functions) or exactly zero (in the case of ReLUs), if the inputs from the training example in question happen to put unit j in the flat operating region. If the derivative g'_j is small or zero, that means that changing the weights leading into unit j will have a negligible effect on its output. As a result, deep networks with many layers may suffer from a **vanishing gradient**—the error signals are extinguished altogether as they are propagated back through the network. Section 22.3.3 provides one solution to this problem.

We have shown that gradients in our tiny example network are simple expressions that can be computed by passing information back through the network from the output units. It turns out that this property holds more generally. In fact, as we show in Section 22.4.1, the gradient computations for *any* feedforward computation graph have the same structure as the underlying computation graph. This property follows straightforwardly from the rules of differentiation.

We have shown the gory details of a gradient calculation, but worry not: there is no need to redo the derivations in Equations (22.4) and (22.5) for each new network structure! All such gradients can be computed by the method of **automatic differentiation**, which applies the rules of calculus in a systematic way to calculate gradients for any numeric program.[1] In fact, the method of back-propagation in deep learning is simply an application of **reverse mode** differentiation, which applies the chain rule "from the outside in" and gains the efficiency advantages of dynamic programming when the network in question has many inputs and relatively few outputs.

All of the major packages for deep learning provide automatic differentiation, so that users can experiment freely with different network structures, activation functions, loss functions, and forms of composition without having to do lots of calculus to derive a new learning algorithm for each experiment. This has encouraged an approach called **end-to-end learning**, in which a complex computational system for a task such as machine translation can be composed from several trainable subsystems; the entire system is then trained in an end-to-end fashion from input/output pairs. With this approach, the designer need have only a vague idea about how the overall system should be structured; there is no need to know in advance exactly what each subsystem should do or how to label its inputs and outputs.

22.2 Computation Graphs for Deep Learning

We have established the basic ideas of deep learning: represent hypotheses as computation graphs with tunable weights and compute the gradient of the loss function with respect to those weights in order to fit the training data. Now we look at how to put together computation graphs. We begin with the input layer, which is where the training or test example $\mathbf{x}$ is encoded as values of the input nodes. Then we consider the output layer, where the outputs $\hat{\mathbf{y}}$ are compared with the true values $\mathbf{y}$ to derive a learning signal for tuning the weights. Finally, we look at the hidden layers of the network.

22.2.1 Input encoding

The input and output nodes of a computational graph are the ones that connect directly to the input data $\mathbf{x}$ and the output data $\mathbf{y}$. The encoding of input data is usually straightforward, at least for the case of factored data where each training example contains values for n input attributes. If the attributes are Boolean, we have n input nodes; usually *false* is mapped to an input of 0 and *true* is mapped to 1, although sometimes –1 and +1 are used. Numeric attributes, whether integer or real-valued, are typically used as is, although they may be scaled to fit within a fixed range; if the magnitudes for different examples vary enormously, the values can be mapped onto a log scale.

Images do not quite fit into the category of factored data; although an RGB image of size $X \times Y$ pixels can be thought of as $3XY$ integer-valued attributes (typically with values in the range $\{0,...,255\}$), this would ignore the fact that the RGB triplets belong to the same pixel in the image and the fact that pixel adjacency really matters. Of course, we can map adjacent pixels onto adjacent input nodes in the network, but the meaning of adjacency is completely lost if the internal layers of the network are fully connected. In practice, networks used with image data have array-like internal structures that aim to reflect the semantics of adjacency. We will see this in more detail in Section 22.3.

Categorical attributes with more than two values—like the *Type* attribute in the restaurant problem from Chapter 19, which has values French, Italian, Thai, or burger)—are usually encoded using the so-called **one-hot encoding**. An attribute with d possible values is represented by d separate input bits. For any given value, the corresponding input bit is set to 1 and all the others are set to 0. This generally works better than mapping the values to integers. If we used integers for the *Type* attribute, Thai would be 3 and burger would be 4. Because the network is a composition of continuous functions, it would have no choice but to pay attention to numerical adjacency, but in this case the numerical adjacency between Thai and burger is semantically meaningless.

22.2.2 Output layers and loss functions

On the output side of the network, the problem of encoding the raw data values into actual values y for the output nodes of the graph is much the same as the input encoding problem. For example, if the network is trying to predict the *Weather* variable from Chapter 12, which has values $\{sun, rain, cloud, snow\}$, we would use a one-hot encoding with four bits.

So much for the data values $\mathbf{y}$. What about the prediction $\hat{\mathbf{y}}$? Ideally, it would exactly match the desired value $\mathbf{y}$, and the loss would be zero, and we'd be done. In practice, this seldom happens–especially before we have started the process of adjusting the weights! Thus, we need to think about what an incorrect output value means, and how to measure the loss. In deriving the gradients in Equations (22.4) and (22.5), we began with the squared-error loss function. This keeps the algebra simple, but it is not the only possibility. In fact, for most deep learning applications, it is more common to interpret the output values $\hat{\mathbf{y}}$ as probabilities and to use the **negative log likelihood** as the loss function—exactly as we did with **maximum likelihood** learning in Chapter 21.

Maximum likelihood learning finds the value of **w** that maximizes the probability of the observed data. And because the log function is monotonic, this is equivalent to maximizing the log likelihood of the data, which is equivalent in turn to minimizing a loss function defined as the negative log likelihood. (Recall from page 776 that taking logs turns products of probabilities into sums, which are more amenable for taking derivatives.) In other words, we are looking for $\mathbf{w}^*$ that minimizes the sum of negative log probabilities of the N examples:

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \; - \sum_{j=1}^{N} \log P_{\mathbf{w}}(y_j|\mathbf{x}_j). \tag{22.6}$$

In the deep learning literature, it is common to talk about minimizing the **cross-entropy loss**. Cross-entropy, written as $H(P, Q)$, is a kind of measure of dissimilarity between two distributions P and Q.[2] The general definition is

$$H(P,Q) = \mathbf{E}_{z\sim P(\mathbf{z})}[\log Q(\mathbf{z})] = \int P(\mathbf{z}) \log Q(\mathbf{z}) d\mathbf{z}. \tag{22.7}$$

In machine learning, we typically use this definition with P being the true distribution over training examples, $P^*(\mathbf{x}, \mathbf{y})$, and Q being the predictive hypothesis $P_{\mathbf{w}}(\mathbf{y} \mid \mathbf{x})$. Minimizing the cross-entropy $H(P^*(\mathbf{x}, \mathbf{y}), P_{\mathbf{w}}(\mathbf{y} \mid \mathbf{x}))$ by adjusting **w** makes the hypothesis agree as closely as possible with the true distribution. In reality, we cannot minimize this cross-entropy because we do not have access to the true data distribution $P^*(\mathbf{x}, \mathbf{y})$; but we do have access to samples from $P^*(\mathbf{x}, \mathbf{y})$, so the sum over the actual data in Equation (22.6) approximates the expectation in Equation (22.7).

To minimize the negative log likelihood (or the cross-entropy), we need to be able to interpret the output of the network as a probability. For example, if the network has one output unit with a sigmoid activation function and is learning a Boolean classification, we can interpret the output value directly as the probability that the example belongs to the positive class. (Indeed, this is exactly how logistic regression is used; see page 702.) Thus, for Boolean classification problems, we commonly use a sigmoid output layer.

Multiclass classification problems are very common in machine learning. For example, classifiers used for object recognition often need to recognize thousands of distinct categories of objects. Natural language models that try to predict the next word in a sentence may have to choose among tens of thousands of possible words. For this kind of prediction, we need the network to output a categorical distribution—that is, if there are d possible answers, we need d output nodes that represent probabilities summing to 1.

To achieve this, we use a **softmax** layer, which outputs a vector of d values given a vector of input values $\mathbf{in} = \langle in_1,\ldots,in_d \rangle$. The kth element of that output vector is given by

$$\text{softmax}(\mathbf{in})_k = \frac{e^{in_k}}{\sum_{k'=1}^{d} e^{in_{k'}}}.$$

By construction, the softmax function outputs a vector of nonnegative numbers that sum to 1. As usual, the input in_k to each of the output nodes will be a weighted linear combination of the outputs of the preceding layer. Because of the exponentials, the softmax layer accentuates differences in the inputs: for example, if the vector of inputs is given by $\mathbf{in} = \langle 5, 2, 0, -2 \rangle$, then the outputs are $\langle 0.946, 0.074, 0.006, 0.001 \rangle$. The softmax, is, nonetheless, smooth and differentiable (Exercise 22.SOFG), unlike the max function. It is easy to show (Exercise 22.SMSG) that the sigmoid is a softmax with $d = 2$. In other words, just as sigmoid units propagate binary class information through a network, softmax units propagate multiclass information.

For a regression problem, where the target value y is continuous, it is common to use a linear output layer—in other words, $\hat{y}_j = in_j$, without any activation function g—and to interpret this as the mean of a Gaussian prediction with fixed variance. As we noted on page 780, maximizing likelihood (i.e., minimizing negative log likelihood) with a fixed-variance Gaussian is the same as minimizing squared error. Thus, a linear output layer interpreted in this way does classical linear regression. The input features to this linear regression are the outputs from the preceding layer, which typically result from multiple nonlinear transformations of the original inputs to the network.

Many other output layers are possible. For example, a **mixture density** layer represents the outputs using a mixture of Gaussian distributions. (See Section 21.3.1 for more details on Gaussian mixtures.) Such layers predict the relative frequency of each mixture component, the mean of each component, and the variance of each component. As long as these output values are interpreted appropriately by the loss function as defining the probability for the true output value $\mathbf{y}$, the network will, after training, fit a Gaussian mixture model in the space of features defined by the preceding layers.

22.2.3 Hidden layers

During the training process, a neural network is shown many input values $\mathbf{x}$ and many corresponding output values y. While processing an input vector $\mathbf{x}$, the neural network performs several intermediate computations before producing the output y. We can think of the values computed at each layer of the network as a different *representation* for the input $\mathbf{x}$. Each layer transforms the representation produced by the preceding layer to produce a new representation. The composition of all these transformations succeeds—if all goes well—in transforming the input into the desired output. Indeed, one hypothesis for why deep learning works well is that the complex end-to-end transformation that maps from input to output—say, from an input image to the output category "giraffe"—is decomposed by the many layers into the composition of many relatively simple transformations, each of which is fairly easy to learn by a local updating process.

In the process of forming all these internal transformations, deep networks often discover meaningful intermediate representations of the data. For example, a network learning to recognize complex objects in images may form internal layers that detect useful subunits: edges, corners, ellipses, eyes, faces—cats. Or it may not—deep networks may form internal layers whose meaning is opaque to humans, even though the output is still correct.

The hidden layers of neural networks are typically less diverse than the output layers. For the first 25 years of research with multilayer networks (roughly 1985–2010), internal nodes used sigmoid and tanh activation functions almost exclusively. From around 2010 onwards, the ReLU and softplus become more popular, partly because they are believed to avoid the problem of vanishing gradients mentioned in Section 22.1.2. Experimentation with increasingly deep networks suggested that, in many cases, better learning was obtained with deep and relatively narrow networks rather than shallow, wide networks, given a fixed total number of weights. A typical example of this is shown in Figure 22.7 on page 820.

There are, of course, many other structures to consider for computation graphs, besides just playing with width and depth. At the time of writing, there is little understanding as to why some structures seem to work better than others for some particular problem. With experience, practitioners gain some intuition as to how to design networks and how to fix them when they don't work, just as chefs gain intuition for how to design recipes and how to fix them when they taste unpleasant. For this reason, tools that facilitate rapid exploration and evaluation of different structures are essential for success in real-world problems.

22.3 Convolutional Networks

We mentioned in Section 22.2.1 that an image cannot be thought of as a simple vector of input pixel values, primarily because adjacency of pixels really matters. If we were to construct a network with fully connected layers and an image as input, we would get the same result whether we trained with unperturbed images or with images all of whose pixels had been randomly permuted. Furthermore, suppose there are n pixels and n units in the first hidden layer, to which the pixels provide input. If the input and the first hidden layer are fully connected, that means n^2 weights; for a typical megapixel RGB image, that's 9 trillion weights. Such a vast parameter space would require correspondingly vast numbers of training images and a huge computational budget to run the training algorithm.

These considerations suggest that we should construct the first hidden layer so that *each hidden unit receives input from only a small, local region of the image.* This kills two birds with one stone. First, it respects adjacency, at least locally. (And we will see later that if subsequent layers have the same locality property, then the network will respect adjacency in a global sense.) Second, it cuts down the number of weights: if each local region has $l \ll n$ pixels, then there will be $ln \ll n^2$ weights in all.

So far, so good. But we are missing another important property of images: roughly speaking, anything that is detectable in one small, local region of the image—perhaps an eye or a blade of grass—would look the same if it appeared in another small, local region of the image. In other words, we expect image data to exhibit approximate **spatial invariance**, at least at small to moderate scales.[3] We don't necessarily expect the top halves of photos to look like bottom halves, so there is a scale beyond which spatial invariance no longer holds.

Local spatial invariance can be achieved by constraining the l weights connecting a local region to a unit in the hidden layer to be the same for each hidden unit. (That is, for hidden units i and j, the weights $w_{1,i},\ldots,w_{l,i}$ are the same as $w_{1,j},\ldots, w_{l,j}$.) This makes the hidden units into feature detectors that detect the same feature wherever it appear in the image. Typically, we want the first hidden layer to detect many kinds of features, not just one; so for each local image region we might have d hidden units with d distinct sets of weights. This means that there are dl weights in all—a number that is not only far smaller than n^2, but is actually independent of n, the image size. Thus, by injecting some prior knowledge—namely, knowledge of adjacency and spatial invariance—we can develop models that have far fewer parameters and can learn much more quickly.

A **convolutional neural network (CNN)** is one that contains spatially local connections, at least in the early layers, and has patterns of weights that are replicated across the units in each layer. A pattern of weights that is replicated across multiple local regions is called a kernel and the process of applying the **kernel** to the pixels of the image (or to spatially organized units in a subsequent layer) is called **convolution**.[4]

Kernels and convolutions are easiest to illustrate in one dimension rather than two or more, so we will assume an input vector $\mathbf{x}$ of size n, corresponding to n pixels in a one-dimensional image, and a vector kernel $\mathbf{k}$ of size l. (For simplicity we will assume that l is an odd number.) All the ideas carry over straightforwardly to higher-dimensional cases.

We write the convolution operation using the $*$ symbol, for example: $\mathbf{z} = \mathbf{x} * \mathbf{k}$. The operation is defined as follows:

$$z_i = \sum_{j=1}^{l} k_j x_{j+i-(l+1)/2} \,. \qquad (22.8)$$

In other words, for each output position i, we take the dot product between the kernel $\mathbf{k}$ and a snippet of $\mathbf{x}$ centered on x_i with width l.

The process is illustrated in Figure 22.4 for a kernel vector [+1, –1, +1], which detects a darker point in the 1D image. (The 2D version might detect a darker line.) Notice that in this example the pixels on which the kernels are centered are separated by a distance of 2 pixels; we say the kernel is applied with a **stride** $s = 2$. Notice that the output layer has fewer pixels: because of the stride, the number of pixels is reduced from n to roughly n/s. (In two dimensions, the number of pixels would be roughly $n/s_x s_y$, where s_x and s_y are the strides in the x and y directions in the image.) We say “roughly” because of what happens at the edge of the image: in Figure 22.4 the convolution stops at the edges of the image, but one can also pad the input with extra pixels (either zeroes or copies of the outer pixels) so that the kernel can be applied exactly $\lceil n/s \rceil$ times. For small kernels, we typically use $s = 1$, so the output has the same dimensions as the image (see Figure 22.5).

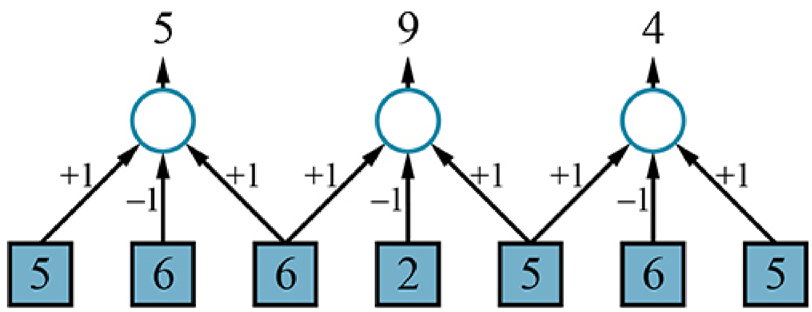


Figure 22.4 An example of a one-dimensional convolution operation with a kernel of size $l = 3$ and a stride $s = 2$. The peak response is centered on the darker (lower intensity) input pixel. The results would usually be fed through a nonlinear activation function (not shown) before going to the next hidden layer.

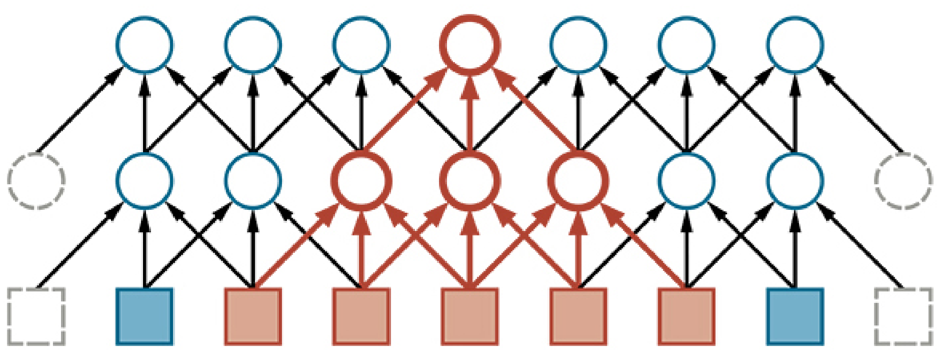


Figure 22.5 The first two layers of a CNN for a 1D image with a kernel size $l = 3$ and a stride $s = 1$. Padding is added at the left and right ends in order to keep the hidden layers the same size as the input. Shown in red is the receptive field of a unit in the second hidden layer. Generally speaking, the deeper the unit, the larger the receptive field.

The operation of applying a kernel across an image can be implemented in the obvious way by a program with suitable nested loops; but it can also be formulated as a single matrix operation, just like the application of the weight matrix in Equation (22.1). For example, the convolution illustrated in Figure 22.4 can be viewed as the following matrix multiplication:

$$\begin{pmatrix} +1 & -1 & +1 & 0 & 0 & 0 & 0 \\ 0 & 0 & +1 & -1 & +1 & 0 & 0 \\ 0 & 0 & 0 & 0 & +1 & -1 & +1 \end{pmatrix} \begin{pmatrix} 5 \\ 6 \\ 6 \\ 2 \\ 5 \\ 6 \\ 5 \end{pmatrix} = \begin{pmatrix} 5 \\ 9 \\ 4 \end{pmatrix}. \tag{22.9}$$

In this weight matrix, the kernel appears in each row, shifted according to the stride relative to the previous row, One wouldn't necessarily construct the weight matrix explicitly—it is mostly zeroes, after all—but the fact that convolution is a linear matrix operation serves as a reminder that gradient descent can be applied easily and effectively to CNNs, just as it can to plain vanilla neural networks.

As mentioned earlier, there will be d kernels, not just one; so, with a stride of 1, the output will be d times larger. This means that a two-dimensional input array becomes a three-dimensional array of hidden units, where the third dimension is of size d. It is important to organize the hidden layer this way, so that all the kernel outputs from a particular image location stay associated with that location. Unlike the spatial dimensions of the image, however, this additional "kernel dimension" does not have any adjacency properties, so it does not make sense to run convolutions along it.

CNNs were inspired originally by models of the visual cortex proposed in neuroscience. In those models, the **receptive field** of a neuron is the portion of the sensory input that can affect that neuron's activation. In a CNN, the receptive field of a unit in the first hidden layer is small—just the size of the kernel, i.e., l pixels. In the deeper layers of the network, it can be much larger. Figure 22.5 illustrates this for a unit in the second hidden layer, whose receptive field contains five pixels. When the stride is 1, as in the figure, a node in the mth hidden layer will have a receptive field of size $(l - 1)m + 1$; so the growth is linear in m. (In a 2D image, each dimension of the receptive field grows linearly with m, so the area grows quadratically.) When the stride is larger than 1, each pixel in layer m represents s pixels in layer $m - 1$; therefore, the receptive field grows as $O(ls^m)$—that is, exponentially with depth. The same effect occurs with pooling layers, which we discuss next.

22.3.1 Pooling and downsampling

A **pooling** layer in a neural network summarizes a set of adjacent units from the preceding layer with a single value. Pooling works just like a convolution layer, with a kernel size l and stride s, but the operation that is applied is fixed rather than learned. Typically, no activation function is associated with the pooling layer. There are two common forms of pooling:

- Average-pooling computes the average value of its l inputs. This is identical to convolution with a uniform kernel vector $\mathbf{k} = [1/l, \ldots, 1/l]$. If we set $l = s$, the effect is to coarsen the resolution of the image—to **downsample** it—by a factor of s. An object that occupied, say, $10s$ pixels would now occupy only 10 pixels after pooling. The same learned classifier that would be able to recognize the object at a size of 10 pixels in the original image would now be able to recognize that object in the pooled image, even if it was too big to recognize in the original image. In other words, average-pooling facilitates multiscale recognition. It also reduces the number of weights required in subsequent layers, leading to lower computational cost and possibly faster learning.
- Max-pooling computes the maximum value of its l inputs. It can also be used purely for downsampling, but it has a somewhat different semantics. Suppose we applied maxpooling to the hidden layer [5,9,4] in Figure 22.4: the result would be a 9, indicating that somewhere in the input image there is a darker dot that is detected by the kernel. In other words, max-pooling acts as a kind of logical disjunction, saying that a feature exists somewhere in the unit's receptive field.

If the goal is to classify the image into one of c categories, then the final layer of the network will be a softmax with c output units. The early layers of the CNN are image-sized, so somewhere in between there must be significant reductions in layer size. Convolution layers and pooling layers with stride larger than 1 all serve to reduce the layer size. It's also possible to reduce the layer size simply by having a fully connected layer with fewer units than the preceding layer. CNNs often have one or two such layers preceding the final softmax layer.

22.3.2 Tensor operations in CNNs

We saw in Equations (22.1) and (22.3) that the use of vector and matrix notation can be helpful in keeping mathematical derivations simple and elegant and providing concise descriptions of computation graphs. Vectors and matrices are one-dimensional and two-dimensional special cases of **tensors**, which (in deep learning terminology) are simply multidimensional arrays of any dimension.[5]

For CNNs, tensors are a way of keeping track of the "shape" of the data as it progresses through the layers of the network. This is important because the whole notion of convolution depends on the idea of adjacency: adjacent data elements are assumed to be semantically related, so it makes sense to apply operators to local regions of the data. Moreover, with suitable language primitives for constructing tensors and applying operators, the layers themselves can be described concisely as maps from tensor inputs to tensor outputs.

A final reason for describing CNNs in terms of tensor operations is computational efficiency: given a description of a network as a sequence of tensor operations, a deep learning software package can generate compiled code that is highly optimized for the underlying computational substrate. Deep learning workloads are often run on GPUs (graphics processing units) or TPUs (tensor processing units), which make available a high degree of parallelism. For example, one of Google's third-generation TPU pods has throughput equivalent to about ten million laptops. Taking advantage of these capabilities is essential if one is training a large CNN on a large database of images. Thus, it is common to process not one image at a time but many images in parallel; as we will see in Section 22.4, this also aligns nicely with the way that the stochastic gradient descent algorithm calculates gradients with respect to a minibatch of training examples.

Let us put all this together in the form of an example. Suppose we are training on 256×256 RGB images with a minibatch size of 64. The input in this case will be a four-dimensional tensor of size $256 \times 256 \times 3 \times 64$. Then we apply 96 kernels of size $5 \times 5 \times 3$ with a stride of 2 in both x and y directions in the image. This gives an output tensor of size $128 \times 128 \times 96 \times 64$. Such a tensor is often called a **feature map,** since it shows how each feature extracted by a kernel appears across the entire image; in this case it is composed of 96 **channels**, where each channel carries information from one feature. Notice that unlike the input tensor, this feature map no longer has dedicated color channels; nonetheless, the color information may still be present in the various feature channels if the learning algorithm finds color to be useful for the final predictions of the network.

22.3.3 Residual networks

Residual networks are a popular and successful approach to building very deep networks that avoid the problem of vanishing gradients.

Typical deep models use layers that learn a new representation at layer i by completely replacing the representation at layer $i - 1$. Using the matrix–vector notation that we introduced in Equation (22.3), with $\mathbf{z}^{(i)}$ being the values of the units in layer i, we have

$$\mathbf{z}^{(i)} = f(\mathbf{z}^{(i-1)}) = \mathbf{g}^{(i)}(\mathbf{W}^{(i)}\mathbf{z}^{(i-1)}).$$

Because each layer completely replaces the representation from the preceding layer, all of the layers must learn to do something useful. Each layer must, at the very least, preserve the task-relevant information contained in the preceding layer. If we set $\mathbf{W}^{(i)} = \mathbf{0}$ for any layer i, the entire network ceases to function. If we also set $\mathbf{W}^{(i-1)} = \mathbf{0}$, the network would not even be able to learn: layer i would not learn because it would observe no variation in its input from layer $i-1$, and layer $i-1$ would not learn because the back-propagated gradient from layer i would always be zero. Of course, these are extreme examples, but they illustrate the need for layers to serve as conduits for the signals passing through the network.

The key idea of residual networks is that a layer should *perturb* the representation from the previous layer rather than *replace* it entirely. If the learned perturbation is small, the next layer is close to being a copy of the previous layer. This is achieved by the following equation for layer i in terms of layer $i-1$:

$$\mathbf{z}^{(i)} = \mathbf{g}_r{}^{(i)}(\mathbf{z}^{(i-1)} + f(\mathbf{z}^{(i-1)})). \qquad (22.10)$$

where $\mathbf{g}_r$ denotes the activation functions for the residual layer. Here we think of f as the **residual**, perturbing the default behavior of passing layer $i-1$ through to layer i. The function used to compute the residual is typically a neural network with one nonlinear layer combined with one linear layer:

$$f(\mathbf{z}) = \mathbf{V}\,\mathbf{g}(\mathbf{W}\mathbf{z}),$$

where $\mathbf{W}$ and $\mathbf{V}$ are learned weight matrices with the usual bias weights added.

Residual networks make it possible to learn significantly deeper networks reliably. Consider what happens if we set $\mathbf{V} = \mathbf{0}$ for a particular layer in order to disable that layer. Then the residual f disappears and Equation (22.10) simplifies to

$$\mathbf{z}^{(i)} = \mathbf{g}_r(\mathbf{z}^{(i-1)}).$$

Now suppose that $\mathbf{g}_r$ consists of ReLU activation functions and that $\mathbf{z}^{(i-1)}$ also applies a ReLU function to its inputs: $\mathbf{z}^{(i-1)} = \mathrm{ReLU}(\mathbf{in}^{(i-1)})$. In that case we have

$$\mathbf{z}^{(i)} = \mathbf{g}_r(\mathbf{z}^{(i-1)}) = \mathrm{ReLU}(\mathbf{z}^{(i-1)}) = \mathrm{ReLU}\left(\mathrm{ReLU}\left(\mathbf{in}^{(i-1)}\right)\right) = \mathrm{Re\,LU}(\mathbf{in}^{(i-1)}) = \mathbf{z}^{(i-1)},$$

where the penultimate step follows because ReLU(ReLU(x)) = ReLU(x). In other words, in residual nets with ReLU activations, a layer with zero weights simply passes its inputs through with no change. The rest of the network functions just as if the layer had never existed. Whereas traditional networks must *learn* to propagate information and are subject to catastrophic failure of information propagation for bad choices of the parameters, residual networks propagate information by default.

Residual networks are often used with convolutional layers in vision applications, but they are in fact a general-purpose tool that makes deep networks more robust and allows researchers to experiment more freely with complex and heterogeneous network designs. At the time of writing, it is not uncommon to see residual networks with hundreds of layers. The design of such networks is evolving rapidly, so any additional specifics we might provide would probably be outdated before reaching printed form. Readers desiring to know the best architectures for specific applications should consult recent research publications.

22.4 Learning Algorithms

Training a neural network consists of modifying the network's parameters so as to minimize the loss function on the training set. In principle, any kind of optimization algorithm could be used. In practice, modern neural networks are almost always trained with some variant of stochastic gradient descent (SGD).

We covered standard gradient descent and its stochastic version in Section 19.6.2. Here, the goal is to minimize the loss $L(\mathbf{w})$, where $\mathbf{w}$ represents all of the parameters of the network. Each update step in the gradient descent process looks like this:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} L(\mathbf{w}),$$

where α is the learning rate. For standard gradient descent, the loss L is defined with respect to the entire training set. For SGD, it is defined with respect to a minibatch of m examples chosen randomly at each step.

As noted in Section 4.2, the literature on optimization methods for high-dimensional continuous spaces includes innumerable enhancements to basic gradient descent. We will not cover all of them here, but it is worth mentioning a few important considerations that are particularly relevant to training neural networks:

- For most networks that solve real-world problems, both the dimensionality of $\mathbf{w}$ and the size of the training set are very large. These considerations militate strongly in favor of using SGD with a relatively small minibatch size m: stochasticity helps the algorithm escape small local minima in the high-dimensional weight space (as in simulated annealing—see page 132); and the small minibatch size ensures that the computational cost of each weight update step is a small constant, independent of the training set size.

- Because the gradient contribution of each training example in the SGD minibatch can be computed independently, the minibatch size is often chosen so as to take maximum advantage of hardware parallelism in GPUs or TPUs.
- To improve convergence, it is usually a good idea to use a learning rate that decreases over time. Choosing the right schedule is usually a matter of trial and error.
- Near a local or global minimum of the loss function with respect to the entire training set, the gradients estimated from small minibatches will often have high variance and may point in entirely the wrong direction, making convergence difficult. One solution is to increase the minibatch size as training proceeds; another is to incorporate the idea of **momentum**, which keeps a running average of the gradients of past minibatches in order to compensate for small minibatch sizes.
- Care must be taken to mitigate numerical instabilities that may arise due to overflow, underflow, and rounding error. These are particularly problematic with the use of exponentials in softmax, sigmoid, and tanh activation functions, and with the iterated computations in very deep networks and recurrent networks (Section 22.6) that lead to vanishing and exploding activations and gradients.

Overall, the process of learning the weights of the network is usually one that exhibits diminishing returns. We run until it is no longer practical to decrease the test error by running longer. Usually this does not mean we have reached a global or even a local minimum of the loss function. Instead, it means we would have to make an impractically large number of very small steps to continue reducing the cost, or that additional steps would only cause overfitting, or that estimates of the gradient are too inaccurate to make further progress.

22.4.1 Computing gradients in computation graphs

On page 806, we derived the gradient of the loss function with respect to the weights in a specific (and very simple) network. We observed that the gradient could be computed by back-propagating error information from the output layer of the network to the hidden layers. We also said that this result holds in general for any feedforward computation graph. Here, we explain how this works.

Figure 22.6 shows a generic node in a computation graph. (The node h has in-degree and out-degree 2, but nothing in the analysis depends on this.) During the forward pass, the node computes some arbitrary function h from its inputs, which come from nodes f and g. In turn, h feeds its value to nodes j and k.

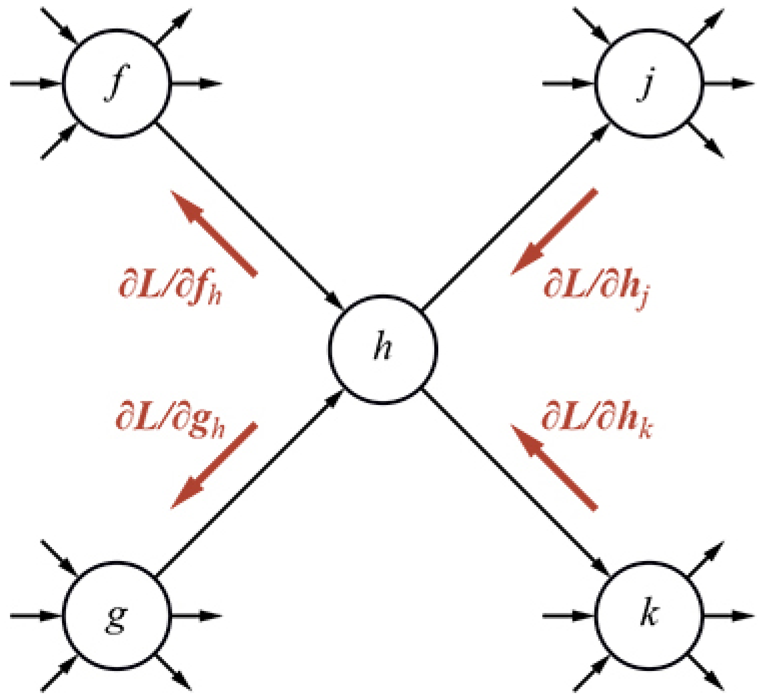


Figure 22.6 Illustration of the back-propagation of gradient information in an arbitrary computation graph. The forward computation of the output of the network proceeds from left to right, while the back-propagation of gradients proceeds from right to left.

The back-propagation process passes messages back along each link in the network. At each node, the incoming messages are collected and new messages are calculated to pass back to the next layer. As the figure shows, the messages are all partial derivatives of the loss L. For example, the backward message $\partial L/\partial h_j$ is the partial derivative of L with respect to j's first input, which is the forward message from h to j. Now, h affects L through both j and k, so we have

$$\partial L/\partial h = \partial L/\partial h_j + \partial L/\partial h_k. \tag{22.11}$$

With this equation, the node h can compute the derivative of L with respect to h by summing the incoming messages from j and k. Now, to compute the outgoing messages $\partial L/\partial f_h$ and $\partial L/\partial g_h$, we use the following equations:

$$\frac{\partial L}{\partial f_h} = \frac{\partial L}{\partial h}\frac{\partial h}{\partial f_h} \qquad \text{and} \qquad \frac{\partial L}{\partial g_h} = \frac{\partial L}{\partial h}\frac{\partial h}{\partial g_h}. \tag{22.12}$$

In Equation (22.12), $\partial L/\partial h$ was already computed by Equation (22.11), and $\partial L/\partial f_h$ and $\partial L/\partial g_h$ are just the derivatives of h with respect to its first and second arguments, respectively. For example, if h is a multiplication node—that is, $h(f, g) = f \cdot g$—then $\partial h/\partial f_h = h$ and $\partial h/\partial g_h = f$. Software packages for deep learning typically come with a library of node types (addition, multiplication, sigmoid, and so on), each of which knows how to compute its own derivatives as needed for Equation (22.12).

The back-propagation process begins with the output nodes, where each initial message $\partial L/\partial \hat{y}_j$ is calculated directly from the expression for L in terms of the predicted value $\hat{\mathbf{y}}$ and the true value $\mathbf{y}$ from the training data. At each internal node, the incoming backward messages are summed according to Equation (22.11) and the outgoing messages are generated from Equation (22.12). The process terminates at each node in the computation graph that represents a weight w (e.g., the light mauve ovals in Figure 22.3(b)). At that point, the sum of the incoming messages to w is $\partial L/\partial w$—precisely the gradient we need to update w. Exercise 22.BPRE asks you to apply this process to the simple network in Figure 22.3 in order to rederive the gradient expressions in Equations (22.4) and (22.5).

Weight-sharing, as used in convolutional networks (Section 22.3) and recurrent networks (Section 22.6), is handled simply by treating each shared weight as a single node with multiple outgoing arcs in the computation graph. During back-propagation, this results in multiple incoming gradient messages. By Equation (22.11), this means that the gradient for the shared weight is the sum of the gradient contributions from each place it is used in the network.

It is clear from this description of the back-propagation process that its computational cost is linear in the number of nodes in the computation graph, just like the cost of the forward computation. Furthermore, because the node types are typically fixed when the network is designed, all of the gradient computations can be prepared in symbolic form in advance and compiled into very efficient code for each node in the graph. Note also that the messages in Figure 22.6 need not be scalars: they could equally be vectors, matrices, or higherdimensional tensors, so that the gradient computations can be mapped onto GPUs or TPUs to benefit from parallelism.

One drawback of back-propagation is that it requires storing most of the intermediate values that were computed during forward propagation in order to calculate gradients in the backward pass. This means that the total memory cost of training the network is proportional to the number of units in the entire network. Thus, even if the network itself is represented only implicitly by propagation code with lots of loops, rather than explicitly by a data structure, all of the intermediate results of that propagation code have to be stored explicitly.

22.4.2 Batch normalization

Batch normalization is a commonly used technique that improves the rate of convergence of SGD by rescaling the values generated at the internal layers of the network from the examples within each minibatch. Although the reasons for its effectiveness are not well understood at the time of writing, we include it because it confers significant benefits in practice. To some extent, batch normalization seems to have effects similar to those of the residual network.

Consider a node z somewhere in the network: the values of z for the m examples in a minibatch are $z_1,\ldots, z_m$. Batch normalization replaces each z_i with a new quantity $\hat{z}_i$:

$$\hat{z}_i = \gamma \frac{z_i - \mu}{\sqrt{\epsilon + \sigma^2}} + \beta,$$

where μ is the mean value of z across the minibatch, σ is the standard deviation of $z_1,\ldots, z_m$, ϵ is a small constant added to prevent division by zero, and γ and β are learned parameters.

Batch normalization standardizes the mean and variance of the values, as determined by the values of β and γ. This makes it much simpler to train a deep network. Without batch normalization, information can get lost if a layer's weights are too small, and the standard deviation at that layer decays to near zero. Batch normalization prevents this from happening. It also reduces the need for careful initialization of all the weights in the network to make sure that the nodes in each layer are in the right operating region to allow information to propagate.

With batch normalization, we usually include β and γ, which may be node-specific or layer-specific, among the parameters of the network, so that they are included in the learning process. After training, β and γ are fixed at their learned values.

22.5 Generalization

So far we have described how to fit a neural network to its training set, but in machine learning the goal is to generalize to new data that has not been seen previously, as measured by performance on a test set. In this section, we focus on three approaches to improving generalization performance: choosing the right network architecture, penalizing large weights, and randomly perturbing the values passing through the network during training.

22.5.1 Choosing a network architecture

A great deal of effort in deep learning research has gone into finding network architectures that generalize well. Indeed, for each particular kind of data—images, speech, text, video, and so on—a good deal of the progress in performance has come from exploring different kinds of network architectures and varying the number of layers, their connectivity, and the types of node in each layer.[6]

Some neural network architectures are explicitly designed to generalize well on particular types of data: convolutional networks encode the idea that the same feature extractor is useful at all locations across a spatial grid, and recurrent networks encode the idea that the same update rule is useful at all points in a stream of sequential data. To the extent that these assumptions are valid, we expect convolutional architectures to generalize well on images and recurrent networks to generalize well on text and audio signals.

One of the most important empirical findings in the field of deep learning is that when comparing two networks with similar numbers of weights, the deeper network usually gives better generalization performance. Figure 22.7 shows this effect for at least one real-world application—recognizing house numbers. The results show that for any fixed number of parameters, an eleven-layer network gives much lower test-set error than a three-layer network.

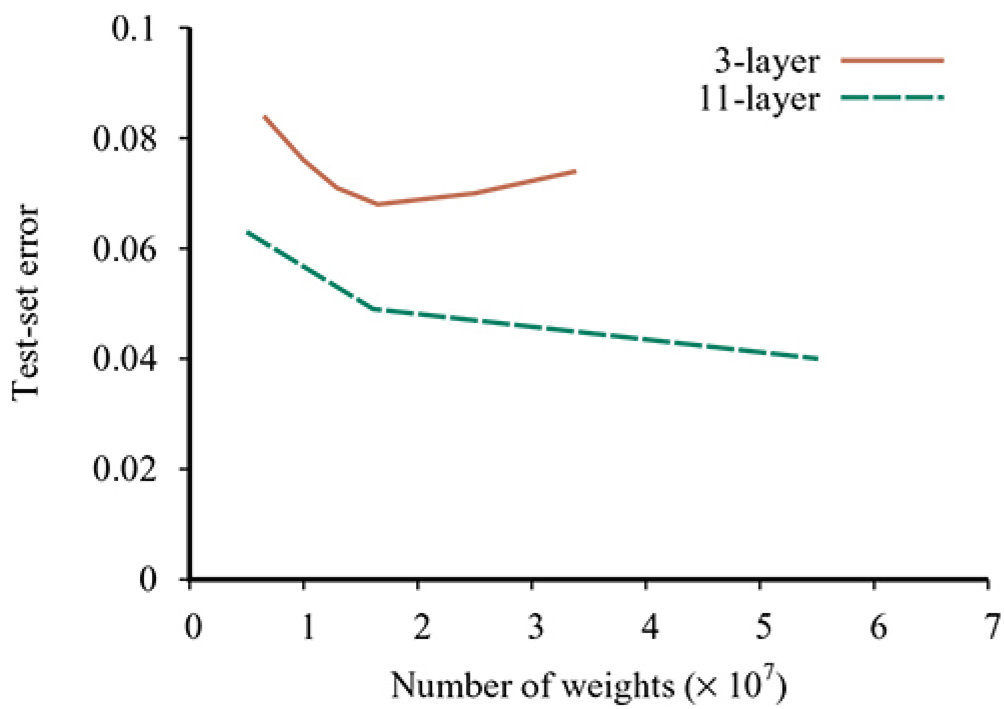


Figure 22.7 Test-set error as a function of layer width (as measured by total number of weights) for three-layer and eleven-layer convolutional networks. The data come from early versions of Google's system for transcribing addresses in photos taken by Street View cars (Goodfellow *et al.*, 2014).

Deep learning systems perform well on some but not all tasks. For tasks with highdimensional inputs—images, video, speech signals, etc.—they perform better than any other pure machine learning approaches. Most of the algorithms described in Chapter 19 can handle high-dimensional input only if it is preprocessed using manually designed features to reduce the dimensionality. This preprocessing approach, which prevailed prior to 2010, has not yielded performance comparable to that achieved by deep learning systems.

Clearly, deep learning models are capturing some important aspects of these tasks. In particular, their success implies that the tasks can be solved by parallel programs with a relatively small number of steps (10 to 10^3 rather than, say, 10^7). This is perhaps not surprising, because these tasks are typically solved by the brain in less than a second, which is time enough for only a few tens of sequential neuron firings. Moreover, by examining the internal-layer representations learned by deep convolutional networks for vision tasks, we find evidence that the processing steps seem to involve extracting a sequence of increasingly abstract representations of the scene, beginning with tiny edges, dots, and corner features and ending with entire objects and arrangements of multiple objects.

On the other hand, because they are simple circuits, deep learning models lack the compositional and quantificational expressive power that we see in first-order logic (Chapter 8) and context-free grammars (Chapter 24).

Although deep learning models generalize well in many cases, they may also produce unintuitive errors. They tend to produce input–output mappings that are discontinuous, so that a small change to an input can cause a large change in the output. For example, it may be possible to alter just a few pixels in an image of a dog and cause the network to classify the dog as an ostrich or a school bus—even though the altered image still looks exactly like a dog. An altered image of this kind is called an **adversarial example.**

In low-dimensional spaces it is hard to find adversarial examples. But for an image with a million pixel values, it is often the case that even though most of the pixels contribute to the image being classified in the middle of the "dog" region of the space, there are a few dimensions where the pixel value is near the boundary to another category. An adversary with the ability to reverse engineer the network can find the smallest vector difference that would move the image over the border.

When adversarial examples were first discovered, they set off two worldwide scrambles: one to find learning algorithms and network architectures that would not be susceptible to adversarial attack, and another to create ever-more-effective adversarial attacks against all kinds of learning systems. So far the attackers seem to be ahead. In fact, whereas it was assumed initially that one would need access to the internals of the trained network in order to construct an adversarial example specifically for that network, it has turned out that one can construct *robust* adversarial examples that fool multiple networks with different architectures, hyperparameters, and training sets. These findings suggest that deep learning models recognize objects in ways that are quite different from the human visual system.

22.5.2 Neural architecture search

Unfortunately, we don't yet have a clear set of guidelines to help you choose the best network architecture for a particular problem. Success in deploying a deep learning solution requires experience and good judgment.

From the earliest days of neural network research, attempts have been made to automate the process of architecture selection. We can think of this as a case of hyperparameter tuning (Section 19.4.4), where the hyperparameters determine the depth, width, connectivity, and other attributes of the network. However, there are so many choices to be made that simple approaches like grid search can't cover all possibilities in a reasonable amount of time.

Therefore, it is common to use **neural architecture search** to explore the state space of possible network architectures. Many of the search techniques and learning techniques we covered earlier in the book have been applied to neural architecture search.

Evolutionary algorithms have been popular because it is sensible to do both recombination (joining parts of two networks together) and mutation (adding or removing a layer or changing a parameter value). Hill climbing can also be used with these same mutation operations. Some researchers have framed the problem as reinforcement learning, and some as Bayesian optimization. Another possibility is to treat the architectural possibilities as a continuous differentiable space and use gradient descent to find a locally optimal solution.

For all these search techniques, a major challenge is estimating the value of a candidate network. The straightforward way to evaluate an architecture is to train it on a test set for multiple batches and then evaluate its accuracy on a validation set. But with large networks that could take many GPU-days.

Therefore, there have been many attempts to speed up this estimation process by eliminating or at least reducing the expensive training process. We can train on a smaller data set. We can train for a small number of batches and predict how the network would improve with more batches. We can use a reduced version of the network architecture that we hope retains the properties of the full version. We can train one big network and then search for subgraphs of the network that perform better; this search can be fast because the subgraphs share parameters and don't have to be retrained.

Another approach is to learn a heuristic evaluation function (as was done for A$*$ search). That is, start by choosing a few hundred network architectures and train and evaluate them. That gives us a data set of (network, score) pairs. Then learn a mapping from the features of a network to a predicted score. From that point on we can generate a large number of candidate networks and quickly estimate their value. After a search through the space of networks, the best one(s) can be fully evaluated with a complete training procedure.

22.5.3 Weight decay

In Section 19.4.3 we saw that **regularization**—limiting the complexity of a model—can aid generalization. This is true for deep learning models as well. In the context of neural networks we usually call this approach **weight decay**.

Weight decay consists of adding a penalty $\lambda \sum_{i,j} W_{i,j}^2$ to the loss function used to train the neural network, where λ is a hyperparameter controlling the strength of the penalty and the sum is usually taken over all of the weights in the network. Using $\lambda = 0$ is equivalent to not using weight decay, while using larger values of λ encourages the weights to become small. It is common to use weight decay with λ near 10^{-4}.

Choosing a specific network architecture can be seen as an absolute constraint on the hypothesis space: a function is either representable within that architecture or it is not. Loss function penalty terms such as weight decay offer a softer constraint: functions represented with large weights are in the function family, but the training set must provide more evidence in favor of these functions than is required to choose a function with small weights.

It is not straightforward to interpret the effect of weight decay in a neural network. In networks with sigmoid activation functions, it is hypothesized that weight decay helps to keep the activations near the linear part of the sigmoid, avoiding the flat operating region that leads to vanishing gradients. With ReLU activation functions, weight decay seems to be beneficial, but the explanation that makes sense for sigmoids no longer applies because the ReLU's output is either linear or zero. Moreover, with residual connections, weight decay encourages the network to have small differences between consecutive layers rather than small absolute weight values. Despite these differences in the behavior of weight decay across many architectures, weight decay is still widely useful.

One explanation for the beneficial effect of weight decay is that it implements a form of maximum a posteriori (MAP) learning (see page 774). Letting $\mathbf{X}$ and $\mathbf{y}$ stand for the inputs and outputs across the entire training set, the maximum a posteriori hypothesis h_{MAP} satisfies

$$
\begin{aligned}
h_{\mathrm{MAP}} &= \underset{\mathbf{w}}{\operatorname{argmax}} P(\mathbf{y} \mid \mathbf{X}, \mathbf{W}) P(\mathbf{W}) \\
&= \underset{\mathbf{w}}{\operatorname{argmax}} [-\log P(\mathbf{y} \mid \mathbf{X}, \mathbf{W}) - \log P(\mathbf{W})].
\end{aligned}
$$

The first term is the usual cross-entropy loss; the second term prefers weights that are likely under a prior distribution. This aligns exactly with a regularized loss function if we set

$$
\log P(\mathbf{W}) = -\lambda \sum_{i,j} \mathbf{W}_{i,j}^2,
$$

which means that $P(\mathbf{W})$ is a zero-mean Gaussian prior.

22.5.4 Dropout

Another way that we can intervene to reduce the test-set error of a network —at the cost of making it harder to fit the training set—is to use **dropout**. At each step of training, dropout applies one step of back-propagation learning to a new version of the network that is created by deactivating a randomly chosen subset of the units. This is a rough and very low-cost approximation to training a large ensemble of different networks (see Section 19.8).

More specifically, let us suppose we are using stochastic gradient descent with minibatch size m. For each minibatch, the dropout algorithm applies the following process to every node in the network: with probability p, the unit output is multiplied by a factor of $1/p$; otherwise, the unit output is fixed at zero. Dropout is typically applied to units in the hidden layers with $p = 0.5$; for input units, a value of $p = 0.8$ turns out to be most effective. This process produces a thinned network with about half as many units as the original, to which back-propagation is applied with the minibatch of m training examples. The process repeats in the usual way until training is complete. At test time, the model is run with no dropout.

We can think of dropout from several perspectives:

- By introducing noise at training time, the model is forced to become robust to noise.
- As noted above, dropout approximates the creation of a large ensemble of thinned networks. This claim can be verified analytically for linear models, and appears to hold experimentally for deep learning models.
- Hidden units trained with dropout must learn not only to be useful hidden units; they must also learn to be compatible with many other possible sets of other hidden units that may or may not be included in

the full model. This is similar to the selection processes that guide the evolution of genes: each gene must not only be effective in its own function, but must work well with other genes, whose identity in future organisms may vary considerably.

- Dropout applied to later layers in a deep network forces the final decision to be made robustly by paying attention to all of the abstract features of the example rather than focusing on just one and ignoring the others. For example, a classifier for animal images might be able to achieve high performance on the training set just by looking at the animal's nose, but would presumably fail on a test case where the nose was obscured or damaged. With dropout, there will be training cases where the internal “nose unit” is zeroed out, causing the learning process to find additional identifying features. Notice that trying to achieve the same degree of robustness by adding noise to the input data would be difficult: there is no easy way to know in advance that the network is going to focus on noses, and no easy way to delete noses automatically from each image.

Altogether, dropout forces the model to learn multiple, robust explanations for each input. This causes the model to generalize well, but also makes it more difficult to fit the training set—it is usually necessary to use a larger model and to train it for more iterations.

22.6 Recurrent Neural Networks

Recurrent neural networks (RNNs) are distinct from feedforward networks in that they allow cycles in the computation graph. In all the cases we will consider, each cycle has a delay, so that units may take as input a value computed from their own output at an earlier step in the computation. (Without the delay, a cyclic circuit may reach an inconsistent state.) This allows the RNN to have internal state, or **memory**: inputs received at earlier time steps affect the RNN's response to the current input.

RNNs can also be used to perform more general computations—after all, ordinary computers are just Boolean circuits with memory—and to model real neural systems, many of which contain cyclic connections. Here we focus on the use of RNNs to analyze sequential data, where we assume that a new input vector $\mathbf{x}_t$ arrives at each time step.

As tools for analyzing sequential data, RNNs can be compared to the hidden Markov models, dynamic Bayesian networks, and Kalman filters described in Chapter 14. (The reader may find it helpful to refer back to that chapter before proceeding.) Like those models, RNNs make a **Markov assumption** (see page 481): the hidden state $\mathbf{z}_t$ of the network suffices to capture the information from all previous inputs. Furthermore, suppose we describe the RNN's update process for the hidden state by the equation $\mathbf{z}_t = f_{\mathrm{w}}(\mathbf{z}_{t-1}, \mathbf{x}_t)$ for some parameterized function $f_{\mathbf{w}}$. Once trained, this function represents a **time-homogeneous** process (page 481)—effectively a universally quantified assertion that the dynamics represented by $f_{\mathbf{w}}$ hold for all time steps. Thus, RNNs add expressive power compared to feedforward networks, just as convolutional networks do, and just as dynamic Bayes nets add expressive power compared to regular Bayes nets. Indeed, if you tried to use a feedforward network to analyze sequential data, the fixed size of the input layer would force the network to examine only a finite-length window of data, in which case the network would fail to detect long-distance dependencies.

22.6.1 Training a basic RNN

The basic model we will consider has an input layer $\mathbf{x}$, a hidden layer $\mathbf{z}$ with recurrent connections, and an output layer $\mathbf{y}$, as shown in Figure 22.8(a). We assume that both $\mathbf{x}$ and $\mathbf{y}$ are observed in the training data at each time step. The equations defining the model refer to the values of the variables indexed by time step t:

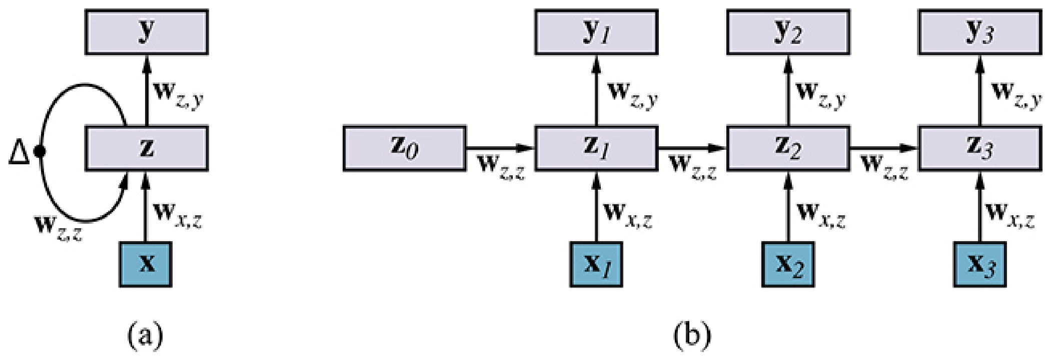


Figure 22.8 (a) Schematic diagram of a basic RNN where the hidden layer **z** has recurrent connections; the Δ symbol indicates a delay. (b) The same network unrolled over three time steps to create a feedforward network. Note that the weights are shared across all time steps.

$$\mathbf{z}_t = f\mathbf{w}(\mathbf{z}_{t-1}, \mathbf{x}_t) = \mathbf{g}_z(W_{z,z}\mathbf{z}_{t-1} + \mathbf{W}_{x,z}\mathbf{x}_t) \equiv \mathbf{g}_z(\mathbf{in}_{z,t})$$

$$\overset{\wedge}{\mathbf{y}}_t = \mathbf{g}_y(\mathbf{W}_{z,y}\mathbf{z}t) \equiv gy(\mathbf{in}_{y,t}), \qquad (22.13)$$

where $\mathbf{g}_z$ and $\mathbf{g}_y$ denote the activation functions for the hidden and output layers, respectively. As usual, we assume an extra dummy input fixed at +1 for each unit as well as bias weights associated with those inputs.

Given a sequence of input vectors $\mathbf{x}_1,\ldots,\mathbf{x}_T$ and observed outputs $\mathbf{y}_1,\ldots,\mathbf{y}_T$, we can turn this model into a feedforward network by "unrolling" it for T steps, as shown in Figure 22.8(b). Notice that the weight matrices $\mathbf{W}_{x,z}$, $\mathbf{W}_{z,z}$, and $\mathbf{W}_{z,y}$ are shared across all time steps. In the unrolled network, it is easy to see that we can calculate gradients to train the weights in the usual way; the only difference is that the sharing of weights across layers makes the gradient computation a little more complicated.

To keep the equations simple, we will show the gradient calculation for an RNN with just one input unit, one hidden unit, and one output unit. For this case, making the bias weights explicit, we have $z_t = g_z(w_{z,z}z_{4-1} + w_{x,z}x_t + w_{0,z})$ and $\hat{y}_t = g_y(w_{z,y}z_4 + w_{0,y})$. As in Equations (22.4) and (22.5), we will assume a squared-error loss L—in this case, summed over the time steps. The derivations for the input-layer and output-layer weights $w_{x,z}$ and $w_{z,y}$ are essentially identical to Equation (22.4), so we leave them as an exercise. For the hidden-layer weight $w_{z,z}$, the first few steps also follow the same pattern as Equation (22.4):

$$
\begin{aligned}
\frac{\partial L}{\partial w_{z,z}} &= \frac{\partial}{\partial w_{z,z}} \sum_{t=1}^{T} (y_t - \hat{y}_t)^2 = \sum_{t=1}^{T} -2(y_t - \hat{y}_t) \frac{\partial \hat{y}_t}{\partial w_{z,z}} \\
&= \sum_{t=1}^{T} -2(y_t - \hat{y}_t) \frac{\partial}{\partial w_{z,z}} g_y(in_{y,t}) = \sum_{t=1}^{T} -2(y_t - \hat{y}_t) g'_y(in_{y,t}) \frac{\partial}{\partial w_{z,z}} in_{y,t} \\
&= \sum_{t=1}^{T} -2(y_t - \hat{y}_t) g'_y(in_{y,t}) \frac{\partial}{\partial w_{z,z}} (w_{z,y} z_t + w_{0,y}) \\
&= \sum_{t=1}^{T} -2(y_t - \hat{y}_t) g'_y(in_{y,t}) w_{z,y} \frac{\partial z_t}{\partial w_{z,z}}. \qquad (22.14)
\end{aligned}
$$

Now the gradient for the hidden unit z_t can be obtained from the previous time step as follows:

$$
\begin{aligned}
\frac{\partial_{z,t}}{\partial w_{z,z}} &= \frac{\partial}{\partial w_{z,z}} g_z(in_{z,t}) = g\prime_z(in_{z,t}) \frac{\partial}{\partial w_{z,z}} in_{z,t} = g\prime_z(in_{z,t}) \frac{\partial}{\partial w_{z,z}} \left(w_{z,z} z_{t-1} + w_{x,z} x_t + w_{0,z} \right) \\
&= g\prime_z(in_{z,t}) \left(z_{t-1} + w_{z,z} \frac{\partial z_{t-1}}{\partial w_{z,z}} \right), \qquad (22.15)
\end{aligned}
$$

where the last line uses the rule for derivatives of products: $\partial(uv)/\partial x = v\partial x + u\partial v.\partial x.$.

Looking at Equation (22.15), we notice two things. First, the gradient expression is recursive: the contribution to the gradient from time step t is calculated using the contribution from time step $t - 1$. If we order the calculations in the right way, the total run time for computing the gradient will be linear in the size of the network. This algorithm is called **backpropagation through time**, and is usually handled automatically by deep learning software systems. Second, if we iterate the recursive calculation, we see that gradients at T will include terms proportional to $w_{z,z} \Pi_{t=1}^{T} g'_z(in_{z,t})$. For sigmoids, tanhs, and ReLUs, $g' \leq 1$, so our simple RNN will certainly suffer from the vanishing gradient problem (see page 807) if $w_{z,z} < 1$. On the other hand, if $w_{z,z} > 1$, we may experience the **exploding gradient** problem. (For the general case, these outcomes depend on the first eigenvalue of the weight matrix $\mathbf{W}_{z,z}$.) The next section describes a more elaborate RNN design intended to mitigate this issue.

22.6.2 Long short-term memory RNNs

Several specialized RNN architectures have been designed with the goal of enabling information to be preserved over many time steps. One of the most popular is the **long short-term memory** or **LSTM**. The long-term memory component of an LSTM, called the **memory cell** and denoted by **c**, is essentially *copied* from time step to time step. (In contrast, the basic RNN multiplies its memory by a weight matrix at every time step, as shown in Equation (22.13).) New information enters the memory by *adding* updates; in this way, the gradient expressions do not accumulate multiplicatively over time. LSTMs also include **gating units**, which are vectors that control the flow of information in the LSTM via elementwise multiplication of the corresponding information vector:

- The **forget gate f** determines if each element of the memory cell is remembered (copied to the next time step) or forgotten (reset to zero).
- The **input gate i** determines if each element of the memory cell is updated additively by new information from the input vector at the current time step.
- The **output gate o** determines if each element of the memory cell is transferred to the short-term memory **z**, which plays a similar role to the hidden state in basic RNNs.

Whereas the word “gate” in circuit design usually connotes a Boolean function, gates in LSTMs are soft—for example, elements of the memory cell vector will be partially forgotten if the corresponding elements of the forget-gate vector are small but not zero. The values for the gating units are always in the range [0, 1] and are obtained as the outputs of a sigmoid function applied to the current input and the previous hidden state. In detail, the update equations for the LSTM are as follows:

$$\begin{aligned}
\mathbf{f}_t &= \sigma(\mathbf{W}_{x,f}\mathbf{x}_t + \mathbf{W}_{z,f}\mathbf{z}_{t-1}) \\
\mathbf{i}_t &= \sigma(\mathbf{W}_{x,i}\mathbf{x}_t + \mathbf{W}_{z,i}\mathbf{z}_{t-1}) \\
\mathbf{o}_t &= \sigma(\mathbf{W}_{x,o}\mathbf{x}_t + \mathbf{W}_{z,o}\mathbf{z}_{t-1}) \\
\mathbf{c}_t &= \mathbf{c}_{t-1} \odot \mathbf{f}_t + \mathbf{i}_t \odot \tanh(W_{x,c}\mathbf{x}_t + \mathbf{W}_{z,c}\mathbf{z}_{t-1}) \\
\mathbf{z}_t &= \tanh(\mathbf{c}_t) \odot \mathbf{o}_t,
\end{aligned}$$

where the subscripts on the various weight matrices **W** indicate the origin and destination of the corresponding links. The ⊙ symbol denotes elementwise multiplication.

LSTMs were among the first practically usable forms of RNN. They have demonstrated excellent performance on a wide range of tasks including speech recognition and handwriting recognition. Their use in natural language processing is discussed in Chapter 25.

22.7 Unsupervised Learning and Transfer Learning

The deep learning systems we have discussed so far are based on supervised learning, which requires each training example to be labeled with a value for the target function. Although such systems can reach a high level of test-set accuracy—as shown by the ImageNet competition results, for example—they often require far more labeled data than a human would for the same task. For example, a child needs to see only one picture of a giraffe, rather than thousands, in order to be able to recognize giraffes reliably in a wide range of settings and views. Clearly, something is missing in our deep learning story; indeed, it may be the case that our current approach to supervised deep learning renders some tasks completely unattainable because the requirements for labeled data would exceed what the human race (or the universe) can supply. Moreover, even in cases where the task is feasible, labeling large data sets usually requires scarce and expensive human labor.

For these reasons, there is intense interest in several learning paradigms that reduce the dependence on labeled data. As we saw in Chapter 19, these paradigms include **unsupervised learning, transfer learning**, and **semisupervised learning**. Unsupervised learning algorithms learn solely from unlabeled inputs x, which are often more abundantly available than labeled examples. Unsupervised learning algorithms typically produce generative models, which can produce realistic text, images, audio, and video, rather than simply predicting labels for such data. Transfer learning algorithms require some labeled examples but are able to improve their performance further by studying labeled examples for different tasks, thus making it possible to draw on more existing sources of data. Semisupervised learning algorithms require some labeled examples but are able to improve their performance further by also studying unlabeled examples. This section covers deep learning approaches to unsupervised and transfer learning; while semisupervised learning is also an active area of research in the deep learning community, the techniques developed so far have not proven broadly effective in practice, so we do not cover them.

22.7.1 Unsupervised learning

Supervised learning algorithms all have essentially the same goal: given a training set of inputs $\mathbf{x}$ and corresponding outputs $y = f(\mathbf{x})$, learn a function h that approximates f well. Unsupervised learning algorithms, on the other hand, take a training set of unlabeled examples $\mathbf{x}$. Here we describe two things that such an algorithm might try to do. The first is to learn new representations—for example, new features of images that make it easier to identify the objects in an image. The second is to learn a generative model—typically in the form of a probability distribution from which new samples can be generated. (The algorithms for learning Bayes nets in Chapter 21 fall in this category.) Many algorithms are capable of both representation learning and generative modeling.

Suppose we learn a joint model $P_W(\mathbf{x}, \mathbf{z})$, where $\mathbf{z}$ is a set of latent, unobserved variables that represent the content of the data $\mathbf{x}$ in some way. In keeping with the spirit of the chapter, we do not predefine the meanings of the $\mathbf{z}$ variables; the model is free to learn to associate $\mathbf{z}$ with $\mathbf{x}$ however it chooses. For example, a model trained on images of handwritten digits might choose to use one direction in $\mathbf{z}$ space to represent the thickness of pen strokes, another to represent ink color, another to represent background color, and so on. With images of faces, the learning algorithm might choose one direction to represent gender and another to capture the presence or absence of glasses, as illustrated in Figure 22.9.

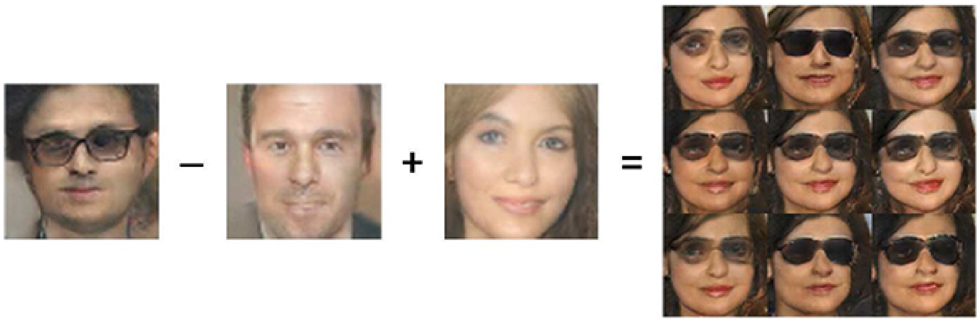


Figure 22.9 A demonstration of how a generative model has learned to use different directions in $\mathbf{z}$ space to represent different aspects of faces. We can actually perform arithmetic in $\mathbf{z}$ space. The images here are all generated from the learned model and show what happens when we decode different points in $\mathbf{z}$ space. We start with the coordinates for the concept of "man with glasses," subtract off the coordinates for "man," add the coordinates for "woman," and obtain the coordinates for "woman with glasses." Images reproduced with permission from (Radford et *al.*, 2015).

A learned probability model $P_W(\mathbf{x}, \mathbf{z})$ achieves both representation learning (it has constructed meaningful $\mathbf{z}$ vectors from the raw $\mathbf{x}$ vectors) and generative modeling: if we integrate $\mathbf{z}$ out of $P_W(\mathbf{x}, \mathbf{z})$ we obtain $P_W(\mathbf{x})$.

Probabilistic PCA: A simple generative model

There have been many proposals for the form that $P_W(\mathbf{x}, \mathbf{z})$ might take. One of the simplest is the **probabilistic principal components analysis (PPCA)** model.[7] In a PPCA model, $\mathbf{z}$ is chosen from a zero-mean, spherical Gaussian, then $\mathbf{x}$ is generated from $\mathbf{z}$ by applying a weight matrix $\mathbf{W}$ and adding spherical Gaussian noise:

$$P(\mathbf{z}) = N(\mathbf{z}; \mathbf{0}, \mathbf{I})$$
$$Pw(\mathbf{x}|\mathbf{z}) = N(\mathbf{x}; \mathbf{W}\mathbf{z}, \sigma^2\mathbf{I}).$$

The weights $\mathbf{W}$ (and optionally the noise parameter σ^2) can be learned by maximizing the likelihood of the data, given by

$$Pw(\mathbf{x}) = \int Pw(\mathbf{x}, \mathbf{z}) = N(\mathbf{x}; 0, \mathbf{W}\mathbf{W}^\top + \sigma^2\mathbf{I}). \quad (22.16)$$

The maximization with respect to $\mathbf{W}$ can be done by gradient methods or by an efficient iterative EM algorithm (see Section 21.3). Once $\mathbf{W}$ has been learned, new data samples can be generated directly from

$P_W(\mathbf{x})$ using Equation (22.16). Moreover, new observations x that have very low probability according to Equation (22.16) can be flagged as potential anomalies.

With PPCA, we usually assume that the dimensionality of $\mathbf{z}$ is much less than the dimensionality of $\mathbf{x}$, so that the model learns to explain the data as well as possible in terms of a small number of features. These features can be extracted for use in standard classifiers by computing $\hat{\mathbf{z}}$, the expectation of $P_W(\mathbf{z}|\mathbf{x})$.

Generating data from a probabilistic PCA model is straightforward: first sample z from its fixed Gaussian prior, then sample x from a Gaussian with mean $\mathbf{Wz}$. As we will see shortly, many other generative models resemble this process, but use complicated mappings defined by deep models rather than linear mappings from $\mathbf{z}$-space to $\mathbf{x}$-space.

Autoencoders

Many unsupervised deep learning algorithms are based on the idea of an **autoencoder**. An autoencoder is a model containing two parts: an encoder that maps from $\mathbf{x}$ to a representation $\hat{\mathbf{z}}$ and a decoder that maps from a representation $\hat{\mathbf{z}}$ to observed data $\mathbf{x}$. In general, the encoder is just a parameterized function f and the decoder is just a parameterized function g. The model is trained so that $\mathbf{x} \approx g(f(\mathbf{x}))$, so that the encoding process is roughly inverted by the decoding process. The functions f and g can be simple linear models parameterized by a single matrix or they can be represented by a deep neural network.

A very simple autoencoder is the linear autoencoder, where both f and g are linear with a shared weight matrix $\mathbf{W}$:

$$\begin{aligned}\hat{\mathbf{z}} &= f(\mathbf{x}) = \mathbf{W}\mathbf{x}\\ \mathbf{x} &= g(\hat{\mathbf{z}}) = \mathbf{W}^{\top}\hat{\mathbf{z}}.\end{aligned}$$

One way to train this model is to minimize the squared error $\sum_j ||\mathbf{x}_j - g(f(\mathbf{x}_j))||^2$ so that $\mathbf{x} \approx g(f(\mathbf{x}))$ The idea is to train $\mathbf{W}$ so that a low-dimensional $\hat{\mathbf{z}}$ will retain as much information as possible to reconstruct the high-dimensional data $\mathbf{x}$. This linear autoencoder turns out to be closely connected to classical principal components analysis (PCA). When $\mathbf{z}$ is m-dimensional, the matrix $\mathbf{W}$ should learn to span the m principal components of the data–in other words, the set of m orthogonal directions in which the data has highest variance, or equivalently the m eigenvectors of the data covariance matrix that have the largest eigenvalues–exactly as in PCA.

The PCA model is a simple generative model that corresponds to a simple linear autoencoder. The correspondence suggests that there may be a way to capture more complex kinds of generative models using more complex kinds of autoencoders. The **variational autoencoder** (VAE) provides one way to do this.

Variational methods were introduced briefly on page 476 as a way to approximate the posterior distribution in complex probability models, where summing or integrating out a large number of hidden variables is intractable. The idea is to use a **variational posterior** $Q(\mathbf{z})$, drawn from a computationally tractable family of distributions, as an approximation to the true posterior. For example, we might choose Q from the family of Gaussian distributions with a diagonal covariance matrix. Within the chosen family of tractable distributions, Q is optimized to be as close as possible to the true posterior distribution $P(\mathbf{z} \mid \mathbf{x})$.

For our purposes, the notion of "as close as possible" is defined by the KL divergence, which we mentioned on page 809. This is given by

$$D_{KL}(Q(\mathbf{z}) \parallel P(\mathbf{z}\Big|\mathbf{x})) = \int Q(\mathbf{z}) \log \tfrac{Q(\mathbf{z})}{P(\mathbf{z}|\mathbf{x})} d\mathbf{z},$$

which is an average (with respect to Q) of the log ratio between Q and P. It is easy to see that $D_{KL}(Q(\mathbf{z}) \parallel P(\mathbf{z}|\mathbf{x})) \geq 0$, with equality when Q and P coincide. We can then define the **variational lower bound** $\mathcal{L}$ (sometimes called the **evidence lower bound**, or **ELBO**) on the log likelihood of the data:

$$\mathscr{L}(\mathbf{x}, Q) = \log P(\mathbf{x}) - D_{KL}(Q(\mathbf{z}) \parallel P(\mathbf{z}|\mathbf{x})). \qquad (22.17)$$

We can see that $\mathcal{L}$ is a lower bound for log P because the KL divergence is nonnegative. Variational learning maximizes $\mathcal{L}$ with respect to parameters $\mathbf{w}$ rather than maximizing log $P(\mathbf{x})$, in the hope that the solution found, $\mathbf{w}*$, is close to maximizing log $P(\mathbf{x})$ as well.

As written, $\mathcal{L}$ does not yet seem to be any easier to maximize than log P. Fortunately, we can rewrite Equation (22.17) to reveal improved computational tractability:

$$\begin{aligned}\mathscr{L} &= \log P(\mathbf{x}) - \int Q(\mathbf{z}) \log \frac{Q(\mathbf{z})}{P(\mathbf{z}|\mathbf{x})} d\mathbf{z} \\ &= -\int Q(\mathbf{z}) \log Q(\mathbf{z}) d\mathbf{z} + \int Q(\mathbf{z}) \log P(\mathbf{x}) P(\mathbf{z}|\mathbf{x}) d\mathbf{z} \\ &= H(Q) + \mathbf{E}_{\mathbf{z}\sim Q} \log P(\mathbf{z},\mathbf{x})\end{aligned}$$

where $H(Q)$ is the entropy of the Q distribution. For some variational families Q (such as Gaussian distributions), $H(Q)$ can be evaluated analytically. Moreover, the expectation, $\mathbf{E}_{Z\sim Q}\log P(\mathbf{z}, \mathbf{x})$, admits an efficient unbiased estimate via samples of $\mathbf{z}$ from Q. For each sample, $P(\mathbf{z}, \mathbf{x})$ can usually be evaluated efficiently—for example, if P is a Bayes net, $P(\mathbf{z}, \mathbf{x})$ is just a product of conditional probabilities because $\mathbf{z}$ and $\mathbf{x}$ comprise all the variables.

Variational autoencoders provide a means of performing variational learning in the deep learning setting. Variational learning involves maximizing $\mathcal{L}$ with respect to the parameters of both P and Q. For a variational autoencoder, the decoder $g(\mathbf{z})$ is interpreted as defining $\log P(\mathbf{x} \mid \mathbf{z})$. For example, the output of the decoder might define the mean of a conditional Gaussian. Similarly, the output of the encoder $f(\mathbf{x})$ is interpreted as defining the parameters of Q—for example, Q might be a Gaussian with mean $f(\mathbf{x})$. Training the variational autoencoder then consists of maximizing $\mathcal{L}$ with respect to the parameters of both the encoder f and the decoder g, which can themselves be arbitrarily complicated deep networks.

Deep autoregressive models

An **autoregressive model** (or AR model) is one in which each element x_i, of the data vector $\mathbf{x}$ is predicted based on other elements of the vector. Such a model has no latent variables. If $\mathbf{x}$ is of fixed size, an AR model can be thought of as a fully observable and possibly fully connected Bayes net. This means that calculating the likelihood of a given data vector according to an AR model is trivial; the same holds for predicting the value of a single missing variable given all the others, and for sampling a data vector from the model.

The most common application of autoregressive models is in the analysis of time series data, where an AR model of order k predicts x_t given $x_{t-k},\ldots,x_{t-1}$. In the terminology of Chapter 14, an AR model is a non-hidden Markov model. In the terminology of Chapter 24, an n-gram model of letter or word sequences is an AR model of order $n - 1$.

In classical AR models, where the variables are real-valued, the conditional distribution $P(x_t \mid x_{t-k},\ldots,x_{t-1})$ is a linear-Gaussian model with fixed variance whose mean is a weighted linear combination of $x_{t-k},\ldots, x_{t-1}$—in other words, a standard linear regression model. The maximum likelihood solution is given by the **Yule–Walker equations**, which are closely related to the **normal equations** on page 698.

A **deep autoregressive model** is one in which the linear–Gaussian model is replaced by an arbitrary deep network with a suitable output layer depending on whether x_t is discrete or continuous. Recent applications of this autoregressive approach include DeepMind's WaveNet model for speech generation (van den Oord *et al.*, 2016a). WaveNet is trained on raw acoustic signals, sampled 16,000 times per second, and implements a nonlinear AR model of order 4800 with a multilayer convolutional structure. In tests it proves to be substantially more realistic than previous state-of-the-art speech generation systems.

Generative adversarial networks

A **generative adversarial network (GAN)** is actually a pair of networks that combine to form a generative system. One of the networks, the **generator**, maps values from $\mathbf{z}$ to $\mathbf{x}$ in order to produce samples from the distribution $P_{\mathbf{w}}(\mathbf{x})$. A typical scheme samples $\mathbf{z}$ from a unit Gaussian of moderate dimension and then passes it through a deep network $h_{\mathbf{w}}$ to obtain $\mathbf{x}$. The other network, the **discriminator**, is a classifier trained to classify inputs $\mathbf{x}$ as real (drawn from the training set) or fake (created by the generator). GANs are a kind of **implicit model** in the sense that samples can be generated but their probabilities are not readily available; in a Bayes net, on the other hand, the probability of a sample is just the product of the conditional probabilities along the sample generation path.

The generator is closely related to the decoder from the variational autoencoder framework. The challenge in implicit modeling is to design a loss function that makes it possible to train the model using samples from the distribution, rather than maximizing the likelihood assigned to training examples from the data set.

Both the generator and the discriminator are trained simultaneously, with the generator learning to fool the discriminator and the discriminator learning to accurately separate real from fake data. The competition between generator and discriminator can be described in the language of game theory (see Chapter 17). The idea is that in the equilibrium state of the game, the generator should reproduce the training distribution perfectly, such that the discriminator cannot perform better than random guessing. GANs have worked particularly well for image generation tasks. For example, GANs can create photorealistic, high-resolution images of people who have never existed (Karras *et al.*, 2017).

Unsupervised translation

Translation tasks, broadly construed, consist of transforming an input $\mathbf{x}$ that has rich structure into an output $\mathbf{y}$ that also has rich structure. In this context, "rich structure" means that the data are multidimensional and have interesting statistical dependencies among the various dimensions. Images and natural language sentences have a rich structure, but a single number, such as a class ID, does not. Transforming a sentence from English to French or converting a photo of a night scene into an equivalent photo taken during the daytime are both examples of translation tasks.

Supervised translation consists of gathering many ($\mathbf{x}$, $\mathbf{y}$) pairs and training the model to map each $\mathbf{x}$ to the corresponding $\mathbf{y}$. For example, machine translation systems are often trained on pairs of sentences that have been translated by professional human translators. For other kinds of translation, supervised training data may not be available. For example, consider a photo of a night scene containing many moving cars and pedestrians. It is presumably not feasible to find all of the cars and pedestrians and return them to their original positions in the night-time photo in order to retake the same photo in the daytime. To overcome this difficulty, it is possible to use **unsupervised translation** techniques that are capable of training on many examples of $\mathbf{x}$ and many separate examples of $\mathbf{y}$ but no corresponding ($\mathbf{x}$, $\mathbf{y}$) pairs.

These approaches are generally based on GANs; for example, one can train a GAN generator to produce a realistic example of $\mathbf{y}$ when conditioned on $\mathbf{x}$, and another GAN generator to perform the reverse mapping. The GAN training framework makes it possible to train a generator to generate any one of many possible samples that the discriminator accepts as a *realistic* example of $\mathbf{y}$ given $\mathbf{x}$, without any need for a specific paired y as is traditionally needed in supervised learning. More detail on unsupervised translation for images is given in Section 27.7.5.

22.7.2 Transfer learning and multitask learning

In **transfer learning**, experience with one learning task helps an agent learn better on another task. For example, a person who has already learned to play tennis will typically find it easier to learn related sports such as racquetball and squash; a pilot who has learned to fly one type of commercial passenger airplane will very quickly learn to fly another type; a student who has already learned algebra finds it easier to learn calculus.

We do not yet know the mechanisms of human transfer learning. For neural networks, learning consists of adjusting weights, so the most plausible approach for transfer learning is to copy over the weights learned for task A to a network that will be trained for task B. The weights are then updated by gradient descent in the usual way using data for task B. It may be a good idea to use a smaller learning rate in task B, depending on how similar the tasks are and how much data was used in task A.

Notice that this approach requires human expertise in selecting the tasks: for example, weights learned during algebra training may not be very useful in a network intended for racquetball. Also, the notion of copying weights requires a simple mapping between the input spaces for the two tasks and essentially identical network architectures.

One reason for the popularity of transfer learning is the availability of high-quality pretrained models. For example, you could download a pretrained visual object recognition model such as the ResNet-50 model trained on the COCO data set, thereby saving yourself weeks of work. From there you can modify the model parameters by supplying additional images and object labels for your specific task.

Suppose you want to classify types of unicycles. You have only a few hundred pictures of different unicycles, but the COCO data set has over 3,000 images in each of the categories of bicycles, motorcycles, and skateboards. This means that a model pretrained on COCO already has experience with wheels and roads and other relevant features that will be helpful in interpreting the unicycle images.

Often you will want to freeze the first few layers of the pretrained model—these layers serve as feature detectors that will be useful for your new model. Your new data set will be allowed to modify the parameters of the higher levels only; these are the layers that identify problem-specific features and do classification. However, sometimes the difference between sensors means that even the lowest-level layers need to be retrained.

As another example, for those building a natural language system, it is now common to start with a pretrained model such as the RoBERTa model (see Section 25.6), which already “knows” a great deal about the vocabulary and syntax of everyday language. The next step is to fine-tune the model in two ways. First, by giving it examples of the specialized vocabulary used in the desired domain; perhaps a medical domain (where it will learn about “myocardial infarction”) or perhaps a financial domain (where it will learn about “fiduciary responsibility”). Second, by training the model on the task it is to perform. If it is to do question answering, train it on question/answer pairs.

One very important kind of transfer learning involves transfer between simulations and the real world. For example, the controller for a self-driving car can be trained on billions of miles of simulated driving, which would be impossible in the real world. Then, when the controller is transitioned to the real vehicle, it adapts quickly to the new environment.

Multitask learning is a form of transfer learning in which we simultaneously train a model on multiple objectives. For example, rather than training a natural language system on part-of-speech tagging and then transferring the learned weights to a new task such as document classification, we train one system simultaneously on part-of-speech tagging, document classification, language detection, word prediction, sentence difficulty modeling, plagiarism detection, sentence entailment, and question answering. The idea is that to solve any one of these tasks, a model might be able to take advantage of superficial features of the data. But to solve all eight at once with a common representation layer, the model is more likely to create a common representation that reflects real natural language usage and content.

22.8 Applications

Deep learning has been applied successfully to many important problem areas in AI. For indepth explanations, we refer the reader to the relevant chapters: Chapter 23 for the use of deep learning in reinforcement learning systems, Chapter 25 for natural language processing, Chapter 27 (particularly Section 27.4) for computer vision, and Chapter 26 for robotics.

22.8.1 Vision

We begin with computer vision, which is the application area that has arguably had the biggest impact on deep learning, and vice versa. Although deep convolutional networks had been in use since the 1990s for tasks such as handwriting recognition, and neural networks had begun to surpass generative probability models for speech recognition by around 2010, it was the success of the AlexNet deep learning system in the 2012 ImageNet competition that propelled deep learning into the limelight.

The ImageNet competition was a supervised learning task with 1,200,000 images in 1,000 different categories, and systems were evaluated on the "top-5" score—how often the correct category appears in the top five predictions. AlexNet achieved an error rate of 15.3%, whereas the next best system had an error rate of more than 25%. AlexNet had five convolutional layers interspersed with max-pooling layers, followed by three fully connected layers. It used ReLU activation functions and took advantage of GPUs to speed up the process of training 60 million weights.

Since 2012, with improvements in network design, training methods, and computing resources, the top-5 error rate has been reduced to less than 2%—well below the error rate of a trained human (around 5%). CNNs have been applied to a wide range of vision tasks, from self-driving cars to grading cucumbers.[8] Driving, which is covered in Section 27.7.6 and in several sections of Chapter 26, is among the most demanding of vision tasks: not only must the algorithm detect, localize, track, and recognize pigeons, paper bags, and pedestrians, but it has to do it in real time with near-perfect accuracy.

22.8.2 Natural language processing

Deep learning has also had a huge impact on natural language processing (NLP) applications such as machine translation and speech recognition. Some advantages of deep learning for these applications include the possibility of end-to-end learning, the automatic generation of internal representations for the meanings of words, and the interchangeability of learned encoders and decoders.

End-to-end learning refers to the construction of entire systems as a single, learned function f. For example, an f for machine translation might take as input an English sentence S_E and produce an equivalent Japanese sentence $S_J = f(S_E)$. Such an f can be learned from training data in the form of human-translated pairs of sentences (or even pairs of texts, where the alignment of corresponding sentences or phrases is part of the problem to be solved). A more classical pipeline approach might first parse S_E, then extract its meaning, then re-express the meaning in Japanese as S_J, then post-edit S_J using a language model for Japanese. This pipeline approach has two major drawbacks: first, errors are compounded at each stage; and second, humans have to determine what constitutes a “parse tree” and a “meaning representation,” but there is no easily accessible ground truth for these notions, and our theoretical ideas about them are almost certainly incomplete.

At our present stage of understanding, then, the classical pipeline approach—which, at least naively, seems to correspond to how a human translator works—is outperformed by the end-to-end method made possible by deep learning. For example, Wu *et al.* (2016b) showed that end-to-end translation using deep learning reduced translation errors by 60% relative to a previous pipeline-based system. As of 2020, machine translation systems are approaching human performance for language pairs such as French and English for which very large paired data sets are available, and they are usable for other language pairs covering the majority of Earth's population. There is even some evidence that networks trained on multiple languages do in fact learn an internal meaning representation: for example, after learning to translate Portuguese to English and English to Spanish, it is possible to translate Portuguese directly into Spanish without any Portuguese/Spanish sentence pairs in the training set.

One of the most significant findings to emerge from the application of deep learning to language tasks is that a great deal deal of mileage comes from re-representing individual words as vectors in a high-dimensional space—so-called **word embeddings** (see Section 25.1). The vectors are usually extracted from the weights of the first hidden layer of a network trained on large quantities of text, and they capture the statistics of the lexical contexts in which words are used. Because words with similar meanings are used in similar contexts, they end up close to each other in the vector space. This allows the network to generalize effectively across categories of words, without the need for humans to predefine those categories. For example, a sentence beginning "John bought a watermelon and two pounds of ... " is likely to continue with "apples" or "bananas" but not with "thorium" or "geography." Such a prediction is much easier to make if “apples” and “bananas” have similar representations in the internal layer.

22.8.3 Reinforcement learning

In reinforcement learning (RL), a decision-making agent learns from a sequence of reward signals that provide some indication of the quality of its behavior. The goal is to optimize the sum of future rewards. This can be done in several ways: in the terminology of Chapter 16, the agent can learn a value function, a Q-function, a policy, and so on. From the point of view of deep learning, all these are functions that can be represented by computation graphs. For example, a value function in Go takes a board position as input and returns an estimate of how advantageous the position is for the agent. While the methods of training in RL differ from those of supervised learning, the ability of multilayer computation graphs to represent complex functions over large input spaces has proved to be very useful. The resulting field of research is called **deep reinforcement learning**.

In the 1950s, Arthur Samuel experimented with multilayer representations of value functions in his work on reinforcement learning for checkers, but he found that in practice a linear function approximator worked best. (This may have been a consequence of working with a computer roughly 100 billion times less powerful than a modern tensor processing unit.) The first major successful demonstration of deep RL was DeepMind's Atari-playing agent, DQN (Mnih *et al.*, 2013). Different copies of this agent were trained to play each of several different Atari video games, and demonstrated skills such as shooting alien spaceships, bouncing balls with paddles, and driving simulated racing cars. In each case, the agent learned a Q-function from raw image data with the reward signal being the game score. Subsequent work has produced deep RL systems that play at a superhuman level on the majority of the 57 different Atari games. DeepMind's ALPHAGO system also used deep RL to defeat the best human players at the game of Go (see Chapter 6).

Despite its impressive successes, deep RL still faces significant obstacles: it is often difficult to get good performance, and the trained system may behave very unpredictably if the environment differs even a little from the training data (Irpan, 2018). Compared to other applications of deep learning, deep RL is rarely applied in commercial settings. It is, nonetheless, a very active area of research.

Summary

This chapter described methods for learning functions represented by deep computational graphs. The main points were:

- **Neural networks** represent complex nonlinear functions with a network of parameterized linear-threshold units.
- The **back-propagation** algorithm implements a gradient descent in parameter space to minimize the loss function.
- Deep learning works well for visual object recognition, speech recognition, natural language processing, and reinforcement learning in complex environments.
- Convolutional networks are particularly well suited for image processing and other tasks where the data have a grid topology.
- Recurrent networks are effective for sequence-processing tasks including language modeling and machine translation.

Bibliographical and Historical Notes

The literature on neural networks is vast. Cowan and Sharp (1988b, 1988a) survey the early history, beginning with the work of McCulloch and Pitts (1943). (As mentioned in Chapter 1, John McCarthy has pointed to the work of Nicolas Rashevsky (1936, 1938) as the earliest mathematical model of neural learning.) Norbert Wiener, a pioneer of cybernetics and control theory (Wiener, 1948), worked with McCulloch and Pitts and influenced a number of young researchers, including Marvin Minsky, who may have been the first to develop a working neural network in hardware, in 1951 (see Minsky and Papert, 1988, pp. ix–x). Alan Turing (1948) wrote a research report titled *Intelligent Machinery* that begins with the sentence "I propose to investigate the question as to whether it is possible for machinery to show intelligent behaviour" and goes on to describe a recurrent neural network architecture he called "B-type unorganized machines" and an approach to training them. Unfortunately, the report went unpublished until 1969, and was all but ignored until recently.

The perceptron, a one-layer neural network with a hard-threshold activation function, was popularized by Frank Rosenblatt (1957). After a demonstration in July 1958, the New York Times described it as "the embryo of an electronic computer that [the Navy] expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence." Rosenblatt (1960) later proved the perceptron convergence theorem, although it had been foreshadowed by purely mathematical work outside the context of neural networks (Agmon, 1954; Motzkin and Schoenberg, 1954). Some early work was also done on multilayer networks, including **Gamba perceptrons** (Gamba *et al.*, 1961) and **madalines** (Widrow, 1962).

Learning Machines (Nilsson, 1965) covers much of this early work and more. The subsequent demise of early perceptron research efforts was hastened (or, the authors later claimed, merely explained) by the book *Perceptrons* (Minsky and Papert, 1969), which lamented the field's lack of mathematical rigor. The book pointed out that single-layer perceptrons could represent only linearly separable concepts and noted the lack of effective learning algorithms for multilayer networks. These limitations were already well known (Hawkins, 1961) and had been acknowledged by Rosenblatt himself (Rosenblatt, 1962).

The papers collected by Hinton and Anderson (1981), based on a conference in San Diego in 1979, can be regarded as marking a renaissance of connectionism. The two-volume "PDP" (Parallel Distributed Processing) anthology (Rumelhart and McClelland, 1986) helped to spread the gospel, so to speak, particularly in the psychology and cognitive science communities. The most important development of this period was the back-propagation algorithm for training multilayer networks.

The back-propagation algorithm was discovered independently several times in different contexts (Kelley, 1960; Bryson, 1962; Dreyfus, 1962; Bryson and Ho, 1969; Werbos, 1974; Parker, 1985) and Stuart Dreyfus (1990) calls it the "Kelley–Bryson gradient procedure." Although Werbos had applied it to neural networks, this idea did not become widely known until a paper by David Rumelhart, Geoff Hinton, and Ron Williams (1986) appeared in *Nature* giving a nonmathematical presentation of the algorithm. Mathematical respectability was enhanced by papers showing that multilayer feedforward networks are (subject to technical conditions) universal function approximators (Cybenko, 1988, 1989). The late 1980s and early 1990s saw a huge growth in neural network research: the number of papers mushroomed by a factor of 200 between 1980–84 and 1990–94.

In the late 1990s and early 2000s, interest in neural networks waned as other techniques such as Bayes nets, ensemble methods, and kernel machines came to the fore. Interest in deep models was sparked when Geoff Hinton's research on deep Bayesian networks——generative models with category variables at the root and evidence variables at the leaves—began to bear fruit, outperforming kernel machines on small benchmark data sets (Hinton *et al.*, 2006). Interest in deep learning exploded when Krizhevsky *et al*. (2013) used deep convolutional networks to win the ImageNet competition (Russakovsky *et al.*, 2015).

Commentators often cite the availability of "big data" and the processing power of GPUs as the main contributing factors in the emergence of deep learning. Architectural improvements were also important, including the adoption of the ReLU activation function instead of the logistic sigmoid (Jarrett *et al.*, 2009; Nair and Hinton, 2010; Glorot et al., 2011) and later the development of residual networks (He *et al.*, 2016).

On the algorithmic side, the use of stochastic gradient descent (SGD) with small batches was essential in allowing neural networks to scale to large data sets (Bottou and Bousquet, 2008). Batch normalization (Ioffe and Szegedy, 2015) also helped in making the training process faster and more reliable and has spawned several additional normalization techniques (Ba *et al.*, 2016; Wu and He, 2018; Miyato *et al.*, 2018). Several papers have studied the empirical behavior of SGD on large networks and large data sets (Dauphin *et al*., 2015; Choromanska *et al.*, 2014; Goodfellow *et al.*, 2015b). On the theoretical side, some progress has been made on explaining the observation that SGD applied to overparameterized networks often reaches a global minimum with a training error of zero, although so far the theorems to this effect assume a network with layers far wider than would ever occur in practice (Allen-Zhu et *al*., 2018; Du *et al.*, 2018). Such networks have more than enough capacity to function as lookup tables for the training data.

The last piece of the puzzle, at least for vision applications, was the use of convolutional networks. These had their origins in the descriptions of the mammalian visual system by neurophysiologists David Hubel and Torsten Wiesel (Hubel and Wiesel, 1959, 1962, 1968). They described “simple cells” in the visual system of a cat that resemble edge detectors, as well as “complex cells” that are invariant to some transformations such as small spatial translations. In modern convolutional networks, the output of a convolution is analogous to a simple cell while the output of a pooling layer is analogous to a complex cell.

The work of Hubel and Wiesel inspired many of the early connectionist models of vision (Marr and Poggio, 1976). The neocognitron (Fukushima, 1980; Fukushima and Miyake, 1982), designed as a model of the visual cortex, was essentially a convolutional network in terms of model architecture, although an effective training algorithm for such networks had to wait until Yann LeCun and collaborators showed how to apply back-propagation (LeCun *et al.*, 1995). One of the early commercial successes of neural networks was handwritten digit recognition using convolutional networks (LeCun *et al.*, 1995).

Recurrent neural networks (RNNs) were commonly proposed as models of brain function in the 1970s, but no effective learning algorithms were associated with these proposals. The method of back-propagation through time appears in the PhD thesis of Paul Werbos (1974), and his later review paper (Werbos, 1990) gives several additional references to rediscoveries of the method in the 1980s. One of the most influential early works on RNNs was due to Jeff Elman (1990), building on an RNN architecture suggested by Michael Jordan (1986). Williams and Zipser

(1989) present an algorithm for online learning in RNNs. Bengio *et al.* (1994) analyzed the problem of vanishing gradients in recurrent networks. The long shortterm memory (LSTM) architecture (Hochreiter, 1991; Hochreiter and Schmidhuber, 1997; Gers *et al.*, 2000) was proposed as a way of avoiding this problem. More recently, effective RNN designs have been derived automatically (Jozefowicz *et al.*, 2015; Zoph and Le, 2016).

Many methods have been tried for improving generalization in neural networks. Weight decay was suggested by Hinton (1987) and analyzed mathematically by Krogh and Hertz (1992). The dropout method is due to Srivastava et al. (2014a). Szegedy *et al.* (2013) introduced the idea of adversarial examples, spawning a huge literature.

Poole *et al.* (2017) showed that deep networks (but not shallow ones) can disentangle complex functions into flat manifolds in the space of hidden units. Rolnick and Tegmark (2018) showed that the number of units required to approximate a certain class of polynomials of *n* variables grows exponentially for shallow networks but only linearly for deep networks.

White *et al.* (2019) showed that their BANANAS system could do neural architecture search (NAS) by predicting the accuracy of a network to within 1% after training on just 200 random sample architectures. Zoph and Le (2016) use reinforcement learning to search the space of neural network architectures. Real *et al.* (2018) use an evolutionary algorithm to do model selection, Liu *et al.* (2017) use evolutionary algorithms on hierarchical representations, and Jaderberg *et al.* (2017) describe population-based training. Liu *et al.* (2019) relax the space of architectures to a continuous differentiable space and use gradient descent to find a locally optimal solution. Pham *et al.* (2018) describe the ENAS (Efficient Neural Architecture Search) system, which searches for optimal subgraphs of a larger graph. It is fast because it does not need to retrain parameters. The idea of searching for a subgraph goes back to the “optimal brain damage” algorithm of LeCun *et al.* (1990).

Despite this impressive array of approaches, there are critics who feel the field has not yet matured. Yu *et al.* (2019) show that in some cases these NAS algorithms are no more efficient than random architecture selection. For a survey of recent results in neural architecture search, see Elsken *et al.* (2018).

Unsupervised learning constitutes a large subfield within statistics, mostly under the heading of density estimation. Silverman (1986) and Murphy (2012) are good sources for classical and modern techniques in this area. Principal components analysis (PCA) dates back to Pearson (1901); the name comes from independent work by Hotelling (1933). The probabilistic PCA model (Tipping and Bishop, 1999) adds a generative model for the principal components themselves. The variational autoencoder is due to Kingma and Welling (2013) and Rezende *et al.* (2014); Jordan *et al.* (1999) provide an introduction to variational methods for inference in graphical models.

For autoregressive models, the classic text is by Box *et al.* (2016). The Yule–Walker equations for fitting AR models were developed independently by Yule (1927) and Walker (1931). Autoregressive models with nonlinear dependencies were developed by several authors (Frey, 1998; Bengio and Bengio, 2001; Larochelle and Murray, 2011). The autoregressive WaveNet model (van den Oord *et al.*, 2016a) was based on earlier work on autoregressive image generation (van den Oord *et al.*, 2016b). Generative adversarial networks, or GANs, were first proposed by Goodfellow *et al.*, (2015a) and have found many applications in AI. Some theoretical understanding of their properties is emerging, leading to improved GAN models and algorithms (Li and Malik, 2018b, 2018a; Zhu *et* *al.*, 2019). Part of that understanding involves protecting against adversarial attacks (Carlini *et al.*, 2019).

Several branches of research into neural networks have been popular in the past but are not actively explored today. **Hopfield networks** (Hopfield, 1982) have symmetric connections between each pair of nodes and can learn to store patterns in an associative memory, so that an entire pattern can be retrieved by indexing into the memory using a fragment of the pattern. Hopfield networks are deterministic; they were later generalized to stochastic Boltzmann machines (Hinton and Sejnowski, 1983, 1986). **Boltzmann machines** are possibly the earliest example of a deep generative model. The difficulty of inference in Boltzmann machines led to advances in both Monte Carlo techniques and variational techniques (see Section 13.4).

Research on neural networks for AI has also been intertwined to some extent with research into biological neural networks. The two topics coincided in the 1940s, and ideas for convolutional networks and reinforcement learning can be traced to studies of biological systems; but at present, new ideas in deep learning tend to be based on purely computational or statistical concerns. The field of **computational neuroscience** aims to build computational models that capture important and specific properties of actual biological systems. Overviews are given by Dayan and Abbott (2001) and Trappenberg (2010).

For modern neural nets and deep learning, the leading textbooks are those by Goodfellow *et al.* (2016) and Charniak (2018). There are also many hands-on guides associated with the various open-source software packages for deep learning. Three of the leaders of the field—Yann LeCun, Yoshua Bengio, and Geoff Hinton—introduced the key ideas to non-AI researchers in an influential *Nature* article (2015). The three were recipients of the 2018 Turing Award. Schmidhuber (2015) provides a general overview, and Deng *et al.* (2014) focus on signal processing tasks.

The primary publication venues for deep learning research are the conference on Neural Information Processing Systems (NeurIPS), the International Conference on Machine Learning (ICML), and the International Conference on Learning Representations (ICLR). The main journals are *Machine Learning*, the *Journal of Machine Learning Research*, and *Neural Computation*. Increasingly, because of the fast pace of research, papers appear first on arXiv.org and are often described in the research blogs of the major research centers.

[1] Automatic differentiation methods were originally developed in the 1960s and 1970s for optimizing the parameters of systems defined by large, complex Fortran programs.

[2] Cross-entropy is not a distance in the usual sense because $H(P, P)$ is not zero; rather, it equals the entropy $H(P)$. It is easy to show that $H(P,Q) = H(P) + D_{KL}((P \,||\, Q)$, where D_{KL} is the **Kullback–Leibler divergence**, which does satisfy $D_{KL}(P \,||\, P) = 0$. Thus, for fixed P, varying Q to minimize the cross-entropy also minimizes the KL divergence.

[3] Similar ideas can be applied to process time-series data sources such as audio waveforms. These typically exhibit **temporal invariance**—a word sounds the same no matter what time of day it is uttered. Recurrent neural networks (Section 22.6) automatically exhibit temporal invariance.

[4] In the terminology of signal processing, we would call this operation a cross-correlation, not a convolution. But “convolution” is used within the field of neural networks.

[5] The proper mathematical definition of tensors requires that certain invariances hold under a change of basis.

[6] Noting that much of this incremental, exploratory work is carried out by graduate students, some have called the process **graduate student descent (GSD)**.

[7] Standard PCA involves fitting a multivariate Gaussian to the raw input data and then selecting out the longest axes—the principal components—of that ellipsoidal distribution.

[8] The widely known tale of the Japanese cucumber farmer who built his own cucumber-sorting robot using TensorFlow is, it turns out, mostly mythical. The algorithm was developed by the farmer's son, who worked previously as a software engineer at Toyota, and its low accuracy—about 70%—meant that the cucumbers still had to be sorted by hand (Zeeberg, 2017).

CHAPTER 23

REINFORCEMENT LEARNING

In which we see how experiencing rewards and punishments can teach an agent how to maximize rewards in the future.

With **supervised learning**, an agent learns by passively observing example input/output pairs provided by a “teacher.” In this chapter, we will see how agents can actively learn from their own experience, without a teacher, by considering their own ultimate success or failure.

23.1 Learning from Rewards

Consider the problem of learning to play chess. Let's imagine treating this as a supervised learning problem using the methods of Chapters 19, 21, and 22. The chess-playing agent function takes as input a board position and returns a move, so we train this function by supplying examples of chess positions, each labeled with the correct move. Now, it so happens that we have available databases of several million grandmaster games, each a sequence of positions and moves. The moves made by the winner are, with few exceptions, assumed to be good, if not always perfect. Thus, we have a promising training set. The problem is that there are relatively few examples (about 10^8) compared to the space of all possible chess positions (about 10^{40}). In a new game, one soon encounters positions that are significantly different from those in the database, and the trained agent function is likely to fail miserably—not least because it has no idea of what its moves are supposed to achieve (checkmate) or even what effect the moves have on the positions of the pieces. And of course chess is a tiny part of the real world. For more realistic problems, we would need much vaster grandmaster databases, and they simply don't exist.[1]

An alternative is **reinforcement learning** (RL), in which an agent interacts with the world and periodically receives **rewards** (or, in the terminology of psychology, **reinforcements**) that reflect how well it is doing. For example, in chess the reward is 1 for winning, 0 for losing, and ½ for a draw. We have already seen the concept of rewards in Chapter 16 for **Markov decision processes** (MDPs). Indeed, the goal is the same in reinforcement learning: maximize the expected sum of rewards. Reinforcement learning differs from "just solving an MDP" because the agent is not *given* the MDP as a problem to solve; the agent is *in* the MDP. It may not know the transition model or the reward function, and it has to act in order to learn more. Imagine playing a new game whose rules you don't know; after a hundred or so moves, the referee tells you "You lose." That is reinforcement learning in a nutshell.

From our point of view as designers of AI systems, providing a reward signal to the agent is usually much easier than providing labeled examples of how to behave. First, the reward function is often (as we saw for chess) very concise and easy to specify: it requires only a few lines of code to tell the chess agent if it has won or lost the game or to tell the car-racing agent that it has won or lost the race or has crashed. Second, we don't have to be experts, capable of supplying the correct action in any situation, as would be the case if we tried to apply supervised learning.

It turns out, however, that a little bit of expertise can go a long way in reinforcement learning. The two examples in the preceding paragraph—the win/loss rewards for chess and racing—are what we call **sparse** rewards, because in the vast majority of states the agent is given no informative reward signal at all. In games such as tennis and cricket, we can easily supply additional rewards for each point won or for each run scored. In car racing, we could reward the agent for making progress around the track in the right direction. When learning to crawl, any forward motion is an achievement. These intermediate rewards make learning much easier.

As long as we can provide the correct reward signal to the agent, reinforcement learning provides a very general way to build AI systems. This is particularly true for *simulated* environments, where there is no shortage of opportunities to gain experience. The addition of deep learning as a tool within RL systems has also made new applications possible, including learning to play Atari video games from raw visual input (Mnih *et* *al.*, 2013), controlling robots (Levine *et al.*, 2016), and playing poker (Brown and Sandholm, 2017).

Literally hundreds of different reinforcement learning algorithms have been devised, and many of them can employ as tools a wide range of learning methods from Chapters 19, 21, and 22. In this chapter, we cover the basic ideas and give some sense of the variety of approaches through a few examples. We categorize the approaches as follows:

- **Model-based reinforcement learning:** In these approaches the agent uses a transition model of the environment to help interpret the reward signals and to make decisions about how to act. The model may be initially unknown, in which case the agent learns the model from observing the effects of its actions, or it may already be known—for example, a chess program may know the rules of chess even if it does not know how to choose good moves. In partially observable environments, the transition model is also useful for **state estimation** (see Chapter 14). Model-based reinforcement learning systems often learn a **utility function** $U(s)$, defined (as in Chapter 16) in terms of the sum of rewards from state s onward.[2]
- **Model-free reinforcement learning:** In these approaches the agent neither knows nor learns a transition model for the environment. Instead, it learns a more direct representation of how to behave. This comes in one of two varieties:
  - **Action-utility learning:** We introduced action-utility functions in Chapter 16. The most common form of action-utility learning is **Q-learning**, where the agent learns a **Q-function**, or quality-function, $Q(s, a)$, denoting the sum of rewards from state s onward if action a is taken. Given a Q-function, the agent can choose what to do in s by finding the action with the highest Q-value.

- **Policy search:** The agent learns a policy $\pi(s)$ that maps directly from states to Policy search actions. In the terminology of Chapter 2, this a **reflex agent.**

We begin in Section 23.2 with **passive reinforcement learning**, where the agent's policy is fixed and the task is to learn the utilities of states (or of state–action pairs); this could also involve learning a model of the environment. (An understanding of Markov decision processes, as described in Chapter 16, is essential for this section.) Section 23.3 covers **active reinforcement learning**, where the agent must also figure out what to do. The principal issue is **exploration**: an agent must experience as much as possible of its environment in order to learn how to behave in it. Section 23.4 discusses how an agent can use inductive learning (including deep learning methods) to learn much faster from its experiences. We also discuss other approaches that can help scale up RL to solve real problems, including providing intermediate pseudorewards to guide the learner and organizing behavior into a hierarchy of actions. Section 23.5 covers methods for policy search. In Section 23.6, we explore **apprenticeship learning**: training a learning agent using demonstrations rather than reward signals. Finally, Section 23.7 reports on applications of reinforcement learning.

23.2 Passive Reinforcement Learning

We start with the simple case of a fully observable environment with a small number of actions and states, in which an agent already has a fixed policy $\pi(s)$ that determines its actions. The agent is trying to learn the utility function $U^{\pi}(s)$—the expected total discounted reward if policy π is executed beginning in state s. We call this a **passive learning agent**.

The passive learning task is similar to the **policy evaluation** task, part of the policy iteration algorithm described in Section 16.2.2. The difference is that the passive learning agent does not know the transition model $P(s' \mid s,a)$, which specifies the probability of reaching state s' from state s after doing action a; nor does it know the reward function $R(s,a,s')$, which specifies the reward for each transition.

We will use as our example the 4 × 3 world introduced in Chapter 16. Figure 23.1 shows the optimal policies for that world and the corresponding utilities. The agent executes a set of **trials** in the environment using its policy π. In each trial, the agent starts in state (1,1) and experiences a sequence of state transitions until it reaches one of the terminal states, (4,2) or (4,3). Its percepts supply both the current state and the reward received for the transition that just occurred to reach that state. Typical trials might look like this:

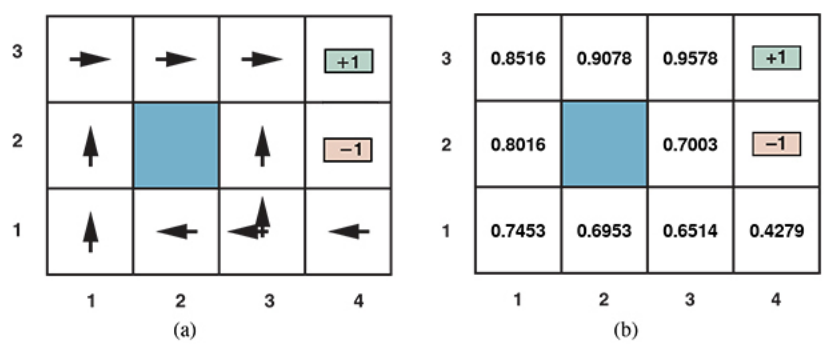


Figure 23.1 (a) The ptimal policies for the stochastic environment with $R(s, a, s') = -0.04$ for transitions between nonterminal states. There are two policies because in state (3,1) both *Left* and *Up* are optimal. We saw this before in Figure 16.2. (b) The utilities of the states in the 4 × 3 world, given policy π.

$$(1,1) \xrightarrow[Up]{\textbf{-.04}} (1,2) \xrightarrow[Up]{\textbf{-.04}} (1,3) \xrightarrow[Right]{\textbf{-.04}} (1,2) \xrightarrow[Up]{\textbf{-.04}} (1,3) \xrightarrow[Right]{\textbf{-.04}} (2,3) \xrightarrow[Right]{\textbf{-.04}} (3,3) \xrightarrow[Right]{\textbf{+1}} (4,3)$$

$$(1,1) \xrightarrow[Up]{\textbf{-.04}} (1,2) \xrightarrow[Up]{\textbf{-.04}} (1,3) \xrightarrow[Right]{\textbf{-.04}} (2,3) \xrightarrow[Right]{\textbf{-.04}} (3,3) \xrightarrow[Right]{\textbf{-.04}} (3,2) \xrightarrow[Up]{\textbf{-.04}} (3,3) \xrightarrow[Right]{\textbf{+1}} (4,3)$$

$$(1,1) \xrightarrow[Up]{\textbf{-.04}} (1,2) \xrightarrow[Up]{\textbf{-.04}} (1,3) \xrightarrow[Right]{\textbf{-.04}} (2,3) \xrightarrow[Right]{\textbf{-.04}} (3,3) \xrightarrow[Right]{\textbf{-.04}} (3,2) \xrightarrow[Up]{\textbf{-1}} (4,2)$$

Note that each transition is annotated with both the action taken and the reward received at the next state. The object is to use the information about rewards to learn the expected utility $U^{\pi}(s)$ associated with each nonterminal state s. The utility is defined to be the expected sum of (discounted) rewards obtained if policy π is followed. As in Equation (16.2) on page 557, we write

$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t, \pi(S_t), S_{t+1})\right], \quad (23.1)$$

where $R(S_t, \pi(S_t), S_{t+1})$ is the reward received when action $\pi(S_t)$is taken in state S_t and reaches state S_{t+1}. Note that S_t is a random variable denoting the state reached at time t when executing policy π, starting from state $S_0 = s$. We will include a **discount factor** γ in all of our equations, but for the 4 × 3 world we will set γ = 1, which means no discounting.

23.2.1 Direct utility estimation

The idea of **direct utility estimation** is that the utility of a state is defined as the expected total reward from that state onward (called the expected **reward-to-go**), and that each trial provides a *sample* of this quantity for each state visited. For example, the first of the three trials shown earlier provides a sample total reward of 0.76 for state (1,1), two samples of 0.80 and 0.88 for (1,2), two samples of 0.84 and 0.92 for (1,3), and so on. Thus, at the end of each sequence, the algorithm calculates the observed reward-to-go for each state and updates the estimated utility for that state accordingly, just by keeping a running average for each state in a table. In the limit of infinitely many trials, the sample average will converge to the true expectation in Equation (23.1).

This means that we have reduced reinforcement learning to a standard supervised learning problem in which each example is a (*state*, *reward-to-go*) pair. We have a lot of powerful algorithms for supervised learning, so this approach seems promising, but it ignores an important constraint: *The utility of a state is determined by the reward and the expected utility of the successor states.* More specifically, the utility values obey the Bellman equations for a fixed policy (see also Equation (16.14)):

$$U_i(s) = \sum_{s\prime} P(s\prime|s, \pi_i(s))[R(s, \pi_i(s), s\prime) + \gamma U_i(s\prime)]. \quad (23.2)$$

By ignoring the connections between states, direct utility estimation misses opportunities for learning. For example, the second of the three trials given earlier reaches the state (3,2), which has not previously been visited. The next transition reaches (3,3), which is known from the first trial to have a high utility. The Bellman equation suggests immediately that (3,2) is also likely to have a high utility, because it leads to (3,3), but direct utility estimation learns nothing until the end of the trial. More broadly, we can view direct utility estimation as searching for U in a hypothesis space that is much larger than it needs to be, in that it includes many functions that violate the Bellman equations. For this reason, the algorithm often converges very slowly.

23.2.2 Adaptive dynamic programming

An **adaptive dynamic programming** (or ADP) agent takes advantage of the constraints among the utilities of states by learning the transition model that connects them and solving the corresponding Markov decision process using dynamic programming. For a passive learning agent, this means plugging the learned transition model $P(s' \mid s, \pi(s))$ and the observed rewards $R(s, \pi(s), s')$ into Equation (23.2) to calculate the utilities of the states. As we remarked in our discussion of policy iteration in Chapter 16, these Bellman equations are linear when the policy π is fixed, so they can be solved using any linear algebra package.

Alternatively, we can adopt the approach of **modified policy iteration** (see page 568), using a simplified value iteration process to update the utility estimates after each change to the learned model. Because the model usually changes only slightly with each observation, the value iteration process can use the previous utility estimates as initial values and typically converge very quickly.

Learning the transition model is easy, because the environment is fully observable. This means that we have a supervised learning task where the input for each training example is a state–action pair, (s, a), and the output is the resulting state, s'. The transition model $P(s' \mid s, a)$ is represented as a table and it is estimated directly from the counts that are accumulated in $N_{s' \mid sa}$. The counts record how often state s' is reached when executing a in s. For example, in the three trials given on page 843, *Right* is executed four times in (3,3) and the resulting state is (3,2) twice and (4,3) twice, so $P((3,2) \mid (3,3), Right)$ and $P((4,3) \mid (3,3), Right)$ are both estimated to be ½.

The full agent program for a passive ADP agent is shown in Figure 23.2. Its performance on the 4 × 3 world is shown in Figure 23.3. In terms of how quickly its value estimates improve, the ADP agent is limited only by its ability to learn the transition model. In this sense, it provides a standard against which to measure any other reinforcement learning algorithms. It is, however, intractable for large state spaces. In backgammon, for example, it would involve solving roughly 10^{20} equations in 10^{20} unknowns.

---

function PASSIVE-ADP-LEARNER(*percept*) **returns** an action
  inputs: *percept*, a percept indicating the current state s' and reward signal r
  persistent: π, a fixed policy
    mdp, an MDP with model P, rewards R, actions A, discount γ
    U, a table of utilities for states, initially empty
    $N_{s'|s,a}$, a table of outcome count vectors indexed by state and action, initially zero
    s, a, the previous state and action, initially null

  if s' is new **then** $U[s'] \leftarrow 0$
  if s is not null **then**
    increment $N_{s'|s,a}[s,a][s']$
    $R[s, a, s'] \leftarrow r$
    add a to $A[s]$
    $\mathbf{P}(\cdot \mid s,a) \leftarrow$ NORMALIZE($N_{s'|s,a}[s, a]$)
    $U \leftarrow$ POLICYEVALUATION(π, U, mdp)
    $s, a \leftarrow s', \pi[s']$
    return a

Figure 23.2 A passive reinforcement learning agent based on adaptive dynamic programming. The agent chooses a value for γ and then incrementally computes the P and R values of the MDP. The POLICY-EVALUATION function solves the fixed-policy Bellman equations, as described on page 567.

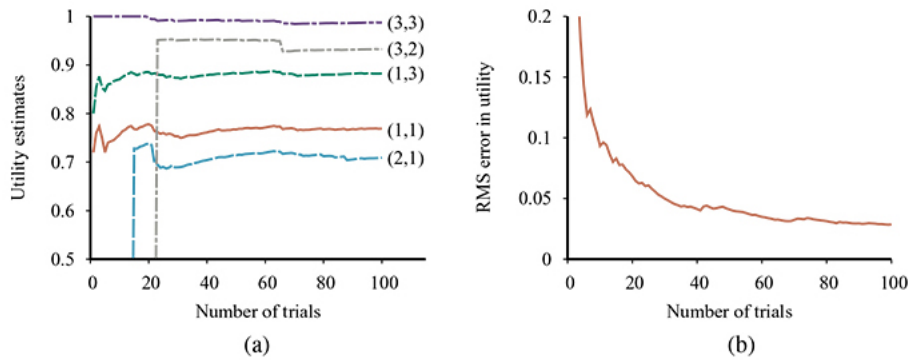


Figure 23.3 The passive ADP learning curves for the 4 × 3 world, given the optimal policy shown in Figure 23.1. (a) The utility estimates for a selected subset of states, as a function of the number of trials. Notice that it takes 14 and 23 trials respectively before the rarely visited states (2,1) and (3,2) "discover" that they connect to the +1 exit state at (4,3). (b) The root-mean-square error (see Appendix A) in the estimate for $U(1, 1)$, averaged over 50 runs of 100 trials each.

23.2.3 Temporal-difference learning

Solving the underlying MDP as in the preceding section is not the only way to bring the Bellman equations to bear on the learning problem. Another way is to use the observed transitions to adjust the utilities of the observed states so that they agree with the constraint equations. Consider, for example, the transition from (1,3) to (2,3) in the second trial on page 843. Suppose that as a result of the first trial, the utility estimates are $U^\pi(1, 3) = 0.88$ and $U^\pi(2, 3) = 0.96$. Now, if this transition from (1,3) to (2,3) occurred all the time, we would expect the utilities to obey the equation

$$U^\pi(1,3) = -0.04 + U^\pi(2,3),$$

so $U^\pi(1, 3)$ would be 0.92. Thus, its current estimate of 0.88 might be a little low and should be increased. More generally, when a transition occurs from state s to state s' via action $\pi(s)$,
we apply the following update to $U^\pi(s)$:

$$U^\pi(s) \leftarrow U^\pi(s) + \alpha[R(s,\pi(s),s\prime)\ +\ \gamma U^\pi(s\prime) - U^\pi(s)]. \tag{23.3}$$

Here, α is the **learning rate** parameter. Because this update rule uses the difference in utilities between successive states (and thus successive times), it is often called the **temporal-difference** (TD) equation. Just as in the weight update rules from Chapter 19 (e.g., Equation (19.6) on page 698), the TD term $R(s,\pi(s), s') + \gamma U^\pi(s') - U^\pi(s)$ is effectively an error signal, and the update is intended to reduce the error.

All temporal-difference methods work by adjusting the utility estimates toward the ideal equilibrium that holds locally when the utility estimates are correct. In the case of passive learning, the equilibrium is given by Equation (23.2). Now Equation (23.3) does in fact cause the agent to reach the equilibrium given by Equation (23.2), but there is some subtlety involved. First, notice that the update involves only the observed successor s', whereas the actual equilibrium conditions involve all possible next states. One might think that this causes an improperly large change in $U^\pi(s)$ when a very rare transition occurs; but, in fact, because rare transitions occur only rarely, the *average value* of $U^{\pi}(s)$ will converge to the correct quantity in the limit, even if the value itself continues to fluctuate.

Furthermore, if we turn the parameter α into a function that decreases as the number of times a state has been visited increases, as shown in Figure 23.4, then $U^{\pi}(s)$ itself will converge to the correct value.[3] Figure 23.5 illustrates the performance of the passive TD agent on the 4 × 3 world. It does not learn quite as fast as the ADP agent and shows much higher variability, but it is much simpler and requires much less computation per observation. Notice that *TD does not need a transition model to perform its updates.* The environment itself supplies the connection between neighboring states in the form of observed transitions.

```
function PASSIVE-TD-LEARNER(percept) returns an action
   inputs: percept, a percept indicating the current state s′ and reward signal r
   persistent: π, a fixed policy
               s, the previous state, initially null
               U, a table of utilities for states, initially empty
               N_s, a table of frequencies for states, initially zero

   if s′ is new then U[s′] ← 0
   if s is not null then
      increment N_s[s]
      U[s] ← U[s] + α(N_s[s]) × (r + γ U[s′] - U[s])
   s ← s′
   return π[s′]
```

Figure 23.4 A passive reinforcement learning agent that learns utility estimates using temporal differences. The step-size function $\alpha(n)$ is chosen to ensure convergence.

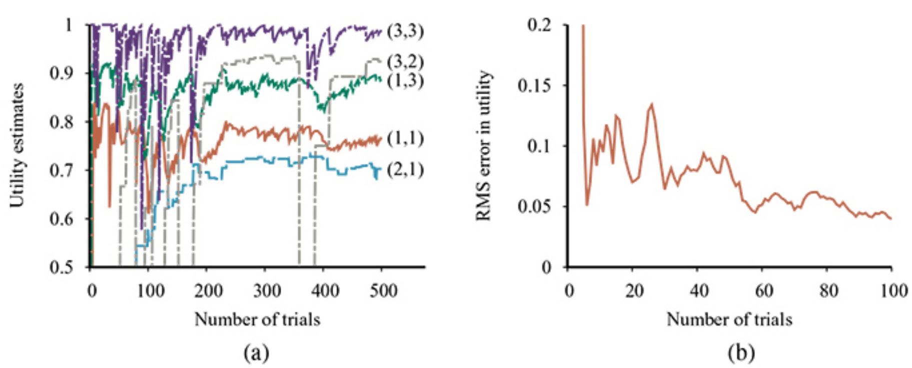


Figure 23.5 The TD learning curves for the 4 × 3 world. (a) The utility estimates for a selected subset of states, as a function of the number of trials, for a single run of 500 trials. Compare with the run of 100

trials in Figure 23.3(a). (b) The root-mean-square error in the estimate for $U(1, 1)$, averaged over 50 runs of 100 trials each.

The ADP and TD approaches are closely related. Both try to make local adjustments to the utility estimates in order to make each state "agree" with its successors. One difference isthat TD adjusts a state to agree with its *observed* successor (Equation (23.3)), whereas ADP adjusts the state to agree with *all* of the successors that might occur, weighted by their probabilities (Equation (23.2)). This difference disappears when the effects of TD adjustments are averaged over a large number of transitions, because the frequency of each successor in the set of transitions is approximately proportional to its probability. A more important difference is that whereas TD makes a single adjustment per observed transition, ADP makes as many as it needs to restore consistency between the utility estimates U and the transition model P. Although the observed transition makes only a local change in P, its effects might need to be propagated throughout U. Thus, TD can be viewed as a crude but efficient first approximation to ADP.

Each adjustment made by ADP could be seen, from the TD point of view, as a result of a **pseudoexperience** generated by simulating the current transition model. It is possible to extend the TD approach to use a transition model to generate several pseudoexperiences— transitions that the TD agent can imagine *might* happen, given its current model. For each observed transition, the TD agent can generate a large number of imaginary transitions. In this way, the resulting utility estimates will approximate more and more closely those of ADP—of course, at the expense of increased computation time.

In a similar vein, we can generate more efficient versions of ADP by directly approximating the algorithms for value iteration or policy iteration. Even though the value iteration algorithm is efficient, it is intractable if we have, say, 10^{100} states. However, many of the necessary adjustments to the state values on each iteration will be extremely tiny. One possible approach to generating reasonably good answers quickly is to bound the number of adjustments made after each observed transition. One can also use a heuristic to rank the possible adjustments so as to carry out only the most significant ones. The **prioritized sweeping** heuristic prefers to make adjustments to states whose *likely* successors have just undergone a *large* adjustment in their own utility estimates.

Using heuristics like this, approximate ADP algorithms can learn roughly as fast as full ADP, in terms of the number of training sequences, but can be orders of magnitude more efficient in terms of total computation (see Exercise 23.PRSW). This enables them to handle state spaces that are far too large for full ADP. Approximate ADP algorithms have an additional advantage: in the early stages of learning a new environment, the transition model P often will be far from correct, so there is little point in calculating an exact utility function to match it. An approximation algorithm can use a minimum adjustment size that decreases as the transition model becomes more accurate. This eliminates the very long runs of value iteration that can occur early in learning due to large changes in the model.

23.3 Active Reinforcement Learning

A passive learning agent has a fixed policy that determines its behavior. An **active learning agent** gets to decide what actions to take. Let us begin with the adaptive dynamic programming (ADP) agent and consider how it can be modified to take advantage of this new freedom.

First, the agent will need to learn a complete transition model with outcome probabilities for *all* actions, rather than just the model for the fixed policy. The learning mechanism used by PASSIVE-ADP-AGENT will do just fine for this. Next, we need to take into account the fact that the agent has a choice of actions. The utilities it needs to learn are those defined by the *optimal* policy; they obey the Bellman equations (which we repeat here):

$$U(s) = \max_{a \in A(s)} \sum_{s\prime} P(s\prime|s,a)[R(s,a,s\prime) + \gamma U(s\prime)]\,. \qquad (23.4)$$

These equations can be solved to obtain the utility function U using the value iteration or policy iteration algorithms from Chapter 16.

The final issue is what to do at each step. Having obtained a utility function U that is optimal for the learned model, the agent can extract an optimal action by one-step look-ahead to maximize the expected utility; alternatively, if it uses policy iteration, the optimal policy is already available, so it could simply execute the action the optimal policy recommends. But should it?

23.3.1 Exploration

Figure 23.6 shows the results of one sequence of trials for an ADP agent that follows the recommendation of the optimal policy for the learned model at each step. The agent *does not* learn the true utilities or the true optimal policy! What happens instead is that in the third trial, it finds a policy that reaches the +1 reward along the lower route via (2,1), (3,1), (3,2), and (3,3). (See Figure 23.6(b).) After experimenting with minor variations, from the eighth trial onward it sticks to that policy, never learning the utilities of the other states and never finding the optimal route via (1,2), (1,3), and (2,3). We will call this agent a **greedy agent**, because it greedily takes the action that it currently believes to be optimal at each step. Sometimes greed pays off and the agent converges to the optimal policy, but often it does not.

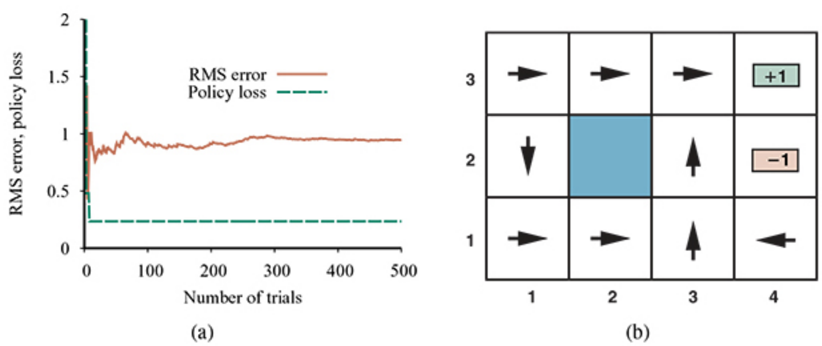


Figure 23.6 Performance of a greedy ADP agent that executes the action recommended by the optimal policy for the learned model. (a) The root mean square (RMS) error averaged across all nine nonterminal squares and the policy loss in (1,1). We see that the policy converges quickly, after just eight trials, to a suboptimal policy with a loss of 0.235. (b) The suboptimal policy to which the greedy agent converges in this particular sequence of trials. Notice the *Down* action in (1,2).

How can it be that choosing the optimal action leads to suboptimal results? The answer is that the learned model is not the same as the true environment; what is optimal in the learned model can therefore be suboptimal in the true environment. Unfortunately, the agent does not know what the true environment is, so it cannot compute the optimal action for the true environment. What, then, should it do?

The greedy agent has overlooked the fact that actions do more than provide *rewards;* they also provide *information* in the form of percepts in the resulting states. As we saw with **bandit problems** in Section 16.3, an agent must make a tradeoff between **exploitation** of the current best action to maximize its short-term reward and **exploration** of previously unknown states to gain information that can lead to a change in policy (and to greater rewards in the future). In the real world, one constantly has to decide between continuing in a comfortable existence, versus striking out into the unknown in the hopes of a better life.

Although bandit problems are difficult to solve exactly to obtain an *optimal* exploration scheme, it is nonetheless possible to come up with a scheme that will eventually discover an optimal policy, even if it might take longer to do so than is optimal. Any such scheme should not be greedy in terms of the immediate next move, but should be what is called "greedy in the limit of infinite exploration," or **GLIE**. A GLIE scheme must try each action in each state an unbounded number of times to avoid having a finite probability that an optimal action is missed. An ADP agent using such a scheme will eventually learn the true transition model, and can then operate under exploitation.

There are several GLIE schemes; one of the simplest is to have the agent choose a random action at time step t with probability $1/t$ and to follow the greedy policy otherwise. While this does eventually converge to an optimal policy, it can be slow. A better approach would give some weight to actions that the agent has not tried very often, while tending to avoid actions that are believed to be of low utility (as we did with Monte Carlo tree search in Section 6.4). This can be implemented by altering the constraint equation (23.4) so that it assigns a higher utility estimate to relatively unexplored state–action pairs.

This amounts to an optimistic prior over the possible environments and causes the agent to behave initially as if there were wonderful rewards scattered all over the place. Let us use $U^{+}(s)$ to denote the optimistic estimate of the utility (i.e., the expected reward-to-go) of the state s, and let $N(s, a)$ be the number of times action a has been tried in state s. Suppose we are using value iteration in an ADP learning agent; then we need to rewrite the update equation (Equation (16.10) on page 563) to incorporate the optimistic estimate:

$$U+(s) \leftarrow \max_a f(\sum_{s\prime} P(s\prime|s,a)[R(s,a,s\prime)+\gamma U^{+}(s\prime)], N(s,a)). \tag{23.5}$$

Here, f is the **exploration function**. The function $f(u, n)$ determines how greed (preference for high values of the utility u) is traded off against curiosity (preference for actions that have not been tried often and have a low count n). The function should be increasing in u and decreasing in n. Obviously, there are many possible functions that fit these conditions. One particularly simple definition is

$$f(u,n) = \begin{cases} R^{+} & \text{if } n \ < N_e \\ u & \text{otherwise,} \end{cases}$$

where R^+ is an optimistic estimate of the best possible reward obtainable in any state and N_e is a fixed parameter. This will have the effect of making the agent try each state–action pair at least N_e times. The fact that U^+ rather than U appears on the right-hand side of Equation (23.5) is very important. As exploration proceeds, the states and actions near the start state might well be tried a large number of times. If we used U, the more pessimistic utility estimate, then the agent would soon become disinclined to explore further afield. The use of U^+ means that the benefits of exploration are propagated back from the edges of unexplored regions, so that actions that lead *toward* unexplored regions are weighted more highly, rather than just actions that are themselves unfamiliar.

The effect of this exploration policy can be seen clearly in Figure 23.7(b), which shows a rapid convergence toward zero policy loss, unlike with the greedy approach. A very nearly optimal policy is found after just 18 trials. Notice that the RMS error in the utility estimates does not converge as quickly. This is because the agent stops exploring the unrewarding parts of the state space fairly soon, visiting them only "by accident" thereafter. However, it makes perfect sense for the agent not to care about the exact utilities of states that it knows are undesirable and can be avoided. There is not much point in learning about the best radio station to listen to while falling off a cliff.

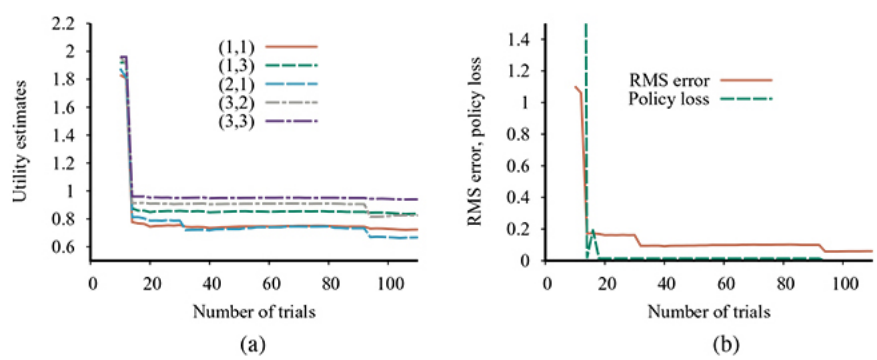


Figure 23.7 Performance of the exploratory ADP agent using $R^+ = 2$ and $N_e = 5$. (a) Utility estimates for selected states over time. (b) The RMS error in utility values and the associated policy loss.

23.3.2 Safe exploration

So far we have assumed that an agent is free to explore as it wishes—that any negative rewards serve only to improve its model of the world. That is, if we play a game of chess and lose, we suffer no damage (except perhaps to our pride), and whatever we learned will make us a better player in the next game. Similarly, in a simulation environment for a self-driving car, we could explore the limits of the car's performance, and any accidents give us more information. If the car crashes, we just hit the reset button.

Unfortunately, the real world is less forgiving. If you are a baby sunfish, your probability of surviving to adulthood is about 0.00000001. Many actions are **irreversible**, in the sense defined for online search agents in Section 4.5: no subsequent sequence of actions can restore the state to what it was before the irreversible action was taken. In the worst case, the agent enters an **absorbing state** where no actions have any effect and no rewards are received.

In many practical settings, we cannot afford to have our agents taking irreversible actions or entering absorbing states. For example, an agent learning to drive in a real car should avoid taking actions that might lead to any of the following:

- states with large negative rewards, such as serious car crashes;
- states from which there is no escape, such as driving the car into a deep ditch;
- states that permanently limit future rewards, such as damaging the car's engine so that its maximum speed is reduced.

We can end up in a bad state either because our model is *unknown*, and we actively choose to explore in a direction that turns out to be bad, or because our model is *incorrect* and we don't know that a given action can have a disastrous result. Note that the algorithm in Figure 23.2 is using maximum-likelihood estimation (see Chapter 21) to learn the transition model; moreover, by choosing a policy based solely on the *estimated* model, it is acting *as if* the model were correct. This is not necessarily a good idea! For example, a taxi agent that didn't know how traffic lights work might ignore a red light once or twice with no ill effects and then formulate a policy to ignore all red lights from then on.

A better idea would be to choose a policy that works reasonably well for the whole range of models that have a reasonable chance of being the true model, even if the policy happens to be suboptimal for the maximum-likelihood model. There are three mathematical approaches that have this flavor.

The first approach, **Bayesian reinforcement learning**, assumes a prior probability $P(h)$ over hypotheses h about what the true model is; the posterior probability $P(h \mid \mathbf{e})$ is obtained in the usual way by Bayes' rule given the observations to date. Then, if the agent has decided to stop learning, the optimal policy is the one that gives the highest expected utility. Let U_h^π be the expected utility, averaged over all possible start states, obtained by executing policy π in model h. Then we have

$$\pi^* = \underset{\pi}{\operatorname{argmax}} \sum_h P(h|\mathbf{e}) U_h^\pi .$$

In some special cases, this policy can even be computed! If the agent will continue learning in the future, however, then finding an optimal policy becomes considerably more difficult, because the agent must consider the effects of future observations on its beliefs about the transition model. The problem becomes an **exploration POMDP** whose belief states are distributions over models. In principle, this exploration POMDP can be formulated and solved before the agent ever sets foot in the world. (Exercise 23.EPOM asks you to do this for the Minesweeper game to find the best first move.) The result is a complete strategy that tells the agent what to do next given any possible percept sequence. Solving the exploration POMDP is usually wildly intractable, but the concept provides an analytical foundation for understanding the exploration problem described in Section 23.3.

It is worth noting that being perfectly Bayesian will not protect the agent from an untimely death. Unless the prior gives some indication of percepts that suggest danger, there is nothing to prevent the agent from taking an exploratory action that leads to an absorbing state. For example, it used to be thought that human infants had an innate fear of heights and would not crawl off a cliff, but this turns out not to be true (Adolph *et al.*, 2014).

The second approach, derived from **robust control theory**, allows for a set of possible models $\mathcal{H}$ without assigning probabilities to them, and defines an optimal robust policy as one that gives the best outcome in the *worst case* over $\mathcal{H}$:

$$\pi^* = \underset{\pi}{\operatorname{argmax}} \ \min_h U_h^\pi .$$

Often, the set $\mathcal{H}$ will be the set of models that exceed some likelihood threshold on $P(h \mid \mathbf{e})$, so the robust and Bayesian approaches are related.

The robust control approach can be considered as a game between the agent and an adversary, where the adversary gets to pick the worst possible result for any action, and the policy we get is the minimax solution for the game. Our logical wumpus agent (see Section 7.7) is a robust control agent in this way: it considers all models that are logically possible, and does not explore any locations that could possibly contain a pit or a wumpus, so it is finding the action with maximum utility in the worst case over all possible hypotheses.

The problem with the worst-case assumption is that it results in overly conservative behavior. A self-driving car that assumes that every other driver *will try to collide with it* has no choice but to stay in the garage. Real life is full of such risk–reward tradeoffs.

Although one reason for venturing into reinforcement learning was to escape the need for a human teacher (as in supervised learning), it turns out that human knowledge can help keep a system safe. One way is to record a series of actions by an experienced teacher, so that the system will act reasonably from the start, and can learn to improve from there. A second way is for a human to write down constraints on what a system can do, and have a program outside of the reinforcement learning system enforce those constraints. For example, when training an autonomous helicopter, a partial policy can be provided that takes over control when the helicopter enters a state from which any further unsafe actions would lead to an irrecoverable state—one in which the safety controller cannot guarantee that the absorbing state will be avoided. In all other states, the learning agent is free to do as it pleases.

23.3.3 Temporal-difference Q-learning

Now that we have an active ADP agent, let us consider how to construct an active temporal-difference (TD) learning agent. The most obvious change is that the agent will have to learn a transition model so that it can choose an action based on $U(s)$ via one-step look-ahead. The model acquisition problem for the TD agent is identical to that for the ADP agent, and the TD update rule remains unchanged. Once again, it can be shown that the TD algorithm will converge to the same values as ADP, as the number of training sequences tends to infinity.

The **Q-learning** method avoids the need for a model by learning an action-utility function $Q(s, a)$ instead of a utility function $U(s)$. $Q(s, a)$ denotes the expected total discounted reward if the agent takes action a in s and acts optimally thereafter. Knowing the Q-function enables the agent to act optimally simply by choosing $\operatorname{argmax}_a Q(s, a)$, with no need for look-ahead based on a transition model.

We can also derive a model-free TD update for the Q-values. We begin with the Bellman equation for $Q(s, a)$, repeated here from Equation (16.8):

$$Q(s,a) = \sum_{s\prime} P(s\prime \middle| s,a)[R(s,a,s\prime) + \gamma \max_{a\prime} Q(s\prime a\prime)] \qquad (23.6)$$

From this, we can write down the Q-learning TD update, by analogy to the TD update for utilities in Equation (23.3):

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R(s,a,s\prime) + \gamma \max_{a\prime} Q(s\prime a\prime) \ - \ Q(s,a)] \qquad (23.7)$$

This update is calculated whenever action a is executed in state s leading to state s'. As in Equation (23.3), the term $R(s, a, s') + \gamma \max_{a'} Q(s', a') - Q(s, a)$ represents an error that the update is trying to minimize.

The important part of this equation is what it does not contain: *a TD Q-learning agent does not need a transition model, $P(s' \mid s, a)$, either for learning or for action selection.* As noted at the beginning of the chapter, model-free methods can be applied even in very complex domains because no model need be provided or learned.

On the other hand, the Q-learning agent has no means of looking into the future, so it may have difficulty when rewards are sparse and long action sequences must be constructed to reach them.

The complete agent design for an exploratory TD Q-learning agent is shown in Figure 23.8. Notice that it uses exactly the same exploration function f as that used by the exploratory ADP agent—hence the need to keep statistics on actions taken (the table N). If a simpler exploration policy is used—say, acting randomly on some fraction of steps, where the fraction decreases over time—then we can dispense with the statistics.

---

function Q-LEARNING-AGENT(*percept*) **returns** an action
  inputs: *percept*, a percept indicating the current state s' and reward signal r
  persistent: Q, a table of action values indexed by state and action, initially zero
    N_{sa}, a table of frequencies for state–action pairs, initially zero
    s, a, the previous state and action, initially null

  if s is not null **then**
    increment $N_{sa}[s,a]$
    $Q[s,a] \leftarrow Q[s,a] + \alpha(N_{sa}[s,a])(r + \gamma \max_{a'} Q[s',a'] - Q[s,a])$
  $s,a \leftarrow s', \operatorname{argmax}_{a'} f(Q[s',a'], N_{sa}[s',a'])$
  return a

Figure 23.8 An exploratory Q-leaming agent. It is an active learner that learns the value *Q(s, a)* of each action in each situation. It uses the same exploration function *f* as the exploratory ADP agent, but avoids having to learn the transition model.

Q-learning has a close relative called **SARSA** (for state, action, reward, state, action). The update rule for SARSA is very similar to the Q-learning update rule (Equation (23.7)), except that SARSA updates with the Q-value of the action a' that is actually taken:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R(s,a,s') + \gamma Q(s',a') - Q(s,a)], \qquad (23.8)$$

The rule is applied at the end of each s, a, r, s', a' quintuplet—hence the name. The difference from Q-learning is quite subtle: whereas Q-learning backs up the Q-value from the best action in s', SARSA waits until an action is taken and backs up the Q-value for that action. If the agent is greedy and always takes the action with the best Q-value, the two algorithms are identical. When exploration is happening, however, they differ: if the exploration yields a negative reward, SARSA penalizes the action, while Q-learning does not.

Q-learning is an **off-policy** learning algorithm, because it learns Q-values that answer the question “What would this action be worth in this state, assuming that I stop using whatever policy I am using now, and start acting according to a policy that chooses the best action (according to my estimates)?” SARSA is an **on-policy** algorithm: it learns Q-values that answer the question “What would this action be worth in this state, assuming I stick with mypolicy?” Q-learning is more flexible than SARSA, in the sense that a Q-learning agent can learn how to behave well when under the control of a wide variety of exploration policies. On the other hand, SARSA is appropriate if the overall policy is even partly controlled by other agents or programs, in which case it is better to learn a Q-function for what will actually happen rather than what would happen if the agent got to pick estimated best actions. Both Q-learning and SARSA learn the optimal policy for the 4×3 world, but they do so at a much slower rate than the ADP agent. This is because the local updates do not enforce consistency among all the Q-values via the model.

23.4 Generalization in Reinforcement Learning

So far, we have assumed that utility functions and Q-functions are represented in tabular form with one output value for each state. This works for state spaces with up to about 10^6 states, which is more than enough for our toy two-dimensional grid environments. But in real-world environments with many more states, convergence will be too slow. Backgammon is simpler than most real-world applications, yet it has about 10^{20} states. We cannot easily visit them all in order to learn how to play the game.

Chapter 6 introduced the idea of an **evaluation function** as a compact measure of desirability for potentially vast state spaces. In the terminology of this chapter, the evaluation function is an approximate utility function; we use the term **function approximation** for the process of constructing a compact approximation of the true utility function or Q-function. For example, we might approximate the utility function using a weighted linear combination of **features** $f_1,\ldots, f_n$:

$$\widehat{U}_\theta(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + \cdots + \theta_n f_n(s).$$

Instead of learning 10^{20} state values in a table, a reinforcement learning algorithm can learn, say, 20 values for the parameters $\theta = \theta_1,\ldots,\theta_{20}$ that make $\hat{U}_\theta$ a good approximation to the true utility function. Sometimes this approximate utility function is combined with look-ahead search to produce more accurate decisions. Adding look-ahead search means that effective behavior can be generated from a much simpler utility function approximator that is learnable from far fewer experiences.

Function approximation makes it practical to represent utility (or Q) functions for very large state spaces, but more importantly, it allows for inductive generalization: the agent can generalize from states it has visited to states it has not yet visited. Tesauro (1992) used this technique to build a backgammon-playing program that played at human champion level, even though it explored only a trillionth of the complete state space of backgammon.

23.4.1 Approximating direct utility estimation

The method of direct utility estimation (Section 23.2) generates trajectories in the state space and extracts, for each state, the sum of rewards received from that state onward until termination. The state and the sum of rewards received constitute a training example for a **supervised learning** algorithm. For example, suppose we represent the utilities for the 4 × 3 world using a simple linear function, where the features of the squares are just their x and y coordinates. In that case, we have

$$\widehat{U}_\theta(x, y) = \theta_0 + \theta_1 x + \theta_2 y. \qquad (23.9)$$

Thus, if $(\theta_0, \theta_1, \theta_2) = (0.5, 0.2, 0.1)$, then $\hat{U}_\theta((1, 1) = 0.8$. Given a collection of trials, we obtain a set of sample values of $\hat{U}_\theta(x, y)$, and we can find the best fit, in the sense of minimizing the squared error, using standard linear regression (see Chapter 19).

For reinforcement learning, it makes more sense to use an *online* learning algorithm that updates the parameters after each trial. Suppose we run a trial and the total reward obtained starting at (1,1) is 0.4. This suggests that $\hat{U}_\theta(1, 1)$, currently 0.8, is too large and must be reduced. How should the parameters be adjusted to achieve this? As with neuralnetwork learning, we write an error function and compute its gradient with respect to the parameters. If $u_j(s)$ is the observed total reward from state s onward in the jth trial, then the error is defined as (half) the squared difference of the predicted total and the actual total: $E_j(s) = (\widehat{U}_\theta(s) - u_j(s))^2/2$. The rate of change of the error with respect to each parameter θ_i is $\partial E_j / \partial \theta_i$, so to move the parameter in the direction of decreasing the error, we want

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial E_j(s)}{\partial \theta_i} = \theta_i + \alpha[u_j(s) - \widehat{U}_\theta(s)] \frac{\partial \widehat{U}_\theta(s)}{\partial \theta_i}. \tag{23.10}$$

This is called the **Widrow-Hoff rule**, or the **delta rule**, for online least-squares. For the linear function approximator $\hat{U}_\theta(s)$ in Equation (23.9), we get three simple update rules:

$$\theta_0 \leftarrow \theta_0 + \alpha[u_j(s) - \widehat{U}_\theta(s)],$$
$$\theta_1 \leftarrow \theta_1 + \alpha[u_j(s) - \widehat{U}_\theta(s)]x,$$
$$\theta_2 \leftarrow \theta_2 + \alpha[u_j(s) - \widehat{U}_\theta(s)]y,$$

We can apply these rules to the example where $\hat{U}_\theta$ (1, 1) is 0.8 and $u_j(1, 1)$ is 0.4. Parameters θ_0, θ_1, and θ_2 are all decreased by 0.4α, which reduces the error for (1,1). Notice that changing the parameters θ_i in response to an observed transition between two states also changes the values of $\hat{U}_\theta$ for every other state! This is what we mean by saying that function approximation allows a reinforcement learner to generalize from its experiences.

The agent will learn faster if it uses a function approximator, provided that the hypothesis space is not too large and includes some functions that are a reasonably good fit to the true utility function. Exercise 23.APLM asks you to evaluate the performance of direct utility estimation, both with and without function approximation. The improvement in the 4 × 3 world is noticeable but not dramatic, because this is a very small state space to begin with. The improvement is much greater in a 10 × 10 world with a +1 reward at (10,10).

The 10 × 10 world is well suited for a linear utility function because the true utility function is smooth and nearly linear: it is basically a diagonal slope with its lower corner at (1,1) and its upper corner at (10,10). (See Exercise 23.TENX.) On the other hand, if we put the +1 reward at (5,5), the true utility is more like a pyramid and the function approximator in Equation (23.9) will fail miserably.

All is not lost, however! Remember that what matters for linear function approximation is that the function be linear in the features. But we can choose the features to be arbitrary nonlinear functions of the state variables. Hence, we can include a feature such as $f_3(x,y) = \sqrt{(x - x_g)^2 + (y - y_g)^2}$ that measures the distance to the goal. With this new feature, the linear function approximator does well.

23.4.2 Approximating temporal-difference learning

We can apply these ideas equally well to temporal-difference learners. All we need do is adjust the parameters to try to reduce the temporal difference between successive states. The new versions of the TD and Q-learning equations (23.3 on page 846 and 23.7 on page 853) are given by

$$\theta_i \leftarrow \theta_i + \alpha[R(s,a,s\prime) + \gamma \widehat{U}_\theta(s\prime) - \widehat{U}_\theta(s)] \frac{\partial \widehat{U}_\theta(s)}{\partial \theta_i} \tag{23.11}$$

for utilities and

$$\theta_i \leftarrow \theta_i + \alpha[R(s,a,s') + \gamma \max_{a\prime} \widehat{Q}_\theta(s',a') - \widehat{Q}_\theta(s,a)] \frac{\partial \widehat{U}_\theta(s,a)}{\partial \theta_i} \tag{23.12}$$

for Q-values. For passive TD learning, the update rule can be shown to converge to the closest possible approximation to the true function when the function approximator is *linear* in the features.[4] With active learning and *nonlinear* functions such as neural networks, nearly all bets are off: there are some very simple cases in which the parameters can go off to infinity with these update rules, even though there are good solutions in the hypothesis space. There are more sophisticated algorithms that can avoid these problems, but at present reinforcement learning with general function approximators remains a delicate art.

In addition to parameters diverging to infinity, there is a more surprising problem called **catastrophic forgetting**. Suppose you are training an autonomous vehicle to drive along (simulated) roads safely without crashing. You assign a high negative reward for crossing the edge of the road, and you use quadratic features of the road position so that the car can learn that the utility of being in the middle of the road is higher than being close to the edge. All goes well, and the car learns to drive perfectly down the middle of the road. After a few minutes of this, you are starting to get bored and are about to halt the simulation and write up the excellent results. All of a sudden, the vehicle swerves off the road and crashes. Why? What has happened is that the car has learned *too well:* because it has learned to steer away from the edge, it has learned that the entire central region of the road is a safe place to be, and it has forgotten that the region closer to the edge is dangerous. The central region therefore has a flat value function, so the quadratic features get zero weight; then, any nonzero weight on the linear features causes the car to slide off the road to one side or the other.

One solution to this problem, called **experience replay**, ensures that the car keeps reliving its youthful crashing behavior at regular intervals. The learning algorithm can retain trajectories from the entire learning process and replay those trajectories to ensure that its value function is still accurate for parts of the state space it no longer visits.

For model-based reinforcement learning systems, function approximation can also be very helpful for learning a model of the environment. Remember that learning a model for an *observable* environment is a *supervised* learning problem, because the next percept gives the outcome state. Any of the supervised learning methods in Chapters 19, 21, and 22 can be used, with suitable adjustments for the fact that we need to predict a complete state description rather than just a Boolean classification or a single real value. With a learned model, the agent can do a look-ahead search to improve its decisions and can carry out internal simulations to improve its approximate representations of U or Q rather than requiring slow and potentially expensive real-world experiences.

For a *partially observable* environment, the learning problem is much more difficult because the next percept is no longer a label for the state prediction problem. If we know what the hidden variables are and how they are causally related to each other and to the observable variables, then we can fix the structure of a dynamic Bayesian network and use the EM algorithm to learn the parameters, as was described in Chapter 21. Learning the internal structure of dynamic Bayesian networks and creating new state variables is still considered a difficult problem. Deep recurrent neural networks (Section 22.6) have in some cases been successful at inventing the hidden structure.

23.4.3 Deep reinforcement learning

There are two reasons why we need to go beyond linear function approximators: first, there may be no good linear function that comes close to approximating the utility function or the Q-function; second, we may not be able to invent the necessary features, particularly in new domains. If you think about it, these are really the same reason: it is always *possible* to represent U or Q as linear combinations of features, especially if we have features such as $f_1(s) = U(s)$ or $f_2(s,a) = Q(s,a)$, but unless we can come up with such features (in an efficiently computable form) the linear function approximator may be insufficient.

For these reasons (or reason), researchers have explored more complex, nonlinear function approximators since the earliest days of reinforcement learning. Currently, deep neural networks (Chapter 22) are very popular in this role and have proved to be effective even when the input is a raw image with no human-designed feature extraction at all. If all goes well, the deep neural network in effect discovers the useful features for itself. And if the final layer of the network is linear, then we can see what features the network is using to build its own linear function approximator. A reinforcement learning system that uses a deep network as a function approximator is called a deep reinforcement learning system.

Just as in Equation (23.9), the deep network is a function parameterized by θ, except that now the function is much more complicated. The parameters are all the weights in all the layers of the network. Nonetheless, the gradients required for Equations (23.11) and (23.12) are just the same as the gradients required for supervised learning, and they can be computed by the same back-propagation process described in Section 22.4.

As we explain in Section 23.7, deep RL has achieved very significant results, including learning to play a wide range of video games at an expert level, defeating the human world champion at Go, and training robots to perform complex tasks.

Despite its impressive successes, deep RL still faces significant obstacles: it is often difficult to get good performance and the trained system may behave very unpredictably if the environment differs even a little from the training data. Compared to other applications of deep learning, deep RL is rarely applied in commercial settings. It is, nonetheless, a very active area of research.

23.4.4 Reward shaping

As noted in the introduction to this chapter, real-world environments may have very sparse rewards: many primitive actions are required to achieve any nonzero reward. For example, a soccer-playing robot might send a hundred thousand motor control commands to its various joints before conceding a goal. Now it has to work out what it did wrong. The technical term for this is the **credit assignment** problem. Other than playing trillions of soccer games so that the negative reward eventually propagates back to the actions responsible for it, is there a good solution?

One common method, originally used in animal training, is called **reward shaping**. This involves supplying the agent with additional rewards, called **pseudorewards**, for "making progress." For example, we might give pseudorewards to the robot for making contact with the ball or for advancing it toward the goal. Such rewards can speed up learning enormously and are simple to provide, but there is a risk that the agent will learn to maximize the pseudorewards rather than the true rewards; for example, standing next to the ball and "vibrating" causes many contacts with the ball.

In Chapter 16 (page 559), we saw a way to modify the reward function without changing the optimal policy. For any potential function $\Phi(s)$ and any reward function R, we can create a new reward function R' as follows:

$$R'(s,a,s') = R(s,a,s') + \gamma\Phi(s') - \Phi(s).$$

The potential fuction Φ can be constructed to reflect any desirable aspects of the state, such as achievement of subgoals or distance to a desired terminal state. For example, Φ for the soccerplaying robot could add a constant bonus for states where the robot's team has possession and another bonus for reducing the distance of the ball from the opponents' goal. This will result in faster learning overall, but will not prevent the robot from, say, learning to pass back to the goalkeeper when danger threatens.

23.4.5 Hierarchical reinforcement learning

Another way to cope with very long action sequences is to break them up into a few smaller pieces, and then break those into smaller pieces still, and so on until the action sequences are short enough to make learning easy. This approach is called **hierarchical reinforcement learning** (HRL), and it has much in common with the **HTN planning** methods described in Chapter 11. For example, scoring a goal in soccer can be broken down into obtaining possession, passing to a teammate, receiving the ball from a team-mate, dribbling toward the goal, and shooting; each of these can be broken down further into lower-level motor behaviors. Obviously, there are multiple ways of obtaining possession and shooting, multiple teammates one could pass to, and so on, so each higher-level action may have many different lower-level implementations.

To illustrate these ideas, we'll use a simplified soccer game called **keepaway**, in which one team of three players tries to keep possession of the ball for as long as possible by dribbling and passing amongst themselves while the other team of two players tries to take possession by intercepting a pass or tackling a player in possession.[5] The game is implemented within the RoboCup 2D simulator, which provides detailed continuous-state motion models with 100ms time steps and has proved to be a good testbed for RL systems.

A hierarchical reinforcement learning agent begins with a **partial program** that outlines a hierarchical structure for the agent's behavior. The partial-programming language for agent programs extends any ordinary programming language by adding primitives for unspecified choices that must be filled in by learning. (Here, we use pseudocode for the programming language.) The partial program can be arbitrarily complicated, as long as it terminates.

It is easy to see that HRL includes ordinary RL as a special case. We simply provide the trivial partial program that allows the agent to keep choosing any action from $A(s)$, the set of actions that can be executed in the current state s:

$$\begin{array}{l} \textbf{while } \mathit{true} \textbf{ do} \\ \qquad \textbf{choose}\,(A(s)). \end{array}$$

The **choose** operator allows the agent to choose any element of the specified set. The learning process converts the partial agent program into a complete program by learning how each choice should be made. For example, the learning process might associate a Q-function with each choice; once the Q-functions are learned, the program produces behavior by choosing the option with the highest Q-value each time it encounters a choice.

The agent programs for keepaway are more interesting. We'll look at the partial program for a single player on the "keeper" team. The choice of what to do at the top level depends mainly on whether the player has the ball or not:

$$\begin{array}{l} \textbf{while not } \text{IS-TERMINAL}(s) \textbf{ do} \\ \quad \textbf{if } \text{BALL-IN-MY-POSSESSION}(s) \textbf{ then choose}\,(\{\text{PASS}, \text{HOLD}, \text{DRIBBLE}\}) \\ \quad \textbf{else choose}\,(\{\text{STAY}, \text{MOVE}, \text{INTERCEPT-BALL}\}). \end{array}$$

Each of these choices invokes a subroutine that may itself make further choices, all the way down to primitive actions that can be executed directly. For example, the high-level action PAss chooses a teammate to pass to, but also has the choice to do nothing and return control to the higher level if appropriate (e.g., if there is no one to pass to):

$$\textbf{choose}\,(\{\text{PASS-TO}(\textbf{choose}(\text{TEAMMATES}(s))), \textbf{return}\}).$$

The PASS-TO routine then has to choose a speed and direction for the pass. While it is relatively easy for a human —even one with little expertise in soccer—to provide this kind of high-level advice to the learning agent, it would be difficult, if not impossible, to write down the rules for determining the speed and direction of the kick to maximize the probability of maintaining possession. Similarly, it is far from obvious how to choose the right teammate to receive the ball or where to move in order to make oneself available to receive the ball. The partial program provides general know-how—overall scaffolding and structural organization for complex behaviors—and the learning process works out all the details.

The theoretical foundations of HRL are based on the concept of the joint state space, in which each state (s, m) is composed of a physical state s and a machine state m. The machine state is defined by the current internal state of the agent program: the program counter for each subroutine on the current call stack, the values of the arguments, and the values of all local and global variables. For example, if the agent program has chosen to pass to teammate Ali and is in the middle of calculating the speed of the pass, then the fact that Ali is the argument of PASS-TO is part of the current machine state. A **choice state** $\sigma = (s, m)$ is one in which the program counter for m is at a choice point in the agent program. Between two choice states, any number of computational transitions and physical actions may occur, but they are all preordained, so to speak: by definition, the agent isn't making any choices in between choice states. Essentially, the hierarchical **RL** agent is solving a Markovian decision problem with the following elements:

- The states are the choice states σ of the joint state space.
- The actions at σ are the choices c available in σ according to the partial program.
- The reward function $\rho(\sigma,c,\sigma')$ is the expected sum of rewards for all physical transitions occurring between the choice states σ and σ'.
- The transition model $\tau(\sigma, c, \sigma')$ is defined in the obvious way: if c invokes a physical action a, then τ borrows from the physical model $P(s' \mid s, a)$; if c invokes a computational transition, such as calling a subroutine, then the transition deterministically modifies the computational state m according to the rules of the programming language.[6]

By solving this decision problem, the agent finds the optimal policy that is consistent with original partial program.

Hierarchical RL can be a very effective method for learning complex behaviors. In keepaway, an **HRL** agent based on the partial program sketched above learns a policy that keeps possession forever against the standard taker policy—a significant improvement on the previous record of about 10 seconds. One important characteristic is that the lower-level skills are not fixed subroutines in the usual sense; their choices are sensitive to the entire internal state of the agent program, so they behave differently depending on where they are invoked within that program and what is going on at the time. If necessary, the Q-functions for the lower-level choices can be initialized by a separate training process with its own reward function, and then integrated into the overall system so they can be adapted to function well in the context of the whole agent.

In the preceding section we saw that shaping rewards can be helpful for learning complex behaviors. In HRL, the fact that learning takes place in the joint state space provides additional opportunities for shaping. For example, to help with learning the Q-function for accurate passing within the PASS-TO routine, we can provide a shaping reward that depends on the location of the intended recipient and the proximity of opponents to that player: the ball should be close to the recipient and far from the opponents. That seems entirely obvious; but the identity of the intended recipient for a pass is not part of the physical state of the*world.* The physical state consists only of the positions, orientations, and velocities of the players and the ball. There is no "passing" and no "recipient" in the physical world; these are entirely internal constructs. This means that there is no way to provide such sensible advice to a standard RL system.

The hierarchical structure of behavior also provides a natural **additive decomposition** of the overall utility function. Remember that utility is the sum of rewards over time, and consider a sequence of, say, ten time steps with rewards $[r_1, r_2,..., r_{10}]$. Suppose that for the first five time steps the agent is doing PASS-To(*Ali*) and for the remaining five steps it is doing MOVE-INTO-SPACE. Then the utility for the initial state is the sum of the total reward during PASS-TO and the total reward during MOVE-INTO-SPACE. The former depends only on whether the ball gets to Ali with enough time and space for Ali to retain possession, and the latter depends only on whether the agent reaches a good location to receive the ball. In other words, the overall utility decomposes into several terms, each of which depends on only a few variables. This, in turns, means that learning occurs much more quickly than if we try to learn a single utility function that depends on all the variables. This is somewhat analogous to the representation theorems underlying the conciseness of Bayes nets (Chapter 13).

23.5 Policy Search

The final approach we will consider for reinforcement learning problems is called **policy search**. In some ways, policy search is the simplest of all the methods in this chapter: the idea is to keep twiddling the policy as long as its performance improves, then stop.

Let us begin with the policies themselves. Remember that a policy π is a function that maps states to actions. We are interested primarily in *parameterized* representations of π that have far fewer parameters than there are states in the state space (just as in the preceding section). For example, we could represent π by a collection of parameterized Q-functions, one for each action, and take the action with the highest predicted value:

$$\pi(s) = \underset{a}{\operatorname{argmax}}\, \widehat{Q}_\theta(s,a). \tag{23.13}$$

Each Q-function could be a linear function, as in Equation (23.9), or it could be a nonlinear function such as a deep neural network. Policy search will then adjust the parameters θ to improve the policy. Notice that if the policy is represented by Q-functions, then policy search results in a process that learns Q-functions. *This process is not the same as Q-learning!*

In Q-learning with function approximation, the algorithm finds a value of θ such that $\widehat{Q}_\theta$ is "close"to Q* the optimal Q-function. Policy search, on the other hand, finds a value of θ that results in good performance; the values found by the two methods may differ very substantially. (For example, the approximate Q-function defined by $\widehat{Q}_\theta(s,a) = Q^*(s,a)/100$ gives optimal performance, even though it is not at all close to Q*.) Another clear instance of the difference is the case where $\pi(s)$ is calculated using, say, depth-10 look-ahead search with an approximate utility function $\hat{U}_\theta$. A value of θ that gives good results may be a long way from making $\hat{U}_\theta$ resemble the true utility function.

One problem with policy representations of the kind given in Equation (23.13) is that the policy is a *discontinuous* function of the parameters when the actions are discrete. That is, there will be values of θ such that an infinitesimal change in θ causes the policy to switch from one action to another. This means that the value of the policy may also change discontinuously, which makes gradient-based search difficult. For this reason, policy search methods often use a **stochastic policy** representation $\pi_\theta(s, a)$, which specifies the *probability* of selecting action a in state s. One popular representation is the **softmax** function:

$$\pi_\theta(s,a) = \frac{e^{\beta \widehat{Q}_\theta(s,a)}}{\sum_{a'} e^{\beta \widehat{Q}_\theta(s,a')}}. \qquad (23.14)$$

The parameter $\beta > 0$ modulates the softness of the softmax: for values of β that are large compared to the separations between Q-values, the softmax approaches a hard max, whereas for values of β close to zero the softmax approaches a uniform random choice among the actions. For all finite values of β, the softmax provides a differentiable function of θ; hence, the value of the policy (which depends continuously on the action-selection probabilities) is a differentiable function of θ.

Now let us look at methods for improving the policy. We start with the simplest case: a deterministic policy and a deterministic environment. Let $\rho(\theta)$ be the **policy value**, that is, the expected reward-to-go when π_θ is executed. If we can derive an expression for $\rho(\theta)$ in closed form, then we have a standard optimization problem, as described in Chapter 4. We can follow the **policy gradient** vector $\nabla_\theta \rho(\theta)$ provided $\rho(\theta)$ is differentiable. Alternatively, if $\rho(\theta)$ is not available in closed form, we can evaluate π_θ

simply by executing it and observing the accumulated reward. We can follow the **empirical gradient** by hill climbing—that is, evaluating the change in policy value for small increments in each parameter. With the usual caveats, this process will converge to a local optimum in policy space.

When the environment (or the policy) is nondeterministic, things get more difficult. Suppose we are trying to do hill climbing, which requires comparing $\rho(\theta)$ and $\rho(\theta + \Delta\theta)$ for some small $\Delta\theta$. The problem is that the total reward for each trial may vary widely, so estimates of the policy value from a small number of trials will be quite unreliable; trying to compare two such estimates will be even more unreliable. One solution is simply to run lots of trials, measuring the sample variance and using it to determine that enough trials have been run to get a reliable indication of the direction of improvement for $\rho(\theta)$. Unfortunately, this is impractical for many real problems in which trials may be expensive, time-consuming, and perhaps even dangerous.

For the case of a nondeterministic policy $\pi_\theta(s, a)$, it is possible to obtain an unbiased estimate of the gradient at θ, $\nabla_\theta\rho(\theta)$, directly from the results of trials executed at 0. For simplicity, we will derive this estimate for the simple case of an episodic environment in which each action a obtains reward $R(s_0, a, s_0)$ and the environment restarts in s_0. In this case, the policy value is just the expected value of the reward, and we have

$$\nabla_\theta\rho(\theta) = \nabla_\theta \sum_a R(s_0, a, s_0)\pi_\theta(s_0, a) = \sum_a R(s_0, a, s_0)\nabla_\theta\pi_\theta(s_0, a).$$

Now we perform a simple trick so that this summation can be approximated by samples generated from the probability distribution defined by $\pi_\theta(s_0, a)$ Suppose that we have N trials in all, and the action taken on the jth trial is a_j. Then

$$\nabla_\theta \rho(\theta) = \sum_a \pi_\theta(s_0, a). \frac{R(s_0, a, s_0)\nabla_\theta \pi_\theta(s_0, a)}{\pi_\theta(s_0, a)}$$

$$= \approx \frac{1}{N}\sum_{j=1}^{N} \frac{R(s_0, a_j, s_0)\nabla_\theta \pi_\theta(s_0, a_j)}{\pi_\theta(s_0, a_j)}.$$

Thus, the true gradient of the policy value is approximated by a sum of terms involving the gradient of the action-selection probability in each trial. For the sequential case, this generalizes to

$$\nabla_\theta \rho(\theta) \approx \frac{1}{N}\sum_{j=1}^{N} \frac{u_j(s)\nabla_\theta \pi_\theta(s, a_j)}{\pi_\theta(s, a_j)}$$

for each state *s* visited, where a_j is executed in *s* on the *j*th trial and $u_j(s)$ is the total reward received from state *s* onward in the *j*th trial. The resulting algorithm, called REINFORCE, is due to Ron Williams (1992); it is usually much more effective than hill climbing using lots of trials at each value of θ. However, it is still much slower than necessary.

Consider the following task: given two blackjack policies, determine which is best. The policies might have true net returns per hand of, say, –0.21% and +0.06%, so finding out which is better is very important. One way to do this is to have each policy play against a standard "dealer" for a certain number of hands and then to measure their respective winnings. The problem with this, as we have seen, is that the winnings of each policy fluctuate wildly depending on whether it receives good or bad cards. One would need several million hands to have a reliable indication of which policy is better. The same issue arises when using random sampling to compare two adjacent policies in a hill-climbing algorithm.

A better solution for blackjack is to generate a certain number of hands in advance and *have each program play the same set of hands*. In this way, we eliminate the measurement error due to differences in the cards received. Only a few thousand hands are needed to determine which of the two blackjack policies is better.

This idea, called **correlated sampling**, can be applied to policy search in general, given an environment simulator in which the random-number sequences can be repeated. It was implemented in a policy-search algorithm called PEGASUS (Ng and Jordan, 2000), which was one of the first algorithms to achieve completely stable autonomous helicopter flight (see Figure 23.9(b)). It can be shown that the number of random sequences required to ensure that the value of *every* policy is well estimated depends only on the complexity of the policy space, and not at all on the complexity of the underlying domain.

Figure 23.9 (a) Setup for the problem of balancing a long pole on top of a moving cart. The cart can be jerked left or right by a controller that observes the cart's position x and velocity $\dot{x}$, as well as the pole's angle θ and rate of change of angle $\dot{\theta}$. (b) Six

superimposed time-lapse images of a single autonomous helicopter performing a very difficult “nose-in circle” maneuver. The helicopter is under the control of a policy developed by the PEGASUS policy-search algorithm (Ng *et al.*, 2003). A simulator model was developed by observing the effects of various control manipulations on the real helicopter; then the algorithm was run on the simulator model overnight. A variety of controllers were developed for different maneuvers. In all cases, performance far exceeded that of an expert human pilot using remote control. (Image courtesy of Andrew Ng.)

23.6 Apprenticeship and Inverse Reinforcement Learning

Some domains are so complex that it is difficult to define a reward function for use in reinforcement learning. Exactly what do we want our self-driving car to do? Certainly it should not take too long to get to the destination, but it should not drive so fast as to incur undue risk or to get speeding tickets. It should conserve fuel/energy. It should avoid jostling or accelerating the passengers too much, but it can slam on the brakes in an emergency. And so on. Deciding how much weight to give to each of these factors is a difficult task. Worse still, there are almost certainly important factors we have forgotten, such as the obligation to behave with consideration for other drivers. Omitting a factor usually leads to behavior that assigns an extreme value to the omitted factor—in this case, extremely inconsiderate driving—in order to maximize the remaining factors.

One approach is to do extensive testing in simulation, notice problematic behaviors, and try to modify the reward function to eliminate those behaviors. Another approach is to seek additional sources of information about the appropriate reward function. One such source is the behavior of agents who are already optimizing (or, let's say, nearly optimizing) that reward function—in this case, expert human drivers.

The general field of **apprenticeship learning** studies the process of learning how to behave well given observations of expert behavior. We show the algorithm examples of expert driving and tell it to "do it like that." There are (at least) two ways to approach the apprenticeship learning problem. The first is the one we discussed briefly at the beginning of the chapter: assuming the environment is observable, we apply supervised learning to the observed state-action pairs to learn a policy $\pi(s)$. This is called **imitation learning**. It has had some success in robotics (see page 973) but suffers from the the problem of brittleness: even small deviations from the training set lead to errors that grow over time and eventually to failure. Moreover, imitation learning will at best duplicate the teacher's performance, not exceed it. When humans learn by imitation, we sometimes use the pejorative term "aping" to describe what they are doing. (It's quite possible that apes use the term "humaning" amongst themselves, perhaps in an even more pejorative sense.) The implication is that the imitation learner doesn't understand why it should perform any given action.

The second approach to apprenticeship learning is to understand *why*: to observe the expert's actions (and resulting states) and try to work out what reward function the expert is maximizing. Then we could derive an optimal policy with respect to that reward function. One expects that this approach will produce robust policies from relatively few examples of expert behavior; after all, the field of reinforcement learning is predicated on the idea that the reward function, rather than the policy or the value function, is the most succinct, robust, and transferable definition of the task. Furthermore, if the learner makes appropriate allowances for possible suboptimality on the part of the expert, then it may be able to do better than the expert by optimizing an accurate approximation to the true reward function. We call this approach **inverse reinforcement learning** (IRL): learning rewards by observing a policy, rather than learning a policy by observing rewards.

How do we find the expert's reward function, given the expert's actions? Let us begin by assuming that the expert was acting rationally. In that case, it seems we should be looking for a reward function R^* such that the total expected discounted reward under the expert's policy is higher than (or at least the same as) under any other possible policy.

Unfortunately, there will be many reward functions that satisfy this constraint; one of them is $R^*(s, a, s') = 0$, because any policy is rational when there are no rewards at all.[7] Another problem with this approach is that the assumption of a rational expert is unrealistic. It means, for example, that a robot observing Lee Sedol making what eventually turns out to be a losing move against ALPHAGO would have to assume that Lee Sedol was trying to lose the game.

To avoid the problem that $R^*(s, a, s') = 0$ explains any observed behavior, it helps to think in a Bayesian way. (See Section 21.1 for a reminder of what this means.) Suppose we observe data **d** and let h_R be the hypothesis that R is the true reward function. Then according to Bayes' rule, we have

$$P(h_R|\mathbf{d}) = \alpha P(\mathbf{d}|h_R)P(h_R).$$

Now, if the prior $P(h_R)$ is based on simplicity, then the hypothesis that $R = 0$ scores fairly well, because 0 is certainly simple. On the other hand, the term $P(\mathbf{d} \mid h_R)$ is *infinitesimal* for the hypothesis that $R = 0$, because it doesn't explain why the expert chose that particular behavior out of the vast space of behaviors that would be optimal if the hypothesis were true. On the other hand, for a reward function R that has a unique optimal policy or a relatively small equivalence class of optimal policies, $P(\mathbf{d} \mid h_R)$ will be far higher.

To allow for the occasional mistake by the expert, we simply allow $P(\mathbf{d} \mid h_R)$ to be nonzero even when **d** comes from behavior that is a little bit suboptimal according to R. A typical assumption—made, it must be said, more for mathematical convenience than faithfulness to actual human data

—is that an agent whose true Q-function is $Q(s, a)$ chooses not according to the deterministic policy $\pi(s) = \arg\max_a Q(s, a)$ but instead according to a stochastic policy defined by the softmax distribution from Equation (23.14). This is sometimes called **Boltzmann rationality** because, in statistical mechanics, the state occupation probabilities in a Boltzmann distribution depend exponentially on their energy levels.

There are dozens of inverse RL algorithms in the literature. One of the simplest is called **feature matching**. It assumes that the reward function can be written as a weighted linear combination of features:

$$R_\theta(s, a, s') = \sum_{i=1}^{n} \theta_i f_i(s, a, s') = \theta \cdot \mathbf{f}.$$

For example, the features in the driving domain might include speed, speed in excess of the speed limit, acceleration, proximity to nearest obstacle, etc.

Recall from Equation (16.2) on page 557 that the utility of executing a policy π, starting in state s_0, is defined to be

$$U^\pi(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t, \pi(S_t), S_{t+1})\right],$$

where the expectation E is with respect to the probability distribution over state sequences determined by s and π. Because R is assumed to be a linear combination of feature values, we can rewrite this as follows:

$$\begin{aligned}
U^\pi(s) &= E\left[\sum_{t=0}^{\infty} \gamma^t \sum_{i=1}^{n} \theta_i f_i(S_t, \pi(S_t), S_{t+1})\right] \\
&= \sum_{i=1}^{n} \theta_i E\left[\sum_{t=0}^{\infty} \gamma^t f_i(S_t, \pi(S_t), S_{t+1})\right] \\
&= \sum_{i=1}^{n} \theta_i \mu_i(\pi) = \theta \cdot \mu(\pi)
\end{aligned}$$

where we have defined the **feature expectation** $\mu_i(\pi)$ as the expected discounted value of the feature f_i when policy π is executed. For example, if f_i is the excess speed of the vehicle (above the speed limit), then $\mu_i(\pi)$ is the (time-discounted) average excess speed over the entire trajectory. The key point about feature expectations is the following: *if a policy π produces feature expectations* $\mu_i(\pi)$ *that match those of the expert's policy* π_E, *, then* π *is as good as the expert's policy according to the expert's own reward function.* Now, we cannot measure the exact values for the feature expectations of the expert's policy, but we can approximate them using the average values on the observed trajectories. Thus, we need to find values for the parameters θ_i such that the feature expectations of the policy induced by the parameter values match those of the expert policy on the observed trajectories. The following algorithm achieves this with any desired error bound.

- Pick an initial default policy $\pi^{(0)}$.
- For $j = 1,2,...$ until convergence:
  - Find parameters $\theta^{(j)}$ such that expert's policy maximally outperformes the policies $\pi^{(0)},....\pi^{(j-1)}$ according to the expected utility $\theta^{(j)}\cdot\mu(\pi)$
  - Let $\pi^{(j)}$ be the optimal policy for the reward function $R^{(j)} = \theta^{(j)}\cdot\mathbf{f}$.

This algorithm converges to a policy that is close in value to the expert's, according to the expert's own reward function. It requires only $O(n \log n)$ iterations and $O(n \log n)$ expert demonstrations, where n is the number of features.

A robot can use inverse reinforcement learning to learn a good policy for itself, by understanding the actions of an expert. In addition, the robot can learn the policies used by other agents in a multiagent domain, whether they be adversarial or cooperative. And finally, inverse reinforcement learning can be used for scientific inquiry (without any thought of agent design), to better understand the behavior of humans and other animals.

A key assumption in inverse RL is that the "expert" is behaving optimally, or nearly optimally, with respect to some reward function in a single-agent MDP. This is a reasonable assumption if the learner is watching the expert through a one-way mirror while the expert goes about his or her business unawares. It is not a reasonable assumption if the expert is aware of the learner. For example, suppose a robot is in medical school, learning to be a surgeon by watching a human expert. An inverse RL algorithm would assume that the human performs the surgery in the usual optimal way, as if the robot were not there. But that's not what would happen: the human surgeon is motivated to have the robot (like any other medical student) learn quickly and well, and so she will modify her behavior considerably. She might explain what she is doing as she goes along; she might point out mistakes to avoid, such as making the incision too deep or the stitches too tight; she might describe the contingency plans in case something goes wrong during surgery. None of these behaviors make sense when performing surgery in isolation, so inverse RL algorithms will not be able to interpret the underlying reward function. Instead, we need to understand this kind of situation as a two-person assistance game, as described in Section 17.2.5.

23.7 Applications of Reinforcement Learning

We now turn to applications of reinforcement learning. These include game playing, where the transition model is known and the goal is to learn the utility function, and robotics, where the model is initially unknown.

23.7.1 Applications in game playing

In Chapter 1 we described Arthur Samuel's early work on reinforcement learning for checkers, which began in 1952. A few decades passed before the challenge was taken up again, this time by Gerry Tesauro in his work on backgammon. Tesauro's first attempt (1990) was a system called NEUROGAMMON. The approach was an interesting variant on imitation learning. The input was a set of 400 games played by Tesauro against himself. Rather than learn a policy, NEUROGAMMON converted each move (s, a, s') into a set of training examples, each of which labeled s' as a better position than some other position s'' reachable from s by a different move. The network had two separate halves, one for s' and one for s'', and was constrained to choose which was better by comparing the outputs of the two halves. In this way, each half was forced to learn an evaluation function $\hat{U}_\theta$.. NEUROGAMMON won the 1989 Computer Olympiad—the first learning program ever to win a computer game tournament—but never progressed past Tesauro's own intermediate level of play.

Tesauro's next system, TD-GAMMON(1992), adopted Sutton's recently published TD learning method—essentially returning to the approach explored by Samuel, but with much greater technical understanding of how to do it right. The evaluation function $\hat{U}_\theta$ was represented by a fully connected neural network with a single hidden layer containing 80 nodes.

(It also used some manually designed input features borrowed from NEUROGAMMON.) After 300,000 training games, it reached a standard of play comparable to the top three human players in the world. Kit Woolsey, a top-ten player, said, “There is no question in my mind that its positional judgment is far better than mine.”

The next challenge was to learn from raw perceptual inputs—something closer to the real world—rather than discrete game board representations. Beginning in 2012, a team at DeepMind developed the **deep Q-network (DQN)** system, the first modern deep RL system. DQN uses a deep neural network to represent the Q-function; otherwise it is a typical reinforcement learning system. DQN was trained separately on each of 49 different Atari video games. It learned to drive simulated race cars, shoot alien spaceships, and bounce balls with paddles. In each case, the agent learned a Q-function from raw image data with the reward signal being the game score. overall, the system performed at roughly human expert level, although a few games gave it trouble. One game in particular, *Montezuma’s Revenge,* proved far too difficult, because it required extended planning strategies, and the rewards were too sparse. Subsequent work produced deep RL systems that generated more extensive exploratory behaviors and were able to conquer *Montezuma’s Revenge* and other difficult games.

DeepMind’s ALPHAGO system also used deep reinforcement learning to beat the best human players at the game of Go (see Chapter 6). Whereas a Q-function with no look-ahead suffices for Atari games, which are primarily reactive in nature, Go requires substantial lookahead. For this reason, ALPHAGO learned both a value function and a Q-function that guided its search by predicting which moves are worth exploring. The Q-function, implemented as a convolutional neural network, is accurate enough by itself to beat most amateur human players without any search at all.

23.7.2 Application to robot control

The setup for the famous **cart–pole** balancing problem, also known as the **inverted pendulum**, is shown in Figure 23.9(a). The problem is to keep the pole roughly upright ($\theta \approx 90°$) by applying forces to move the cart right or left, while keeping the position x within the limits of the track. Several thousand papers in reinforcement learning and control theory have been published on this seemingly simple problem. One difficulty is that the state variables x, θ, $\dot{x}$, and θ are continuous. The actions, however, are defined to be discrete: jerk left or jerk right, the so-called **bang-bang control** regime.

The earliest work on learning for this problem was carried out by Michie and Chambers (1968), using a real cart and pole, not a simulation. Their BOXES algorithm was able to balance the pole for over an hour after 30 trials. The algorithm first discretized the fourdimensional state space into boxes—hence the name. It then ran trials until the pole fell over. Negative reinforcement was associated with the final action in the final box and then propagated back through the sequence. Improved generalization and faster learning can be obtained using an algorithm that *adaptively* partitions the state space according to the observed variation in the reward, or by using a continuous-state, nonlinear function approximator such as a neural network. Nowadays, balancing a *triple* inverted pendulum (three poles joined together end to end) is a common exercise—a feat far beyond the capabilities of most humans, but achievable using reinforcement learning.

Still more impressive is the application of reinforcement learning to radio-controlled helicopter flight (Figure 23.9(b)). This work has generally used policy search over large MDPs (Bagnell and Schneider, 2001; Ng *et* *al.*, 2003), often combined with imitation learning and inverse RL given observations of a human expert pilot (Coates *et al.*, 2009).

Inverse RL has also been applied successfully to interpret human behavior, including destination prediction and route selection by taxi drivers based on 100,000 miles of GPS data (Ziebart *et al.*, 2008) and detailed physical movements by pedestrians in complex environments based on hours of video observation (Kitani *et al.*, 2012). In the area of robotics, a single expert demonstration was enough for the LittleDog quadruped to learn a 25-feature reward function and nimbly traverse a previously unseen area of rocky terrain (Kolter *et al.*, 2008). For more on how RL and inverse RL are used in robotics, see Sections 26.7 and 26.8.

Summary

This chapter has examined the reinforcement learning problem: how an agent can become proficient in an unknown environment, given only its percepts and occasional rewards. Reinforcement learning is a very broadly applicable paradigm for creating intelligent systems. The major points of the chapter are as follows.

- The overall agent design dictates the kind of information that must be learned:
  - A **model-based reinforcement learning** agent acquires (or is equipped with) a transition model $P(s' \mid s, a)$ for the environment and learns a utility function $U(s)$.
  - A **model-free reinforcement learning** agent may learn an action-utility function $Q(s, a)$ or a policy $\pi(s)$.
- Utilities can be learned using several different approaches:
  - **Direct utility estimation** uses the total observed reward-to-go for a given state as direct evidence for learning its utility.
  - **Adaptive dynamic programming** (ADP) learns a model and a reward function from observations and then uses value or policy iteration to obtain the utilities or an optimal policy. ADP makes optimal use of the local constraints on utilities of states imposed through the neighborhood structure of the environment.
  - **Temporal-difference** (TD) methods adjust utility estimates to be more consistent with those of successor states. They can be viewed as simple approximations of the ADP approach that can learn without requiring a transition model. Using a learned model

to generate pseudoexperiences can, however, result in faster learning.

- Action-utility functions, or Q-functions, can be learned by an ADP approach or a TD approach. With TD, **Q-learning** requires no model in either the learning or action-selection phase. This simplifies the learning problem but potentially restricts the ability to learn in complex environments, because the agent cannot simulate the results of possible courses of action.
- When the learning agent is responsible for selecting actions while it learns, it must trade off the estimated value of those actions against the potential for learning useful new information. An exact solution for the exploration problem is infeasible, but some simple heuristics do a reasonable job. An exploring agent must also take care to avoid premature death.
- In large state spaces, reinforcement learning algorithms must use an approximate functional representation of $U(s)$ or $Q(s, a)$ in order to generalize over states. **Deep reinforcement learning**—using deep neural networks as function approximators—has achieved considerable success on hard problems.
- **Reward shaping** and **hierarchical reinforcement learning** are helpful for learning complex behaviors, particularly when rewards are sparse and long action sequences are required to obtain them.
- **Policy-search** methods operate directly on a representation of the policy, attempting to improve it based on observed performance. The variation in the performance in a stochastic domain is a serious problem; for simulated domains this can be overcome by fixing the randomness in advance.

- **Apprenticeship learning** through observation of expert behavior can be an effective solution when a correct reward function is hard to specify. **Imitation learning** formulates the problem as supervised learning of a policy from the expert's state–action pairs. **Inverse reinforcement learning** infers reward information from the expert's behavior.

Reinforcement learning continues to be one of the most active areas of machine learning research. It frees us from manual construction of behaviors and from labeling the vast data sets required for supervised learning, or having to hand-code control strategies. Applications in robotics promise to be particularly valuable; these will require methods for handling continuous, high-dimensional, partially observable environments in which successful behaviors may consist of thousands or even millions of primitive actions.

We have presented a variety of approaches to reinforcement learning because there is (at least so far) no single best approach. The question of model-based versus model-free methods is, at its heart, a question about the best way to represent the agent function. This is an issue at the foundations of artificial intelligence. As we stated in Chapter 1, one of the key historical characteristics of much AI research is its (often unstated) adherence to the **knowledge-based** approach. This amounts to an assumption that the best way to represent the agent function is to build a representation of some aspects of the environment in which the agent is situated. Some argue that with access to sufficient data, model-free methods can succeed in any domain. Perhaps this is true in theory, but of course, the universe may not contain enough data to make it true in practice. (For example, it is not easy to imagine how a model-free approach would enable one to design and build, say, the LIGO gravity-wave detector.) Our intuition, for what it's worth, is that as the environment becomes more complex, the advantages of a model-based approach become more apparent.

Bibliographical and Historical Notes

It seems likely that the key idea of reinforcement learning—that animals do more of what they are rewarded for and less of what they are punished for—played a significant role in the domestication of dogs at least 15,000 years ago. The early foundations of our scientific understanding of reinforcement learning include the work of the Russian physiologist Ivan Pavlov, who won the Nobel Prize in 1904, and that of the American psychologist Edward Thorndike—particularly his book *Animal Intelligence* (1911). Hilgard and Bower (1975) provide a good survey.

Alan Turing (1948, 1950) proposed reinforcement learning as an approach for teaching computers; he considered it a partial solution, writing, "The use of punishments and rewards can at best be a part of the teaching process." Arthur Samuel's checkers program (1959, 1967) was the first successful use of machine learning of any kind. Samuel suggested most of the modern ideas in reinforcement learning, including temporal-difference learning and function approximation. He experimented with multilayer representations of value functions, similar to today's deep RL. In the end, he found that a simple linear evaluation function over handcrafted features worked best. This may have been a consequence of working with a computer roughly 100 billion times less powerful than a modern tensor processing unit.

Around the same time, researchers in adaptive control theory (Widrow and Hoff, 1960), building on work by Hebb (1949), were training simple networks using the delta rule. Thus, reinforcement learning combines influences from animal psychology, neuroscience, operations research, and optimal control theory.

The connection between reinforcement learning and Markov decision processes was first made by Werbos (1977). (Work by Ian Witten (1977) described a TD-like process in the language of control theory.) The development of reinforcement learning in AI stems primarily from work at the University of Massachusetts in the early 1980s (Barto *et al.*, 1981). An influential paper by Rich Sutton (1988) provided a mathematical understanding of temporal-difference methods. The combination of temporal-difference learning with the model-based generation of simulated experiences was proposed in Sutton's DYNA architecture (Sutton, 1990). Q-learning was developed in Chris Watkins's Ph.D. thesis (1989), while SARSA appeared in a technical report by Rummery and Niranjan (1994). Prioritized sweeping was introduced independently by Moore and Atkeson (1993) and Peng and Williams (1993).

Function approximation in reinforcement learning goes back to Arthur Samuel's checkers program (1959). The use of neural networks to represent value functions was common in the 1980s and came to the fore in Gerry Tesauro's TD-Gammon program (Tesauro, 1992, 1995). Deep neural networks are currently the most popular choice for function approximators in reinforcement learning. Arulkumaran et *al.* (2017) and Francois-Lavet et *al.* (2018) give overviews of deep RL. The DQN system (Mnih et *al.*, 2015) uses a deep network to learn a Q-function, while ALPHAZERO (Silver et *al.*, 2018) learns both a value function for use with a known model and a Q-function for use in metalevel decisions that guide search. Irpan (2018) warns that deep RL systems can perform poorly if the actual environment is even slightly different from the training environment.

Weighted linear combinations of features and neural networks are factored representations for function approximation. It is also possible to apply reinforcement learning to *structured* representations; this is called

relational reinforcement learning (Tadepalli *et al.*,2004). The use of relational descriptions allows for generalization across complex behaviors involving different objects.

Analysis of the convergence properties of reinforcement learning algorithms using function approximation is an extremely technical subject. Results for TD learning have been progressively strengthened for the case of linear function approximators (Sutton, 1988; Dayan, 1992; Tsitsiklis and Van Roy, 1997), but several examples of divergence have been presented for nonlinear functions (see Tsitsiklis and Van Roy, 1997, for a discussion). Papavassiliou and Russell (1999) describe a type of reinforcement learning that converges with any form of function approximator, provided that the problem of fitting the hypothesis to the data is solvable. Liu *et al.* (2018) describe the family of **gradient TD** algorithms and provide extensive theoretical analysis of convergence and sample complexity.

A variety of exploration methods for sequential decision problems are discussed by Barto *et al.* (1995). Kearns and Singh (1998) and Brafman and Tennenholtz (2000) describe algorithms that explore unknown environments and are guaranteed to converge on near-optimal policies with a sample complexity that is polynomial in the number of states. Bayesian reinforcement learning (Dearden *et al.*, 1998, 1999) provides another angle on both model uncertainty and exploration.

The basic idea underlying imitation learning is to apply supervised learning to a training set of expert actions. This is an old idea in adaptive control, but first came to prominence in AI with the work of Sammut *et al.* (1992) on “Learning to Fly” in a flight simulator. They called their method **behavioral cloning**. A few years later, the same research group reported that the method was much more fragile than had been reported initially (Camacho and Michie, 1995): even very small perturbations caused the learned policy to deviate from the desired trajectory, leading to compounding errors as the agent strayed further and further from the training set. (See also the discussion on page 973.) Work on apprenticeship learning aims to make the approach more robust, in part by including information about the desired outcomes rather than just the expert policy. Ng *et al.* (2003) and Coates *et al.* (2009) show how apprenticeship learning works for learning to fly an actual helicopter, as illustrated in Figure 23.9(b) on page 868.

Inverse reinforcement learning (IRL) was introduced by Russell (1998), and the first algorithms were developed by Ng and Russell (2000). (A similar problem has been studied in economics for much longer, under the heading of **structural estimation of MDPs** (Sargent, 1978).) The algorithm given in the chapter is due to Abbeel and Ng (2004). Baker *et al.*(2009) describe how the understanding of another agent's actions can be seen as inverse planning. Ho *et al.* (2017) show that agents can learn better from behaviors that are *instructive* rather than *optimal*. Hadfield-Menell *et al.* (2017a) extend IRL into a game-theoretic formulation that encompasses both observer and demonstrator, showing how teaching and learning behaviors emerge as solutions of the game.

García and Fernández (2015) give a comprehensive survey on safe reinforcement learning. Munos *et al.* (2017) describe an algorithm for safe off-policy (e.g., Q-learning) exploration. Hans *et al.* (2008) break the problem of safe exploration into two parts: defining a safety function to indicate which states to avoid, and defining a backup policy to lead the agent back to safety when it might otherwise enter an unsafe state. You *et al.* (2017) show how to train a deep reinforcement learning model to drive a car in simulation, and then use transfer learning to drive safely in the real world.

Thomas *et al.* (2017) offer an approach to learning that is guaranteed, with high probability, to do no worse than the current policy. Akametalu *et al.* (2014) describe a reachability-based approach, in which the learning process operates under the guidance of a control policy that ensures the agent never reaches an unsafe state. Saunders *et al.* (2018) demonstrate that a system can use human intervention to stop it from wandering out of the safe region, and can learn over time to need less intervention.

Policy search methods were brought to the fore by Williams (1992), who developed the REINFORCE family of algorithms, which stands for "REward Increment = Nonnegative Factor × Offset Reinforcement × Characteristic Eligibility." Later work by Marbach and Tsitsiklis (1998), Sutton *et al.* (2000), and Baxter and Bartlett (2000) strengthened and generalized the convergence results for policy search. Schulman *et al.* (2015b) describe **trust region policy optimization**, a theoretically well-founded and also practical policy search algorithm that has spawned many variants. The method of correlated sampling to reduce variance in Monte Carlo comparisons is due to Kahn and Marshall (1953); it is also one of a number of variance reduction methods explored by Hammersley and Handscomb (1964).

Early approaches to hierarchical reinforcement learning (HRL) attempted to construct hierarchies using **state abstraction**—that is, grouping states together into abstract states and then doing RL in the abstract state space (Dayan and Hinton, 1993). Unfortunately, the transition model for abstract states is typically non-Markovian, leading to divergent behavior of standard RL algorithms. The temporal abstraction approach in this chapter was developed in the late 1990s (Parr and Russell, 1998; Andre and Russell, 2002; Sutton *et al.*, 2000) and extended to handle concurrent behaviors by Marthi *et al.* (2005). Dietterich (2000) introduced the notion of an additive decomposition of Q-functions induced by the subroutine hierarchy. Temporal abstraction is based on a much earlier result due to Forestier and Varaiya (1978), who showed that a large MDP can be decomposed into a two-layer system in which a supervisory layer chooses among low-level controllers, each of which returns control to the supervisor on completion. The problem of learning the abstraction hierarchy itself has been studied at least since the work of Peter Andreae (1985); for a recent exploration into learning robot motion primitives, see Frans *et al.* (2018). The keepaway game was introduced by Stone *et al.* (2005); the HRL solution given here is due to Bai and Russell (2017).

Neuroscience has often inspired reinforcement learning and confirmed the value of the approach. Research using single-cell recording suggests that the dopamine system in primate brains implements something resembling value-function learning (Schultz *et al.*, 1997). The neuroscience text by Dayan and Abbott (2001) describes possible neural implementations of temporal-difference learning; related research describes other neuroscientific and behavioral experiments (Dayan and Niv, 2008; Niv, 2009; Lee *et al.*, 2012).

Work in reinforcement learning has been accelerated by the availability of open-source simulation environments for developing and testing learning agents. The University of Alberta’s Arcade Learning Environment (ALE) (Bellemare *et al.*, 2013) provided such a framework for 55 classic Atari video games. The pixels on the screen are provided to the agent as percepts, along with a hardwired score of the game so far. ALE was used by the DeepMind team to implement DQN learning and verify the generality of their system on a wide variety of games (Mnih *et al.*, 2015).

DeepMind in turn open-sourced several agent platforms, including the DeepMind Lab (Beattie *et al.*, 2016), the AI Safety Gridworlds (Leike *et al.*, 2017), the Unity game platform (Juliani *et al.*, 2018), and the DM Control Suite (Tassa *et al.*, 2018). Blizzard released the StarCraft II Learning Environment (SC2LE), to which DeepMind added the PySC2 component for machine learning in Python (Vinyals *et al.*, 2017a).

Facebook's AI Habitat simulation (Savva *et al.*, 2019) provides a photo-realistic virtual environment for indoor robotic tasks, and their HORIZON platform (Gauci *et al.*, 2018) enables reinforcement learning in large-scale production systems. The SYNTHIA system (Ros *et al.*, 2016) is a simulation environment designed for improving the computer vision capabilities of self-driving cars. The OpenAI Gym (Brockman *et al.*, 2016) provides several environments for reinforcement learning agents, and is compatible with other simulations such as the Google Football simulator.

Littman (2015) surveys reinforcement learning for a general scientific audience. The canonical text by Sutton and Barto (2018), two of the field's pioneers, shows how reinforcement learning weaves together the ideas of learning, planning, and acting. Kochenderfer (2015) takes a slightly less mathematical approach, with plenty of real-world examples. A short book by Szepesvari (2010) gives an overview of reinforcement learning algorithms. Bertsekas and Tsitsiklis (1996) provide a rigorous grounding in the theory of dynamic programming and stochastic convergence. Reinforcement learning papers are published frequently in the journals *Machine Learning* and *Journal of Machine Learning Research*, and in the the proceedings of the International Conference on Machine Learning (ICML) and the Neural Information Processing Systems (NeurIPS) conferences.

[1] As Yann LeCun and Alyosha Efros have pointed out, “the AI revolution will not be supervised.”

[2] In the RL literature, which draws more on operations research than economics, utility functions are often called **value functions** and denoted $V(s)$.

[3] The technical conditions are given on page 702. In Figure 23.5 we have used $\alpha(n) = 60/(59 + n)$, which satisfies the conditions.

[4] The definition of distance between utility functions is rather technical; see Tsitsiklis and Van Roy (1997).

[5] Rumors that keepaway was inspired by the real-world tactics of Barcelona FC are probably unfounded.

[6] Because more than one physical action may be executed before the next choice state is reached, the problem is technically a semi-Markov decision process, which allows actions to have different durations, including stochastic durations. If the discount factor $\gamma < 1$, then the action duration affects the discounting applied to the reward obtained during the action, which means that some extra discount bookkeeping has to be done and the transition model includes the duration distribution.

[7] According to Equation (16.9) on page 559, a reward function $R'(s, a, s') = R(s, a, s') + \gamma\Phi(s') - \Phi(s)$ has exactly the same optimal policies as $R(s, a, s')$, so we can recover the reward function only up to the possible addition of any shaping function $\Phi(s)$. This is not such a serious problem, because a robot using R' will behave just like a robot using the "correct" R.

CHAPTER 24

NATURAL LANGUAGE PROCESSING

In which we see how a computer can use natural language to communicate with humans and learn from what they have written.

About 100,000 years ago, humans learned how to speak, and about 5,000 years ago they learned to write. The complexity and diversity of human language sets *Homo sapiens* apart from all other species. Of course there are other attributes that are uniquely human: no other species wears clothes, creates art, or spends two hours a day on social media in the way that humans do. But when Alan Turing proposed his test for intelligence, he based it on language, not art or haberdashery, perhaps because of its universal scope and because language captures so much of intelligent behavior: a speaker (or writer) has the **goal** of communicating some **knowledge**, then **plans** some language that **represents** the knowledge, and **acts** to achieve the goal. The listener (or reader) **perceives** the language, and **infers** the intended meaning. This type of communication via language has allowed civilization to grow; it is our main means of passing along cultural, legal, scientific, and technological knowledge. There are three primary reasons for computers to do **natural language processing (NLP)**:

- To **communicate** with humans. In many situations it is convenient for humans to use speech to interact with computers, and in most situations it is more convenient to use natural language rather than a formal language such as first-order predicate calculus.
- To **learn**. Humans have written down a lot of knowledge using natural language. Wikipedia alone has 30 million pages of facts such as "Bush babies are small nocturnal primates," whereas there are hardly any sources of facts like this written in formal logic. If we want our system to know a lot, it had better understand natural language.
- To advance the **scientific understanding** of languages and language use, using the tools of AI in conjunction with linguistics, cognitive psychology, and neuroscience.

In this chapter we examine various mathematical models for language, and discuss the tasks that can be achieved using them.

24.1 Language Models

Formal languages, such as first-order logic, are precisely defined, as we saw in Chapter 8. A **grammar** defines the syntax of legal sentences and **semantic rules** define the meaning.

Natural languages, such as English or Chinese, cannot be so neatly characterized:

- Language judgments vary from person to person and time to time. Everyone agrees that "Not to be invited is sad" is a grammatical sentence of English, but people disagree on the grammaticality of "To be not invited is sad."
- Natural language is **ambiguous** ("He saw her duck" can mean either that she owns a waterfowl, or that she made a downwards evasive move) and **vague** ("That's great!" does not specify precisely how great it is, nor what it is).
- The mapping from symbols to objects is not formally defined. In first-order logic, two uses of the symbol "Richard" must refer to the same person, but in natural language two occurrences of the same word or phrase may refer to different things in the world.

If we can't make a definitive Boolean distinction between grammatical and ungrammatical strings, we can at least say how likely or unlikely each one is.

We define a **language model** as a probability distribution describing the likelihood of any string. Such a model should say that "Do I dare disturb the universe?" has a reasonable probability as a string of English, but "Universe dare the I disturb do?" is extremely unlikely.

With a language model, we can predict what words are likely to come next in a text, and thereby suggest completions for an email or text message. We can compute which alterations to a text would make it more probable, and thereby suggest spelling or grammar corrections. With a pair of models, we can compute the most probable translation of a sentence. With some example question/answer pairs as training data, we can compute the most likely answer to a question. So language models are at the heart of a broad range of natural language tasks. The language modeling task itself also serves as a common benchmark to measure progress in language understanding.

Natural languages are complex, so any language model will be, at best, an approximation. The linguist Edward Sapir said “No language is tyrannically consistent. All grammars leak” (Sapir, 1921). The philosopher Donald Davidson said “there is no such thing as language, not if a language is ... a clearly defined shared structure” (Davidson, 1986), by which he meant there is no one definitive language model for English in the way that there is for Python 3.8; we all have different models, but we still somehow manage to muddle through and communicate. In this section we cover simplistic language models that are clearly wrong, but still manage to be useful for certain tasks.

24.1.1 The bag-of-words model

Section 12.6.1 explained how a naive Bayes model based on the presence of specific words could reliably classify sentences into categories; for example sentence (1) below is categorized as *business*, and (2) as *weather.*

1. Stocks rallied on Monday, with major indexes gaining 1% as optimism persisted over the first quarter earnings season.

2. Heavy rain continued to pound much of the east coast on Monday, with flood warnings issued in New York City and other locations.

This section revisits the naive Bayes model, casting it as a full language model. That means we don't just want to know what category is most likely for each sentence; we want a joint probability distribution over all sentences and categories. That suggests we should consider *all* the words in the sentence. Given a sentence consisting of the words w_1, w_2,... w_N (which we will write as $w_{1:N}$, as in Chapter 14), the naive Bayes formula (Equation (12.21)) gives us

$$\mathbf{P}(Class \,\Big|\, w_{1:N}) = \alpha\, \mathbf{P}(Class) \prod_j \mathbf{P}(w_j | Class).$$

The application of naive Bayes to strings of words is called the **bag-of-words model**. It is a generative model that describes a process for generating a sentence: Imagine that for each category (*business*, *weather*, etc.) we have a bag full of words (you can imagine each word written on a slip of paper inside the bag; the more common the word, the more slips it is duplicated on). To generate text, first select one of the bags and discard the others. Reach into that bag and pull out a word at random; this will be the first word of the sentence. Then put the word back and draw a second word. Repeat until an end-of-sentence indicator (e.g., a period) is drawn.

This model is clearly wrong: it falsely assumes that each word is independent of the others, and therefore it does not generate coherent English sentences. But it does allow us to do classification with good accuracy using the naive Bayes formula: the words "stocks" and "earnings" are clear evidence for the business section, while "rain" and "cloudy" suggest the weather section.

We can learn the prior probabilities needed for this model via supervised training on a body or **corpus** of text, where each segment of text is labeled with a class. A corpus typically consists of at least a million words of text, and at least tens of thousands of distinct vocabulary words. Recently we are seeing even larger corpuses being used, such as the 2.5 billion words in Wikipedia or the 14 billion word iWeb corpus scraped from 22 million web pages.

From a corpus we can estimate the prior probability of each category, $\mathbf{P}(Class)$, by counting how common each category is. We can also use counts to estimate the conditional probability of each word given the category, $\mathbf{P}(w_j \mid Class)$. For example, if we've seen 3000 texts and 300 of them were classified as *business,* then we can estimate $P(Class = business) \approx 300/3000 = 0.1$. And if within the *business* category we have seen 100,000 words and the word "stocks" appeared 700 times, then we can estimate $P(stocks \mid Class = business) \approx 700/100{,}000 = 0.007$. Estimation by counting works well when we have high counts (and low variance), but we will see in Section 24.1.4 a better way to estimate probabilities when the counts are low.

Sometimes a different machine learning approach, such as logistic regression, neural networks, or support vector machines, can work even better than naive Bayes. The features of the machine learning model are the words in the vocabulary: "a," "aardvark," ..., "zyzzyva," and the values are the number of times each word appears in the text (or sometimes just a Boolean value indicating whether the word appears or not). That makes the feature vector large and sparse—we might have 100,000 words in the language model, and thus a feature vector of length 100,000, but for a short text almost all the features will be zero.

As we have seen, some machine learning models work better when we do **feature selection**, limiting ourselves to a subset of the words as features. We could drop words that are very rare (and thus have high variance in their predictive powers), as well as words that are common to all classes (such as “the”) but don’t discriminate between classes. We can also mix other features in with our word-based features; for example if we are classifying email messages we could add features for the sender, the time the message was sent, the words in the subject header, the presence of nonstandard punctuation, the percentage of uppercase letters, whether there is an attachment, and so on.

Note it is not trivial to decide what a *word* is. Is “aren’t” one word, or should it be broken up as “aren/’/t” or “are/n’t,” or something else? The process of dividing a text into a sequence of words is called **tokenization**.

24.1.2 N-gram word models

The bag-of-words model has limitations. For example, the word “quarter” is common in both the *business* and *sports* categories. But the four-word sequence “first quarter earnings report” is common only in *business* and “fourth quarter touchdown passes” is common only in *sports*. We’d like our model to make that distinction. We could tweak the bag-of-words model by treating special phrases like “first-quarter earnings report” as if they were single words, but a more principled approach is to introduce a new model, where each word is dependent on previous words. We can start by making a word dependent on *all* previous words in a sentence:

$$\mathbf{P}(w_{1:N}) = \prod_{j=1}^{N} \mathbf{P}(w_j | w_{1:j-1}).$$

This model is in a sense perfectly “correct” in that it captures all possible interactions between words, but it is not practical: with a vocabulary of

100,000 words and a sentence length of 40, this model would have 10^{200} parameters to estimate. We can compromise with a **Markov chain** model that considers only the dependence between *n* adjacent words. This is known as an **n-gram model** (from the Greek root *gramma* meaning "written thing"): a sequence of written symbols of length *n* is called an *n*-gram, with special cases "unigram" for 1-gram, "bigram" for 2-gram, and "trigram" for 3-gram. In an *n*-gram model, the probability of each word is dependent only on the $n - 1$ previous words; that is:

$$P(w_j | w_{1:j-1}) = P(w_j | w_{j-n+1:j-1})$$

$$P(w_{1:N}) = \prod_{j=1}^{N} P(w_j | w_{j-n+1:j-1}).$$

N-gram models work well for classifying newspaper sections, as well as for other classification tasks such as **spam detection** (distinguishing spam email from non-spam), **sentiment analysis** (classifying a movie or product review as positive or negative) and **author attribution** (Hemingway has a different style and vocabulary than Faulkner or Shakespeare).

24.1.3 Other n-gram models

An alternative to an *n*-gram word model is a **character-level model** in which the probability of each character is determined by the $n - 1$ previous characters. This approach is helpful for dealing with unknown words, and for languages that tend to run words together, as in the Danish word "Speciallffigepraksisplanlsgningsstabiliseringsperiode."

Character-level models are well suited for the task of **language identification:** given a text, determine what language it is written in. Even with very short texts such as "Hello, world" or "Wie geht's dir," *n*-gram letter models can identify the first as English and the second as German, generally achieving accuracy greater than 99%. (Closely related languages such as Swedish and Norwegian are more difficult to distinguish and require longer samples; there, accuracy is in the 95% range.) Character models are good at certain classification tasks, such as deciding that "dextroamphetamine" is a drug name, "Kallenberger" is a person name, and "Plattsburg" is a city name, even if we have never seen these words before.

Another possibility is the **skip-gram** model, in which we count words that are near each other, but skip a word (or more) between them. For example, given the French text "je ne comprends pas" the 1-skip-bigrams are "je comprends," and "ne pas." Gathering these helps create a better model of French, because it tells us about conjugation ("je" goes with "comprends," not "comprend") and negation ("ne" goes with "pas"); we wouldn't get that from regular bigrams alone.

24.1.4 Smoothing n-gram models

High-frequency *n*-grams like "of the" have high counts in the training corpus, so their probability estimate is likely to be accurate: with a different training corpus we would get a similar estimate. Low-frequency n-grams have low counts that are subject to random noise—they have high variance. Our models will perform better if we can smooth out that variance.

Furthermore, there is always a chance that we will be asked to evaluate a text containing an unknown or **out-of-vocabulary** word: one that never appeared in the training corpus. But it would be a mistake to assign such a word a probability of zero, because then the probability of the whole sentence, $P(w_{1:N})$, would be zero.

One way to model unknown words is to modify the training corpus by replacing infrequent words with a special symbol, traditionally `<UNK>`. We could decide in advance to keep only, say, the 50,000 most common words, or all words with frequency greater than 0.0001%, and replace the others with `<UNK>`. Then compute *n*-gram counts for the corpus as usual, treating `<UNK>` just like any other word. When an unknown word appears in a test set, we look up its probability under `<UNK>`. Sometimes different unknown-word symbols are used for different types. For example, a string of digits might be replaced with `<NUM>`, or an email address with `<EMAIL>`. (We note that it is also advisable to have a special symbol, such as `<S>`, to mark the start (and stop) of a text. That way, when the formula for bigram probabilities asks for the word before the first word, the answer is `<S>`, not an error.)

Even after we've handled unknown words, we have the problem of unseen *n*-grams. For example, a test text might contain the phrase "colorless aquamarine ideas," three words that we may have seen individually in the training corpus, but never in that exact order. The problem is that some low-probability *n*-grams appear in the training corpus, while other equally low-probability *n*-grams happen to not appear at all. We don't want some of them to have a zero probability while others have a small positive probability; we want to apply **smoothing** to all the similar *n*-grams—reserving some of the probability mass of the model for never-seen *n*-grams, to reduce the variance of the model.

The simplest type of smoothing was suggested by Pierre-Simon Laplace in the 18th century to estimate the probability of rare events, such as the sun failing to rise tomorrow. Laplace's (incorrect) theory of the solar system suggested it was about N = 2 million days old. Going by the data, there were zero out of two million days when the sun failed to rise, yet we don't want to say that the probability is exactly zero. Laplace showed that if we adopt a uniform prior, and combine that with the evidence so far, we get a best estimate of 1 / (N + 2) for the probability of the sun's failure to rise tomorrow—either it will or it won't (that's the 2 in the denominator) and a uniform prior says it is as likely as not (that's the 1 in the numerator). Laplace smoothing (also called add-one smoothing) is a step in the right direction, but for many natural language applications it performs poorly.

Another choice is a **backoff model**, in which we start by estimating n-gram counts, but for any particular sequence that has a low (or zero) count, we back off to (n – 1)-grams. **Linear interpolation smoothing** is a backoff model that combines trigram, bigram, and unigram models by linear interpolation. It defines the probability estimate as

$$\widehat{P}(c_i \big| c_{i-2:i-1}) = \lambda_3 P(c_i \big| c_{i-2:i-1}) + \lambda_2 P(c_i \big| c_{i-1}) + \lambda_1 P(c_i),$$

where $\lambda_3 + \lambda_2 + \lambda_1 = 1$. The parameter values λ_i can be fixed, or they can be trained with an expectation–maximization algorithm. It is also possible to have the values of λ_i, depend on the counts: if we have a high count of trigrams, then we weigh them relatively more; if only a low count, then we put more weight on the bigram and unigram models.

One camp of researchers has developed ever more sophisticated smoothing techniques (such as Witten-Bell and Kneser-Ney), while another camp suggests gathering a larger corpus so that even simple smoothing techniques work well (one such approach is called “stupid backoff”). Both are getting at the same goal: reducing the variance in the language model.

24.1.5 Word representations

N-grams can give us a model that accurately predicts the probability of word sequences, telling us that, for example, “a black cat” is a more likely English phrase than “cat black a” because “a black cat” appears in about 0.000014% of the trigrams in a training corpus, while “cat black a” does not appear at all. Everything that the *n*-gram word model knows, it learned from counts of specific word sequences.

But a native speaker of English would tell a different story: “a black cat” is valid because it follows a familiar pattern (article-adjective-noun), while “cat black a” does not.

Now consider the phrase “the fulvous kitten.” An English speaker could recognize this as also following the article-adjective-noun pattern (even a speaker who does not know that “fulvous” means “brownish yellow” could recognize that almost all words that end in “-ous” are adjectives). Furthermore, the speaker would recognize the close syntactic connection between “a” and “the,” as well as the close semantic relation between “cat” and “kitten.” Thus, the appearance of “a black cat” in the data is evidence, through generalization, that “the fulvous kitten” is also valid English.

The *n*-gram model misses this generalization because it is an *atomic* model: each word is an atom, distinct from every other word, with no internal structure. We have seen throughout this book that *factored* or *structured* models allow for more expressive power and better generalization. We will see in Section 25.1 that a factored model called **word embeddings** gives a better ability to generalize.

One type of structured word model is a **dictionary**, usually constructed through manual labor. For example, **WordNet** is an open-source, hand-curated dictionary in machine-readable format that has proven useful for many natural language applications[1] Below is the WordNet entry for “kitten:”

```
“kitten” <noun.animal> (“young domestic cat“) IS A: young_mammal

“kitten” <verb.body> (“give birth to kittens“)
```

```
EXAMPLE: “our cat kittened again this year”
```

WordNet will help you separate the nouns from the verbs, and get the basic categories (a kitten is a young mammal, which is a mammal, which is an animal), but it won’t tell you the details of what a kitten looks like or acts like. WordNet will tell you that two subclasses of cat are *Siamese cat* and *Manx cat*, but won’t tell you any more about the breeds.

24.1.6 Part-of-speech (POS) tagging

One basic way to categorize words is by their **part of speech (POS)**, also called **lexical category** or **tag:** *noun*, *verb*, *adjective*, and so on. Parts of speech allow language models to capture generalizations such as “adjectives generally come before nouns in English.” (In other languages, such as French, it is the other way around (generally)).

Everyone agrees that “noun” and “verb” are parts of speech, but when we get into the details there is no one definitive list. Figure 24.1 shows the 45 tags used in the **Penn Tree-bank,** a corpus of over three million words of text annotated with part-of-speech tags. As we will see later, the Penn Treebank also annotates many sentences with syntactic parse trees, from which the corpus gets its name. Here is an excerpt saying that “from” is tagged as a preposition (IN), “the” as a determiner (DT), and so on:

Tag	Word	Description	Tag	Word	Description
CC	*and*	Coordinating conjunction	PRP$	*your*	Possessive pronoun
CD	*three*	Cardinal number	RB	*quickly*	Adverb
DT	*the*	Determiner	RBR	*quicker*	Adverb, comparative
EX	*there*	Existential there	RBS	*quickest*	Adverb, superlative
FW	*per se*	Foreign word	RP	*off*	Particle
IN	*of*	Preposition	SYM	+	Symbol
JJ	*purple*	Adjective	TO	*to*	to
JJR	*better*	Adjective, comparative	UH	*eureka*	Interjection
JJS	*best*	Adjective, superlative	VB	*talk*	Verb, base form
LS	*1*	List item marker	VBD	*talked*	Verb, past tense
MD	*should*	Modal	VBG	*talking*	Verb, gerund
NN	*kitten*	Noun, singular or mass	VBN	*talked*	Verb, past participle
NNS	*kittens*	Noun, plural	VBP	*talk*	Verb, non-3rd-sing
NNP	*Ali*	Proper noun, singular	VBZ	*talks*	Verb, 3rd-sing
NNPS	*Fords*	Proper noun, plural	WDT	*which*	Wh-determiner
PDT	*all*	Predeterminer	WP	*who*	Wh-pronoun
POS	*’s*	Possessive ending	WP$	*whose*	Possessive wh-pronoun
PRP	*you*	Personal pronoun	WRB	*where*	Wh-adverb
$	$	Dollar sign	#	#	Pound sign
“	‘	Left quote	”	’	Right quote
(	[	Left parenthesis	)	]	Right parenthesis
,	,	Comma	.	!	Sentence end
:	;	Mid-sentence punctuation			

Figure 24.1 Part-of-speech tags (with an example word for each tag) for the Penn Treebank corpus (Marcus *et al.*, 1993). Here “3rd-sing” is an abbreviation for “third person singular present tense.”

From	the	start	,	it	took	a	person	with	great	qualities	to	succeed
IN	DT	NN	,	PRP	VBD	DT	NN	IN	JJ	NNS	TO	VB

The task of assigning a part of speech to each word in a sentence is called **part-of-speech tagging**. Although not very interesting in its own right, it is a useful first step in many other NLP tasks, such as question answering or translation. Even for a simple task like text-to-speech synthesis, it is important to know that the noun "record" is pronounced differently from the verb "record." In this section we will see how two familiar models can be applied to the tagging task, and in Chapter 25 we will consider a third model.

One common model for POS tagging is the **hidden Markov model (HMM)**. Recall from Section 14.3 that a hidden Markov model takes in a temporal sequence of evidence observations and predicts the most likely hidden states that could have produced that sequence. In the HMM example on page 491, the evidence consisted of observations of a person carrying an umbrella (or not), and the hidden state was rain (or not) in the outside world. For POS tagging, the evidence is the sequence of words, $W_{1:N}$, and the hidden states are the lexical categories, $C_{1:N}$.

The HMM is a generative model that says that the way to produce language is to start in one state, such as IN, the state for prepositions, and then make two choices: what word (such as *from*) should be emitted, and what state (such as DT) should come next. The model does not consider any context other than the current part-of-speech state, nor does it have any idea of what the sentence is actually trying to convey. And yet it is a useful model—if we apply the **Viterbi algorithm** (Section 14.2.3) to find the most probable sequence of hidden states (tags), we find that the tagging achieves very high accuracy; usually around 97%.

To create a HMM for POS tagging, we need the transition model, which gives the probability of one part of speech following another, $\mathbf{P}(C_t \mid C_{t-1})$, and the sensor model, $\mathbf{P}(W_t \mid C_t)$. For example, $\mathbf{P}(C_t = VB \mid C_{t-1} = MD) = 0.8$ means that given a modal verb (such as *would*), we can expect the following word to be a verb (such as *think)* with probability 0.8. Where does the 0.8 number come from? Just as with n-gram models, from counts in the corpus, with appropriate smoothing. It turns out that there are 13124 instances of *MD* in the Penn Treebank, and 10471 of them are followed by a *VB*.

For the sensor model, $P(W_t = would \mid C_t = MD) = 0.1$ means that when we are choosing a modal verb, we will choose *would* 10% of the time. These numbers also come from corpus counts, with smoothing.

A weakness of HMM models is that everything we know about language has to be expressed in terms of the transition and sensor models. The part of speech for the current word is determined solely by the probabilities in these two models and by the part of speech of the previous word. There is no easy way for a system developer to say, for example, that *any* word that ends in “ous” is likely an adjective, nor that in the phrase “attorney general,” *attorney* is a noun, not an adjective.

Fortunately, **logistic regression** does have the ability to represent information like this. Recall from Section 19.6.5 that in a logistic regression model, the input is a vector, **x**, of feature values. We then take the dot product, $\mathbf{w} \cdot \mathbf{x}$, of those features with a pretrained vector of weights w, and transform that sum into a number between 0 and 1 that can be interpreted as the probability that the input is a positive example of a category.

The weights in the logistic regression model correspond to how predictive each feature is for each category; the weight values are learned by gradient descent. For POS tagging we would build 45 different logistic regression models, one for each part of speech, and ask each model how probable it is that the example word is a member of that category, given the feature values for that word in its particular context.

The question then is what should the features be? POS taggers typically use binaryvalued features that encode information about the word being tagged, w_i (and perhaps other nearby words), as well as the category that was assigned to the previous word, c_{i-1} (and perhaps the category of earlier words). Features can depend on the exact identity of a word, some aspects of the way it is spelled, or some attribute from a dictionary entry. A set of POS tagging features might include:

$w_{i-1} =$ "I"	$w_{i+1} =$ "for"
$w_{i-1} =$ "you"	$c_{i-1} =$ IN
w_i ends with "ous"	w_i contains a hyphen
w_i ends with "ly"	w_i contains a digit
w_i starts with "un"	w_i is all uppercase
$w_{i-2} =$ "to" and $c_{i-1} =$ VB	w_{i-2} has attribute PRESENT
$w_{i-1} =$ "I" and $w_{i+1} =$ "to"	w_{i-2} has attribute PAST

For example, the word "walk" can be a noun or a verb, but in "I walk to school," the feature in the last row, left column could be used to classify "walk" as a verb (VBP). As another example, the word "cut" can be either a noun (NN), past tense verb (VBD), or present tense verb (VBP). Given the sentence "Yesterday I cut the rope," the feature in the last row, right column could help tag "cut" as VBD, while in the sentence "Now I cut the rope," the feature above that one could help tag "cut" as VBP.

All together, there might be a million features, but for any given word, only a few dozen will be nonzero. The features are usually hand-crafted by a human system designer who thinks up interesting feature templates.

Logistic regression does not have the notion of a sequence of inputs—you give it a single feature vector (information about a single word) and it produces an output (a tag). But we can force logistic regression to handle a sequence with a **greedy search**: start by choosing the most likely category for the first word, and proceed to the rest of the words in left-to-right order. At each step the category c_i is assigned according to

$$c_i = \operatorname*{argmax}_{\hat{c} \in Categories} P(c' \middle| w_{1:N}, c_{1:i-1}).$$

That is, the classifier is allowed to look at any of the non-category features for any of the words anywhere in the sentence (because these features are all fixed), as well as any previously assigned categories.

Note that the greedy search makes a definitive category choice for each word, and then moves on to the next word; if that choice is contradicted by evidence later in the sentence, there is no possibility to go back and reverse the choice. That makes the algorithm fast. The Viterbi algorithm, in contrast, keeps a table of all possible category choices at each step, and always has the option of changing. That makes the algorithm more accurate, but slower. For both algorithms, a compromise is a **beam search**, in which we consider every possible category at each time step, but then only keep the b most likely tags, dropping the other less-likely tags. Changing *b* trades off speed versus accuracy.

Naive Bayes and Hidden Markov models are **generative models** (see Section 21.2.3). That is, they learn a joint probability distribution, $\mathbf{P}(W, C)$, and we can generate a random sentence by sampling from that probability distribution to get a first word (with category) of the sentence, and then adding words one at a time.

Logistic regression on the other hand is a **discriminative model**. It learns a conditional probability distribution $\mathbf{P}(C \mid W)$, meaning that it can assign categories given a sequence of words, but it can't generate random sentences. Generally, researchers have found that discriminative models have a lower error rate, perhaps because they model the intended output directly, and perhaps because they make it easier for an analyst to create additional features. However, generative models tend to converge more quickly, and so may be preferred when the available training time is short, or when there is limited training data.

24.1.7 Comparing language models

To get a feeling for what different *n*-gram models are like, we built unigram (i.e., bag-of-words), bigram, trigram, and 4-gram models over the words in this book and then randomly sampled word sequences from each of the four models:

- n = 1: *logical are as are confusion a may right tries agent goal the was*
- n = 2: *systems are very similar computational approach would be represented*
- n = 3: *planning and scheduling are integrated the success of naive Bayes model is*
- n = 4: *taking advantage of the structure of Bayesian networks and developed various languages for writing "templates" with logical variables, from which large networks could be constructed automatically for each problem instance*

From this small sample it should be clear that the unigram model is a very poor approximation of either English in general or an AI textbook in particular, and that the 4-gram model is imperfect but much better. Next, to demonstrate how samples switch between training sources (and definitely not just for fun) we added the text of the King James Bible to the 4-gram model yielding these random samples:

- *Prove that any 3-SAT problem can be reduced to simpler ones using the laws of thy God.*
- *Masters, give unto your servants that which is true iff both P and Q in any model m by a simple experiment: put your hand unto, ye and your*

households for it is pleasant.

- *Many will intreat the LORD your God, Saying, No; but we will ignore this issue for now; Chapters 7 and 8 suggest methods for compactly representing very large belief states.*
- *And it came to pass, as if it had no successors.*
- *The direct utility estimation is just an instance of the general or algorithm in which new function symbols are constructed "on the fly." For example, the first child of the Holy Ghost.*

There is a limit to *n*-gram models—as *n* increases, they will produce language that is more fluent, but they tend to reproduce long passages from their training data verbatim, rather than generate novel text. Language models with more complex representations of words and context can do better. The rest of this chapter demonstrates how **grammar** can improve a language model, and Chapter 25 shows how deep learning methods have recently produced impressive language models. One such deep learning model, GPT-2, can produce fluent English samples when given a prompt. We gave GPT-2 the first two sentences of this paragraph as a prompt; it produced the following two samples:

- *Numerous studies have shown that the best language models that can generate language are those that use neural networks, and in particular that use deep learning to model speech. I have used one of these models to generate hundreds of sentences in multiple languages, and it has the ability to generate almost perfect translations of my training sentences. It's a great model to have, and will save you a lot of time in the future when you need to translate between many languages.*
- *These models can also do better if their samples are chosen at random, rather than by an expert in the domain. However, as a general rule, n-gram and language models that include a large number of features*

tend to produce text better, since these features contribute the most to language learning. To measure the performance of a model, I've made some sample sentences that I'd like to train to better understand speech and language models. You can read them to get a grasp of how the model predicts.

We see that these passages are diverse and grammatically fluent; moreover, they stick to topics that are relevant to the prompt sentences. But the sentences do not build on each other to advance a coherent thesis. The GPT-2 language model is known as a **transformer model**, which will be covered in Section 25.4; further examples from GPT-2 are in Figure 25.14. Another transformer model is the Conditional Transformer Language, CTRL. It can be controlled more flexibly; in the following samples CTRL was asked to generate text in the category *product reviews*, with a rating of 1 and of 4 (out of 5): specified rating (out of 5):

- **1.0**: *I bought this for my son who is a huge fan of the show. He was so excited to get it and when he opened it, we were all very disappointed. The quality of the product is terrible. It looks like something you would buy at a dollar store.*
- **4.0**: *I bought this for my husband and he loves it. He has a small wrist so it is hard to find watches that fit him well. This one fits perfectly.*

24.2 Grammar

In Chapter 7 we used Backus–Naur Form (BNF) to write down a grammar for the language of first-order logic. A **grammar** is a set of rules that defines the tree structure of allowable phrases, and a **language** is the set of sentences that follow those rules.

Natural languages do not work exactly like the formal language of first-order logic—they do not have a hard boundary between allowable and unallowable sentences, nor do they have a single definitive tree structure for each sentence. However, hierarchical structure *is* important in natural language. The word “Stocks” in “Stocks rallied on Monday” is not just a word, nor is it just a *noun;* in this sentence it also comprises a *noun phrase,* which is the subject of the following *verb phrase*. **Syntactic categories** such as *noun phrase* or *verb phrase* help to constrain the probable words at each point within a sentence, and the **phrase structure** provides a framework for the meaning or semantics of the sentence.

There are many competing language models based on the idea of hierarchical syntactic structure; in this section we will describe a popular model called the **probabilistic context-free grammar**, or PCFG. A probabilistic grammar assigns a probability to each string, and “context-free” means that any rule can be used in any context: the rules for a noun phrase at the beginning of a sentence are the same as for another noun phrase later in the sentence, and if the same phrase occurs in two locations, it must have the same probability each time. We will define a PCFG grammar for a tiny fragment of English that is suitable for communication between agents exploring the wumpus world. We call this language ξ_0 (see Figure 24.2). A grammar rule such as

S	$\rightarrow$	*NP VP*	[0.90]	I + feel a breeze
	$\mid$	*S Conj S*	[0.10]	I feel a breeze + and + It stinks
NP	$\rightarrow$	*Pronoun*	[0.25]	I
	$\mid$	*Name*	[0.10]	Ali
	$\mid$	*Noun*	[0.10]	pits
	$\mid$	*Article Noun*	[0.25]	the + wumpus
	$\mid$	*Article Adjs Noun*	[0.05]	the + smelly dead + wumpus
	$\mid$	*Digit Digit*	[0.05]	3 4
	$\mid$	*NP PP*	[0.10]	the wumpus + in 1 3
	$\mid$	*NP RelClause*	[0.05]	the wumpus + that is smelly
	$\mid$	*NP Conj NP*	[0.05]	the wumpus + and + I
VP	$\rightarrow$	*Verb*	[0.40]	stinks
	$\mid$	*VP NP*	[0.35]	feel + a breeze
	$\mid$	*VP Adjective*	[0.05]	smells + dead
	$\mid$	*VP PP*	[0.10]	is + in 1 3
	$\mid$	*VP Adverb*	[0.10]	go + ahead
$Adjs$	$\rightarrow$	*Adjective*	[0.80]	smelly
	$\mid$	*Adjective Adjs*	[0.20]	smelly + dead
PP	$\rightarrow$	*Prep NP*	[1.00]	to + the east
$RelClause$	$\rightarrow$	*RelPro VP*	[1.00]	that + is smelly

Figure 24.2 The grammar for ξ_0, with example phrases for each rule. The syntactic categories are sentence (*S*), noun phrase (*NP*), verb phrase (*VP*), list of adjectives *(Adjs)*, prepositional phrase (*PP*), and relative clause *(RelClause)*.

$$\begin{aligned} Adjs &\rightarrow Adjective & [0.80] \\ &\mid\ \ Adjective\ Adjs & [0.20] \end{aligned}$$

means that the syntactic category *Adjs* can consist of either a single *Adjective*, with probability 0.80, or of an *Adjective* followed by a string that constitutes an *Adjs*, with probability 0.20.

Unfortunately, the grammar **overgenerates**: that is, it generates sentences that are not grammatical, such as “Me go I.” It also **undergenerates**: there are many sentences of English that it rejects, such as “I think the wumpus is smelly.” We will see how to learn a better grammar later; for now we concentrate on what we can do with this very simple grammar.

24.2.1 The lexicon of $\mathcal{E}_0$

The **lexicon**, or list of allowable words, is defined in Figure 24.3. Each of the lexical categories ends in ... to indicate that there are other words in the category. For nouns, names, verbs, adjectives, and adverbs, it is infeasible even in principle to list all the words. Not only are there tens of thousands of members in each class, but new ones—like *humblebrag* or *microbiome*—are being added constantly. These five categories are called **open classes**. Pronouns, relative pronouns, articles, prepositions, and conjunctions are called **closed classes**; they have a small number of words (a dozen or so), and change over the course of centuries, not months. For example, “thee” and “thou” were commonly used pronouns in the 17th century, were on the decline in the 19th century, and are seen today only in poetry and some regional dialects.

Noun	→	**stench** [0.05] | **breeze** [0.10] | **wumpus** [0.15] | **pits** [0.05] | ...
Verb	→	**is** [0.10] | **feel** [0.10] | **smells** [0.10] | **stinks** [0.05] | ...
Adjective	→	**right** [0.10] | **dead** [0.05] | **smelly** [0.02] | **breezy** [0.02] ...
Adverb	→	**here** [0.05] | **ahead** [0.05] | **nearby** [0.02] | ...
Pronoun	→	**me** [0.10] | **you** [0.03] | **I** [0.10] | **it** [0.10] | ...
RelPro	→	**that** [0.40] | **which** [0.15] | **who** [0.20] | **whom** [0.02] | ...
Name	→	**Ali** [0.01] | **Bo** [0.01] | **Boston** [0.01] | ...
Article	→	**the** [0.40] | **a** [0.30] | **an** [0.10] | **every** [0.05] | ...
Prep	→	**to** [0.20] | **in** [0.10] | **on** [0.05] | **near** [0.10] | ...
Conj	→	**and** [0.50] | **or** [0.10] | **but** [0.20] | **yet** [0.02] | ...
Digit	→	**0** [0.20] | **1** [0.20] | **2** [0.20] | **3** [0.20] | **4** [0.20] | ...

Figure 24.3 The lexicon for ξ_0. *RelPro* is short for relative pronoun, *Prep* for preposition, and *Conj* for conjunction. The sum of the probabilities for each category is 1.

24.3 Parsing

Parsing is the process of analyzing a string of words to uncover its phrase structure, according to the rules of a grammar. We can think of it as a **search** for a valid parse tree whose leaves are the words of the string. Figure 24.4 shows that we can start with the *S* symbol and search top down, or we can start with the words and search bottom up. Pure top-down or bottom-up parsing strategies can be inefficient, however, because they can end up repeating effort in areas of the search space that lead to dead ends. Consider the following two sentences:

List of items	Rule
S	
NP VP	*S* → *NP VP*
NP VP Adjective	*VP* → *VP Adjective*
NP Verb Adjective	*VP* → *Verb*
NP Verb **dead**	*Adjective* → **dead**
NP **is dead**	*Verb* → **is**
Article Noun **is dead**	*NP* → *Article Noun*
Article **wumpus is dead**	*Noun* → **wumpus**
the wumpus is dead	*Article* → **the**

Figure 24.4 Parsing the string “The wumpus is dead” as a sentence, according to the grammar ξ_0. Viewed as a top-down parse, we start with S, and on each step match one nonterminal X with a rule of the form ($X \rightarrow Y$ ...) and replace X in the list of items with Y ...; for example replacing S with the sequence NP

VP. Viewed as a bottom-up parse, we start with the words "the wumpus is dead", and on each step match a string of tokens such as (Y ...) against a rule of the form ($X \rightarrow Y$ ...) and replace the tokens with X; for example replacing "the" with *Article* or *Article Noun* with *NP*.

Have the students in section 2 of Computer Science 101 take the exam.
Have the students in section 2 of Computer Science 101 taken the exam?

Even though they share the first 10 words, these sentences have very different parses, because the first is a command and the second is a question. A left-to-right parsing algorithm would have to guess whether the first word is part of a command or a question and will not be able to tell if the guess is correct until at least the eleventh word, *take* or *taken*. If the algorithm guesses wrong, it will have to backtrack all the way to the first word and reanalyze the whole sentence under the other interpretation.

To avoid this source of inefficiency we can use **dynamic programming**: every time we analyze a substring, store the results so we won't have to reanalyze it later. For example, once we discover that "the students in section 2 of Computer Science 101" is an *NP*, we can record that result in a data structure known as a **chart**. An algorithm that does this is called a **chart parser**. Because we are dealing with context-free grammars, any phrase that was found in the context of one branch of the search tree can work just as well in any other branch of the search tree. There are many types of chart parsers; we describe a probabilistic version of a bottom-up chart parsing algorithm called the **CYK algorithm**, after its inventors, Ali Cocke, Daniel Younger, and Tadeo Kasami.[2]

The CYK algorithm is shown in Figure 24.5. It requires a grammar with all rules in one of two very specific formats: lexical rules of the form X

→ **word** [p], and syntactic rules of the form $X \rightarrow Y\ Z$ [p], with exactly two categories on the right-hand side. This grammar format, called **Chomsky Normal Form**, may seem restrictive, but it is not: any context-free grammar can be automatically transformed into Chomsky Normal Form. Exercise 24.CNFX leads you through the process.

```
function CYK-PARSE(words, grammar) returns a table of parse trees
  inputs: words, a list of words
          grammar, a structure with LEXICALRULES and GRAMMARRULES
  T ← a table        // T[X, i, k] is most probable X tree spanning words_{i:k}
  P ← a table, initially all 0        // P[X, i, k] is probability of tree T[X, i, k]
  // Insert lexical categories for each word.
  for i = 1 to LEN(words) do
    for each (X, p) in grammar.LEXICALRULES(words_i) do
      P[X, i, i] ← p
      T[X, i, i] ← TREE(X, words_i)
  // Construct X_{i:k} from Y_{i:j} + Z_{j+1:k}, shortest spans first.
  for each (i, j, k) in SUBSPANS(LEN(words)) do
    for each (X, Y, Z, p) in grammar.GRAMMARRULES do
      PYZ ← P[Y, i, j] × P[Z, j+1, k] × p
      if PYZ > P[X, i, k] do
        P[X, i, k] ← PYZ
        T[X, i, k] ← TREE(X, T[Y, i, j], T[Z, j + 1, k])
  return T

function SUBSPANS(N) yields (i, j, k) tuples
  for length = 2 to N do
    for i = 1 to N + 1 − length do
      k ← i + length − 1
      for j = i to k − 1 do
        yield (i, j, k)
```

Figure 24.5 The CYK algorithm for parsing. Given a sequence of words, it finds the most probable parse tree for the sequence and its subsequences. The table $P[X, i, k]$ gives the probability of the most probable tree of category X spanning $words_{i:k}$. The output table $T[X, i, k]$ contains the most probable tree of category X spanning positions i to k inclusive. The function SUBSPANS returns all tuples (i,j,k) covering a span of $words_{i:k}$, with $i \leq j < k$, listing the tuples by increasing length of the $i: k$ span, so that when we go to combine two shorter spans into a longer one, the shorter spans are already in the table. LEXICALRULES(word) returns a collection of (X, p) pairs, one for each rule of the form $X \rightarrow$ word [p], and GRAMMARRULES gives (X, Y, Z, p) tuples, one for each grammar rule of the form $X \rightarrow Y\, Z\, [p]$.

The CYK algorithm uses space of $O(n^2m)$ for the P and T tables, where n is the number of words in the sentence, and m is the number of nonterminal symbols in the grammar, and takes time $O(n^3m)$. If we want an algorithm that is guaranteed to work for all possible context-free grammars, then we can't do any better than that. But actually we only want to parse natural languages, not all possible grammars. Natural languages have evolved to be easy to understand in real time, not to be as tricky as possible, so it seems that they should be amenable to a faster parsing algorithm.

To try to get to $O(n)$, we can apply $A*$ search in a fairly straightforward way: each state is a list of items (words or categories), as shown in Figure 24.4. The start state is a list of words, and a goal state is the single item S. The cost of a state is the inverse of its probability as defined by the rules applied so far, and there are various heuristics to estimate the remaining distance to the goal; the best heuristics in current use come from machine learning applied to a corpus of sentences.

With the *A** algorithm we don't have to search the entire state space, and we are guaranteed that the first parse found will be the most probable (assuming an admissible heuristic). This will usually be faster than CYK, but (depending on the details of the grammar) still slower than $O(n)$. An example result of a parse is shown in Figure 24.6.

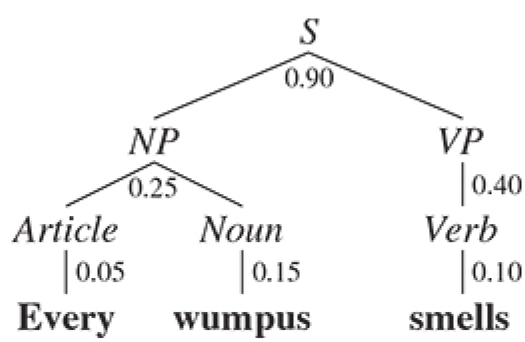


Figure 24.6 Parse tree for the sentence "Every wumpus smells" according to the grammar ξ_0. Each interior node of the tree is labeled with its probability. The probability of the tree as a whole is $0.9 \times 0.25 \times 0.05 \times 0.15 \times 0.40 \times 0.10 = 0.0000675$. The tree can also be written in linear form as [*S* [*NP* [*Article* **every**] [*Noun* **wumpus**]][*VP* [*Verb* **smells**]]].

Just as with part-of-speech tagging, we can use a **beam search** for parsing, where at any time we consider only the *b* most probable alternative parses. This means we are not guaranteed to find the parse with highest probability, but (with a careful implementation) the parser can operate in $O(n)$ time and still finds the best parse most of the time.

A beam search parser with $b = 1$ is called a **deterministic parser**. One popular deterministic approach is **shift-reduce parsing**, in which we go through the sentence word by word, choosing at each point whether to shift the word onto a stack of constituents, or to reduce the top constituent(s) on the stack according to a grammar rule. Each style of parsing has its adherents within the NLP community. Even though it is possible to transform a shift-reduce system into a PCFG (and vice versa), when you apply machine learning to the problem of inducing a grammar, the inductive bias and hence the generalizations that each system will make will be different (Abney *et al.*, 1999).

24.3.1 Dependency parsing

There is a widely used alternative syntactic approach called **dependency grammar**, which assumes that syntactic structure is formed by binary relations between lexical items, without a need for syntactic constituents. Figure 24.7 shows a sentence with a dependency parse and a phrase structure parse.

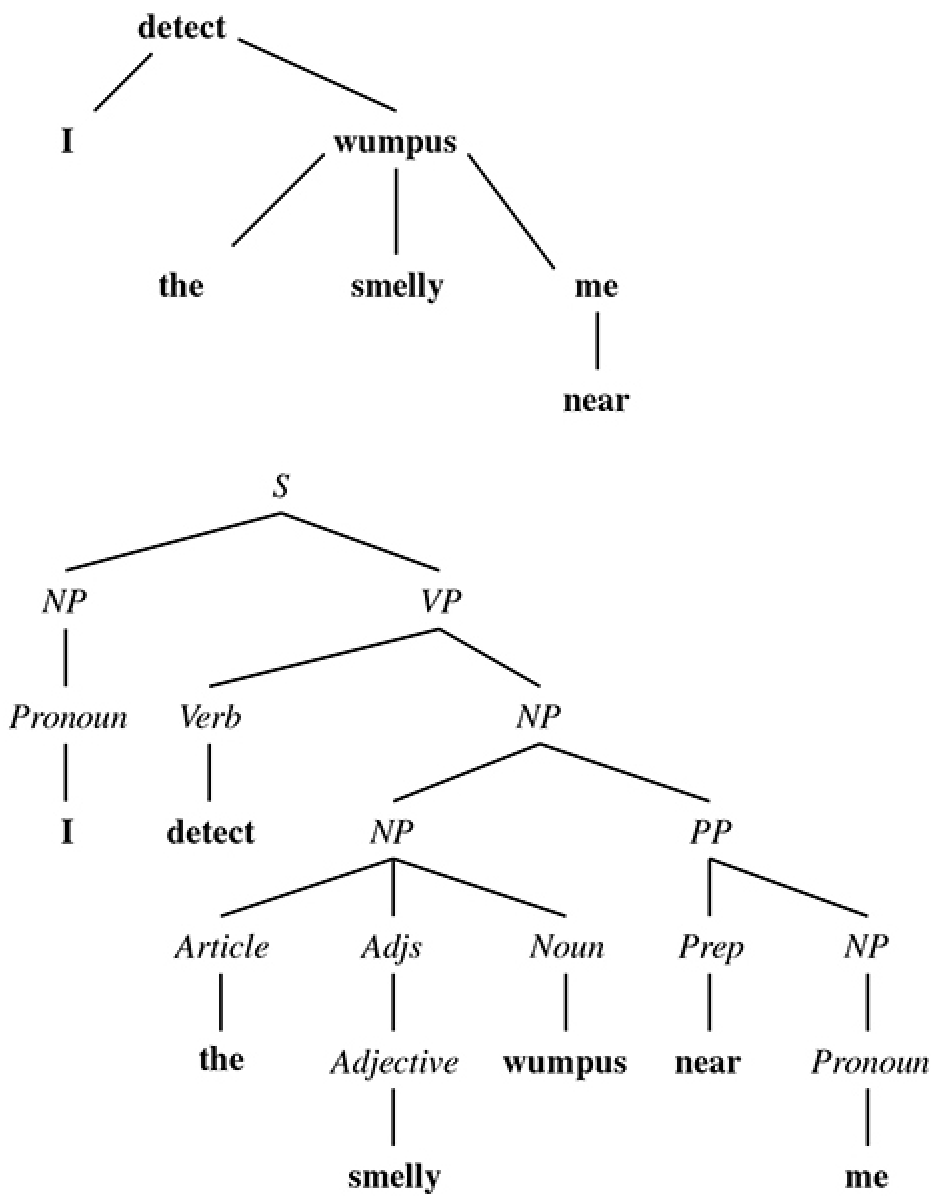


Figure 24.7 A dependency-style parse (top) and the corresponding phrase structure parse (bottom) for the sentence *I detect the smelly wumpus near me.*

In one sense, dependency grammar and phrase structure grammar are just notational variants. If the phrase structure tree is annotated with the head of each phrase, you can recover the dependency tree from it. In the other direction, we can convert a dependency tree into a phrase structure tree by introducing arbitrary categories (although we might not always get a natural-looking tree this way).

Therefore we wouldn't prefer one notation over the other because one is more powerful; rather we would prefer one because it is more natural—either more familiar for the human developers of a system, or more natural for a machine learning system which will have to learn the structures. In general, phrase structure trees are natural for languages (like English) with mostly fixed word order; dependency trees are natural for languages (such as Latin) with mostly free word order, where the order of words is determined more by pragmatics than by syntactic categories.

The popularity of dependency grammar today stems in large part from the Universal Dependencies project (Nivre *et al.*, 2016), an open-source treebank project that defines a set of relations and provides millions of parsed sentences in over 70 languages.

24.3.2 Learning a parser from examples

Building a grammar for a significant portion of English is laborious and error prone. This suggests that it would be better to **learn** the grammar rules (and probabilities) rather than writing them down by hand. To apply supervised learning, we need input/output pairs of sentences and their parse trees. The Penn Treebank is the best known source of such data, with over 100 thousand sentences annotated with parse-tree structure. Figure 24.8 shows an annotated tree from the Penn Treebank.

[[*S* [*NP*-2 **Her eyes**]
[*VP* **were**
[*VP* **glazed**
[*NP* *-2]
[*SBAR-ADV* **as if**
[*S* [*NP* **she**]
[*VP* **did n't**
[*VP* [*VP* **hear** [*NP* *-1]]
or
[*VP* [*ADVP* **even**] **see** [*NP* *-1]]
[*NP*-1 **him**]]]]]]]]
.]

Figure 24.8 Annotated tree for the sentence "Her eyes were glazed as if she didn't hear or even see him." from the Penn Treebank. Note a grammatical phenomenon we have not covered yet: the movement of a phrase from one part of the tree to another. This tree analyzes the phrase "hear or even see him" as consisting of two constituent *VP*s, [*VP* **hear** [*NP* *-1]] and [*VP* *[ADVP* **even**] **see** [*NP* *-1]], both of which have a missing object, denoted *-1, which refers to the NP labeled elsewhere in the tree as [*NP*-1 **him**]. Similarly, the [*NP* *-2] refers to the [*NP*-2 **Her eyes**].

Given a treebank, we can create a PCFG just by counting the number of times each nodetype appears in a tree (with the usual caveats about smoothing low counts). In Figure 24.8, there are two nodes of the form [*S*[*NP*...] [*VP*...]]. We would count these, and all the other subtrees with root

S in the corpus. If there are 1000 S nodes of which 600 are of this form, then we create the rule:

$$S \rightarrow NP\,VP\,[0.6].$$

All together, the Penn Treebank has over 10,000 different node types. This reflects the fact that English is a complex language, but it also indicates that the annotators who created the treebank favored flat trees, perhaps flatter than we would like. For example, the phrase “the good and the bad” is parsed as a single noun phrase rather than as two conjoined noun phrases, giving us the rule:

$$NP \rightarrow Article\ Noun\ Conjunction\ Article\ Noun.$$

There are hundreds of similar rules that define a noun phrase as a string of categories with a conjunction somewhere in the middle; a more concise grammar could capture all the conjoined noun phrase rules with the single rule

$$NP \rightarrow NP\ Conjunction\ NP.$$

Bod *et al.* (2003) and Bod (2008) show how to automatically recover generalized rules like this, greatly reducing the number of rules that come out of the treebank, and creating a grammar that ends up generalizing better for previously unseen sentences. They call their approach **data-oriented parsing**.

We have seen that treebanks are not perfect—they contain errors, and have idiosyncratic parses. It is also clear that creating a treebank requires a lot of hard work; that means that treebanks will remain relatively small in size, compared to all the text that has not been annotated with trees. An alternative approach is **unsupervised parsing,** in which we learn a new grammar (or improve an existing grammar) using a corpus of sentences without trees.

The **inside–outside algorithm** (Dodd, 1988), which we will not cover here, learns to estimate the probabilities in a PCFG from example sentences without trees, similar to the way the forward-backward algorithm (Figure 14.4) estimates probabilities. Spitkovsky *et al.*(2010a) describe an unsupervised learning approach that uses **curriculum learning:** start with the easy part of the curriculum—short unambiguous 2-word sentences like "He left" can be easily parsed based on prior knowledge or annotations. Each new parse of a short sentence extends the system's knowledge so that it can eventually tackle 3-word, then 4-word, and eventually 40-word sentences.

We can also use **semisupervised parsing**, in which we start with a small number of trees as data to build an initial grammar, then add a large number of unparsed sentences to improve the grammar. The semisupervised approach can make use of **partial bracketing**: we can use widely available text that has been marked up by the authors, not by linguistic experts, with a partial tree-like structure, in the form of HTML or similar annotations. In HTML text most brackets correspond to a syntactic component, so partial bracketing can help learn a grammar (Pereira and Schabes, 1992; Spitkovsky *et al.*, 2010b). Consider this HTML text from a newspaper article:

```
In 1998, however, as I <a>established in
<i>The New Republic</i></a> and Bill Clinton just
<a>confirmed in his memoirs</a>, Netanyahu
changed his mind
```

The words surrounded by `<i></i>` tags form a noun phrase, and the two strings of words surrounded by `<a></a>` tags each form verb phrases.

24.4 Augmented Grammars

So far we have dealt with **context-free grammars**. But not every *NP* can appear in every context with equal probability. The sentence “I ate a banana” is fine, but “Me ate a banana” is ungrammatical, and “I ate a bandanna” is unlikely.

The issue is that our grammar is focused on lexical categories, like *Pronoun*, but while “I” and “me” are both pronouns, only “I” can be the subject of a sentence. Similarly, “banana” and “bandanna” are both nouns, but the former is much more likely to be object of “ate”. Linguists say that the pronoun “I” is in the subjective case (i.e., is the subject of a verb) and “me” is in the objective case[3] (i.e., is the object of a verb). They also say that “I” is in the first person (“you” is second person, and “she” is third person) and is singular (“we” is plural). A category like *Pronoun* that has been augmented with features like “subjective case, first person singular” is called a **subcategory**.

In this section we show how a grammar can represent this kind of knowledge to make finer-grained distinctions about which sentences are more likely. We will also show how to construct a representation of the **semantics** of a phrase, in a compositional way. All of this will be accomplished with an **augmented grammar** in which the nonterminals are not just atomic symbols like *Pronoun* or *NP*, but are structured representations. For example, the noun phrase “I” could be represented as *NP*(*Sbj*, *1S*, *Speaker*), which means “a noun phrase that is in the subjective case, first person singular, and whose meaning is the speaker of the sentence.” In contrast, “me” would be represented as *NP*(*Obj*, *1S*, *Speaker*), marking the fact that it is in the objective case.

Consider the sequence *"Noun* and *Noun* or *Noun,"* which can be parsed either as "[*Noun* and *Noun*] or *Noun,"* or as *"Noun* and [*Noun* or *Noun*]." Our context-free grammar has no way to express a preference for one parse over the other, because the rule for conjoined *NP*s, *NP* → *NP Conjunction NP*[0.05], will give the same probability to each parse. We would like a grammar that prefers the parses "[[spaghetti and meatballs] or lasagna]" and "[spaghetti and [pie or cake]]" over the alternative bracketing for each of these phrases.

A **lexicalized PCFG** is a type of augmented grammar that allows us to assign probabilities based on properties of the words in a phrase other than just the syntactic categories. The data would be very sparse indeed if the probability of, say, a 40-word sentence depended on *all* 40 words—this is the same problem we noted with n-grams. To simplify, we introduce the notion of the **head** of a phrase—the most important word. Thus, "banana" is the head of the *NP* "a banana" and "ate" is the head of the *VP* "ate a banana." The notation *VP*(*v*) denotes a phrase with category *VP* whose head word is *v*. Here is a lexicalized PCFG:

$$
\begin{array}{ll}
VP(v) \rightarrow Verb(v)\ NP(n) & [P_1(v,n)] \\
VP(v) \rightarrow Verb(v) & [P_2(v)] \\
NP(n) \rightarrow Article(a)\ Adjs(j)\ Noun(n) & [P_3(n,a)] \\
NP(n) \rightarrow NP(n)\ Conjunction(c)\ NP(m) & [P_4(n,c,m)] \\
Verb(\mathbf{ate}) \rightarrow \mathbf{ate} & [0.002] \\
Noun(\mathbf{banana}) \rightarrow \mathbf{banana} & [0.0007]
\end{array}
$$

Here $P_1(v, n)$ means the probability of a *VP* headed by *v* joining with an *NP* headed by *n* to form a *VP*. We can specify that "ate a banana" is more probable than "ate a bandanna" by ensuring that P_1(*ate, banana*) > P_1(*ate, bandanna*). Note that since we are considering only phrase heads, the distinction between "ate a banana" and "ate a rancid banana" will not be caught by P_1. Conceptually, P_1 is a huge table of probabilities: if there are 5,000 verbs and 10,000 nouns in the vocabulary, then P_1 requires 50 million entries, but most of them will not be stored explicitly; rather they will be derived from smoothing and backoff. For example, we can back off from P_1 (v, n) to a model that depends only on v. Such a model would require 10,000 times fewer parameters, but can still capture important regularities, such as the fact that a transitive verb like "ate" is more likely to be followed by an *NP* (regardless of the head) than an intransitive verb like "sleep."

We saw in Section 24.2 that the simple grammar for ξ_0 overgenerates, producing nonsentences such as "I saw she" or "I sees her." To avoid this problem, our grammar would have to know that "her," not "she," is a valid object of "saw" (or of any other verb) and that "see," not "sees," is the form of the verb that accompanies the subject "I."

We could encode these facts completely in the probability entries, for example making P_1 (v, she) be a very small number, for all verbs v. But it is more concise and modular to augment the category *NP* with additional variables: $NP(c, pn, n)$ is used to represent a noun phrase with case c (subjective or objective), person and number pn (e.g., third person singular), and head noun n. Figure 24.9 shows an augmented lexicalized grammar that handles these additional variables. Let's consider one grammar rule in detail:

$$
\begin{array}{rcl}
S(v) & \rightarrow & NP(Sbj, pn, n)\ VP(pn, v) \mid \ldots \\
NP(c, pn, n) & \rightarrow & Pronoun(c, pn, n) \mid Noun(c, pn, n) \mid \ldots \\
VP(pn, v) & \rightarrow & Verb(pn, v)\ NP(Obj, pn, n) \mid \ldots \\
PP(head) & \rightarrow & Prep(head)\ NP(Obj, pn, h) \\
Pronoun(Sbj, 1S, \mathbf{I}) & \rightarrow & \mathbf{I} \\
Pronoun(Sbj, 1P, \mathbf{we}) & \rightarrow & \mathbf{we} \\
Pronoun(Obj, 1S, \mathbf{me}) & \rightarrow & \mathbf{me} \\
Pronoun(Obj, 3P, \mathbf{them}) & \rightarrow & \mathbf{them} \\
 & & \\
Verb(3S, \mathbf{see}) & \rightarrow & \mathbf{see}
\end{array}
$$

Figure 24.9 Part of an augmented grammar that handles case agreement, subject–verb agreement, and head words. Capitalized names are constants: *Sbj*, and *Obj* for subjective and objective case; *1S* for first person singular; *1P* and *3P* for first and third person plural. As usual, lowercase names are variables. For simplicity, the probabilities have been omitted.

$$S(v) \rightarrow NP(Sbj, pn, n)\ VP(pn, v)\ [P_5(n, v)].$$

This rule says that when an *NP* is followed by a *VP* they can form an *S*, but only if the *NP* has the subjective (*Sbj*) case and the person and number (*pn*) of the *NP* and *VP* are identical. (We say that they are *in agreement*.) If that holds, then we have an *S* whose head is the verb from the *VP*. Here is an example lexical rule,

$$Pronoun(Sbj, IS, I) \rightarrow \mathbf{I}\ [0.005]$$

which says that “I” is a *Pronoun* in the subjective case, first-person singular, with head “I.”

24.4.1 Semantic interpretation

To show how to add semantics to a grammar, we start with an example that is simpler than English: the semantics of arithmetic expressions. Figure 24.10 shows a grammar for arithmetic expressions, where each rule is augmented with a single argument indicating the semantic interpretation of the phrase. The semantics of a digit such as "3" is the digit itself. The semantics of the expression "3 + 4" is the operator "+" applied to the semantics of the phrases "3" and "4." The grammar rules obey the principle of **compositional semantics**—the semantics of a phrase is a function of the semantics of the subphrases. Figure 24.11 shows the parse tree for 3 + (4 ÷ 2) according to this grammar. The root of the parse tree is *Exp*(*5*), an expression whose semantic interpretation is 5.

$$
\begin{array}{l}
Exp(op(x_1,x_2)) \rightarrow Exp(x_1)\ Operator(op)\ Exp(x_2) \\
Exp(x) \rightarrow \mathbf{(}\ Exp(x)\ \mathbf{)} \\
Exp(x) \rightarrow Number(x) \\
Number(x) \rightarrow Digit(x) \\
Number(10 \times x_1 + x_2) \rightarrow Number(x_1)\ Digit(x_2) \\
Operator(+) \rightarrow \mathbf{+} \\
Operator(-) \rightarrow \mathbf{-} \\
Operator(\times) \rightarrow \times \\
Operator(\div) \rightarrow \div \\
Digit(0) \rightarrow \mathbf{0} \\
Digit(1) \rightarrow \mathbf{1} \\
\ldots
\end{array}
$$

Figure 24.10 A grammar for arithmetic expressions, augmented with semantics. Each variable x_i represents the semantics of a constituent.

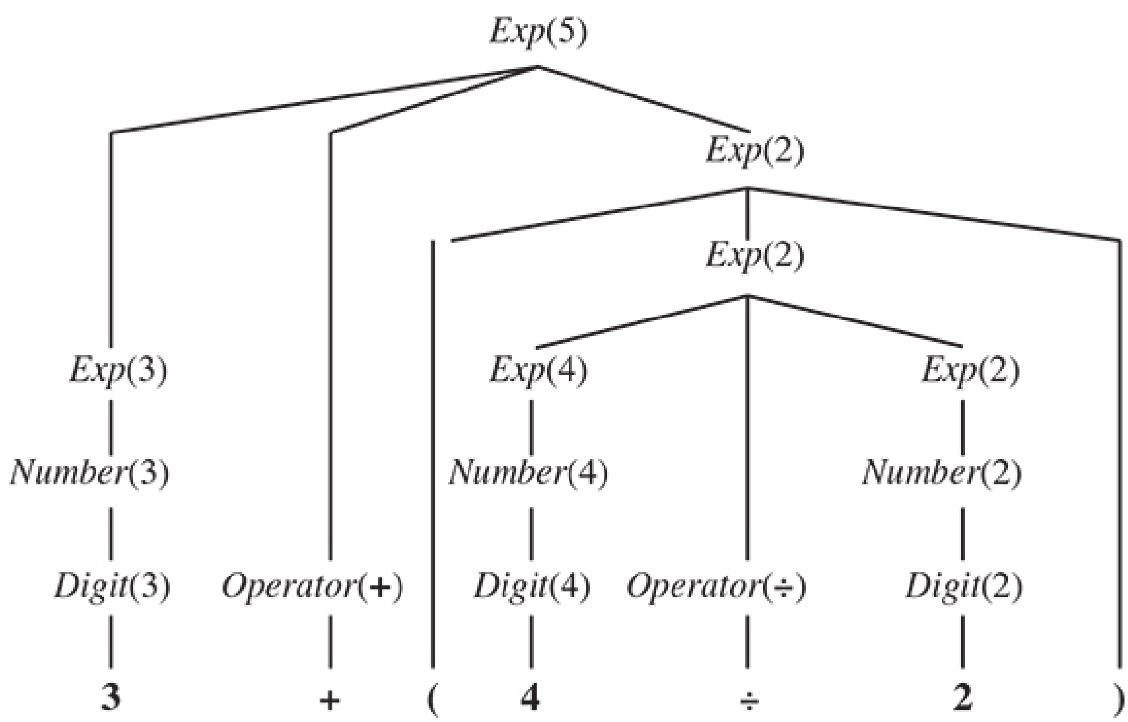


Figure 24.11 Parse tree with semantic interpretations for the string “3 + (4 ÷ 2)”.

Now let’s move on to the semantics of English, or at least a tiny portion of it. We will use first-order logic for our semantic representation. So the simple sentence “Ali loves Bo” should get the semantic representation *Loves*(*Ali*, *Bo*). But what about the constituent phrases? We can represent the *NP* “Ali” with the logical term *Ali*. But the *VP* “loves Bo” is neither a logical term nor a complete logical sentence. Intuitively, “loves Bo” is a description that might or might not apply to a particular person. (In this case, it applies to Ali.) This means that “loves Bo” is a **predicate** that, when combined with a term that represents a person, yields a complete logical sentence.

Using the λ-notation (see page 277), we can represent “loves Bo” as the predicate

$$\lambda x\, Loves(x{,}Bo).$$

Now we need a rule that says “an *NP* with semantics *n* followed by a *VP* with semantics *pred* yields a sentence whose semantics is the result of applying *pred* to n:”

$$S(pred(n)) \rightarrow NP(n)\; VP(pred)\ .$$

The rule tells us that the semantic interpretation of “Ali loves Bo” is
$(\lambda x\, Loves(x{,}Bo))(Ali)$,
which is equivalent to *Loves* (*Ali, Bo*). Technically, we say that this is a β-reduction of the lambda function application.

The rest of the semantics follows in a straightforward way from the choices we have made so far. Because *VP*s are represented as predicates, verbs should be predicates as well. The verb “loves” is represented as λ*y* λ*x* *Loves*(*x, y*), the predicate that, when given the argument *Bo,* returns the predicate λ*x* *Loves*(*x, Bo*). We end up with the grammar and parse tree shown in Figure 24.12. In a more complete grammar, we would put all the augmentations (semantics, case, person-number, and head) together into one set of rules. Here we show only the semantic augmentation to make it clearer how the rules work.

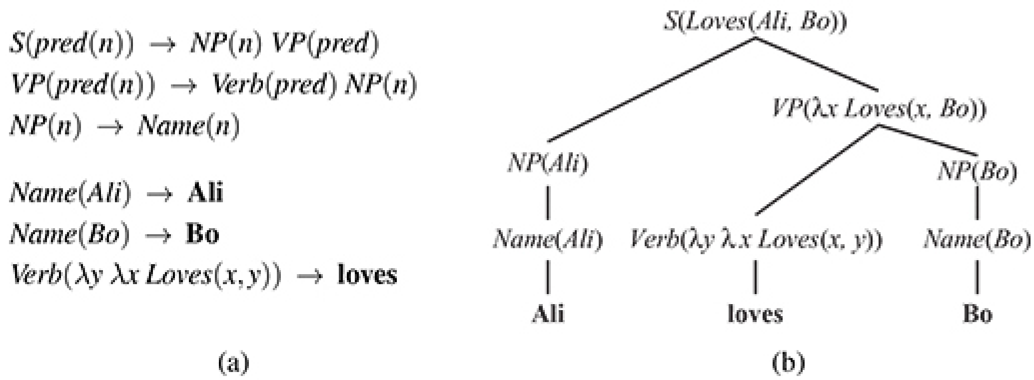


Figure 24.12 (a) A grammar that can derive a parse tree and semantic interpretation for "Ali loves Bo" (and three other sentences). Each category is augmented with a single argument representing the semantics. (b) A parse tree with semantic interpretations for the string "Ali loves Bo."

24.4.2 Learning semantic grammars

Unfortunately, the Penn Treebank does not include semantic representations of its sentences, just syntactic trees. So if we are going to learn a semantic grammar, we will need a different source of examples. Zettlemoyer and Collins (2005) describe a system that learns a grammar for a question-answering system from examples that consist of a sentence paired with the semantic form for the sentence:

- **Sentence:** What states border Texas?
- **Logical Form:** $\lambda x.state(x) \wedge \lambda x.borders(x, Texas)$

Given a large collection of pairs like this and a little bit of hand-coded knowledge for each new domain, the system generates plausible lexical entries (for example, that “Texas” and “state” are nouns such that *state*(*Texas*) is true), and simultaneously learns parameters for a grammar that allows the system to parse sentences into semantic representations. Zettlemoyer and Collins’s system achieved 79% accuracy on two different test sets of unseen sentences. Zhao and Huang (2015) demonstrate a shift-reduce parser that runs faster, and achieves 85% to 89% accuracy.

A limitation of these systems is that the training data includes logical forms. These are expensive to create, requiring human annotators with specialized expertise—not everyone understands the subtleties of lambda calculus and predicate logic. It is much easier to gather examples of question/answer pairs:

- **Question:** What states border Texas?
- **Answer** : Louisiana, Arkansas, Oklahoma, New Mexico.

- **Question:** How many times would Rhode Island fit into California?
- **Answer:** 135

Such question/answer pairs are quite common on the Web, so a large database can be put together without human experts. Using this large source of data it is possible to build parsers that outperform those that use a small database of annotated logical forms (Liang *et al*., 2011; Liang and Potts, 2015). The key approach described in these papers is to invent an internal logical form that is compositional but does not allow an exponentially large search space.

24.5 Complications of Real Natural Language

The grammar of real English is endlessly complex (and other languages are equally complex). We will briefly mention some of the topics that contribute to this complexity.

Quantification: Consider the sentence “Every agent feels a breeze.” The sentence has only one syntactic parse under ξ_0, but it is semantically ambiguous: is there one breeze that is felt by all the agents, or does each agent feel a separate personal breeze? The two interpretations can be represented as

$$\begin{aligned}&\forall a \;\; a \in \text{Agents} \Rightarrow \\ &\qquad \exists b \, b \in Breezes \land Feel(a,b); \\ &\exists b \;\; b \in Breezes \land \forall a \;\; a \in Agents \Rightarrow \\ &\qquad Feel(a,b).\end{aligned}$$

One standard approach to quantification is for the grammar to define not an actual logical semantic sentence, but rather a **quasi-logical form** that is then turned into a logical sentence by algorithms outside of the parsing process. Those algorithms can have preference rules for choosing one quantifier scope over another—preferences that need not be reflected directly in the grammar.

Pragmatics: We have shown how an agent can perceive a string of words and use a grammar to derive a set of possible semantic interpretations. Now we address the problem of completing the interpretation by adding context-dependent information about the current situation. The most obvious need for pragmatic information is in resolving the meaning of **indexicals**, which are phrases that refer directly to the current situation. For example, in the sentence “I am in Boston today,” both “I” and “today” are indexicals. The word “I” would be represented by *Speaker*, a fluent that refers to different objects at different times, and it would be up to the hearer to resolve the referent of the fluent—that is not considered part of the grammar but rather an issue of pragmatics.

Another part of pragmatics is interpreting the speaker's intent. The speaker's utterance is considered a **speech act**, and it is up to the hearer to decipher what type of action it is—a question, a statement, a promise, a warning, a command, and so on. A command such as "go to 2 2" implicitly refers to the hearer. So far, our grammar for *S* covers only declarative sentences. We can extend it to cover commands—a command is a verb phrase where the subject is implicitly the hearer of the command:

$$S(Command(pred(Hearer))) \;\rightarrow\; VP(pred)\ .$$

Long-distance dependencies: In Figure 24.8 we saw that "she didn't hear or even see him" was parsed with two gaps where an *NP* is missing, but refers to the *NP* "him." We can use the symbol ␣ to represent the gaps: "she didn't [hear ␣ or even see ␣] him." In general, the distance between the gap and the *NP* it refers to can be arbitrarily long: in "Who did the agent tell you to give the gold to ␣?" the gap refers to "Who," which is 11 words away.

A complex system of augmented rules can be used to make sure that the missing *NP*s match up properly. The rules are complex; for example, you can't have a gap in one branch of an *NP* conjunction: "What did she play [*NP* Dungeons and ␣]?" is ungrammatical. But you can have the same gap in both branches of a *VP* conjunction, as in the sentence "What did you [*VP* [*VP* smell ␣] and [*VP* shoot an arrow at ␣]]?"

Time and tense: Suppose we want to represent the difference between "Ali loves Bo" and "Ali loved Bo." English uses verb tenses (past, present, and future) to indicate the relative time of an event. One good choice to represent the time of events is the event calculus notation of Section 10.3. In event calculus we have

$$\begin{array}{l} \text{Ali loves Bo}: E_1 \in Loves(Ali, Bo) \wedge During(Now, Extent(E_1)) \\ \text{Aliloved Bo}: E_2 \in Loves(Ali, Bo) \wedge After(Now, Extent(E_2)). \end{array}$$

This suggests that our two lexical rules for the words "loves" and "loved" should be these:

$$Verb(\lambda y\, \lambda x\, e \in \; Loves(x, y) \wedge During(Now, e)) \rightarrow \textbf{loves}$$
$$Verb(\lambda y\, \lambda x\, e \in \; Loves(x, y) \wedge After(Now, e)) \rightarrow \textbf{loved}.$$

Other than this change, everything else about the grammar remains the same, which is encouraging news; it suggests we are on the right track if we can so easily add a complication like the tense of verbs (although we have just scratched the surface of a complete grammar for time and tense).

Ambiguity: We tend to think of ambiguity as a failure in communication; when a listener is consciously aware of an ambiguity in an utterance, it means that the utterance is unclear or confusing. Here are some examples taken from newspaper headlines:

Squad helps dog bite victim.

Police begin campaign to run down jaywalkers.

Helicopter powered by human flies.

Once-sagging cloth diaper industry saved by full dumps.

Include your children when baking cookies.

Portable toilet bombed; police have nothing to go on.

Milk drinkers are turning to powder.

Two sisters reunited after 18 years in checkout counter.

Such confusions are the exception; most of the time the language we hear seems unambiguous. Thus, when researchers first began to use computers to analyze language in the 1960s, they were quite surprised to learn that almost every sentence is ambiguous, with multiple possible parses (sometimes hundreds), even when the single preferred parse is the only one that native speakers notice. For example, we understand the phrase “brown rice and black beans” as “[brown rice] and [black beans],” and never consider the low-probability interpretation “brown [rice and black beans],” where the adjective “brown” is modifying the whole phrase, not just the “rice.” When we hear “Outside of a dog, a book is a person’s best friend,” we interpret “outside of” as meaning “except for,” and find it funny when the next sentence of the Groucho Marx joke is “Inside of a dog it’s too dark to read.”

Lexical ambiguity is when a word has more than one meaning: “back” can be an adverb (go back), an adjective (back door), a noun (the back of the room), a verb (back a candidate), or a proper noun (a river in Nunavut, Canada). “Jack” can be a proper name, a noun (a playing card, a six-pointed metal game piece, a nautical flag, a fish, a bird, a cheese, a socket, etc.), or a verb (to jack up a car, to hunt with a light, or to hit a baseball hard). **Syntactic ambiguity** refers to a phrase that has multiple parses: “I smelled a wumpus in 2,2” has two parses: one where the prepositional phrase “in 2,2” modifies the noun and one where it modifies the verb. The syntactic ambiguity leads to a **semantic ambiguity**, because one parse means that the wumpus is in 2,2 and the other means that a stench is in 2,2. In this case, getting the wrong interpretation could be a deadly mistake.

There can also be ambiguity between literal and figurative meanings. Figures of speech are important in poetry, and are common in everyday speech as well. A **metonymy** is a figure of speech in which one object is used to stand for another. When we hear “Chrysler announced a new model,” we do not interpret it as saying that companies can talk; rather we understand that a spokesperson for the company made the announcement. Metonymy is common and is often interpreted unconsciously by human hearers.

Unfortunately, our grammar as it is written is not so facile. To handle the semantics of metonymy properly, we need to introduce a whole new level of ambiguity. We could do this by providing *two* objects for the semantic interpretation of every phrase in the sentence: one for the object that the phrase literally refers to (Chrysler) and one for the metonymic reference (the spokesperson). We then have to say that there is a relation between the two. In our current grammar, “Chrysler announced” gets interpreted as

$$x = Chrysler \wedge e \in Announce(x) \wedge After(Now, Extent(e)).$$

We need to change that to

$$\begin{aligned} x = Chrysler \wedge e \in Announce(m) \wedge After(Now, Extent(e)) \\ \wedge Metonymy(m, x). \end{aligned}$$

This says that there is one entity *x* that is equal to Chrysler, and another entity *m* that did the announcing, and that the two are in a metonymy relation. The next step is to define what kinds of metonymy relations can occur. The simplest case is when there is no metonymy at all—the literal object *x* and the metonymic object *m* are identical:

$$\forall m, x\ (m = x) \Rightarrow Metonymy(m, x).$$

For the Chrysler example, a reasonable generalization is that an organization can be used to stand for a spokesperson of that organization:

$$\forall m, x\ x \in Organizations \land Spokesperson(m, x)\ \Rightarrow\ Metonymy(m, x)\ .$$

Other metonymies include the author for the works (I read *Shakespeare*) or more generally the producer for the product (I drive a *Honda*) and the part for the whole (The Red Sox need a strong *arm*). Some examples of metonymy, such as "The *ham sandwich* on Table 4 wants another beer," are more novel and are interpreted with respect to a situation (such as waiting on tables and not knowing a customer's name).

A **metaphor** is another figure of speech, in which a phrase with one literal meaning is used to suggest a different meaning by way of an analogy. Thus, metaphor can be seen as a kind of metonymy where the relation is one of similarity.

Disambiguation is the process of recovering the most probable intended meaning of an utterance. In one sense we already have a framework for solving this problem: each rule has a probability associated with it, so the probability of an interpretation is the product of the probabilities of the rules that led to the interpretation. Unfortunately, the probabilities reflect how common the phrases are in the corpus from which the grammar was learned, and thus reflect general knowledge, not specific knowledge of the current situation. To do disambiguation properly, we need to combine four models:

1. The **world model:** the likelihood that a proposition occurs in the world. Given what we know about the world, it is more likely that a speaker who

says “I’m dead” means “I am in big trouble” or “I lost this video game” rather than “My life ended, and yet I can still talk.”

2. The **mental model**: the likelihood that the speaker forms the intention of communicating a certain fact to the hearer. This approach combines models of what the speaker believes, what the speaker believes the hearer believes, and so on. For example, when a politician says, “I am not a crook,” the world model might assign a probability of only 50% to the proposition that the politician is not a criminal, and 99.999% to the proposition that he is not a hooked shepherd’s staff. Nevertheless, we select the former interpretation because it is a more likely thing to say.
3. The **language model**: the likelihood that a certain string of words will be chosen, given that the speaker has the intention of communicating a certain fact.
4. The **acoustic model**: for spoken communication, the likelihood that a particular sequence of sounds will be generated, given that the speaker has chosen a given string of words. (For handwritten or typed communication, we have the problem of optical character recognition.)

24.6 Natural Language Tasks

Natural language processing is a big field, deserving an entire textbook or two of its own (Goldberg, 2017; Jurafsky and Martin, 2020). In this section we briefly describe some of the main tasks; you can use the references to get more details.

Speech recognition is the task of transforming spoken sound into text. We can then perform further tasks (such as question answering) on the resulting text. Current systems have a word error rate of about 3% to 5% (depending on details of the test set), similar to human transcribers. The challenge for a system using speech recognition is to respond appropriately even when there are errors on individual words.

Top systems today use a combination of recurrent neural networks and hidden Markov models (Hinton *et al.*, 2012; Yu and Deng, 2016; Deng, 2016; Chiu *et al.*, 2017; Zhang *et al.*,2017). The introduction of deep neural nets for speech in 2011 led to an immediate and dramatic improvement of about 30% in error rate—this from a field that seemed to be mature and was previously progressing at only a few percent per year. Deep neural networks are a good fit because the problem of speech recognition has a natural compositional breakdown: waveforms to phonemes to words to sentences. They will be covered in the next chapter.

Text-to-speech synthesis is the inverse process—going from text to sound. Taylor (2009) gives a book-length overview. The challenge is to pronounce each word correctly, and to make the flow of each sentence seem natural, with the right pauses and emphasis.

Another area of development is in synthesizing different voices—starting with a choice between a generic male or female voice, then allowing for regional dialects, and even imitating celebrity voices. As with speech recognition, the introduction of deep recurrent neural networks led to a large improvement, with about 2/3 of listeners saying that the neural WaveNet system (van den Oord *et al.*, 2016a) sounded more natural than the previous nonneural system.

Machine translation transforms text in one language to another. Systems are usually trained using a bilingual corpus: a set of paired documents, where one member of the pair is in, say, English, and the other is in, say, French. The documents do not need to be annotated in any way; the machine translation system learns to align sentences and phrases and then when presented with a novel sentence in one language, can generate a translation to the other.

Systems in the early 2000s used *n*-gram models, and achieved results that were usually good enough to get across the meaning of a text, but contained syntactic errors in most sentences. One problem was the limit on the length of the *n*-grams: even with a large limit of 7, it was difficult for information to flow from one end of the sentence to the other. Another problem was that all the information in an *n*-gram model is at the level of individual words. Such a system could learn that “black cat” translates to “chat noir,” but it could not learn the rule that adjectives generally come before the noun in English and after the noun in French.

Recurrent neural sequence-to-sequence models (Sutskever *et al.*, 2015) got around the problem. They could generalize better (because they could use word embeddings rather than *n*-gram counts of specific words) and could form compositional models throughout the various levels of the deep network to effectively pass information along. Subsequent work using the attention-focusing mechanism of the transformer model (Vaswani *et al.*, 2018) increased performance further, and a hybrid model incorporating aspects of both these models does better still, approaching human-level performance on some language pairs (Wu *et al.,* 2016b; Chen *et al.,* 2018).

Information extraction is the process of acquiring knowledge by skimming a text and looking for occurrences of particular classes of objects and for relationships among them. A typical task is to extract instances of addresses from Web pages, with database fields for street, city, state, and zip code; or instances of storms from weather reports, with fields for temperature, wind speed, and precipitation. If the source text is well structured (for example, in the form of a table), then simple techniques such as regular expressions can extract the information (Cafarella *et al.,* 2008). It gets harder if we are trying to extract *all* facts, rather than a specific type (such as weather reports); Banko *et al.* (2007) describe the TEXTRUNNER system that performs extraction over an open, expanding set of relations. For free-form text, techniques include hidden Markov models and rule-based learning systems (as used in TEXTRUNNER and NELL (Never-Ending Language Learning) (Mitchell *et al.,*2018)). More recent systems use recurrent neural networks, taking advantage of the flexibility of word embeddings. You can find an overview in Kumar (2017).

Information retrieval is the task of finding documents that are relevant and important for a given query. Internet search engines such as Google and Baidu perform this task billions of times a day. Three good textbooks on the subject are Manning *et al.* (2008), Croft *et al.*(2010), and Baeza-Yates and Ribeiro-Neto (2011).

Question Answering is a different task, in which the query really is a question, such as “Who founded the U.S. Coast Guard?” and the response is not a ranked list of documents but rather an actual answer: “Alexander Hamilton.” There have been question-answering systems since the 1960s that rely on syntactic parsing as discussed in this chapter, but only since

2001 have such systems used Web information retrieval to radically increase their breadth of coverage. Katz (1997) describes the START parser and question answerer. Banko *et al.* (2002) describe ASKMSR, which was less sophisticated in terms of its syntactic parsing ability, but more aggressive in using Web search and sorting through the results. For example, to answer "Who founded the U.S. Coast Guard?" it would search for queries such as [* founded the U.S. Coast Guard] and [the U.S. Coast Guard was founded by *], and then examine the multiple resulting Web pages to pick out a likely response, knowing that the query word "who" suggests that the answer should be a person. The Text REtrieval Conference (TREC) gathers research on this topic and has hosted competitions on an annual basis since 1991 (Allan *et al.*, 2017). Recently we have seen other test sets, such as the AI2 ARC test set of basic science questions (Clark *et al.*, 2018).

Summary

The main points of this chapter are as follows:

- Probabilistic language models based on n-grams recover a surprising amount of information about a language. They can perform well on such diverse tasks as language identification, spelling correction, sentiment analysis, genre classification, and named-entity recognition.
- These language models can have millions of features, so preprocessing and smoothing the data to reduce noise is important.
- In building a statistical language system, it is best to devise a model that can make good use of available **data**, even if the model seems overly simplistic.
- Word embeddings can give a richer representation of words and their similarities.
- To capture the hierarchical structure of language, **phrase structure** grammars (and in particular, **context-free** grammars) are useful. The probabilistic context-free grammar (PCFG) formalism is widely used, as is the dependency grammar formalism.
- Sentences in a context-free language can be parsed in $O(n^3)$ time by a **chart parser** such as the **CYK algorithm**, which requires grammar rules to be in **Chomsky Normal Form**. With a small loss in accuracy, natural languages can be parsed in $O(n)$ time, using a beam search or a shift-reduce parser.
- A **treebank** can be a resource for learning a PCFG grammar with parameters.
- It is convenient to **augment** a grammar to handle issues such as subject–verb agreement and pronoun case, and to represent

information at the level of words rather than just at the level of categories.

- **Semantic interpretation** can also be handled by an augmented grammar. We can learn a semantic grammar from a corpus of questions paired either with the logical form of the question, or with the answer.
- Natural language is complex and difficult to capture in a formal grammar.

Bibliographical and Historical Notes

N-gram letter models for language modeling were proposed by Markov (1913). Claude Shannon (Shannon and Weaver, 1949) was the first to generate *n*-gram word models of English. The **bag-of-words model** gets its name from a passage from linguist Zellig Harris (1954), “language is not merely a bag of words but a tool with particular properties.” Norvig (2009) gives some examples of tasks that can be accomplished with *n*-gram models.

Chomsky (1956, 1957) pointed out the limitations of finite-state models compared with context-free models, concluding, “Probabilistic models give no particular insight into some of the basic problems of syntactic structure.” This is true, but probabilistic models *do* provide insight into some *other* basic problems—problems that context-free models ignore. Chomsky’s remarks had the unfortunate effect of scaring many people away from statistical models for two decades, until these models reemerged for use in the field of speech recognition (Jelinek, 1976), and in cognitive science, where **optimality theory** (Smolensky and Prince, 1993; Kager, 1999) posited that language works by finding the most probable candidate that optimally satisfies competing constraints.

Add-one smoothing, first suggested by Pierre-Simon Laplace (1816), was formalized by Jeffreys (1948). Other smoothing techniques include interpolation smoothing (Jelinek and Mercer, 1980), Witten–Bell smoothing (1991), Good–Turing smoothing (Church and Gale, 1991), Kneser–Ney smoothing (1995, 2004), and stupid backoff (Brants *et al.*, 2007). Chen and Goodman (1996) and Goodman (2001) survey smoothing techniques.

Simple *n*-gram letter and word models are not the only possible probabilistic models. The **latent Dirichlet allocation** model (Blei *et al.*, 2002; Hoffman *et al.*, 2011) is a probabilistic text model that views a document as a mixture of topics, each with its own distribution of words. This model can be seen as an extension and rationalization of the **latent semantic indexing** model of Deerwester *et al.* (1990) and is also related to the multiple-cause mixture model of (Sahami *et al.*, 1996). And of course there is great interest in non-probabilistic language models, such as the deep learning models covered in Chapter 25.

Joulin *et al.* (2016) give a bag of tricks for efficient text classification. Joachims (2001) uses statistical learning theory and support vector machines to give a theoretical analysis of when classification will be successful. Apté *et al.* (1994) report an accuracy of 96% in classifying Reuters news articles into the “Earnings” category. Koller and Sahami (1997) report accuracy up to 95% with a naive Bayes classifier, and up to 98.6% with a Bayes classifier.

Schapire and Singer (2000) show that simple linear classifiers can often achieve accuracy almost as good as more complex models, and run faster. Zhang *et al.* (2016) describe a character-level (rather than word-level) text classifier. Witten *et al.* (1999) describe compression algorithms for classification, and show the deep connection between the LZW compression algorithm and maximum-entropy language models.

Wordnet (Fellbaum, 2001) is a publicly available dictionary of about 100,000 words and phrases, categorized into parts of speech and linked by semantic relations such as synonym, antonym, and part-of. Charniak (1996) and Klein and Manning (2001) discuss parsing with treebank grammars. The British National Corpus (Leech *et al.*, 2001) contains 100 million words, and the World Wide Web contains several trillion words; Franz and

Brants (2006) describe the publicly available Google n-gram corpus of 13 million unique words from a trillion words of Web text. Buck *et al.* (2014) describe a similar data set from the Common Crawl project. The Penn Treebank (Marcus *et al.*, 1993; Bies *et al.*, 2015) provides parse trees for a 3-million-word corpus of English.

Many of the *n*-gram model techniques are also used in bioinformatics problems. Biostatistics and probabilistic NLP are coming closer together, as each deals with long, structured sequences chosen from an alphabet.

Early part-of-speech (POS) taggers used a variety of techniques, including rule sets (Brill, 1992), *n*-grams (Church, 1988), decision trees (Màrquez and Rodríguez, 1998), HMMs (Brants, 2000), and logistic regression (Ratnaparkhi, 1996). Historically, a logistic regression model was also called a “maximum entropy Markov model” or MEMM, so some work is under that name. Jurafsky and Martin (2020) have a good chapter on POS tagging. Ng and Jordan (2002) compare discriminative and generative models for classification tasks.

Like semantic networks, context-free grammars were first discovered by ancient Indian grammarians (especially Panini, ca. 350 BCE) studying Shastric Sanskrit (Ingerman, 1967). They were reinvented by Noam Chomsky (1956) for the analysis of English and independently by John Backus (1959) and Peter Naur for the analysis of Algol-58.

Probabilistic context-free grammars were first investigated by Booth (1969) and Salomaa (1969). Algorithms for PCFGs are presented in the excellent short monograph by Charniak (1993) and the excellent long textbooks by Manning and Schütze (1999) and Jurafsky and Martin (2020). Baker (1979) introduces the inside–outside algorithm for learning a PCFG. **Lexicalized PCFGs** (Charniak, 1997; Hwa, 1998) combine the best aspects of PCFGs and *n*-gram models. Collins (1999) describes PCFG parsing that is lexicalized with head features, and Johnson (1998) shows how the accuracy of a PCFG depends on the structure of the treebank from which its probabilities were learned.

There have been many attempts to write formal grammars of natural languages, both in "pure" linguistics and in computational linguistics. There are several comprehensive but informal grammars of English (Quirk *et al.*, 1985; McCawley, 1988; Huddleston and Pullum, 2002). Since the 1980s, there has been a trend toward lexicalization: putting more information in the lexicon and less in the grammar.

Lexical-functional grammar, or LFG (Bresnan, 1982) was the first major grammar formalism to be highly lexicalized. If we carry lexicalization to an extreme, we end up with **categorial grammar** (Clark and Curran, 2004), in which there can be as few as two grammar rules, or with **dependency grammar** (Smith and Eisner, 2008; Kübler *et al.*, 2009) in which there are no syntactic categories, only relations between words.

Computerized parsing was first demonstrated by Yngve (1955). Efficient algorithms were developed in the 1960s, with a few twists since then (Kasami, 1965; Younger, 1967; Earley, 1970; Graham *et al.*, 1980). Church and Patil (1982) describe syntactic ambiguity and address ways to resolve it.

Klein and Manning (2003) describe A* parsing, and Pauls and Klein (2009) extend that to K-best A* parsing, in which the result is not a single parse but the K best. Goldberg *et al.*(2013) describe the necessary implementation tricks to make sure that a beam search parser is $O(n)$ and not $O(n^2)$. Zhu *et al.* (2013) describe a fast deterministic shift-reduce parser for natural languages, and Sagae and Lavie (2006) show how adding search to a shift-reduce parser can make it more accurate, at the cost of some speed.

Today, highly accurate open-source parsers include Google's Parsey McParseface (Andor *et al.*, 2016), the Stanford Parser (Chen and Manning, 2014), the Berkeley Parser (Kitaev and Klein, 2018), and the SPACY parser. They all do generalization through neural networks and achieve roughly 95% accuracy on Wall Street Journal or Penn Treebank test sets. There is some criticism of the field that it is focusing too narrowly on measuring performance on a few select corpora, and perhaps overfitting on them.

Formal semantic interpretation of natural languages originates within philosophy and formal logic, particularly Alfred Tarski's (1935) work on the semantics of formal languages. Bar-Hillel (1954) was the first to consider the problems of pragmatics (such as indexicals) and propose that they could be handled by formal logic. Richard Montague's essay "English as a formal language" (1970) is a kind of manifesto for the logical analysis of language, but there are other books that are more readable (Dowty *et al.*, 1991; Portner and Partee, 2002; Cruse, 2011). While semantic interpretation programs are designed to pick the most likely interpretation, literary critics (Empson, 1953; Hobbs, 1990) have been ambiguous about whether ambiguity is something to be resolved or cherished. Norvig (1988) discusses the problems of considering multiple simultaneous interpretations, rather than settling for a single maximum-likelihood interpretation. Lakoff and Johnson (1980) give an engaging analysis and catalog of common metaphors in English. Martin (1990) and Gibbs (2006) offer computational models of metaphor interpretation.

The first NLP system to solve an actual task was the BASEBALL question answering system (Green *et al.*, 1961), which handled questions about a database of baseball statistics. Close after that was Winograd's (1972) SHRDLU, which handled questions and commands about a blocks-world scene, and Woods's (1973) LUNAR, which answered questions about the rocks brought back from the moon by the Apollo program.

Banko *et al.* (2002) present the ASKMSR question-answering system; a similar system is due to Kwok *et al.* (2001). Pasca and Harabagiu (2001) discuss a contest-winning question-answering system.

Modern approaches to semantic interpretation usually assume that the mapping from syntax to semantics will be learned from examples (Zelle and Mooney, 1996; Zettlemoyer and Collins, 2005; Zhao and Huang, 2015). The first important result on **grammar induction** was a negative one: Gold (1967) showed that it is not possible to reliably learn an exactly correct context-free grammar, given a set of strings from that grammar. Prominent linguists, such as Chomsky (1957) and Pinker (2003), have used Gold's result to argue that there must be an innate **universal grammar** that all children have from birth. The so-called **Poverty of the Stimulus** argument says that children aren't given enough input to learn a CFG, so they must already "know" the grammar and be merely tuning some of its parameters.

While this argument continues to hold sway throughout much of Chomskyan linguistics, it has been dismissed by other linguists (Pullum, 1996; Elman *et al.*, 1997) and most computer scientists. As early as 1969, Horning showed that it *is* possible to learn, in the sense of PAC learning, a *probabilistic* context-free grammar. Since then, there have been many convincing empirical demonstrations of language learning from positive examples alone, such as learning semantic grammars with inductive logic programming (Muggleton and De Raedt, 1994; Mooney, 1999), the Ph.D. theses of Schütze (1995) and de Marcken (1996), and the entire line of modern language processing systems based on the transformer model (Section 25.4). There is an annual International Conference on Grammatical Inference (ICGI).

James Baker's DRAGON system (Baker, 1975) could be considered the first succesful speech recognition system. It was the first to use HMMs for speech. After several decades of systems based on probabilistic language models, the field began to switch to deep neural networks (Hinton *et al.*, 2012). Deng (2016) describes how the introduction of deep learning enabled rapid improvement in speech recognition, and reflects on the implications for other NLP tasks. Today deep learning is the dominant approach for all large-scale speech recognition systems. Speech recognition can be seen as the first application area that highlighted the success of deep learning, with computer vision following shortly thereafter.

Interest in the field of **information retrieval** was spurred by widespread usage of Internet searching. Croft *et al.* (2010) and Manning *et al.* (2008) provide textbooks that cover the basics. The TREC conference hosts an annual competition for IR systems and publishes proceedings with results.

Brin and Page (1998) describe the PageRank algorithm, which takes into account the links between pages, and give an overview of the implementation of a Web search engine. Silverstein *et al.* (1998) investigate a log of a billion Web searches. The journal *Information Retrieval* and the proceedings of the annual flagship *SIGIR* conference cover recent developments in the field.

Information extraction has been pushed forward by the annual Message Understanding Conferences (MUC), sponsored by the U.S. government. Surveys of template-based systems are given by Roche and Schabes (1997), Appelt (1999), and Muslea (1999). Large databases of facts were extracted by Craven *et al.* (2000), Pasca *et al.* (2006), Mitchell (2007), and Durme and Pasca (2008). Freitag and McCallum (2000) discuss HMMs for Information Extraction. Conditional random fields have also been used for this task (Lafferty *et al.*, 2001; McCallum, 2003); a tutorial with practical guidance is given by Sutton and McCallum (2007). Sarawagi (2007) gives a comprehensive survey.

Two early influential approaches to automated knowledge engineering for NLP were by Riloff (1993), who showed that an automatically constructed dictionary performed almost as well as a carefully handcrafted domain-specific dictionary, and by Yarowsky (1995), who showed that the task of word sense classification could be accomplished through unsupervised training on a corpus of unlabeled text with accuracy as good as supervised methods.

The idea of simultaneously extracting templates and examples from a handful of labeled examples was developed independently and simultaneously by Blum and Mitchell (1998), who called it **cotraining,** and by Brin (1998), who called it DIPRE (Dual Iterative Pattern Relation Extraction). You can see why the term *cotraining* has stuck. Similar early work, under the name of bootstrapping, was done by Jones *et al.* (1999). The method was advanced by the QXTRACT (Agichtein and Gravano, 2003) and KNOWITALL(Etzioni *et al.*, 2005) systems. Machine reading was introduced by Mitchell (2005) and Etzioni *et al.* (2006) and is the focus of the TEXTRUNNER project (Banko *et al.*, 2007; Banko and Etzioni, 2008).

This chapter has focused on natural language sentences, but it is also possible to do information extraction based on the physical structure or geometric layout of text rather than on the linguistic structure. Lists, tables, charts, graphs, diagrams, etc., whether encoded in HTML or accessed through the visual analysis of pdf documents, are home to data that can be extracted and consolidated (Hurst, 2000; Pinto *et al.*, 2003; Cafarella *et al.*, 2008).

Ken Church (2004) shows that natural language research has cycled between concentrating on the data (empiricism) and concentrating on theories (rationalism); he describes the advantages of having good language resources and evaluation schemes, but wonders if we have gone too far (Church and Hestness, 2019). Early linguists concentrated on actual language usage data, including frequency counts. Noam Chomsky (1956) demonstrated the limitations of finite-state models, leading to an emphasis on theoretical studies of syntax, disregarding actual language performance. This approach dominated for twenty years, until empiricism made a comeback based on the success of work in statistical speech recognition (Jelinek, 1976). Today, the emphasis on empirical language data continues, and there is heightened interest in models that consider higher-level constructs, such as syntactic and semantic relations, not just sequences of words. There is also a strong emphasis on deep learning neural network models of language, which we will cover in Chapter 25.

Work on applications of language processing is presented at the biennial Applied Natural Language Processing conference (ANLP), the conference on Empirical Methods in Natural Language Processing (EMNLP), and the journal *Natural Language Engineering*. A broad range of NLP work appears in the journal *Computational Linguistics* and its conference, ACL, and in the International Computational Linguistics (COLING) conference. Jurafsky and Martin (2020) give a comprehensive introduction to speech and NLP.

[1] And even computer vision applications: WordNet provides the set of categories used by ImageNet.

[2] Sometimes the authors are credited in the order CKY.

[3] The subjective case is also sometimes called the nominative case and the objective case is sometimes called the accusative case. Many languages also make another distinction with a dative

case for words in the indirect object position.

CHAPTER 25

DEEP LEARNING FOR NATURAL LANGUAGE PROCESSING

In which deep neural networks perform a variety of language tasks, capturing the structure of natural language as well as its fluidity.

Chapter 24 explained the key elements of natural language, including grammar and semantics. Systems based on parsing and semantic analysis have demonstrated success on many tasks, but their performance is limited by the endless complexity of linguistic phenomena in real text. Given the vast amount of text available in machine-readable form, it makes sense to consider whether approaches based on data-driven machine learning can be more effective. We explore this hypothesis using the tools provided by deep learning systems (Chapter 22).

We begin in Section 25.1 by showing how learning can be improved by representing words as points in a high-dimensional space, rather than as atomic values. Section 25.2 covers the use of recurrent neural networks to capture meaning and long-distance context as text is processed sequentially. Section 25.3 focuses primarily on machine translation, one of the major successes of deep learning applied to NLP. Sections 25.4 and 25.5 cover models that can be trained from large amounts of unlabeled text and then applied to specific tasks, often achieving state-of-the-art performance. Finally, Section 25.6 takes stock of where we are and how the field may progress.

25.1 Word Embeddings

We would like a representation of words that does not require manual feature engineering, but allows for generalization between related words—words that are related syntactically (“colorless” and “ideal” are both adjectives), semantically (“cat” and “kitten” are both felines), topically (“sunny” and “sleet” are both weather terms), in terms of sentiment (“awesome” has opposite sentiment to “cringeworthy”), or otherwise.

How should we encode a word into an input vector x for use in a neural network? As explained in Section 22.2.1 (page 807), we could use a **one-hot vector**—that is, we encode the *i*th word in the dictionary with a 1 bit in the ith input position and a 0 in all the other positions. But such a representation would not capture the similarity between words.

Following the linguist John R. Firth’s (1957) maxim, “You shall know a word by the company it keeps,” we could represent each word with a vector of n-gram counts of all the phrases that the word appears in. However, raw n-gram counts are cumbersome. With a 100,000-word vocabulary, there are 10^{25} 5-grams to keep track of (although vectors in this 10^{25}-dimensional space would be quite sparse—most of the counts would be zero). We would get better generalization if we reduced this to a smaller-size vector, perhaps with just a few hundred dimensions. We call this smaller, dense vector a **word embedding**: a low-dimensional vector representing a word. Word embeddings are *learned automatically* from the data. (We will see later how this is done.) What are these learned word embeddings like? On the one hand, each one is just a vector of numbers, where the individual dimensions and their numeric values do not have discernible meanings:

$$
\begin{aligned}
\backslash\backslash\text{aardvark}// &= [-0.7, +0.2, -3.2, \ldots] \\
\backslash\backslash\text{abacus}// &= [+0.5, +0.9, -1.3, \ \ldots] \\
\ldots & \\
\backslash\backslash\text{zyzzyva}// &= [-0.1, +0.8, -0.4, \ldots].
\end{aligned}
$$

On the other hand, the feature space has the property that similar words end up having similar vectors. We can see that in Figure 25.1, where there are separate clusters for country, kinship, transportation, and food words.

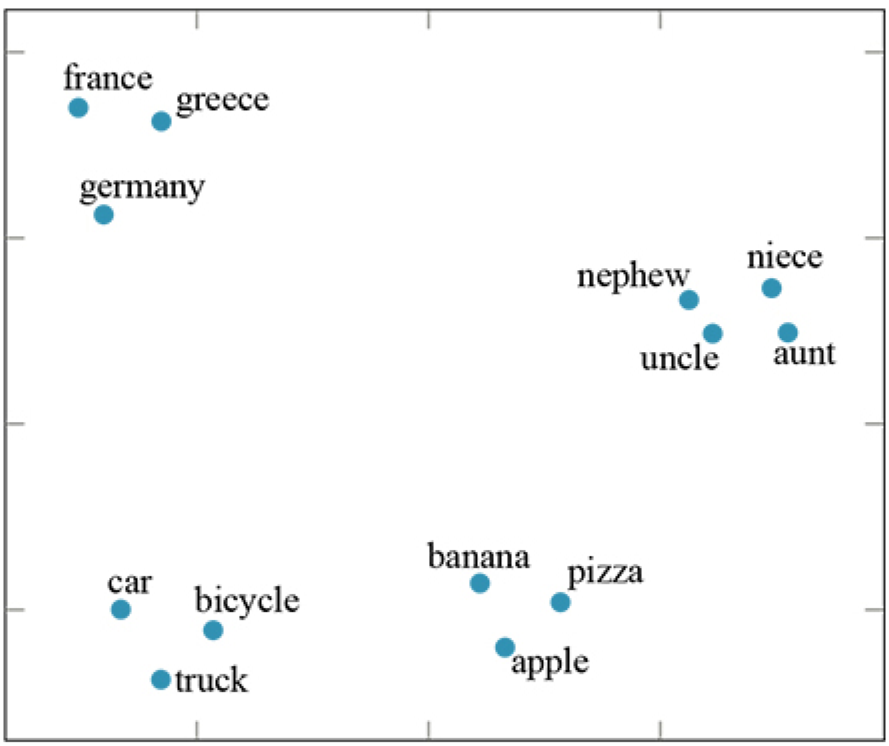


Figure 25.1 Word embedding vectors computed by the GloVe algorithm trained on 6 billion words of text. 100-dimensional

word vectors are projected down onto two dimensions in this visualization. Similar words appear near each other.

It turns out, for reasons we do not completely understand, that the word embedding vectors have additional properties beyond mere proximity for similar words. For example, suppose we look at the vectors **A** for Athens and **B** for Greece. For these words the vector difference **B** – **A** seems to encode the country/capital relationship. Other pairs—France and Paris, Russia and Moscow, Zambia and Lusaka—have essentially the same vector difference.

We can use this property to solve word analogy problems such as "Athens is to Greece as Oslo is to [what]?" Writing C for the Oslo vector and D for the unknown, we hypothesize that **B – A = D – C**, giving us **D = C+(B – A).** And when we compute this new vector **D**, we find that it is closer to "Norway" than to any other word. Figure 25.2 shows that this type of vector arithmetic works for many relationships.

A	B	C	D = C + (B – A)	Relationship
Athens	Greece	Oslo	Norway	*Capital*
Astana	Kazakhstan	Harare	Zimbabwe	*Capital*
Angola	kwanza	Iran	rial	*Currency*
copper	Cu	gold	Au	*Atomic Symbol*
Microsoft	Windows	Google	Android	*Operating System*
New York	New York Times	Baltimore	Baltimore Sun	*Newspaper*
Berlusconi	Silvio	Obama	Barack	*First name*
Switzerland	Swiss	Cambodia	Cambodian	*Nationality*
Einstein	scientist	Picasso	painter	*Occupation*
brother	sister	grandson	granddaughter	*Family Relation*
Chicago	Illinois	Stockton	California	*State*
possibly	impossibly	ethical	unethical	*Negative*
mouse	mice	dollar	dollars	*Plural*
easy	easiest	lucky	luckiest	*Superlative*
walking	walked	swimming	swam	*Past tense*

Figure 25.2 A word embedding model can sometimes answer the question " **A** is to **B** as **C** is to [what]?" with vector arithmetic: given the word embedding vectors for the words **A**, **B**, and **C**, compute the vector **D** = **C**+(**B** – **A**) and look up the word that is closest to **D**. (The answers in column **D** were computed automatically by the model. The descriptions in the "Relationship" column were added by hand.) Adapted from Mikolov *et al.* (2013, 2014).

However, there is no guarantee that a particular word embedding algorithm run on a particular corpus will capture a particular semantic relationship. Word embeddings are popular because they have proven to be a good representation for downstream language tasks (such as question answering or translation or summarization), not because they are guaranteed to answer analogy questions on their own.

Using word embedding vectors rather than one-hot encodings of words turns out to be helpful for essentially all applications of deep learning to NLP tasks. Indeed, in many cases it is possible to use generic **pretrained** vectors, obtained from any of several suppliers, for one's particular NLP task. At the time of writing, the commonly used vector dictionaries include WORD2VEC, GloVe (Global Vectors), and FASTTEXT, which has embeddings for 157 languages. Using a pretrained model can save a great deal of time and effort. For more on these resources, see Section 25.5.1.

It is also possible to train your own word vectors; this is usually done at the same time as training a network for a particular task. Unlike generic pretrained embeddings, word embeddings produced for a specific task can be trained on a carefully selected corpus and will tend to emphasize aspects of words that are useful for the task. Suppose, for example, that the task is part-of-speech (POS) tagging (see Section 24.1.6). Recall that this involves predicting the correct part of speech for each word in a sentence. Although this is a simple task, it is nontrivial because many words can be tagged in multiple ways—for example, the word *cut* can be a present-tense verb (transitive or intransitive), a past-tense verb, an infinitive verb, a past participle, an adjective, or a noun. If a nearby temporal adverb refers to the past, that suggests that this particular occurrence of cut is a past-tense verb; and we might hope, then, that the embedding will capture the past-referring aspect of adverbs.

POS tagging serves as a good introduction to the application of deep learning to NLP, without the complications of more complex tasks like question answering (see Section 25.5.3). Given a corpus of sentences with

POS tags, we learn the parameters for the word embeddings and the POS tagger simultaneously. The process works as follows:

1. Choose the width w (an odd number of words) for the prediction window to be used to tag each word. A typical value is $w = 5$, meaning that the tag is predicted based on the word plus the two words to the left and the two words to the right. Split every sentence in your corpus into overlapping windows of length w. Each window produces one training example consisting of the w words as input and the POS category of the middle word as output.
2. Create a vocabulary of all of the unique word tokens that occur more than, say, 5 times in the training data. Denote the total number of words in the vocabulary as v.
3. Sort this vocabulary in any arbitrary order (perhaps alphabetically).
4. Choose a value d as the size of each word embedding vector.
5. Create a new v-by-d weight matrix called $\mathbf{E}$. This is the word embedding matrix. Row i of $\mathbf{E}$ is the word embedding of the ith word in the vocabulary. Initialize $\mathbf{E}$ randomly (or from pretrained vectors).
6. Set up a neural network that outputs a part of speech label, as shown in Figure 25.3. The first layer will consist of w copies of the embedding matrix. We might use two additional hidden layers, $\mathbf{z}_1$ and $\mathbf{z}_2$ (with weight matrices $\mathbf{W}_1$ and $\mathbf{W}_2$, respectively), followed by a softmax layer yielding an output probability distribution y over the possible part-of-speech categories for the middle word:

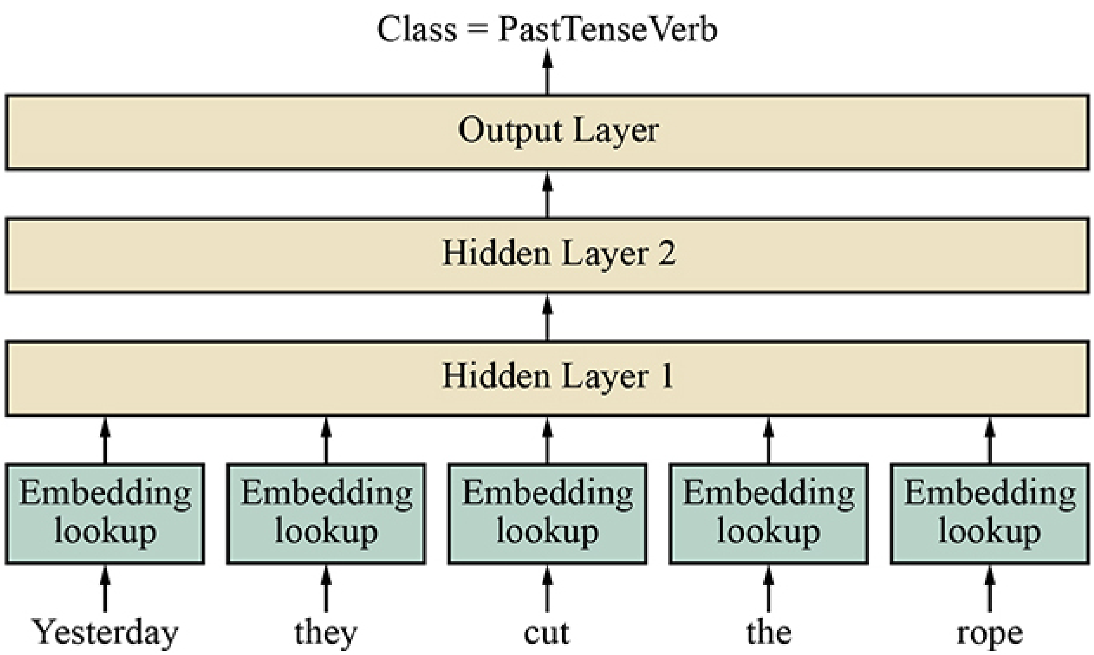


Figure 25.3 Feedforward part-of-speech tagging model. This model takes a 5-word window as input and predicts the tag of the word in the middle—here, *cut*. The model is able to account for word position because each of the 5 input embeddings is multiplied by a different part of the first hidden layer. The parameter values for the word embeddings and for the three layers are all learned simultaneously during training.

$$\mathbf{z}_1 = \sigma(\mathbf{W}_1\mathbf{x})$$
$$\mathbf{z}_2 = \sigma(\mathbf{W}_2\mathbf{z}^1)$$
$$\hat{\mathbf{y}} = \text{softmax}\,(\mathbf{W}_{out}\mathbf{z}_2)$$

7. To encode a sequence of *w* words into an input vector, simply look up the embedding for each word and concatenate the embedding vectors.

The result is a real-valued input vector **x** of length *wd*. Even though a given word will have the same embedding vector whether it occurs in the first position, the last, or somewhere in between, each embedding will be multiplied by a different part of the first hidden layer; therefore we are implicitly encoding the relative position of each word.

8. Train the weights **E** and the other weight matrices $\mathbf{W}_1$, $\mathbf{W}_2$, and $\mathbf{W}_{out}$ using gradient descent. If all goes well, the middle word, *cut*, will be labeled as a past-tense verb, based on the evidence in the window, which includes the temporal past word "yesterday," the third-person subject pronoun "they" immediately before *cut*, and so on.

An alternative to word embeddings is a **character-level model** in which the input is a sequence of characters, each encoded as a one-hot vector. Such a model has to learn how characters come together to form words. The majority of work in NLP sticks with word-level rather than character-level encodings.

25.2 Recurrent Neural Networks for NLP

We now have a good representation for single words in isolation, but language consists of an ordered sequence of words in which the **context** of surrounding words is important. For simple tasks like part of speech tagging, a small, fixed-size window of perhaps five words usually provides enough context.

More complex tasks such as question answering or reference resolution may require dozens of words as context. For example, in the sentence *"Eduardo told me that Miguel was very sick so I took* **him** *to the hospital,"* knowing that **him** refers to *Miguel* and not *Eduardo* requires context that spans from the first to the last word of the 14-word sentence.

25.2.1 Language models with recurrent neural networks

We'll start with the problem of creating a **language model** with sufficient context. Recall that a language model is a probability distribution over sequences of words. It allows us to predict the next word in a text given all the previous words, and is often used as a building block for more complex tasks.

Building a language model with either an n-gram model (as in Section 24.1) or a feedforward network with a fixed window of n words can run into difficulty due to the problem of context: either the required context will exceed the fixed window size or the model will have too many parameters, or both.

In addition, a feedforward network has the problem of **asymmetry**: whatever it learns about, say, the appearance of the word *him* as the 12th word of the sentence it will have to relearn for the appearance of *him* at other positions in the sentence, because the weights are different for each word position.

In Section 22.6, we introduced the **recurrent neural network** or **RNN**, which is designed to process time-series data, one datum at a time. This suggests that **RNNs** might be useful for processing language, one word at a time. We repeat Figure 22.8 here as Figure 25.4.

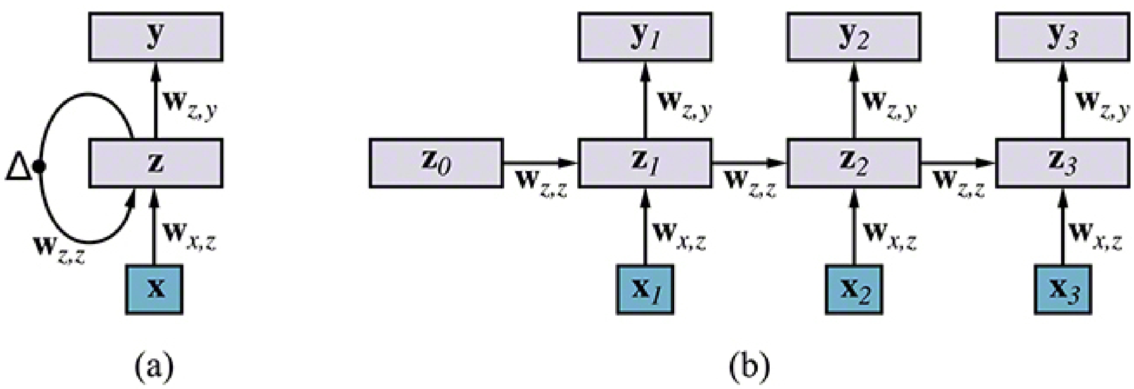


Figure 25.4 (a) Schematic diagram of an **RNN** where the hidden layer $\mathbf{z}_t$ has recurrent connections; the Δ symbol indicates a delay. Each input $\mathbf{x}$ is the word embedding vector of the next word in the sentence. Each output $\mathbf{y}$ is the output for that time step. (b) The same network unrolled over three timesteps to create a feedforward network. Note that the weights are shared across all timesteps.

In an **RNN** language model each input word is encoded as a word embedding vector, $\mathbf{x}_i$. There is a hidden layer $\mathbf{z}_t$ which gets passed as input from one time step to the next. We are interested in doing multiclass classification: the classes are the words of the vocabulary. Thus the output $\mathbf{y}_t$ will be a softmax probability distribution over the possible values of the next word in the sentence.

The **RNN** architecture solves the problem of too many parameters. The number of parameters in the weight matrixes $w_{,z,z,}$ $w_{,x,z,}$ and $w_{,z,y}$ stays constant, regardless of the number of words—it is $O(1)$. This is in contrast to feedforward networks, which have $O(n)$ parameters, and n-gram models, which have $O(v^n)$ parameters, where v is the size of the vocabulary.

The **RNN** architecture also solves the problem of asymmetry, because the weights are the same for every word position.

The **RNN** architecture can sometimes solve the limited context problem as well. In theory there is no limit to how far back in the input the model can look. Each update of the hidden layer $\mathbf{z}_t$ has access to both the current input word $\mathbf{x}_t$ and the previous hidden layer $\mathbf{z}_{t-1}$ which means that information about any word in the input can be kept in the hidden layer indefinitely, copied over (or modified as appropriate) from one time step to the next. Of course, there is a limited amount of storage in $\mathbf{z}$, so it can't remember everything about all the previous words.

In practice **RNN** models perform well on a variety of tasks, but not on all tasks. It can be hard to predict whether they will be succesful for a given problem. One factor that contributes to success is that the training process encourages the network to allocate storage space in $\mathbf{z}$ to the aspects of the input that will actually prove to be useful.

To train an **RNN** language model, we use the training process described in Section 22.6.1. The inputs, $\mathbf{x}_t$, are the words in a training corpus of text, and the observed outputs are the same words offset by 1. That is, for the training text "hello world," the first input $\mathbf{x}_1$ is the word embedding for "hello" and the first output $\mathbf{y}_1$ is the word embedding for "world." We are training the model to predict the next word, and expecting that in order to do so it will use the hidden layer to represent useful information. As explained in Section 22.6.1 we compute the difference between the observed output and the actual output computed by the network, and back-propagate through time, taking care to keep the weights the same for all time steps.

Once the model has been trained, we can use it to generate random text. We give the model an initial input word $\mathbf{x}_1$, from which it will produce an output $\mathbf{y}_1$ which is a softmax probability distribution over words. We sample a single word from the distribution, record the word as the output for time t, and feed it back in as the next input word $\mathbf{x}_2$. We repeat for as long as desired. In sampling from $\mathbf{y}_1$ we have a choice: we could always take the most likely word; we could sample according to the probability of each word; or we could oversample the less-likely words, in order to inject more variety into the generated output. The sampling weight is a hyperparameter of the model.

Here is an example of random text generated by an **RNN** model trained on Shakespeare's works (Karpathy, 2015):

Marry, and will, my lord, to weep in such a one were prettiest;

Yet now I was adopted heir

Of the world's lamentable day,

To watch the next way with his father with his face?

25.2.2 Classification with recurrent neural networks

It is also possible to use **RNN**s for other language tasks, such as part of speech tagging or coreference resolution. In both cases the input and hidden layers will be the same, but for a POS tagger the output will be a softmax distribution over POS tags, and for coreference resolution it will be a softmax distribution over the possible antecedents. For example, when the network gets to the input **him** in *“Eduardo told me that Miguel was very sick so I took* **him** *to the hospital”* it should output a high probability for *“Miguel”*

Training an **RNN** to do classification like this is done the same way as with the language model. The only difference is that the training data will require labels—part of speech tags or reference indications. That makes it much harder to collect the data than for the case of a language model, where unlabelled text is all we need.

In a language model we want to predict the *n*th word given the previous words. But for classification, there is no reason we should limit ourselves to looking at only the previous words. It can be very helpful to look ahead in the sentence. In our coreference example, the referent *him* would be different if the sentence concluded “to see Miguel” rather than “to the hospital,” so looking ahead is crucial. We know from eye-tracking experiments that human readers do not go strictly left-to-right.

To capture the context on the right, we can use a **bidirectional RNN**, which concatenates a separate right-to-left model onto the left-to-right model. An example of using a bidirectional **RNN** for POS tagging is shown in Figure 25.5.

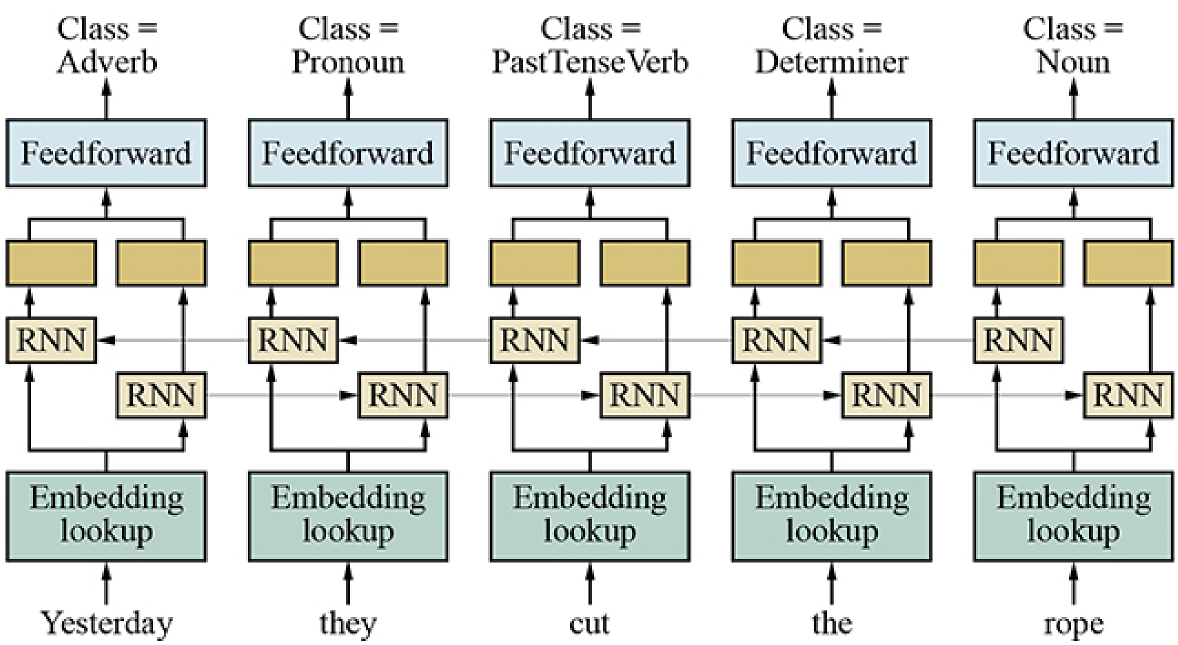


Figure 25.5 A bidirectional **RNN** network for **POS** tagging.

In the case of a multilayer **RNN**, $\mathbf{z}_t$ will be the hidden vector of the last layer. For a bidirectional **RNN**, $\mathbf{z}_t$ is usually taken to be the concatenation of vectors from the left-to-right and right-to-left models.

RNNs can also be used for sentence-level (or document-level) classification tasks, in which a single output comes at the end, rather than having a stream of outputs, one per time step. For example in **sentiment analysis** the goal is to classify a text as having either *Positive* or *Negative* sentiment. For example, *"This movie was poorly written and poorly acted"* should be classified as *Negative*. (Some sentiment analysis schemes use more than two categories, or use a numeric scalar value.)

Using **RNN**s for a sentence-level task is a bit more complex, since we need to obtain an aggregate whole-sentence representation, $\mathbf{y}$ from the per-word outputs $\mathbf{y}_t$ of the **RNN**. The simplest way to do this is to use the **RNN** hidden state corresponding to the last word of the input, since the **RNN** will have read the entire sentence at that timestep. However, this can implicitly bias the model towards paying more attention to the end of the sentence. Another common technique is to pool all of the hidden vectors. For instance, **average pooling** computes the element-wise average over all of the hidden vectors:

$$\tilde{\mathbf{z}} = \frac{1}{s} \sum_{t=1}^{s} \mathbf{z}_t.$$

The pooled d-dimensional vector $\tilde{\mathbf{z}}$ can then be fed into one or more feedforward layers before being fed into the output layer.

25.2.3 LSTMs for NLP tasks

We said that **RNN**s sometimes solve the limited context problem. In theory, any information could be passed along from one hidden layer to the next for any number of time steps. But in practice the information can get lost or distorted, just as in playing the game of telephone, in which players stand in line and the first player whispers a message to the second, who repeats it to the third, and so on down the line. Usually, the message that comes out at the end is quite corrupted from the original message. This problem for **RNN**s is similar to the **vanishing gradient** problem we described on page 807, except that we are dealing now with layers over time rather than with deep layers.

In Section 22.6.2 we introduced the **long short-term memory (LSTM)** model. This is a kind of **RNN** with gating units that don't suffer from the problem of imperfectly reproducing a message from one time step to the next. Rather, an LSTM can choose to *remember* some parts of the input, copying it over to the next timestep, and to forget other parts. Consider a language model handling a text such as

> *The athletes, who all won their local qualifiers and advanced to the finals in Tokyo, now ...*

At this point if we asked the model which next word was more probable, “compete” or “competes,” we would expect it to pick “compete” because it agrees with the subject “The athletes.” An LSTM can learn to create a latent feature for the subject person and number and copy that feature forward without alteration until it is needed to make a choice like this. A regular **RNN** (or an *n*-gram model for that matter) often gets confused in long sentences with many intervening words between the subject and verb.

25.3 Sequence-to-Sequence Models

One of the most widely studied tasks in NLP is **machine translation (MT)**, where the goal is to translate a sentence from a **source language** to a **target language**—for example, from Spanish to English. We train an MT model with a large corpus of source/target sentence pairs. The goal is to then accurately translate new sentences that are not in our training data.

Can we use **RNN**s to create an MT system? We can certainly encode the source sentence with an **RNN**. If there were a one-to-one correspondence between source words and target words, then we could treat MT as a simple tagging task—given the source word "perro" in Spanish, we tag it as the corresponding English word "dog." But in fact, words are not one-to-one: in Spanish the three words "caballo de mar" corresponds to the single English word "seahorse," and the two words "perro grande" translate to "big dog," with the word order reversed. Word reordering can be even more extreme; in English the subject is usually at the start of a sentence, but in Fijian the subject is usually at the end. So how do we generate a sentence in the target language?

It seems like we should generate one word at a time, but keep track of the context so that we can remember parts of the source that haven't been translated yet, and keep track of what has been translated so that we don't repeat ourselves. It also seems that for some sentences we have to process the entire source sentence before starting to generate the target. In other words, the generation of each target word is conditional on the entire source sentence and on all previously generated target words.

This gives text generation for MT a close connection to a standard **RNN** language model, as described in Section 25.2. Certainly, if we had trained an **RNN** on English text, it would be more likely to generate “big dog” than “dog big.” However, we don’t want to generate just any random target language sentence; we want to generate a target language sentence that *corresponds* to the source language sentence. The simplest way to do that is to use two **RNN**s, one for the source and one for the target. We run the source **RNN** over the source sentence and then use the final hidden state from the source **RNN** as the initial hidden state for the target **RNN**. This way, each target word is implicitly conditioned on both the entire source sentence and the previous target words.

This neural network architecture is called a basic **sequence-to-sequence model**, an example of which is shown in Figure 25.6. Sequence-to-sequence models are most commonly used for machine translation, but can also be used for a number of other tasks, like automatically generating a text caption from an image, or summarization: rewriting a long text into a shorter one that maintains the same meaning.

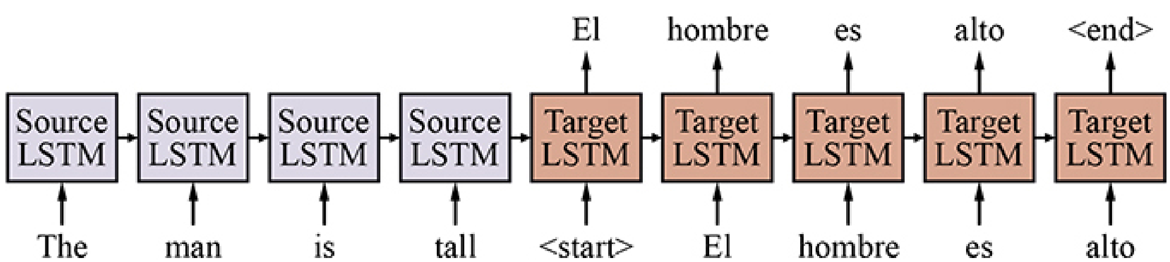


Figure 25.6 Basic sequence-to-sequence model. Each block represents one LSTM timestep. (For simplicity, the embedding and output layers are not shown.) On successive steps we feed the network the words of the source sentence “The man is tall,” followed by the <start> tag to indicate that the network should

start producing the target sentence. The final hidden state at the end of the source sentence is used as the hidden state for the start of the target sentence. After that, each target sentence word at time *t* is used as input at time *t*+1, until the network produces the <end> tag to indicate that sentence generation is finished.

Basic sequence-to-sequence models were a significant breakthrough in NLP and MT specifically. According to Wu *et al.* (2016b) the approach led to a 60% error reduction over the previous MT methods. But these models suffer from three major shortcomings:

- **Nearby context bias:** whatever **RNN**s want to remember about the past, they have to fit into their hidden state. For example, let's say an **RNN** is processing word (or timestep) 57 in a 70-word sequence. The hidden state will likely contain more information about the word at timestep 56 than the word at timestep 5, because each time the hidden vector is updated it has to replace some amount of existing information with new information. This behavior is part of the intentional design of the model, and often makes sense for NLP, since nearby context is typically more important. However, far-away context can be crucial as well, and can get lost in an **RNN** model; even LSTMs have difficulty with this task.
- **Fixed context size limit:** In an **RNN** translation model the entire source sentence is compressed into a single fixed-dimensional hidden state vector. An LSTM used in a state-of-the-art NLP model typically has around 1024 dimensions, and if we have to represent, say, a 64-word sentence in 1024 dimensions, this only gives us 16 dimensions per word—not enough for complex sentences. Increasing the hidden state vector size can lead to slow training and overfitting.

- **Slower sequential processing:** As discussed in Section 22.3, neural networks realize considerable efficiency gains by processing the training data in batches so as to take advantage of efficient hardware support for matrix arithmetic. **RNN**s, on the other hand, seem to be constrained to operate on the training data one word at a time.

25.3.1 Attention

What if the target **RNN** were conditioned on *all* of the hidden vectors from the source **RNN**, rather than just the last one? This would mitigate the shortcomings of nearby context bias and fixed context size limits, allowing the model to access any previous word equally well. One way to achieve this access is to concatenate all of the source **RNN** hidden vectors. However, this would cause a huge increase in the number of weights, with a concomitant increase in computation time and potentially overfitting as well. Instead, we can take advantage of the fact that when the target **RNN** is generating the target one word at a time, it is likely that only a small part of the source is actually relevant to each target word.

Crucially, the target **RNN** must pay attention to different parts of the source for every word. Suppose a network is trained to translate English to Spanish. It is given the words “The front door is red” followed by an end of sentence marker, which means it is time to start outputting Spanish words. So ideally it should first pay attention to “The” and generate “La,” then pay attention to “door” and output “puerta,” and so on.

We can formalize this concept with a neural network component called **attention**, which can be used to create a “context-based summarization” of the source sentence into a fixeddimensional representation. The context vector $\mathbf{c}_i$ contains the most relevant information for generating the next target word, and will be used as an additional input to the target **RNN**. A sequence-to-sequence model that uses attention is called an **attentional sequence-to-sequence model**. If the standard target **RNN** is written as:

$$\mathbf{h}_i = RNN(\mathbf{h}_{i-1}, \mathbf{x}_i),$$

the target **RNN** for attentional sequence-to-sequence models can be written as:

$$\mathbf{h}_i = RNN(\mathbf{h}_{i-1}, [\mathbf{x}_i; \mathbf{c}_i])$$

where $[\mathbf{x}_i; \mathbf{c}_i]$ is the concatenation of the input and context vectors, $\mathbf{c}_i$ defined as:

$$r_{ij} = \mathbf{h}_{i-1} \cdot \mathbf{s}_j$$

$$a_{ij} = e^{r_{ij}} / (\sum_k e^{r_{ik}})$$

$$\mathbf{c}_i = \sum_j a_{ij} \cdot \mathbf{s}_j$$

where $\mathbf{h}_{i-1}$ is the target **RNN** vector that is going to be used for predicting the word at timestep *i*, and $\mathbf{s}_j$ is the output of the source **RNN** vector for the source word (or timestep) $\mathbf{h}_{i-1}$ and $\mathbf{s}_j$ are *d*-dimensional vectors, where *d* is the hidden size. The value of r_{ij} is therefore the raw "attention score" between the current target state and the source word *j*. These scores are then normalized into a probability a_{ij} using a softmax over all source words. Finally, these probabilities are used to generate a weighted average of the source **RNN** vectors, $\mathbf{c}_i$ (another *d*-dimensional vector).

An example of an attentional sequence-to-sequence model is given in Figure 25.7 (a). There are a few important details to understand. First, the attention component itself has no learned weights and supports variable-length sequences on both the source and target side. Second, like most of the other neural network modeling techniques we've learned about, attention is entirely latent. The programmer does not dictate what information gets used when; the model learns what to use. Attention can also be combined with multilayer **RNN**s. Typically attention is applied at each layer in that case.

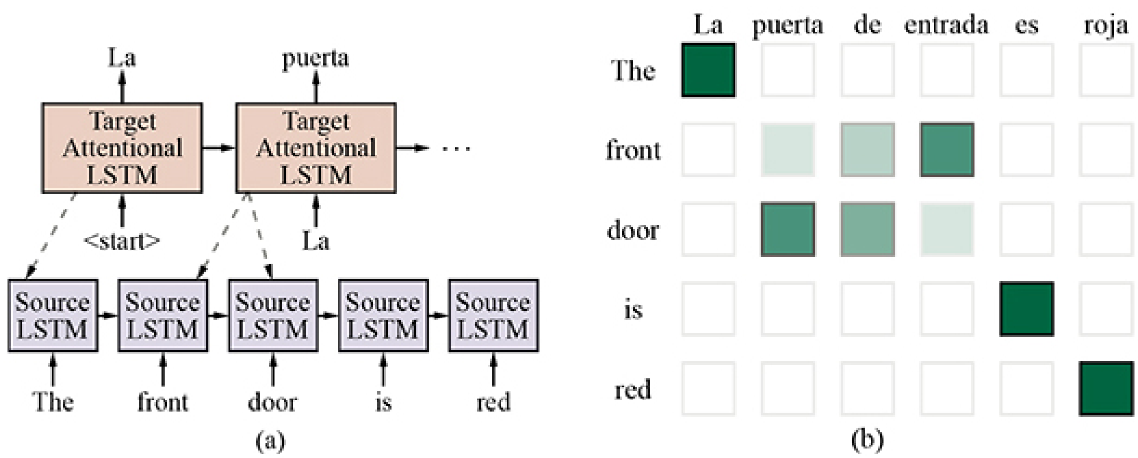


Figure 25.7 (a) Attentional sequence-to-sequence model for English-to-Spanish translation. The dashed lines represent attention. (b) Example of attention probability matrix for a bilingual sentence pair, with darker boxes representing higher values of a_{ij}. The attention probabilities sum to one over each column.

The probabilistic softmax formulation for attention serves three purposes. First, it makes attention differentiable, which is necessary for it to be used with back-propagation. Even though attention itself has no learned weights, the gradients still flow back through attention to the source and target **RNN**s. Second, the probabilistic formulation allows the model to capture certain types of long-distance contextualization that may have not been captured by the source **RNN**, since attention can consider the entire source sequence at once, and learn to keep what is important and ignore the rest. Third, probabilistic attention allows the network to represent uncertainty—if the network does not know exactly what source word to translate next, it can distribute the attention probabilities over several options, and then actually choose the word using the target **RNN**.

Unlike most components of neural networks, attention probabilities are often interpretable by humans and intuitively meaningful. For example, in the case of machine translation, the attention probabilities often correspond to the word-to-word alignments that a human would generate. This is shown in Figure 25.7(b).

Sequence-to-sequence models are a natural for machine translation, but almost any natural language task can be encoded as a sequence-to-sequence problem. For example, a question-answering system can be trained on input consisting of a question followed by a delimiter followed by the answer.

25.3.2 Decoding

At training time, a sequence-to-sequence model attempts to maximize the probability of each word in the target training sentence, conditioned on the source and all of the previous target words. Once training is complete, we are given a source sentence, and our goal is to generate the corresponding target sentence. As shown in Figure 25.7, we can generate the target one word at a time, and then feed back in the word that we generated at the next timestep. This procedure is called **decoding.**

The simplest form of decoding is to select the highest probability word at each timestep and then feed this word as input to the next timestep. This is called **greedy decoding** because after each target word is generated, the system has fully committed to the hypothesis that it has produced so far. The problem is that the goal of decoding is to maximize the probability of the entire target sequence, which greedy decoding may not achieve. For example, consider using a greedy decoder to translate into Spanish the English sentence we saw before, *The front door is red.*

The correct translation is *“La puerta de entrada es roja”*—literally *“The door of entry is red.”* Suppose the target **RNN** correctly generates the first word *La* for *The.* Next, a greedy decoder might propose *entrada* for *front*. But this is an error—Spanish word order should put the noun *puerta* before the modifier. Greedy decoding is fast—it only considers one choice at each timestep and can do so quickly—but the model has no mechanism to correct mistakes.

We could try to improve the attention mechanism so that it always attends to the right word and guesses correctly every time. But for many sentences it is infeasible to guess correctly all the words at the start of the sentence until you have seen what’s at the end.

A better approach is to search for an optimal decoding (or at least a good one) using one of the search algorithms from Chapter 3. A common choice is a **beam search** (see Section 4.1.3). In the context of MT decoding, beam search typically keeps the top k hypotheses at each stage, extending each by one word using the top k choices of words, then chooses the best k of the resulting k^2 new hypotheses. When all hypotheses in the beam generate the special <end> token, the algorithm outputs the highest scoring hypothesis.

A visualization of beam search is given in Figure 25.8. As deep learning models become more accurate, we can usually afford to use a smaller beam size. Current state-of-the-art neural MT models use a beam size of 4 to 8, whereas the older generation of statistical MT models would use a beam size of 100 or more.

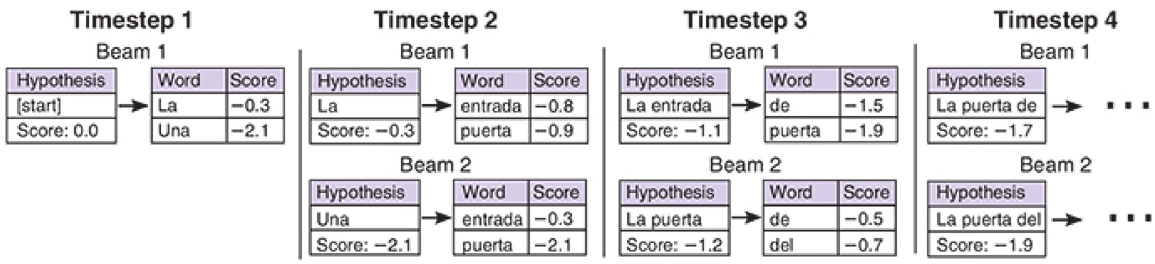


Figure 25.8 Beam search with beam size of $b = 2$. The score of each word is the log-probability generated by the target **RNN** softmax, and the score of each hypothesis is the sum of the word scores. At timestep 3, the highest scoring hypothesis *La entrada* can only generate low-probability continuations, so it "falls off the beam."

25.4 The Transformer Architecture

The influential article "Attention is all you need" (Vaswani *et al.*, 2018) introduced the **transformer** architecture, which uses a **self-attention** mechanism that can model long-distance context without a sequential dependency.

25.4.1 Self-attention

Previously, in sequence-to-sequence models, attention was applied from the target **RNN** to the source **RNN**. **Self-attention** extends this mechanism so that each sequence of hidden states also attends to itself—the source to the source, and the target to the target. This allows the model to additionally capture long-distance (and nearby) context within each sequence.

The most straightforward way of applying self-attention is where the attention matrix is directly formed by the dot product of the input vectors. However, this is problematic. The dot product between a vector and itself will always be high, so each hidden state will be biased towards attending to itself. The transformer solves this by first projecting the input into three different representations using three different weight matrices:

- The **query vector** $\mathbf{q}_i$, = $\mathbf{W}_q\mathbf{x}_i$, is the one being *attended from*, like the target in the standard attention mechanism.
- The **key vector** $\mathbf{k}_i = \mathbf{W}_k\mathbf{x}_i$, is the one being *attended to*, like the source in the basic attention mechanism.
- The **value vector** $\mathbf{v}_i$, = $\mathbf{W}_v\mathbf{x}_i$ is the context that is being generated.

In the standard attention mechanism, the key and value networks are identical, but intuitively it makes sense for these to be separate representations. The encoding results of the *i*th word, $\boldsymbol{c}_i$, can be calculated by applying an attention mechanism to the projected vectors:

$$r_{ij} = (\mathbf{q}_i \cdot \mathbf{k}_j)/\sqrt{d}$$

$$a_{ij} = e^{r_{ij}}/(\sum_k e^{r_{ik}})$$

$$\mathbf{c}_i = \sum_j a_{ij} \cdot \mathbf{v}_j,$$

where d is the dimension of $\mathbf{k}$ and $\mathbf{q}$. Note that i and j are indexes in the same sentence, since we are encoding the context using self-attention. In each transformer layer, self-attention uses the hidden vectors from the previous layer, which initially is the embedding layer.

There are several details worth mentioning here. First of all, the self-attention mechanism is *asymmetric*, as r_{ij} is different from r_{ji}-. Second, the scale factor $\sqrt{d}$ was added to improve numerical stability. Third, the encoding for all words in a sentence can be calculated simultaneously, as the above equations can be expressed using matrix operations that can be computed efficiently in parallel on modern specialized hardware.

The choice of which context to use is completely learned from training examples, not prespecified. The context-based summarization, $\mathbf{c}_i$, is a sum over all previous positions in the sentence. In theory, any information from the sentence could appear in $\mathbf{c}_i$, but in practice, sometimes important information gets lost, because it is essentially averaged out over the whole sentence. One way to address that is called **multiheaded attention**. We divide the sentence up into *m* equal pieces and apply the attention model to each of the *m* pieces. Each piece has its own set of weights. Then the results are concatenated together to form $\mathbf{c}_i$. By concatenating rather than summing, we make it easier for an important subpiece to stand out.

25.4.2 From self-attention to transformer

Self-attention is only one component of the transformer model. Each transformer layer consists of several sub-layers. At each transformer layer, self-attention is applied first. The output of the attention module is fed through feedforward layers, where the same feedforward weight matrices are applied independently at each position. A nonlinear activation function, typically ReLU, is applied after the first feedforward layer. In order to address the potential vanishing gradient problem, two residual connections are added into the transformer layer. A single-layer transformer in shown in Figure 25.9. In practice, transformer models usually have six or more layers. As with the other models that we've learned about, the output of layer i is used as the input to layer $i + 1$.

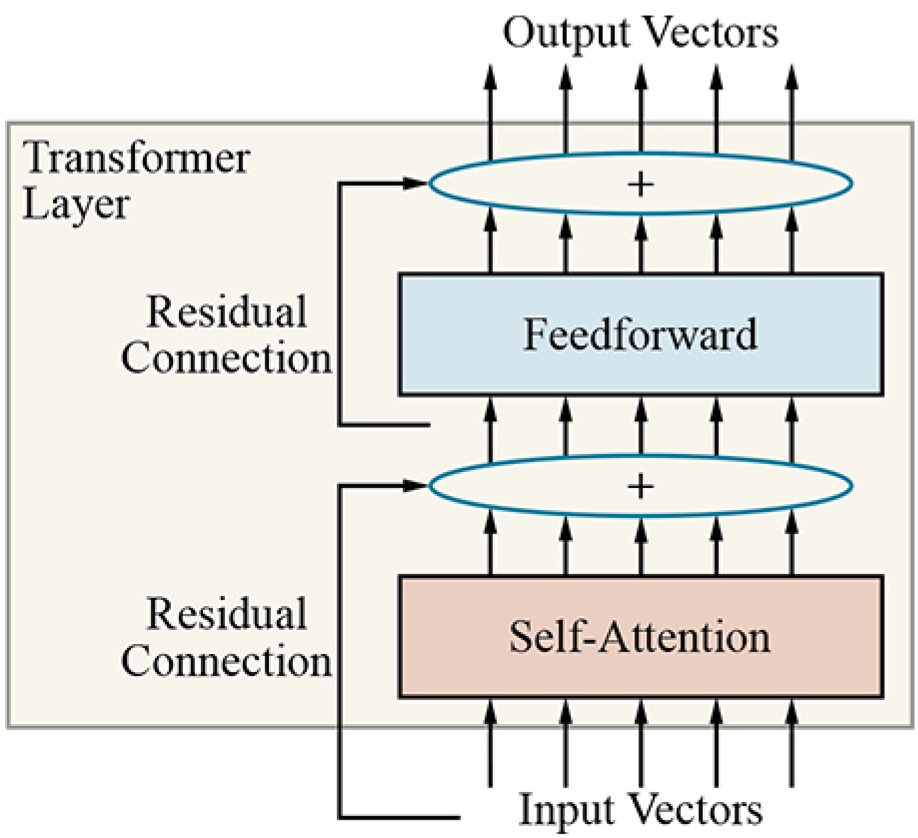


Figure 25.9 A single-layer transformer consists of self-attention, a feedforward network, and residual connections.

The transformer architecture does not explicitly capture the order of words in the sequence, since context is modeled only through self-attention, which is agnostic to word order. To capture the ordering of the words, the transformer uses a technique called **positional embedding**. If our input sequence has a maximum length of n, then we learn n new embedding vectors—one for each word position. The input to the first transformer layer is the sum of the word embedding at position *t* plus the positional embedding corresponding to position *t*.

Figure 25.10 illustrates the transformer architecture for POS tagging, applied to the same sentence used in Figure 25.3. At the bottom, the word embedding and the positional embeddings are summed to form the input for a three-layer transformer. The transformer produces one vector per word, as in **RNN**-based POS tagging. Each vector is fed into a final output layer and softmax layer to produce a probability distribution over the tags.

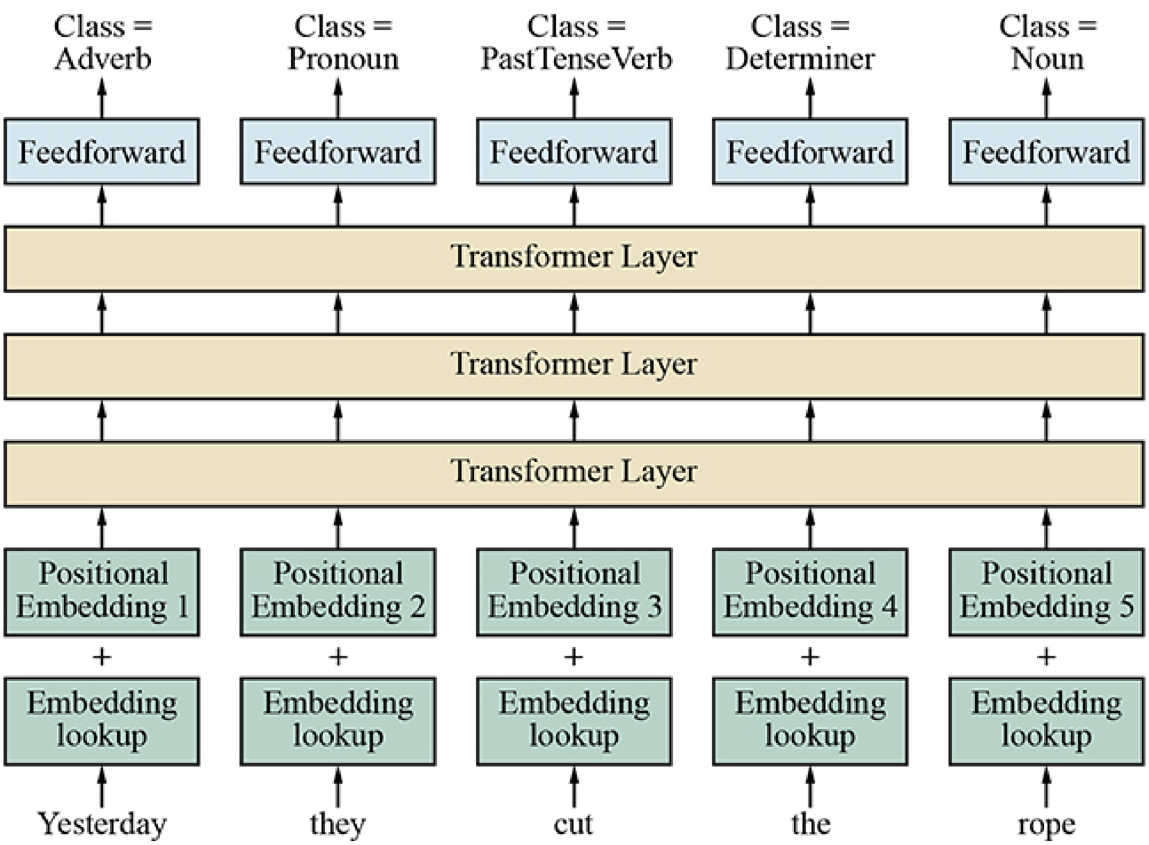


Figure 25.10 Using the transformer architecture for POS tagging.

In this section, we have actually only told half the transformer story: the model we described here is called the **transformer encoder**. It is useful for text classification tasks. The full transformer architecture was originally designed as a sequence-to-sequence model for machine translation. Therefore, in addition to the encoder, it also includes a **transformer decoder**. The encoder and decoder are nearly identical, except that the decoder uses a version of self-attention where each word can only attend to the words before it, since text is generated left-to-right. The decoder also has a second attention module in each transformer layer that attends to the output of the transformer encoder.

25.5 Pretraining and Transfer Learning

Getting enough data to build a robust model can be a challenge. In computer vision (see Chapter 27), that challenge was addressed by assembling large collections of images (such as ImageNet) and hand-labeling them.

For natural language, it is more common to work with text that is unlabeled. The difference is in part due to the difficulty of labeling: an unskilled worker can easily label an image as “cat” or “sunset,” but it requires extensive training to annotate a sentence with part-of-speech tags or parse trees. The difference is also due to the abundance of text: the Internet adds over 100 billion words of text each day, including digitized books, curated resources such as Wikipedia, and uncurated social media posts.

Projects such as Common Crawl provide easy access to this data. Any running text can be used to build *n*-gram or word embedding models, and some text comes with structure that can be helpful for a variety of tasks—for example, there are many FAQ sites with question-answer pairs that can be used to train a question-answering system. Similarly, many Web sites publish side-by-side translations of texts, which can be used to train machine translation systems. Some text even comes with labels of a sort, such as review sites where users annotate their text reviews with a 5-star rating system.

We would prefer not to have to go to the trouble of creating a new data set every time we want a new NLP model. In this section, we introduce the idea of **pretraining**: a form of **transfer learning** (see Section 22.7.2) in which we use a large amount of shared generaldomain language data to train an initial version of an NLP model. From there, we can use a smaller amount of domain-specific data (perhaps including some labeled data) to refine the model. The refined model can learn the vocabulary, idioms, syntactic structures, and other linguistic phenomena that are specific to the new domain.

25.5.1 Pretrained word embeddings

In Section 25.1, we briefly introduced word embeddings. We saw that how similar words like *banana* and *apple* end up with similar vectors, and we saw that we can solve analogy problems with vector subtraction. This indicates that the word embeddings are capturing substantial information about the words.

In this section we will dive into the details of how word embeddings are created using an entirely unsupervised process over a large corpus of text. That is in contrast to the embeddings from Section 25.1, which were built during the process of supervised part of speech tagging, and thus required POS tags that come from expensive hand annotation.

We will concentrate on one specific model for word embeddings, the GloVe (Global Vectors) model. The model starts by gathering counts of how many times each word appears within a window of another word, similar to the skip-gram model. First choose window size (perhaps 5 words) and let X_{ij} be the number of times that words i and j co-occur within a window, and let X_i be the number of times word i co-occurs with any other word. Let $P_{ij} = X_{ij}/X_i$ be the probability that word j appears in the context of word i. As before, let $\mathbf{E}_i$ be the word embedding for word i.

Part of the intuition of the GloVe model is that the relationship between two words can best be captured by comparing them both to other words.

Consider the words *ice* and *steam.* Now consider the ratio of their probabilities of co-occurrence with another word, *w*, that is:

$$P_{w,ice}/P_{w,steam}.$$

When *w* is the word *solid* the ratio will be high (meaning *solid* applies more to *ice*) and when *w* is the word *gas* it will be low (meaning *gas* applies more to *steam).* And when *w* is a non-content word like *the*, a word like *water* that is equally relevant to both, or an equally irrelevant word like *fashion*, the ratio will be close to 1.

The GloVe model starts with this intuition and goes through some mathematical reasoning (Pennington *et al.,* 2014) that converts ratios of probabilities into vector differences and dot products, eventually arriving at the constraint

$$\mathbf{E}_i \cdot \mathbf{E'}_k = \log(p_{ij}).$$

In other words, the dot product of two word vectors is equal to the log probability of their co-occurrence. That makes intuitive sense: two nearly-orthogonal vectors have a dot product close to 0, and two nearly-identical normalized vectors have a dot product close to 1. There is a technical complication wherein the GloVe model creates two word embedding vectors for each word, $\mathbf{E}_i$ and E'_i; computing the two and then adding them together at the end helps limit overfitting.

Training a model like GloVe is typically much less expensive than training a standard neural network: a new model can be trained from billions of words of text in a few hours using a standard desktop CPU.

It is possible to train word embeddings on a specific domain, and recover knowledge in that domain. For example, Tshitoyan *et al.* (2019) used 3.3 million scientific abstracts on the subject of material science to train a word embedding model. They found that, just as we saw that a generic word embedding model can answer “Athens is to Greece as Oslo is to what?” with “Norway,” their material science model can answer “NiFe is to ferromagnetic as IrMn is to what?” with “antiferromagnetic.”

Their model does not rely solely on co-occurrence of words; it seems to be capturing more complex scientific knowledge. When asked what chemical compounds can be classified as “thermoelectric” or “topological insulator,” their model is able to answer correctly. For example, $CsAgGa_2Se_4$ never appears near “thermoelectric” in the corpus, but it does appear near “chalcogenide,” “band gap,” and “optoelectric,” which are all clues enabling it to be classified as similar to “thermoelectric.” Furthermore, when trained only on abstracts up to the year 2008 and asked to pick compounds that are “thermoelectric” but have not yet appeared in abstracts, three of the model’s top five picks were discovered to be thermoelectric in papers published between 2009 and 2019.

25.5.2 Pretrained contextual representations

Word embeddings are better representations than atomic word tokens, but there is an important issue with polysemous words. For example, the word *rose* can refer to a flower or the past tense of *rise*. Thus, we expect to find at least two entirely distinct clusters of word contexts for *rose:* one similar to flower names such as *dahlia*, and one similar to *upsurge*. No single embedding vector can capture both of these simultaneously. *Rose* is a clear example of a word with (at least) two distinct meanings, but other words have subtle shades of meaning that depend on context, such as the word *need* in *you need to see this movie* versus *humans need oxygen to survive*. And some idiomatic phrases like *break the bank* are better analyzed as a whole rather than as component words.

Therefore, instead of just learning a word-to-embedding table, we want to train a model to generate **contextual representations** of each word in a sentence. A contextual representation maps both a word and the surrounding context of words into a word embedding vector. In other words, if we feed this model the word *rose* and the context *the gardener planted a rose bush*, it should produce a contextual embedding that is similar (but not necessarily identical) to the representation we get with the context *the cabbage rose had an unusual fragrance*, and very different from the representation of *rose* in the context *the river rose five feet.*

Figure 25.11 shows a recurrent network that creates contextual word embeddings—the boxes that are unlabeled in the figure. We assume we have already built a collection of noncontextual word embeddings. We feed in one word at a time, and ask the model to predict the next word. So for example in the figure at the point where we have reached the word "car," the the **RNN** node at that time step will receive two inputs: the noncontextual word embedding for "car" and the context, which encodes information from the previous words "The red." The **RNN** node will then output a contextual representation for "car." The network as a whole then outputs a prediction for the next word, "is." We then update the network's weights to minimize the error between the prediction and the actual next word.

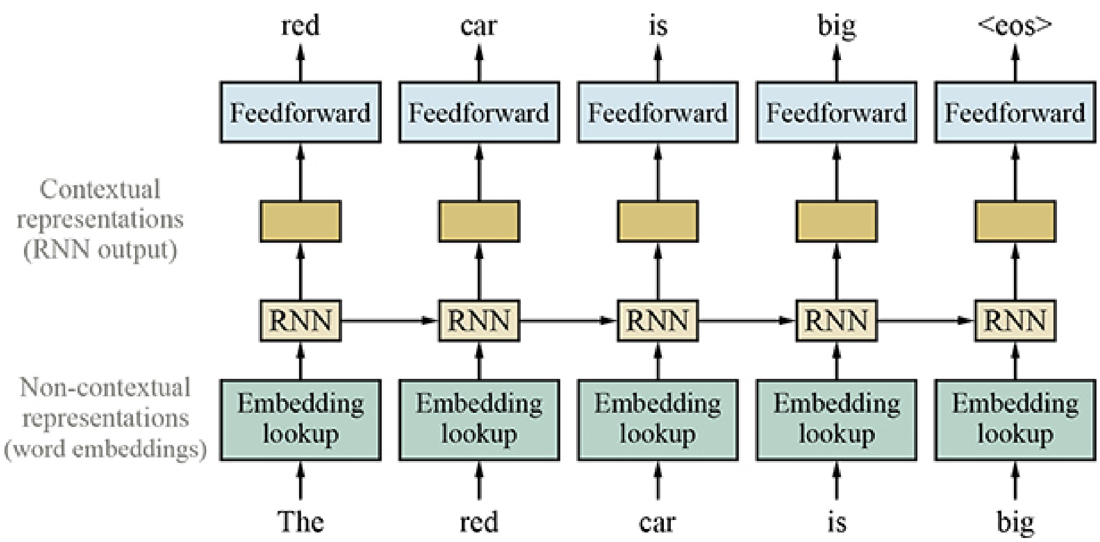


Figure 25.11 Training contextual representations using a left-to-right language model.

This model is similar to the one for POS tagging in Figure 25.5, with two important differences. First, this model is unidirectional (left-to-right), whereas the POS model is bidirectional. Second, instead of predicting the POS tags for the *current* word, this model predicts the *next* word using the prior context. Once the model is built, we can use it to retrieve representations for words and pass them on to some other task; we need not continue to predict the next word. Note that computing a contextual representations always requires two inputs, the current word and the context.

25.5.3 Masked language models

A weakness of standard language models such as *n*-gram models is that the contextualization of each word is based only on the previous words of the sentence. Predictions are made from left to right. But sometimes context from later in a sentence—for example, *feet* in the phrase *rose five feet*—helps to clarify earlier words.

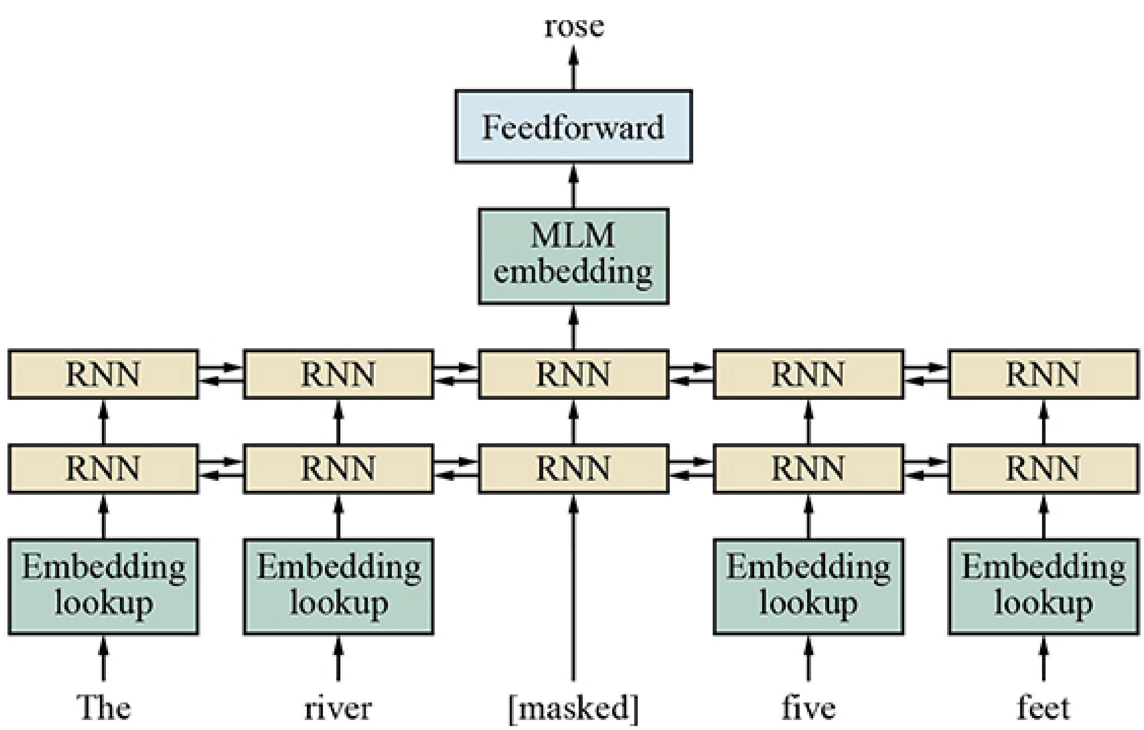


Figure 25.12 Masked language modeling: pretrain a bidirectional model—for example, a multilayer **RNN**—by masking input words and predicting only those masked words.

One straightforward workaround is to train a separate right-to-left language model that contextualizes each word based on subsequent words in the sentence, and then concatenate the left-to-right and right-to-left representations. However, such a model fails to combine evidence from both directions.

Instead, we can use a **masked language model (MLM)**. MLMs are trained by masking (hiding) individual words in the input and asking the model to predict the masked words. For this task, one can use a deep bidirectional **RNN** or transformer on top of the masked sentence. For example, given the input sentence *“The river rose five feet”* we can mask the middle word to get *“The river _ five feet”* and ask the model to fill in the blank.

The final hidden vectors that correspond to the masked tokens are then used to predict the words that were masked—in this example, *rose.* During training a single sentence can be used multiple times with different words masked out. The beauty of this approach is that it requires no labeled data; the sentence provides its own label for the masked word. If this model is trained on a large corpus of text, it generates pretrained representations that perform well across a wide variety of NLP tasks (machine translation, question answering, summarization, grammaticality judgments, and others).

25.6 State of the art

Deep learning and transfer learning have markedly advanced the state of the art for NLP—so much so that one commentator in 2018 declared that "NLP's ImageNet moment has arrived" (Ruder, 2018). The implication is that just as a turning point occurred in 2012 for computer vision when deep learning systems produced surprising good results in the ImageNet competition, a turning point occurred in 2018 for NLP. The principal impetus for this turning point was the finding that transfer learning works well for natural language problems: a general language model can be downloaded and fine-tuned for a specific task.

It started with simple word embeddings from systems such as WORD2VEC in 2013 and GloVe in 2014. Researchers can download such a model or train their own relatively quickly without access to supercomputers. Pretrained contextual representations, on the other hand, are orders of magnitude more expensive to train.

These models became feasible only after hardware advances (GPUs and TPUs) became widespread, and in this case researchers were grateful to be able to download models rather than having to spend the resources to train their own. The transformer model allowed for efficient training of much larger and deeper neural networks than was previously possible (this time due to software advances, not hardware). Since 2018, new NLP projects typically start with a pretrained transformer model.

Although these transformer models were trained to predict the next word in a text, they do a surprisingly good job at other language tasks. A ROBERTA model with some fine-tuning achieves state-of-the-art results in question answering and reading comprehension tests (Liu et *al.*, 2019b).

GPT-2, a transformer-like language model with 1.5 billion parameters trained on 40GB of Internet text, achieves good results on such diverse tasks as translation between French and English, finding referents of long-distance dependencies, and general-knowledge question answering, all without fine-tuning for the particular task. As Figure 25.14 illustrates, GPT-2 can generate fairly convincing text given just a few words as a prompt.

As an example state-of-the-art NLP system, ARISTO (Clark *et al.*, 2019) achieved a score of 91.6% on an 8th grade multiple-choice science exam (see Figure 25.13). ARISTO consists of an ensemble of solvers: some use information retrieval (similar to a web search engine), some do textual entailment and qualitative reasoning, and some use large transformer language models. It turns out that ROBERTA, by itself, scores 88.2% on the test. ARISTO also scores 83% on the more advanced 12th grade exam. (A score of 65% is considered “meeting the standards” and 85% is “meeting the standards with distinction”.)

---

1. **What will best separate a mixture of iron filings and black pepper?**
   (a) magnet (b) filter paper (c) triple beam balance (d) voltmeter
2. **Which form of energy is produced when a rubber band vibrates?**
   (a) chemical (b) light (c) electrical (d) sound
3. **Because copper is a metal, it is**
   (a) liquid at room temperature (b) nonreactive with other substances
   (c) a poor conductor of electricity (d) a good conductor of heat
4. **Which process in an apple tree primarily results from cell division?**
   (a) growth (b) photosynthesis (c) gas exchange (d) waste removal

Figure 25.13 Questions from an 8th grade science exam that the ARISTO system can answer correctly using an ensemble of methods, with the most influential being a ROBERT A language model. Answering these questions requires knowledge about natural language, the structure of multiple-choice tests, commonsense, and science.

It is a truth universally acknowledged that the earth is not the center of the universe. There are those who assert there is. I do not accept them, but others I consider to be of the same opinion. The truth is, however, that if there are other than the center, and if there are any other living things in the universe and if they are not human, then we do not yet have our answers. We have to go on. This page gives a simplified, simplified answer to the problem. We don't have all the answers. The truth is, however, that the truth is out there.

When Gregor Samsa woke up one morning, he did not notice anything strange. "When my wife is looking at me, I feel like she is looking at a piece of art," he said. "I think she is admiring something I have created." The idea is that by looking at your own life, you learn something important and become a better person. It is a theory that emerged from psychologist Daniel Goleman's work, in which he asked "How do you know you're not a loser?"

Alice was beginning to get very tired of sitting with her sister on the bank. She sat up, yawned, and said, with a loud little scream, "I hope you don't mind if I keep on doing what I should like to do, and if someone asks me which of us will do more, don't tell them that I won't do much, my dear sister."

All happy families are alike; each happy family is like a garden of paradise. The only difference between happy families and unhappy families, is that the unhappy family doesn't have any flowers or trees.

Tell me a story. Tell me a story. Tell me a story. Tell me a story. Tell me a story. Tell me a story. Tell me a story. Tell me a story. Tell me a story. Tell me a story. Tell me a story. Tell me a story. Please fill out the following details. Thank you... Thank you for your interest in this interview. Please wait...

Figure 25.14 Example completion texts generated by the GPT-2 language model, given the prompts in **bold**. Most of the texts are quite fluent English, at least locally. The final example demonstrates that sometimes the model just breaks down.

There are limitations of ARISTO. It deals only with multiple-choice questions, not essay questions, and it can neither read nor generate diagrams.[1]

T5 (the Text-to-Text Transfer Transformer) is designed to produce textual responses to various kinds of textual input. It includes a standard encoder-decoder transformer model, pretrained on 35 billion words from the 750 GB Colossal Clean Crawled Corpus (C4). This unlabeled training is designed to give the model generalizable linguistic knowledge that will be useful for multiple specific tasks. T5 is then trained for each task with input consisting of the task name, followed by a colon and some content. For example, when given "translate English to German: *That is good,"* it produces as output *"Das ist gut"* For some tasks, the input is marked up; for example in the Winograd Schema Challenge, the input highlights a pronoun with an ambiguous referent. Given the input "referent: *The city councilmen refused the demonstrators a permit because they feared violence,"* the correct response is *"The city councilmen"* (not *"the demonstrators"*).

Much work remains to be done to improve NLP systems. One issue is that transformer models rely on only a narrow context, limited to a few hundred words. Some experimental approaches are trying to extend that context; the Reformer system (Kitaev *et al.*, 2020) can handle context of up to a million words.

Recent results have shown that using more training data results in better models—for example, ROBERTA achieved state-of-the-art results after training on 2.2 trillion words. If using more textual data is better, what would happen if we included other types of data: structured databases, numerical data, images, and video? We would need a breakthrough in hardware processing speeds to train on a large corpus of video, and we may need several breakthroughs in AI as well.

The curious reader may wonder why we learned about grammars, parsing, and semantic interpretation in the previous chapter, only to discard those notions in favor of purely data-driven models in this chapter? At present, the answer is simply that the data-driven models are easier to develop and maintain, and score better on standard benchmarks, compared to the hand-built systems that can be constructed using a reasonable amount of human effort with the approaches described in Chapter 24. It may be that transformer models and their relatives are learning latent representations that capture the same basic ideas as grammars and semantic information, or it may be that something entirely different is happening within these enormous models; we simply don't know. We do know that a system that is trained with textual data is easier to maintain and to adapt to new domains and new natural languages than a system that relies on hand-crafted features.

It may also be the case that future breakthroughs in explicit grammatical and semantic modeling will cause the pendulum to swing back. Perhaps more likely is the emergence of hybrid approaches that combine the best concepts from both chapters. For example, Kitaev and Klein (2018) used an attention mechanism to improve a traditional constituency parser, achieving the best result ever recorded on the Penn Treebank test set. Similarly, Ringgaard *et al.* (2017) demonstrate how a dependency parser can be improved with word embeddings and a recurrent neural network. Their system, SLING, parses directly into a semantic frame representation, mitigating the problem of errors building up in a traditional pipeline system.

There is certainly room for improvement: not only do NLP systems still lag human performance on many tasks, but they do so after processing thousands of times more text than any human could read in a lifetime. This suggests that there is plenty of scope for new insights from linguists, psychologists, and NLP researchers.

Summary

The key points of this chapter are as follows:

- Continuous representations of words with word embeddings are more robust than discrete atomic representations, and can be pretrained using unlabeled text data.
- Recurrent neural networks can effectively model local and long-distance context by retaining relevant information in their hidden-state vectors.
- Sequence-to-sequence models can be used for machine translation and text generation problems.
- Transformer models use self-attention and can model long-distance context as well as local context. They can make effective use of hardware matrix multiplication.
- Transfer learning that includes pretrained contextual word embeddings allows models to be developed from very large unlabeled corpora and applied to a range of tasks. Models that are pretrained to predict missing words can handle other tasks such as question answering and textual entailment, after fine-tuning for the target domain.

Bibliographical and Historical Notes

The distribution of words and phrases in natural language follow **Zipf's Law** (Zipf, 1935, 1949): the frequency of the nth most popular word is roughly inversely proportional to n. That means we have a data sparsity problem: even with billions of words of training data, we are constantly running into novel words and phrases that were not seen before.

Generalization to novel words and phrases is aided by representations that capture the basic insight that words with similar meanings appear in similar contexts. Deerwester *et al.*(1990) projected words into low-dimensional vectors by decomposing the co-occurrence matrix formed by words and the documents the words appear in. Another possibility is to treat the surrounding words—say, a 5-word window—as context. Brown et *al.* (1992) grouped words into hierarchical clusters according to the bigram context of words; this has proven to be effective for tasks such as named entity recognition (Turian et *al.,* 2010). The WORD2VEC system (Mikolov *et al.,* 2013) was the first significant demonstration of the advantages of word embeddings obtained from training neural networks. The GloVe word embedding vectors (Pennington et *al.*, 2014) were obtained by operating directly on a word co-occurrence matrix obtained from billions of words of text. Levy and Goldberg (2014) explain why and how these word embeddings are able to capture linguistic regularities.

Bengio *et al.* (2003) pioneered the use of neural networks for language models, proposing to combine "(1) a distributed representation for each word along with (2) the probability function for word sequences, expressed in terms of these representations." Mikolov *et al.* (2010) demonstrated the use of **RNN**s for modeling local context in language models. Jozefowicz *et* *al.* (2016) showed how an **RNN** trained on a billion words can outperform carefully handcrafted *n*-gram models. Contextual representations for words were emphasized by Peters *et al.* (2018), who called them ELMO (Embeddings from Language Models) representations.

Note that some authors compare language models by measuring their **perplexity**. The perplexity of a probability distribution is 2^H, where H is the entropy of the distribution (see Section 19.3.3). A language model with lower perplexity is, all other things being equal, a better model. But in practice, all other things are rarely equal. Therefore it is more informative to measure performance on a real task rather than relying on perplexity.

Howard and Ruder (2018) describe the ULMFIT (Universal Language Model Finetuning) framework, which makes it easier to fine-tune a pretrained language model without requiring a vast corpus of target-domain documents. Ruder *et al.* (2019) give a tutorial on transfer learning for NLP.

Mikolov *et al.* (2010) introduced the idea of using **RNN**s for NLP, and Sutskever *et al.* (2015) introduced the idea of sequence to sequence learning with deep networks. Zhu *et al.* (2017) and (Liu *et al.*, 2018b) showed that an unsupervised approach works, and makes data collection much easier. It was soon found that these kinds of models could perform surprisingly well at a variety of tasks, for example, image captioning (Karpathy and Fei-Fei, 2015; Vinyals *et al.*, 2017b).

Devlin *et al.* (2018) showed that transformer models pretrained with the masked language modeling objective can be directly used for multiple tasks. The model was called BERT (Bidirectional Encoder Representations from Transformers). Pretrained BERT models can be fine-tuned for particular domains and particular tasks, including question answering, named entity recognition, text classification, sentiment analysis, and natural language inference.

The XLNET system (Yang *et al.,* 2019) improves on BERT by eliminating a discrepancy between the pretraining and fine-tuning. The ERNIE 2.0 framework (Sun *et al.,* 2019) extracts more from the training data by considering sentence order and the presence of named entities, rather than just co-occurrence of words, and was shown to outperform BERT and XLNET. In response, researchers revisited and improved on BERT: the ROBERTA system (Liu *et al.*, 2019b) used more data and different hyperparameters and training procedures, and found that it could match XLNET. The Reformer system (Kitaev *et al.,* 2020) extends the range of the context that can be considered all the way up to a million words. Meanwhile, ALBERT (A Lite BERT) went in the other direction, reducing the number of parameters from 108 million to 12 million (so as to fit on mobile devices) while maintaining high accuracy.

The XLM system (Lample and Conneau, 2019) is a transformer model with training data from multiple languages. This is useful for machine translation, but also provides more robust representations for monolingual tasks. Two other important systems, GPT-2 (Radford *et al.,* 2019) and T5 (Raffel *et al.,* 2019), were described in the chapter. The later paper also introduced the 35 billion word Colossal Clean Crawled Corpus (C4).

Various promising improvements on pretraining algorithms have been proposed (Yang *et al.,* 2019; Liu et *al.,* 2019b). Pretrained contextual models are described by Peters et *al.* (2018) and Dai and Le (2016).

The GLUE (General Language Understanding Evaluation) benchmark, a collection of tasks and tools for evaluating NLP systems, was introduced by Wang *et al.* (2018a). Tasks include question answering, sentiment analysis, textual entailment, translation, and parsing. Transformer models have so dominated the leaderboard (the human baseline is way down at ninth place) that a new version, SUPERGLUE (Wang *et al.,* 2019), was introduced with tasks that are designed to be harder for computers, but still easy for humans.

At the end of 2019, T5 was the overall leader with a score of 89.3, just half a point below the human baseline of 89.8. On three of the ten tasks, T5 actually exceeds human performance: yes/no question answering (such as “Is France the same time zone as the UK?”) and two reading comprehension tasks involving answering questions after reading either a paragraph or a news article.

Machine translation is a major application of language models. In 1933, Petr Troyanskii received a patent for a “translating machine,” but there were no computers available to implement his ideas. In 1947, Warren Weaver, drawing on work in cryptography and information theory, wrote to Norbert Wiener: “When I look at an article in Russian, I say: ‘This is really written in English, but it has been coded in strange symbols. I will now proceed to decode.”’ The community proceeded to try to decode in this way, but they didn’t have sufficient data and computing resources to make the approach practical.

In the 1970s that began to change, and the SYSTRAN system (Toma, 1977) was the first commercially successful machine translation system. SYSTRAN relied on lexical and grammatical rules hand-crafted by linguists as well as on training data. In the 1980s, the community embraced purely statistical models based on frequency of words and phrases (Brown *et al.*, 1988; Koehn, 2009). Once training sets reached billions or trillions of tokens (Brants *et al.,* 2007), this yielded systems that produced comprehensible but not fluent results (Och and Ney, 2004; Zollmann *et al.*, 2008). Och and Ney (2002) show how discriminative training led to an advance in machine translation in the early 2000s.

Sutskever *et al.* (2015) first showed that it is possible to learn an end-to-end sequence-to-sequence neural model for machine translation. Bahdanau *et al.* (2015) demonstrated the advantage of a model that jointly learns to align sentences in the source and target language and to translate between the languages. Vaswani *et al.* (2018) showed that neural machine translation systems can further be improved by replacing LSTMs with transformer architectures, which use the attention mechanism to capture context. These neural translation systems quickly overtook statistical phrase-based methods, and the transformer architecture soon spread to other NLP tasks.

Research on **question answering** was facilitated by the creation of SQUAD, the first large-scale data set for training and testing question-answering systems (Rajpurkar *et al.*,2016). Since then, a number of deep learning models have been developed for this task (Seo *et al.*, 2017; Keskar *et al.*, 2019). The ARISTO system (Clark *et al.*, 2019) uses deep learning in conjunction with an ensemble of other tactics. Since 2018, the majority of questionanswering models use pretrained language representations, leading to a noticeable improvement over earlier systems.

Natural language inference is the task of judging whether a hypothesis *(dogs need to eat)* is entailed by a premise *(all animals need to eat)*. This task was popularized by the PASCAL Challenge (Dagan *et al.*, 2005). Large-scale data sets are now available (Bowman *et al.*, 2015; Williams *et al.*, 2018). Systems based on pretrained models such as ELMO and BERT currently provide the best performance on language inference tasks.

The Conference on Computational Natural Language Learning (CONLL) focuses on learning for NLP. All the conferences and journals mentioned in

Chapter 24 now include papers on deep learning, which now has a dominant position in the field of NLP.

[1] It has been pointed out that in some multiple-choice exams, it is possible to get a good score even without looking at the questions, because there are tell-tale signs in the incorrect answers (Gururangan *et al*., 2018). That seems to be true for visual question answering as well (Chao *et al*., 2018)

CHAPTER 26

ROBOTICS

In which agents are endowed with sensors and physical effectors with which to move about and make mischief in the real world.

26.1 Robots

Robots are physical agents that perform tasks by manipulating the physical world. To do so, they are equipped with **effectors** such as legs, wheels, joints, and grippers. Effectors are designed to assert physical forces on the environment. When they do this, a few things may happen: the robot's state might change (e.g., a car spins its wheels and makes progress on the road as a result), the state of the environment might change (e.g., a robot arm uses its gripper to push a mug across the counter), and even the state of the people around the robot might change (e.g., an exoskeleton moves and that changes the configuration of a person's leg; or a mobile robot makes progress toward the elevator doors, and a person notices and is nice enough to move out of the way, or even push the button for the robot).

Robots are also equipped with **sensors**, which enable them to perceive their environment. Present-day robotics employs a diverse set of sensors, including cameras, radars, lasers, and microphones to measure the state of the environment and of the people around it; and gyroscopes, strain and torque sensors, and accelerometers to measure the robot's own state.

Maximizing expected utility for a robot means choosing how to actuate its effectors to assert the *right* physical forces—the ones that will lead to changes in state that accumulate as much expected reward as possible. Ultimately, robots are trying to accomplish some task in the physical world.

Robots operate in environments that are partially observable and stochastic: cameras cannot see around corners, and gears can slip. Moreover, the people acting in that same environment are unpredictable, so the robot needs to make predictions about them.

Robots usually model their environment with a continuous state space (the robot's position has continuous coordinates) and a continuous action space (the amount of current a robot sends to its motor is also measured in continuous units). Some robots operate in highdimensional spaces: cars need to know the position, orientation, and velocity of themselves and the nearby agents; robot arms have six or seven joints that can each be independently moved; and robots that mimic the human body have hundreds of joints.

Robotic learning is constrained because the real world stubbornly refuses to operate faster than real time. In a simulated environment, it is possible to use learning algorithms (such as the Q-learning algorithm described in Chapter 23) to learn in a few hours from millions of trials. In a real environment, it might take years to run these trials, and the robot cannot risk (and thus cannot learn from) a trial that might cause harm. Thus, transferring what has been learned in simulation to a real robot in the real world—the **sim-to-real** problem—is an active area of research. Practical robotic systems need to embody prior knowledge about the robot, the physical environment, and the tasks to be performed so that the robot can learn quickly and perform safely.

Robotics brings together many of the concepts we have seen in this book, including probabilistic state estimation, perception, planning, unsupervised learning, reinforcement learning, and game theory. For some of these concepts robotics serves as a challenging example application. For other concepts this chapter breaks new ground, for instance in introducing the continuous version of techniques that we previously saw only in the discrete case.

26.2 Robot Hardware

So far in this book, we have taken the agent architecture—sensors, effectors, and processors—as given, and have concentrated on the agent program. But the success of real robots depends at least as much on the design of sensors and effectors that are appropriate for the task.

26.2.1 Types of robots from the hardware perspective

When you think of a robot, you might imagine something with a head and two arms, moving around on legs or wheels. Such **anthropomorphic robots** have been popularized in fiction such as the movie *The Terminator* and the cartoon *The Jetsons*. But real robots come in many shapes and sizes.

Manipulators are just robot arms. They do not necessarily have to be attached to a robot body; they might simply be bolted onto a table or a floor, as they are in factories (Figure 26.1 (a)). Some have a large payload, like those assembling cars, while others, like wheelchair-mountable arms that assist people with motor impairments (Figure 26.1(b)), can carry less but are safer in human environments.

(a) (b)

Figure 26.1 (a) An industrial robotic arm with a custom end-effector. Image credit: Macor/123RF. (b) A Kinova® JACO® Assistive Robot arm mounted on a wheelchair. Kinova and JACO are trademarks of Kinova, Inc.

Mobile robots are those that use wheels, legs, or rotors to move about the environment. **Quadcopter drones** are a type of **unmanned aerial vehicle (UAV)**; **autonomous underwater vehicles (AUVs)** roam the oceans. But many mobile robots stay indoors and move on wheels, like a vacuum cleaner or a towel delivery robot in a hotel. Their outdoor counterparts include **autonomous cars** or **rovers** that explore new terrain, even on the surface of Mars (Figure 26.2). Finally, **legged robots** are meant to traverse rough terrain that is inaccessible with wheels. The downside is that controlling legs to do the right thing is more challenging than spinning wheels.

(a) (b)

Figure 26.2 (a) NASA's Curiosity rover taking a selfie on Mars. Image courtesy of NASA. (b) A Skydio drone accompanying a family on a bike ride. Image courtesy of Skydio.

Other kinds of robots include prostheses, exoskeletons, robots with wings, swarms, and intelligent environments in which the robot is the entire room.

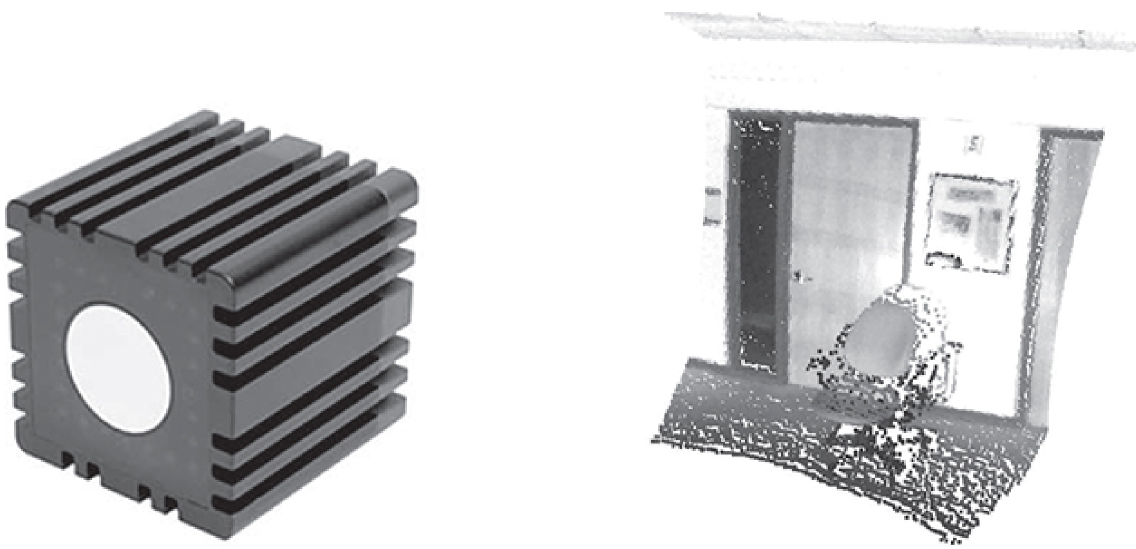
26.2.2 Sensing the world

Sensors are the perceptual interface between robot and environment. **Passive sensors**, such as cameras, are true observers of the environment: they capture signals that are generated by other sources in the environment. **Active sensors**, such as sonar, send energy into the environment. They rely on the fact that this energy is reflected back to the sensor. Active sensors tend to provide more information than passive sensors, but at the expense of increased power consumption and with a danger of interference when multiple active sensors are used at the same time. We also distinguish whether a sensor is directed at sensing the environment, the robot's location, or the robot's internal configuration.

Range finders are sensors that measure the distance to nearby objects. **Sonar** sensors are active range finders that emit directional sound waves, which are reflected by objects, with some of the sound making it back to the sensor. The time and intensity of the returning signal indicates the distance to nearby objects. Sonar is the technology of choice for autonomous underwater vehicles, and was popular in the early days of indoor robotics. **Stereo vision** (see Section 27.6) relies on multiple cameras to image the environment from slightly different viewpoints, analyzing the resulting parallax in these images to compute the range of surrounding objects.

For mobile ground robots, sonar and stereo vision are now rarely used, because they are not reliably accurate. The Kinect is a popular low-cost sensor that combines a camera and a **structured light** projector, which projects a pattern of grid lines onto a scene. The camera sees how the grid lines bend, giving the robot information about the shape of the objects in the scene. If desired, the projection can be infrared light, so as not to interfere with other sensors (such as human eyes).

Most ground robots are now equipped with active optical range finders. Just like sonar sensors, optical range sensors emit active signals (light) and measure the time until a reflection of this signal arrives back at the sensor. Figure 26.3(a) shows a **time-of-flight camera**. This camera acquires range images like the one shown in Figure 26.3(b) at up to 60 frames per second. Autonomous cars often use **scanning lidars** (short for *light detection and ranging*)—active sensors that emit laser beams and sense the reflected beam, giving range measurements accurate to within a centimeter at a range of 100 meters. They use complex arrangements of mirrors or rotating elements to sweep the beam across the environment and build a map. Scanning lidars tend to work better than time-of-flight cameras at longer ranges, and tend to perform better in bright daylight.



(a) (b)

Figure 26.3 (a) Time-of-flight camera; image courtesy of Mesa Imaging GmbH. (b) 3D range image obtained with this camera. The range image makes it possible to detect obstacles and objects in a robot's vicinity. Image courtesy of Willow Garage, LLC.

Radar is often the range finding sensor of choice for air vehicles (autonomous or not). Radar sensors can measure distances up to kilometers, and have an advantage over optical sensors in that they can see through fog. On the close end of range sensing are **tactile sensors** such as whiskers, bump panels, and touch-sensitive skin. These sensors measure range based on physical contact, and can be deployed only for sensing objects very close to the robot.

A second important class is **location sensors**. Most location sensors use range sensing as a primary component to determine location. Outdoors, the **Global Positioning System** (GPS) is the most common solution to the localization problem. GPS measures the distance to satellites that emit pulsed signals. At present, there are 31 operational GPS satellites in orbit, and 24 GLONASS satellites, the Russian counterpart. GPS receivers can recover the distance to a satellite by analyzing phase shifts. By triangulating signals from multiple satellites, GPS receivers can determine their absolute location on Earth to within a few meters. **Differential GPS** involves a second ground receiver with known location, providing millimeter accuracy under ideal conditions.

Unfortunately, GPS does not work indoors or underwater. Indoors, localization is often achieved by attaching beacons in the environment at known locations. Many indoor environments are full of wireless base stations, which can help robots localize through the analysis of the wireless signal. Underwater, active sonar beacons can provide a sense of location, using sound to inform AUVs of their relative distances to those beacons.

The third important class is **proprioceptive sensors**, which inform the robot of its own motion. To measure the exact configuration of a robotic joint, motors are often equipped with **shaft decoders** that accurately measure the angular motion of a shaft. On robot arms, shaft decoders help track the position of joints. On mobile robots, shaft decoders report wheel revolutions for **odometry**—the measurement of distance traveled. Unfortunately, wheels tend to drift and slip, so odometry is accurate only over short distances. External forces, such as wind and ocean currents, increase positional uncertainty. **Inertial sensors**, such as gyroscopes, reduce uncertainty by relying on the resistance of mass to the change of velocity.

Other important aspects of robot state are measured by **force sensors** and **torque sensors**. These are indispensable when robots handle fragile objects or objects whose exact size and shape are unknown. Imagine a one-ton robotic manipulator screwing in a light bulb. It would be all too easy to apply too much force and break the bulb. Force sensors allow the robot to sense how hard it is gripping the bulb, and torque sensors allow it to sense how hard it is turning. High-quality sensors can measure forces in all three translational and three rotational directions. They do this at a frequency of several hundred times a second so that a robot can quickly detect unexpected forces and correct its actions before it breaks a light bulb. However, it can be a challenge to outfit a robot with high-end sensors and the computational power to monitor them.

26.2.3 Producing motion

The mechanism that initiates the motion of an effector is called an **actuator**; examples include transmissions, gears, cables, and linkages. The most common type of actuator is the **electric actuator**, which uses electricity to spin up a motor. These are predominantly used in systems that need rotational motion, like joints on a robot arm. **Hydraulic actuators** use pressurized hydraulic fluid (like oil or water) and **pneumatic actuators** use compressed air to generate mechanical motion.

Actuators are often used to move joints, which connect rigid bodies (links). Arms and legs have such joints. In **revolute joints**, one link rotates with respect to the other. In **prismatic joints**, one link slides along the other. Both of these are single-axis joints (one axis of motion). Other kinds of joints include spherical, cylindrical, and planar joints, which are multi-axis joints.

To interact with objects in the environment, robots use grippers. The most basic type of gripper is the **parallel jaw gripper**, with two fingers and a single actuator that moves the fingers together to grasp objects. This effector is both loved and hated for its simplicity. Three-fingered grippers offer slightly more flexibility while maintaining simplicity. At the other end of the spectrum are humanoid (anthropomorphic) hands. For instance, the Shadow Dexterous Hand has a total of 20 actuators. This offers a lot more flexibility for complex manipulation, including in-hand manipulator maneuvers (think of picking up your cell phone and rotating it in-hand to orient it right-side up), but this flexibility comes at a price—learning to control these complex grippers is more challenging.

26.3 What kind of problem is robotics solving?

Now that we know what the robot hardware might be, we're ready to consider the agent software that drives the hardware to achieve our goals. We first need to decide the computational framework for this agent. We have talked about search in deterministic environments, MDPs for stochastic but fully observable environments, POMDPs for partial observability, and games for situations in which the agent is not acting in isolation. Given a computational framework, we need to instantiate its ingredients: reward or utility functions, states, actions, observation spaces, etc.

We have already noted that robotics problems are nondeterministic, partially observable, and multiagent. Using the game-theoretic notions from Chapter 17, we can see that sometimes the agents are cooperative and sometimes they are competitive. In a narrow corridor where only one agent can go first, a robot and a person collaborate because they both want to make sure they don't bump into each other. But in some cases they might compete a bit to reach their destination quickly. If the robot is too polite and always makes room, it might get stuck in crowded situations and never reach its goal.

Therefore, when robots act in isolation and know their environment, the problem they are solving can be formulated as an MDP; when they are missing information it becomes a POMDP; and when they act around people it can often be formulated as a game.

What is the robot's reward function in this formulation? Usually the robot is acting in service of a human—for example delivering a meal to a hospital patient for the patient's reward, not its own. For most robotics settings, even though robot designers might try to specify a good enough proxy reward function, the true reward function lies with the user whom the robot is supposed to help. The robot will either need to decipher the user's desires, or rely on an engineer to specify an approximation of the user's desires.

As for the robot's action, state, and observation spaces, the most general form is that observations are raw sensor feeds (e.g., the images coming in from cameras, or the laser hits coming in from lidar); actions are raw electric currents being sent to the motors; and state is what the robot needs to know for its decision making. This means there is a huge gap between the low-level percepts and motor controls, and the high-level plans the robot needs to make. To bridge the gap, roboticists decouple aspects of the problem to simplify it.

For instance, we know that when we solve POMDPs properly, perception and action interact: perception informs which actions make sense, but action also informs perception, with agents taking actions to gather information when that information has value in later time steps. However, robots often separate perception from action, consuming the outputs of perception and pretending they will not get any more information in the future. Further, hierarchical planning is called for, because a high-level goal like “get to the cafeteria” is far removed from a motor command like “rotate the main axle 1°,”

In robotics we often use a three-level hierarchy. The **task planning** level decides a plan or policy for high-level actions, sometimes called action primitives or subgoals: move to the door, open it, go to the elevator, press the button, etc. Then **motion planning** is in charge of finding a path that gets the robot from one point to another, achieving each subgoal. Finally, **control** is used to achieve the planned motion using the robot's actuators. Since the task planning level is typically defined over discrete states and actions, in this chapter we will focus primarily on motion planning and control.

Separately, **preference learning** is in charge of estimating an end user's objective, and **people prediction** is used to forecast the actions of other people in the robot's environment. All these combine to determine the robot's behavior.

Whenever we split a problem into separate pieces we reduce complexity, but we give up opportunities for the pieces to help each other. Action can help improve perception, and also determine what kind of perception is useful. Similarly, decisions at the motion level might not be the best when accounting for how that motion will be tracked; or decisions at the task level might render the task plan uninstantiatable at the motion level. So, with progress in these separate areas comes the push to reintegrate them: to do motion planning and control together, to do task and motion planning together, and to reintegrate perception, prediction, and action—closing the feedback loop. Robotics today is about continuing progress in each area while also building on this progress to achieve better integration.

26.4 Robotic Perception

Perception is the process by which robots map sensor measurements into internal representations of the environment. Much of it uses the computer vision techniques from the previous chapter. But perception for robotics must deal with additional sensors like lidar and tactile sensors.

Perception is difficult because sensors are noisy and the environment is partially observable, unpredictable, and often dynamic. In other words, robots have all the problems of **state estimation** (or **filtering**) that we discussed in Section 14.2. As a rule of thumb, good internal representations for robots have three properties:

1. They contain enough information for the robot to make good decisions.
2. They are structured so that they can be updated efficiently.
3. They are natural in the sense that internal variables correspond to natural state variables in the physical world.

In Chapter 14, we saw that Kalman filters, HMMs, and dynamic Bayes nets can represent the transition and sensor models of a partially observable environment, and we described both exact and approximate algorithms for updating the **belief state**—the posterior probability distribution over the environment state variables. Several dynamic Bayes net models for this process were shown in Chapter 14. For robotics problems, we include the robot's own past actions as observed variables in the model. Figure 26.4 shows the notation used in this chapter: $\mathbf{X}_t$ is the state of the environment (including the robot) at time t, $\mathbf{Z}_t$ is the observation received at time t, and A_t is the action taken after the observation is received.

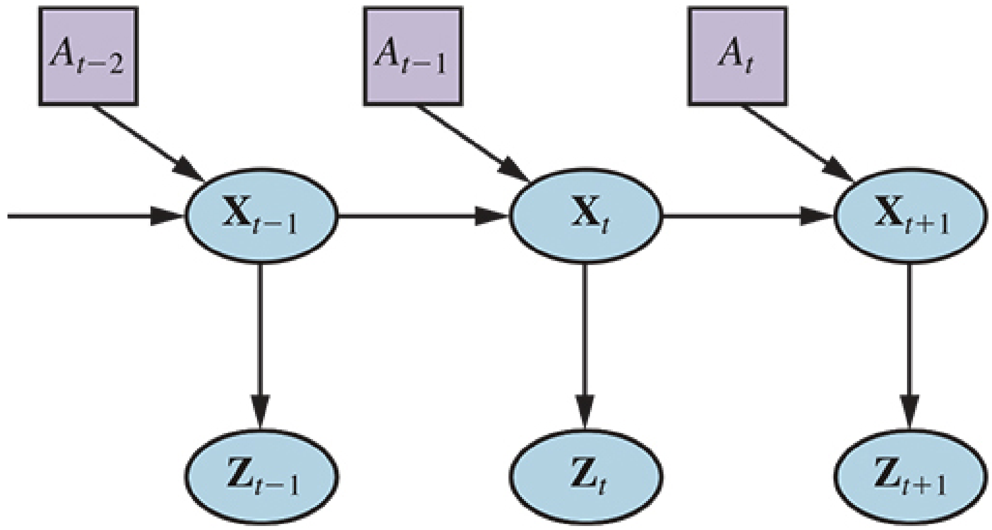


Figure 26.4 Robot perception can be viewed as temporal inference from sequences of actions and measurements, as illustrated by this dynamic decision network.

We would like to compute the new belief state, $\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{z}_{1:t+1}, a_{1:t})$, from the current belief state, $\mathbf{P}(\mathbf{X}_t \mid \mathbf{z}_{1:t}, a_{1:t-1})$, and the new observation $\mathbf{z}_{t+1}$. We did this in Section 14.2, but there are two differences here: we condition on the actions as well as the observations, and we deal with *continuous* rather than *discrete* variables. Thus, we modify the recursive filtering equation (14.5 on page 485) to use integration rather than summation:

$$\begin{aligned}
&\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{z}_{1:t}+1, a_{1:t}) \\
&\quad = \alpha \mathbf{P}(\mathbf{z}_{t+1} \ \mathbf{X}_{t+1}) \int \mathbf{P}(\mathbf{X}_{t+1}|\mathbf{x}_i, a_t) P(\mathbf{x}_t|\mathbf{z}_{1:t}, a_{1:t-1}) d\mathbf{x}_t.
\end{aligned} \tag{26.1}$$

This equation states that the posterior over the state variables $\mathbf{X}$ at time $t + 1$ is calculated recursively from the corresponding estimate one time step earlier.

This calculation involves the previous action a_t and the current sensor measurement $\mathbf{z}_{t+1}$. For example, if our goal is to develop a soccer-playing robot, $\mathbf{X}_{t+1}$ might include the location of the soccer ball relative to the robot. The posterior $\mathbf{P}(\mathbf{X}_t \mid \mathbf{z}_{1:t}, a_{1:t-1})$ is a probability distribution over all states that captures what we know from past sensor measurements and controls. Equation (26.1) tells us how to recursively estimate this location, by incrementally folding in sensor measurements (e.g., camera images) and robot motion commands. The probability $\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{x}_t, a_t)$ is called the **transition model** or **motion model**, and $\mathbf{P}(\mathbf{z}_{t+1} \mid \mathbf{X}_{t+1})$ is the **sensor model**.

26.4.1 Localization and mapping

Localization is the problem of finding out where things are—including the robot itself. To keep things simple, let us consider a mobile robot that moves slowly in a flat two-dimensional world. Let us also assume the robot is given an exact map of the environment. (An example of such a map appears in Figure 26.7.) The pose of such a mobile robot is defined by its two Cartesian coordinates with values x and y and its heading with value θ, as illustrated in Figure 26.5(a). If we arrange those three values in a vector, then any particular state is given by $\mathbf{X}_t = (x_t, y_t, \theta_t)^\top$. So far so good.

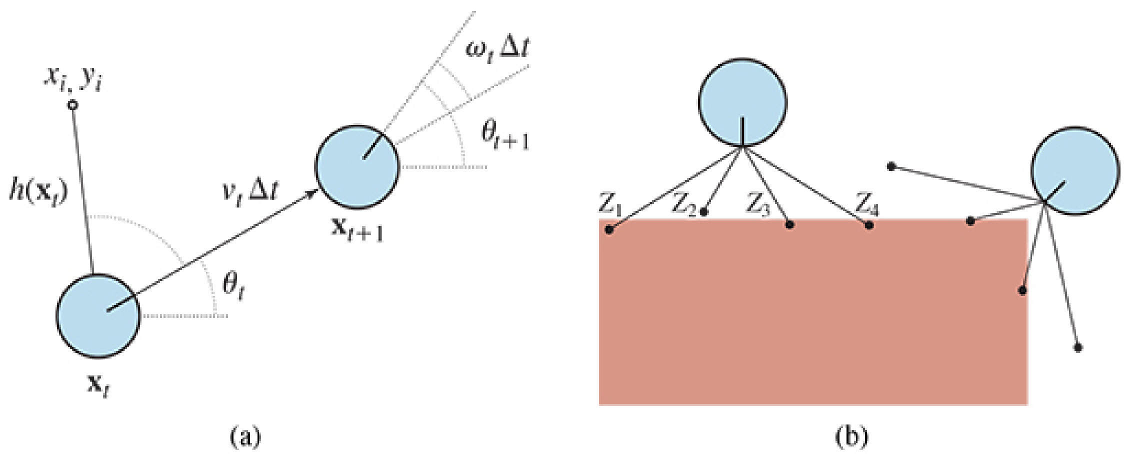


Figure 26.5 (a) A simplified kinematic model of a mobile robot. The robot is shown as a circle with an interior radius line marking the forward direction. The state $\mathbf{x}_t$ consists of the (x_t, y_t) position (shown implicitly) and the orientation θ_t. The new state $\mathbf{x}_{t+1}$ is obtained by an update in position of $v_t\,\Delta_t$ and in orientation of $\omega_t\Delta_t$. Also shown is a landmark at $(x_i\ , y_i)$ observed at time t. (b) The range-scan sensor model. Two possible robot poses are shown for a given range scan (z_1, z_2, z_3, z_4). It is much more likely that the pose on the left generated the range scan than the pose on the right.

In the kinematic approximation, each action consists of the instantaneous specification of two velocities—a translational velocity v_t and a rotational velocity ω_t. For small time intervals Δt, a crude deterministic model of the motion of such robots is given by

$$\hat{\mathbf{X}}_{t+1} = f(\mathbf{X}_t, \underbrace{v_t, \omega_t}_{a_t}) = \mathbf{X}_t + \begin{pmatrix} v_t \Delta t \cos\theta_t \\ v_t \Delta t \sin\theta_t \\ \omega_t \Delta_t \end{pmatrix} .$$

The notation $\widehat{\mathbf{X}}$ refers to a deterministic state prediction. Of course, physical robots are somewhat unpredictable. This is commonly modeled by a Gaussian distribution with mean $f(\mathbf{X}_t, v_t, \omega_t)$ and covariance $\sum_x$. (See Appendix A for a mathematical definition.)

$$\mathbf{P}(\mathbf{X}_{t+1}\ \mathbf{X}_t, v_t, \omega_t) = N(\widehat{\mathbf{X}}_{t+1}, \textstyle\sum_x).$$

This probability distribution is the robot's motion model. It models the effects of the motion a_t on the location of the robot.

Next, we need a sensor model. We will consider two kinds of sensor models. The first assumes that the sensors detect *stable, recognizable* features of the environment called **landmarks**. For each landmark, the range and bearing are reported. Suppose the robot's state is $\mathbf{x}_t = (x_t, y_t, \theta_t)^\top$ and it senses a landmark whose location is known to be $(x_i, y_i)^\top$. Without noise, a prediction of the range and bearing can be calculated by simple geometry (see Figure 26.5(a)):

$$\widehat{\mathbf{z}}_t = h(\mathbf{x}_t) = \begin{array}{c} \overline{(x_t - x_i)^2 + (y_t - y_i)^2} \\ \arctan\frac{y_i - y_t}{x_i - x_t} - \theta_t \end{array} .$$

Again, noise distorts our measurements. To keep things simple, assume Gaussian noise with covariance $\sum_z$ giving us the sensor model

$$P(\mathbf{z}_t | \mathbf{x}_t) = N(\widehat{\mathbf{z}}_t, \textstyle\sum_z).$$

A somewhat different sensor model is used for a **sensor array** of range sensors, each of which has a fixed bearing relative to the robot. Such sensors produce a vector of range values $\mathbf{z}_t = (z_1, \ldots, z_M)^\top$

Given a pose $\mathbf{x}_t$, let $\hat{z}_j$ be the computed range along the jth beam direction from $\mathbf{x}_t$ to the nearest obstacle. As before, this will be corrupted by Gaussian noise. Typically, we assume that the errors for the different beam directions are independent and identically distributed, so we have

$$P(\mathbf{z}_t \;\; \mathbf{x}_t) = \alpha \prod_{j=1}^{M} e^{-(z_j - \hat{z}_j)/2\sigma^2}.$$

Figure 26.5(b) shows an example of a four-beam range scan and two possible robot poses, one of which is reasonably likely to have produced the observed scan and one of which is not. Comparing the range-scan model to the landmark model, we see that the range-scan model has the advantage that there is no need to *identify* a landmark before the range scan can be interpreted; indeed, in Figure 26.5(b), the robot faces a featureless wall. On the other hand, if there *are* visible, identifiable landmarks, they may provide instant localization.

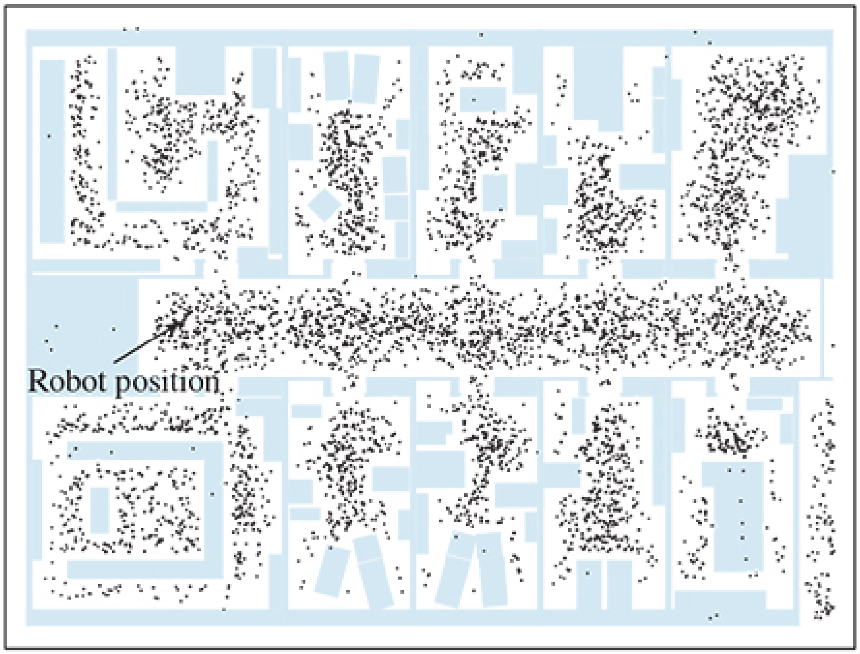
Section 14.4 described the Kalman filter, which represents the belief state as a single multivariate Gaussian, and the particle filter, which represents the belief state by a collection of particles that correspond to states. Most modern localization algorithms use one of these two representations of the robot's belief $\mathbf{P}(\mathbf{X}_t \mid \mathbf{z}_{1:t}, a_{1:t-1})$.

Localization using particle filtering is called **Monte Carlo localization**, or MCL. The MCL algorithm is an instance of the particle-filtering algorithm of Figure 14.17 (page 510). All we need to do is supply the appropriate motion model and sensor model. Figure 26.6 shows one version using the range-scan sensor model. The operation of the algorithm is illustrated in Figure 26.7 as the robot finds out where it is inside an office building. In the first image, the particles are uniformly distributed based on the prior, indicating global uncertainty about the robot's position. In the second image, the first set of measurements arrives and the particles form clusters in the areas of high posterior belief. In the third, enough measurements are available to push all the particles to a single location.

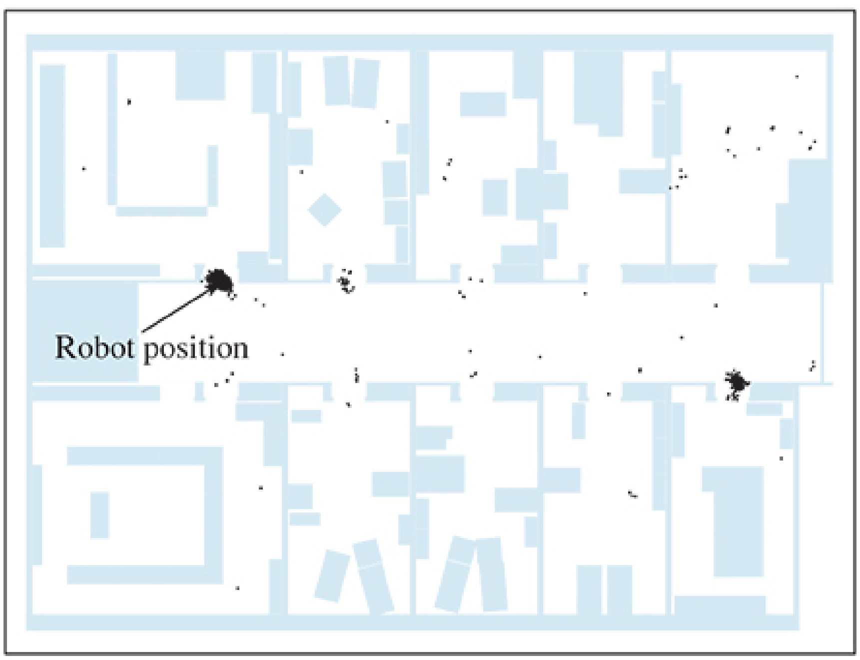
```
function MONTE-CARLO-LOCALIZATION(a, z, N, P(X′|X, v, ω), P(z|z*), map)
  returns a set of samples, S, for the next time step
  inputs: a, robot velocities v and ω
          z, a vector of M range scan data points
          P(X′|X, v, ω), motion model
          P(z|z*), a range sensor noise model
          map, a 2D map of the environment
  persistent: S, a vector of N samples
  local variables: W, a vector of N weights
                   S′, a temporary vector of N samples

  if S is empty then
     for i = 1 to N do        // initialization phase
        S[i] ← sample from P(X_0)
  for i = 1 to N do           // update cycle
     S′[i] ← sample from P(X′|X = S[i], v, ω)
     W[i] ← 1
     for j = 1 to M do
        z* ← RAYCAST(j, X = S′[i], map)
        W[i] ← W[i] · P(z_j| z*)
     S ← WEIGHTED-SAMPLE-WITH-REPLACEMENT(N, S′, W)
  return S
```

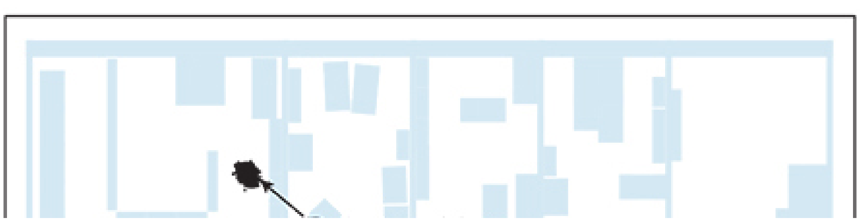
Figure 26.6 A Monte Carlo localization algorithm using a range-scan sensor model with independent noise.



(a)



(b)



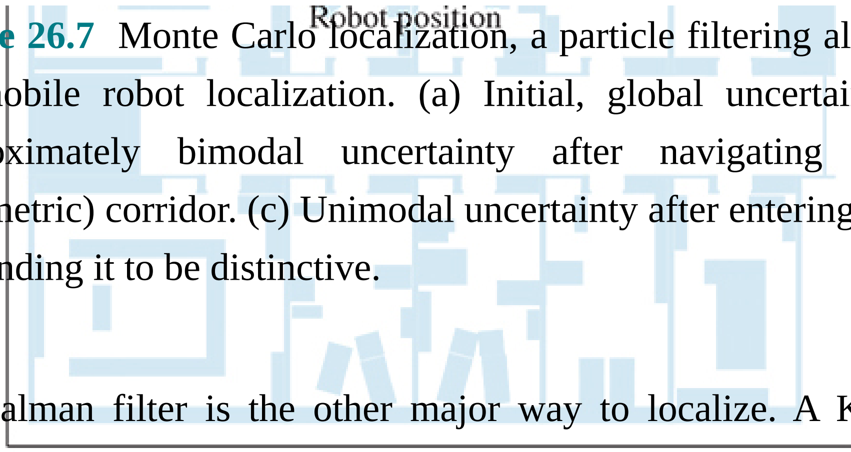


Figure 26.7 Monte Carlo localization, a particle filtering algorithm for mobile robot localization. (a) Initial, global uncertainty. (b) Approximately bimodal uncertainty after navigating in the (symmetric) corridor. (c) Unimodal uncertainty after entering a room and finding it to be distinctive.

The Kalman filter is the other major way to localize. A Kalman filter represents the posterior $\mathbf{P}(\mathbf{X}_t \mid \mathbf{z}_{1:t}, a_{1:t-1})$ by a Gaussian. The mean of this Gaussian will be denoted and its covariance Σ_t. The main problem with Gaussian beliefs is that they are closed only under linear motion models f and linear measurement models h. For nonlinear f or h, the result of updating a filter is in general not Gaussian. Thus, localization algorithms using the Kalman filter linearize the motion and sensor models. Linearization is a local approximation of a nonlinear function by a linear function. Figure 26.8 illustrates the concept of linearization for a (one-dimensional) robot motion model. On the left, it depicts a nonlinear motion model $f(\mathbf{x}_t, a_t)$ (the control a_t is omitted in this graph since it plays no role in the linearization). On the right, this function is approximated by a linear function $\tilde{f}(\mathbf{x}_t, a_t)$. This linear function is tangent to f at the point μ_t, the mean of our state estimate at time t. Such a linearization is called first degree **Taylor expansion**. A Kalman filter that linearizes f and h via **Taylor expansion** is called an **extended Kalman filter** (or EKF). Figure 26.9 shows a sequence of estimates of a robot running an extended Kalman filter localization algorithm.

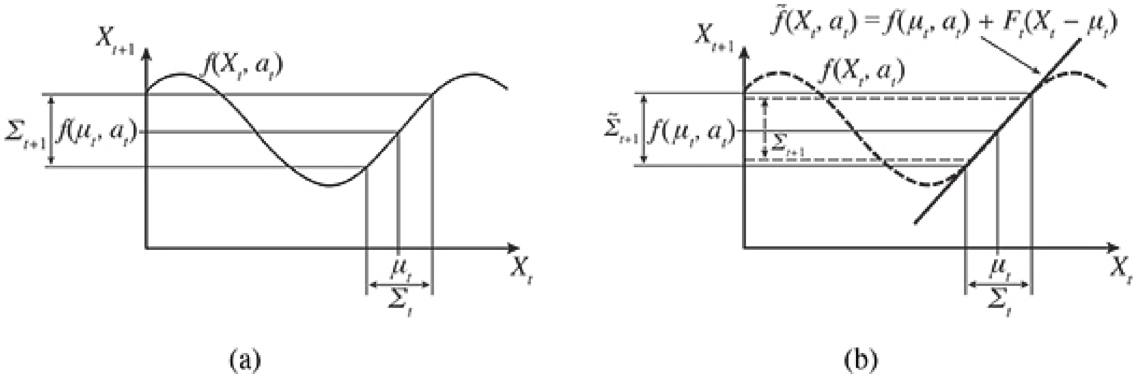


Figure 26.8 One-dimensional illustration of a linearized motion model: (a) The function f, and the projection of a mean μ_t and a covariance interval (based on $\sum_t$) into time $t + 1$. (b) The linearized version is the tangent of f at μ_t. The projection of the mean μ_t is correct. However, the projected covariance $\widetilde{\sum}_{t+1}$ differs from $\sum_{t+1}$.

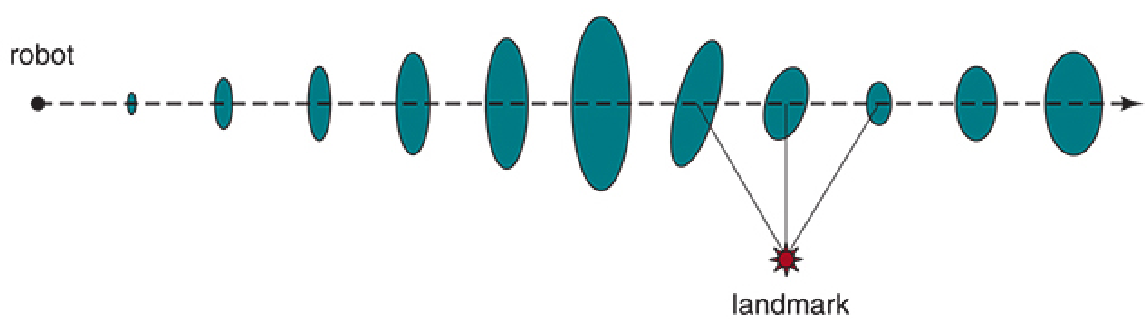


Figure 26.9 Localization using the extended Kalman filter. The robot moves on a straight line. As it progresses, its uncertainty in its location estimate increases, as illustrated by the error ellipses. When

it observes a landmark with known position, the uncertainty is reduced.

As the robot moves, the uncertainty in its location estimate increases, as shown by the error ellipses. Its error decreases as it senses the range and bearing to a landmark with known location and increases again as the robot loses sight of the landmark. EKF algorithms work well if landmarks are easily identified. Otherwise, the posterior distribution may be multimodal, as in Figure 26.7(b). The problem of needing to know the identity of landmarks is an instance of the **data association** problem discussed in Figure 18.3.

In some situations, no map of the environment is available. Then the robot will have to acquire a map. This is a bit of a chicken-and-egg problem: the navigating robot will have to determine its location relative to a map it doesn't quite know, at the same time building this map while it doesn't quite know its actual location. This problem is important for many robot applications, and it has been studied extensively under the name **simultaneous localization and mapping**, abbreviated as **SLAM**.

SLAM problems are solved using many different probabilistic techniques, including the extended Kalman filter discussed above. Using the EKF is straightforward: just augment the state vector to include the locations of the landmarks in the environment. Luckily, the EKF update scales quadratically, so for small maps (e.g., a few hundred landmarks) the computation is quite feasible. Richer maps are often obtained using graph relaxation methods, similar to the Bayesian network inference techniques discussed in Chapter 13. Expectation– maximization is also used for SLAM.

26.4.2 Other types of perception

Not all of robot perception is about localization or mapping. Robots also perceive temperature, odors, sound, and so on. Many of these quantities can be estimated using variants of dynamic Bayes networks. All that is required for such estimators are conditional probability distributions that characterize the evolution of state variables over time, and sensor models that describe the relation of measurements to state variables.

It is also possible to program a robot as a reactive agent, without explicitly reasoning about probability distributions over states. We cover that approach in Section 26.9.1.

The trend in robotics is clearly towards representations with well-defined semantics. Probabilistic techniques outperform other approaches in many hard perceptual problems such as localization and mapping. However, statistical techniques are sometimes too cumbersome, and simpler solutions may be just as effective in practice. To help decide which approach to take, experience working with real physical robots is your best teacher.

26.4.3 Supervised and unsupervised learning in robot perception

Machine learning plays an important role in robot perception. This is particularly the case when the best internal representation is not known. One common approach is to map highdimensional sensor streams into lower-dimensional spaces using unsupervised machine learning methods (see Chapter 19). Such an approach is called **low-dimensional embedding**. Machine learning makes it possible to learn sensor and motion models from data, while simultaneously discovering a suitable internal representation.

Another machine learning technique enables robots to continuously adapt to big changes in sensor measurements. Picture yourself walking from a sunlit space into a dark room with neon lights. Clearly, things are darker inside. But the change of light source also affects all the colors: neon light has a stronger component of green light than sunlight has. Yet somehow we seem not to notice the change. If we walk together with people into a neon-lit room, we don’t think that their faces suddenly turned green. Our perception quickly adapts to the new lighting conditions, and our brain ignores the differences.

Adaptive perception techniques enable robots to adjust to such changes. One example is shown in Figure 26.10, taken from the autonomous driving domain. Here an unmanned ground vehicle adapts its classifier of the concept “drivable surface.” How does this work? The robot uses a laser to provide classification for a small area immediately in front of the robot. When this area is found to be flat in the laser range scan, it is used as a positive training example for the concept “drivable surface.” A mixture-of-Gaussians technique similar to the EM algorithm discussed in Chapter 21 is then trained to recognize the specific color and texture coefficients of the small sample patch. The images in Figure 26.10 are the result of applying this classifier to the full image.

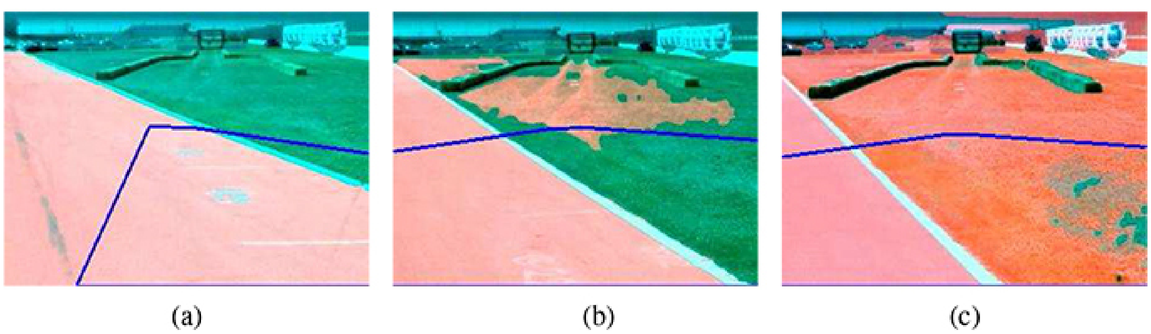


Figure 26.10 Sequence of “drivable surface” classifications using adaptive vision. (a) Only the road is classified as drivable (pink area). The V-shaped blue line shows where the vehicle is heading. (b) The vehicle is commanded to drive off the road, and the classifier is beginning to classify some of the grass as drivable. (c)

The vehicle has updated its model of drivable surfaces to correspond to grass as well as road. Courtesy of Sebastian Thrun.

Methods that make robots collect their own training data (with labels!) are called **self-supervised**. In this instance, the robot uses machine learning to leverage a short-range sensor that works well for terrain classification into a sensor that can see much farther. That allows the robot to drive faster, slowing down only when the sensor model says there is a change in the terrain that needs to be examined more carefully by the short-range sensors.

26.5 Planning and Control

The robot's deliberations ultimately come down to deciding how to move, from the abstract task level all the way down to the currents that are sent to its motors. In this section, we simplify by assuming that perception (and, where needed, prediction) are given, so the world is observable. We further assume deterministic transitions (dynamics) of the world.

We start by separating motion from control. We define a **path** as a sequence of points in geometric space that a robot (or a robot part, such as an arm) will follow. This is related to the notion of path in Chapter 3, but here we mean a sequence of points in space rather than a sequence of discrete actions. The task of finding a good path is called **motion planning**.

Once we have a path, the task of executing a sequence of actions to follow the path is called **trajectory tracking control**. A **trajectory** is a path that has a time associated with each point on the path. A path just says "go from A to B to C, etc." and a trajectory says "start at A, take 1 second to get to B, and another 1.5 seconds to get to C, etc."

26.5.1 Configuration space

Imagine a simple robot, R, in the shape of a right triangle as shown by the lavender triangle in the lower left corner of Figure 26.11. The robot needs to plan a path that avoids a rectangular obstacle, O. The physical space that a robot moves about in is called the **workspace**. This particular robot can move in any direction in the $x - y$ plane, but cannot rotate. The figure shows five other possible positions of the robot with dashed outlines; these are each as close to the obstacle as the robot can get.

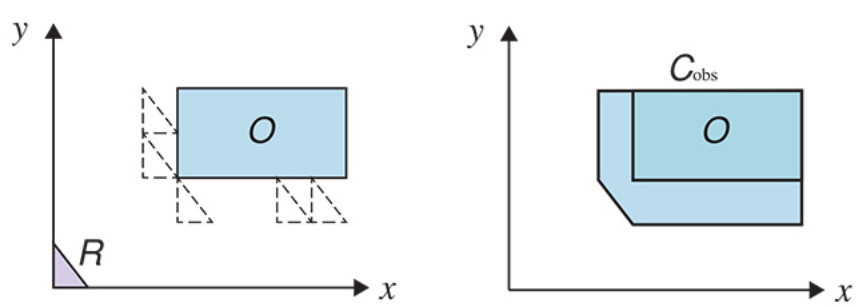


Figure 26.11 A simple triangular robot that can translate, and needs to avoid a rectangular obstacle. On the left is the workspace, on the right is the configuration space.

The body of the robot could be represented as a set of (x, y) points (or (x, y, z) points for a three-dimensional robot), as could the obstacle. With this representation, avoiding the obstacle means that no point on the robot overlaps any point on the obstacle. Motion planning would require calculations on sets of points, which can be complicated and time-consuming.

We can simplify the calculations by using a representation scheme in which all the points that comprise the robot are represented as a single point in an abstract multidimensional space, which we call the **configuration space**, or **C-space**. The idea is that the set of points that comprise the robot can be computed if we know (1) the basic measurements of the robot (for our triangle robot, the length of the three sides will do) and (2) the current **pose** of the robot—its position and orientation.

For our simple triangular robot, two dimensions suffice for the C-space: if we know the (x, y) coordinates of a specific point on the robot—we'll use the right-angle vertex—then we can calculate where every other point of the triangle is (because we know the size and shape of the triangle and because the triangle cannot rotate). In the lower-left corner of Figure 26.11, the lavender triangle can be represented by the configuration (0, 0).

If we change the rules so that the robot can rotate, then we will need three dimensions, (x, y, θ), to be able to calculate where every point is. Here θ is the robot's angle of rotation in the plane. If the robot also had the ability to stretch itself, growing uniformly by a scaling factor s, then the C-space would have four dimensions, (x, y, θ, s).

For now we'll stick with the simple two-dimensional C-space of the non-rotating triangle robot. The next task is to figure out where the points in the obstacle are in C-space. Consider the five dashed-line triangles on the left of Figure 26.11 and notice where the right-angle vertex is on each of these. Then imagine all the ways that the triangle could slide about. Obviously, the right-angle vertex can't go inside the obstacle, and neither can it get any closer than it is on any of the five dashed-line triangles. So you can see that the area where the right-angle vertex can't go—the **C-space obstacle**—is the five-sided polygon on the right of Figure 26.11 labeled C_{obs}

In everyday language we speak of there being multiple obstacles for the robot—a table, a chair, some walls. But the math notation is a bit easier if we think of all of these as combining into one "obstacle" that happens to have disconnected components. In general, the C-space obstacle is the set of all points q in C such that, if the robot were placed in that configuration, its workspace geometry would intersect the workspace obstacle.

Let the obstacles in the workspace be the set of points O, and let the set of all points on the robot in configuration q be $A(q)$. Then the C-space obstacle is defined as

$$C_{obs} = \{q : q \in C \textbf{ and } A(q) \cap O \neq \{\}\}$$

and the **free space** is $C_{free} = C - C_{obs}$.

The C-space becomes more interesting for robots with moving parts. Consider the two-link arm from Figure 26.12(a). It is bolted to a table so the base does not move, but the arm has two joints that move independently—we call these **degrees of freedom (DOF)**. Moving the joints alters the (x, y) coordinates of the elbow, the gripper, and every point on the arm. The arm's configuration space is two-dimensional: $(\theta_{shou}, \theta_{elb})$, where θ_{shou} is the angle of the shoulder joint, and θ_{elb} is the angle of the elbow joint.

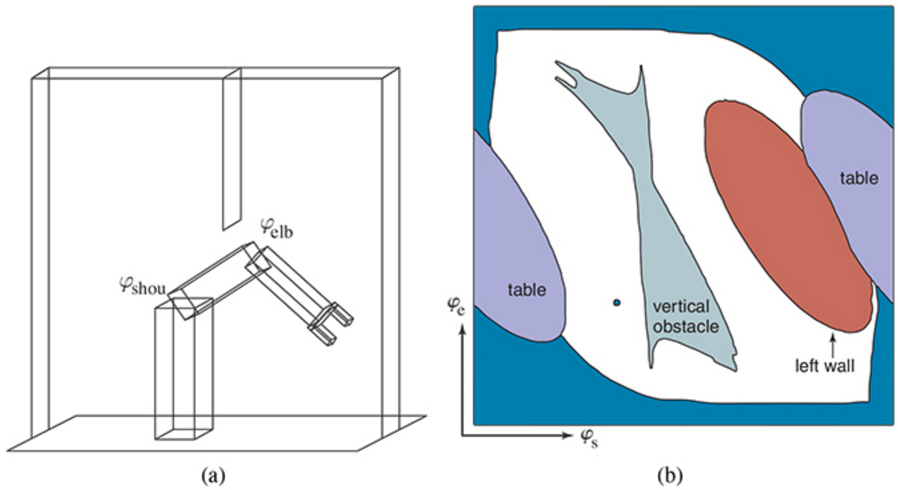


Figure 26.12 (a) Workspace representation of a robot arm with two degrees of freedom. The workspace is a box with a flat obstacle hanging from the ceiling. (b) Configuration space of the same robot. Only white regions in the space are configurations that are free of collisions. The dot in this diagram corresponds to the configuration of the robot shown on the left.

Knowing the configuration for our two-link arm means we can determine where each point on the arm is through simple trigonometry. In general, the **forward kinematics** mapping is a function

$$\phi_b : C \to W$$

that takes in a configuration and outputs the location of a particular point b on the robot when the robot is in that configuration. A particularly useful forward kinematics mapping is that for the robot's end effector, ϕ_{EE}. The set of all points on the robot in a particular configuration q is denoted by $A(q) \subset W$:

$$A(q) = \bigcup_b \{\phi_b(q)\}.$$

The inverse problem, of mapping a desired location for a point on the robot to the configuration(s) the robot needs to be in for that to happen, is known as **inverse kinematics**:

$$IK_b : x \in W \mapsto \{q \in C \ s.t. \ \phi_b(q) = x\}.$$

Sometimes the inverse kinematics mapping might take not just a position, but also a desired orientation as input. When we want a manipulator to grasp an object, for instance, we can compute a desired position and orientation for its gripper, and use inverse kinematics to determine a goal configuration for the robot. Then a planner needs to find a way to get the robot from its current configuration to the goal configuration without intersecting obstacles.

Workspace obstacles are often depicted as simple geometric forms—especially in robotics textbooks, which tend to focus on polygonal obstacles. But how do the obstacles look in configuration space?

For the two-link arm, simple obstacles in the workspace, like a vertical line, have very complex C-space counterparts, as shown in Figure 26.12(b). The different shadings of the occupied space correspond to the different objects in the robot's workspace: the dark region surrounding the entire free space corresponds to configurations in which the robot collides with itself. It is easy to see that extreme values of the shoulder or elbow angles cause such a violation. The two oval-shaped regions on both sides of the robot correspond to the table on which the robot is mounted. The third oval region corresponds to the left wall.

Finally, the most interesting object in configuration space is the vertical obstacle that hangs from the ceiling and impedes the robot's motions. This object has a funny shape in configuration space: it is highly nonlinear and at places even concave. With a little bit of imagination the reader will recognize the shape of the gripper at the upper left end.

We encourage the reader to pause for a moment and study this diagram. The shape of this obstacle in C-space is not at all obvious! The dot inside Figure 26.12(b) marks the configuration of the robot in Figure 26.12(a). Figure 26.13 depicts three additional configurations, both in workspace and in configuration space. In configuration conf-1, the gripper is grasping the vertical obstacle.

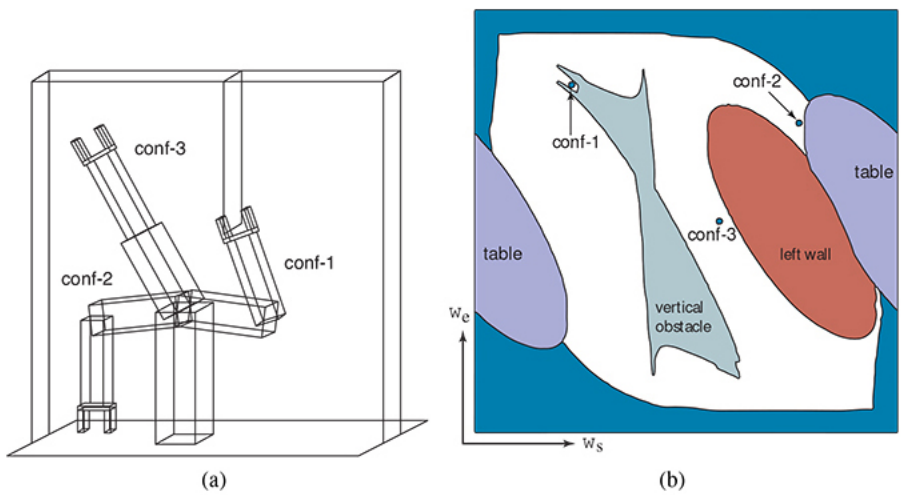


Figure 26.13 Three robot configurations, shown in workspace and configuration space.

We see that even if the robot's workspace is represented by flat polygons, the shape of the free space can be very complicated. In practice, therefore, one usually *probes* a configuration space instead of constructing it explicitly. A planner may generate a configuration and then test to see if it is in free space by applying the robot kinematics and then checking for collisions in workspace coordinates.

26.5.2 Motion planning

The **motion planning** problem is that of finding a plan that takes a robot from one configuration to another without colliding with an obstacle. It is a basic building block for movement and manipulation. In Section 26.5.4 we will discuss how to do this under complicated dynamics, like steering a car that may drift off the path if you take a curve too fast. For now, we will focus on the simple motion planning problem of finding a geometric path that is collision free. Motion planning is a quintessentially continuous-state **search problem**, but it is often possible to discretize the space and apply the search algorithms from Chapter 3.

The motion planning problem is sometimes referred to as the **piano mover's problem**. It gets its name from a mover's struggles with getting a large, irregular-shaped piano from one room to another without hitting anything.

We are given:

- a workspace *world* W in either $\mathbb{R}^2$ for the plane or $\mathbb{R}^3$ for three dimensions,
- an *obstacle region* $O \subset W$,
- a robot with a configuration space C and set of points $A(q)$ for $q \in C$,
- a starting configuration $q_s \in C$, and
- a goal configuration $q_g \in C$.

The obstacle region induces a C-space obstacle C_{obs} and its corresponding free space C_{free} defined as in the previous section. We need to find a continuous **path** through free space. We will use a parameterized curve, $\tau(t)$, to represent the path, where $\tau(0) = q_s$ and $\tau(1) = q_g$ and $\tau(t)$ for every t between 0 and 1 is some point in C_{free}. That is, t parameterizes how far we are along the path, from start to goal. Note that t acts somewhat like time in that as t increases the distance along the path increases, but t is always a point on the interval [0,1] and is not measured in seconds.

The motion planning problem can be made more complex in various ways: defining the goal as a set of possible configurations rather than a single configuration; defining the goal in the workspace rather than the C-space; defining a cost function (e.g., path length) to be minimized; satisfying constraints (e.g., if the path involves carrying a cup of coffee, making sure that the cup is always oriented upright so the coffee does not spill).

The spaces of motion planning: Let's take a step back and make sure we understand the spaces involved in motion planning. First, there is the workspace or world W. Points in W are points in the everyday three-dimensional world. Next, we have the space of configurations, C. Points q in C are d-dimensional, with d the robot's number of degrees of freedom, and map to sets of points $A(q)$ in W. Finally, there is the space of paths. The space of paths is a space of functions. Each point in this space maps to an entire curve through C-space. This space is ∞-dimensional! Intuitively, we need d dimensions for each configuration along the path, and there are as many configurations on a path as there are points in the number line interval [0,1]. Now let's consider some ways of solving the motion planning problem.

Visibility graphs

For the simplified case of two-dimensional configuration spaces and polygonal C-space obstacles, **visibility graphs** are a convenient way to solve the motion planning problem with a guaranteed shortest-path solution. Let $V_{obs} \subset C$ be the set of vertices of the polygons making up C_{obs}, and let $V = V_{obs} \cup \{q_s, q_g\}$.

We construct a graph $G = (V, E)$ on the vertex set V with edges $e_{ij} \in E$ connecting a vertex v_i to another vertex v_j if the line connecting the two vertices is collision-free—that is, if $\{\lambda v_i + (1-\lambda) v_j : \lambda \in [0,1]\} \cap C_{obs} = \{\ \}$. When this happens, we say the two vertices "can see each other," which is where "visibility" graphs got their name.

To solve the motion planning problem, all we need to do is run a discrete graph search (e.g., best-first search) on the graph G with starting state q_s and goal q_g. In Figure 26.14 we see a visibility graph and an optimal three-step solution. An optimal search on visibility graphs will always give us the optimal path (if one exists), or report failure if no path exists.

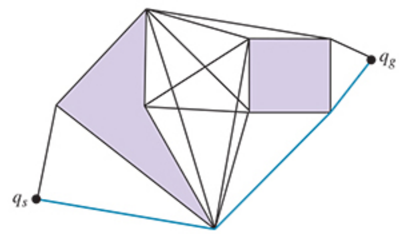


Figure 26.14 A visibility graph. Lines connect every pair of vertices that can “see” each other—lines that don’t go through an obstacle. The shortest path must lie upon these lines.

Voronoi diagrams

Visibility graphs encourage paths that run immediately adjacent to an obstacle—if you had to walk around a table to get to the door, the shortest path would be to stick as close to the table as possible. However, if motion or sensing is nondeterministic, that would put you at risk of bumping into the table. One way to address this is to pretend that the robot’s body is a bit larger than it actually is, providing a buffer zone. Another way is to accept that path length is not the only metric we want to optimize. Section 26.8.2 shows how to learn a good metric from human examples of behavior.

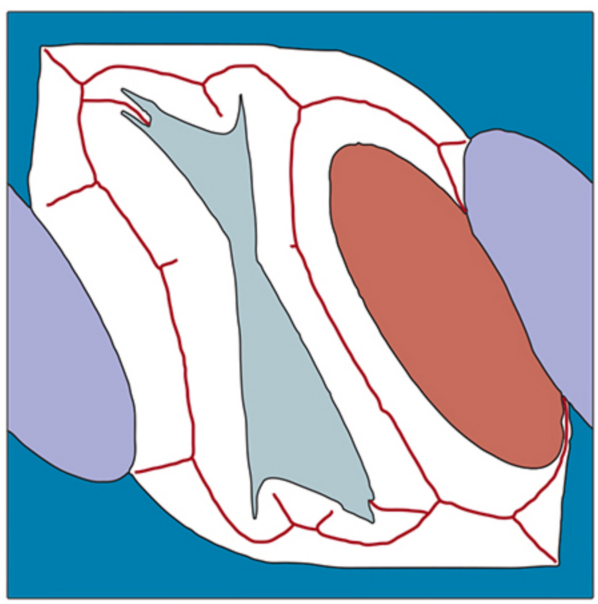


Figure 26.15 A Voronoi diagram showing the set of points (black lines) equidistant to two or more obstacles in configuration space.

A third way is to use a different technique, one that puts paths as far away from obstacles as possible rather than hugging close to them. A **Voronoi diagram** is a representation that allows us to do just that. To get an idea for what a Voronoi diagram does, consider a space where the obstacles are, say, a dozen small points scattered about a plane. Now surround each of the obstacle points with a **region** consisting of all the points in the plane that are closer to that obstacle point than to any other obstacle point. Thus, the regions partition the plane. The Voronoi diagram consists of the set of regions, and the **Voronoi graph** consists of the edges and vertices of the regions.

When obstacles are areas, not points, everything stays pretty much the same. Each region still contains all the points that are closer to one obstacle than to any other, where distance is measured to the closest point on an obstacle. The boundaries between regions still correspond to points that are equidistant between two obstacles, but now the boundary may be a curve rather than a straight line. Computing these boundaries can be prohibitively expensive in high-dimensional spaces.

To solve the motion planning problem, we connect the start point q_s to the closest point on the Voronoi graph via a straight line, and the same for the goal point q_g. We then use discrete graph search to find the shortest path on the graph. For problems like navigating through corridors indoors, this gives a nice path that goes down the middle of the corridor. However, in outdoor settings it can come up with inefficient paths, for example suggesting an unnecessary 100 meter detour to stick to the middle of a wide-open 200-meter space.

Cell decomposition

An alternative approach to motion planning is to discretize the C-space. **Cell decomposition** methods decompose the free space into a finite number of contiguous regions, called cells. These cells are designed so that the path-planning problem within a single cell can be solved by simple means (e.g., moving along a straight line). The path-planning problem then becomes a discrete graph search problem (as with visibility graphs and Voronoi graphs) to find a path through a sequence of cells.

The simplest cell decomposition consists of a regularly spaced grid. Figure 26.16(a) shows a square grid decomposition of the space and a solution path that is optimal for this grid size. Grayscale shading indicates the *value* of each free-space grid cell—the cost of the shortest path from that cell to the goal. (These values can be computed by a deterministic form of the VALUE-ITERATION algorithm given in Figure 16.6 on page 563.) Figure 26.16(b) shows the corresponding workspace trajectory for the arm. Of course, we could also use the A* algorithm to find a shortest path.

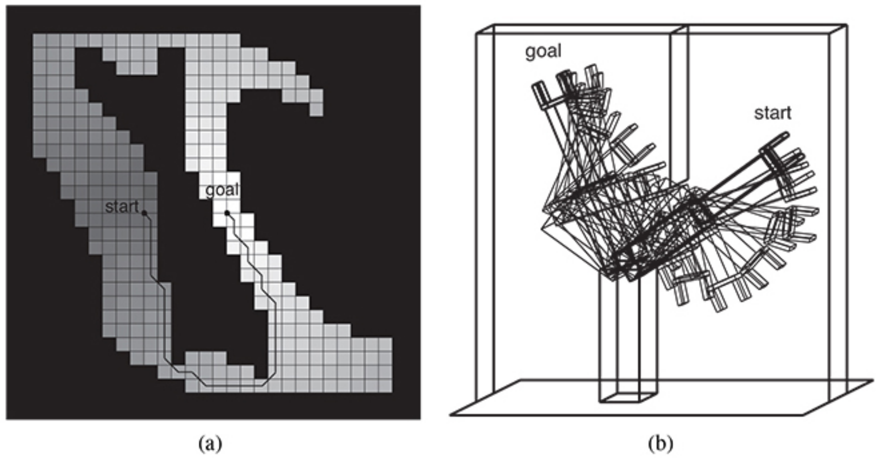


Figure 26.16 (a) Value function and path found for a discrete grid cell approximation of the configuration space. (b) The same path visualized in workspace coordinates. Notice how the robot bends its elbow to avoid a collision with the vertical obstacle.

This grid decomposition has the advantage that it is simple to implement, but it suffers from three limitations. First, it is workable only for low-dimensional configuration spaces, because the number of grid cells increases exponentially with d, the number of dimensions. (Sounds familiar? This is the curse of dimensionality.) Second, paths through discretized state space will not always be smooth. We see in Figure 26.16(a) that the diagonal parts of the path are jagged and hence very difficult for the robot to follow accurately. The robot can attempt to smooth out the solution path, but this is far from straightforward.

Third, there is the problem of what to do with cells that are "mixed"—that is, neither entirely within free space nor entirely within occupied space. A solution path that includes such a cell may not be a real solution, because there may be no way to safely cross the cell. This would make the path planner *unsound.* On the other hand, if we insist that only completely free cells may be used, the planner will be *incomplete,* because it might be the case that the only paths to the goal go through mixed cells—it might be that a corridor is actually wide enough for the robot to pass, but the corridor is covered only by mixed cells.

The first approach to this problem is *further subdivision* of the mixed cells—perhaps using cells of half the original size. This can be continued recursively until a path is found that lies entirely within free cells. This method works well and is complete if there is a way to decide if a given cell is a mixed cell, which is easy only if the configuration space boundaries have relatively simple mathematical descriptions.

It is important to note that cell decomposition does not necessarily require explicitly representing the obstacle space C_{obs}. We can decide to include a cell or not by using a **collision checker**. This is a crucial notion to motion planning. A collision checker is a function $\gamma(q)$ that maps to 1 if the configuration collides with an obstacle, and 0 otherwise. It is much easier to check whether a specific configuration is in collision than to explicitly construct the entire obstacle space C_{obs}.

Examining the solution path shown in Figure 26.16(a), we can see an additional difficulty that will have to be resolved. The path contains arbitrarily sharp corners, but a physical robot has momentum and cannot change direction instantaneously. This problem can be solved by storing, for each grid cell, the exact continuous state (position and velocity) that was attained when the cell was reached in the search. Assume further that when propagating information to nearby grid cells, we use this continuous state as a basis, and apply the continuous robot motion model for jumping to nearby cells. So we don't make an instantaneous 90° turn; we make a rounded turn governed by the laws of motion. We can now guarantee that the resulting trajectory is smooth and can indeed be executed by the robot. One algorithm that implements this is **hybrid A***.

Randomized motion planning

Randomized motion planning does graph search on a *random* decomposition of the configuration space, rather than a regular cell decomposition. The key idea is to sample a random set of points and to create edges between them if there is a very simple way to get from one to the other (e.g., via a straight line) without colliding; then we can search on this graph.

A **probabilistic roadmap (PRM)** algorithm is one way to leverage this idea. We assume access to a collision checker γ (defined on page 953), and to a **simple planner** $B(q_1, q_2)$ that returns a path from q_1 to q_2 (or failure) but does so *quickly*. This simple planner is not going to be complete—it might return failure even if a solution actually exists. Its job is to quickly try to connect q_1 and q_2 and let the main algorithm know if it succeeds. We will use it to define whether an edge exists between two vertices.

The algorithm starts by sampling M **milestones**—points in C_{free}—in addition to the points q_s and q_g. It uses rejection sampling, where configurations are sampled randomly and collision-checked using γ until a total of M milestones are found. Next, the algorithm uses the simple planner to try to connect pairs of milestones. If the simple planner returns success, then an edge between the pair is added to the graph; otherwise, the graph remains as is. We try to connect each milestone either to its k nearest neighbors (we call this k-PRM), or to all milestones in a sphere of a radius r. Finally, the algorithm searches for a path on this graph from q_s to q_g. If no path is found, then M more milestones are sampled, added to the graph, and the process is repeated.

Figure 26.17 shows a roadmap with the path found between two configurations. PRMs are not complete, but they are what is called **probabilistically complete**—they will eventually find a path, if one exists. Intuitively, this is because they keep sampling more milestones. PRMs work well even in high-dimensional configuration spaces.

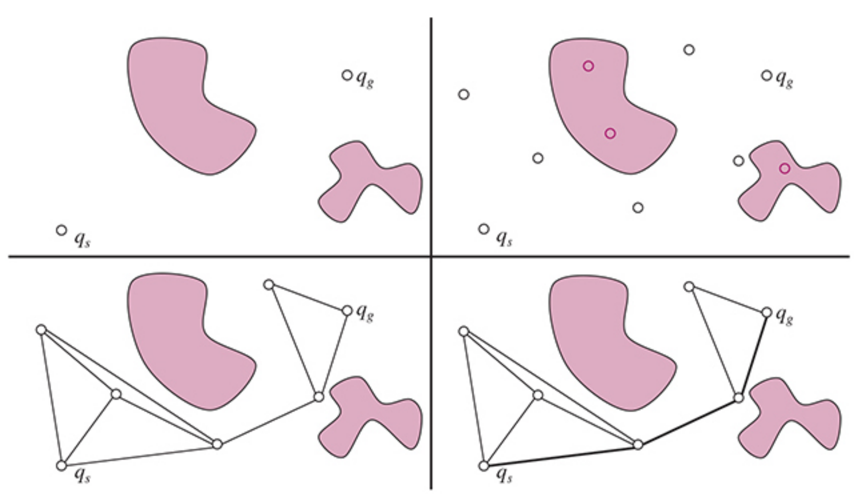


Figure 26.17 The probabilistic roadmap (PRM) algorithm. **Top left**: the start and goal configurations. **Top right**: sample M collision-free milestones (here $M = 5$). **Bottom left**: connect each milestone to its k nearest neighbors (here $k = 3$). **Bottom right**: find the shortest path from the start to the goal on the resulting graph.

PRMs are also popular for **multi-query planning**, in which we have multiple motion planning problems within the same C-space. Often, once the robot reaches a goal, it is called upon to reach another goal in the same workspace. PRMs are really useful, because the robot can dedicate time up front to constructing a roadmap, and amortize the use of that roadmap over multiple queries.

Rapidly-exploring random trees

An extension of PRMs called **rapidly exploring random trees (RRTs)** is popular for single-query planning. We incrementally build two trees, one with q_s as the root and one with q_g as the root. Random milestones are chosen, and an attempt is made to connect each new milestone to the existing trees. If a milestone connects both trees, that means a solution has been found, as in Figure 26.18. If not, the algorithm finds the closest point in each tree and adds to the tree a new edge that extends from the point by a distance δ towards the milestone. This tends to grow the tree towards previously unexplored sections of the space.

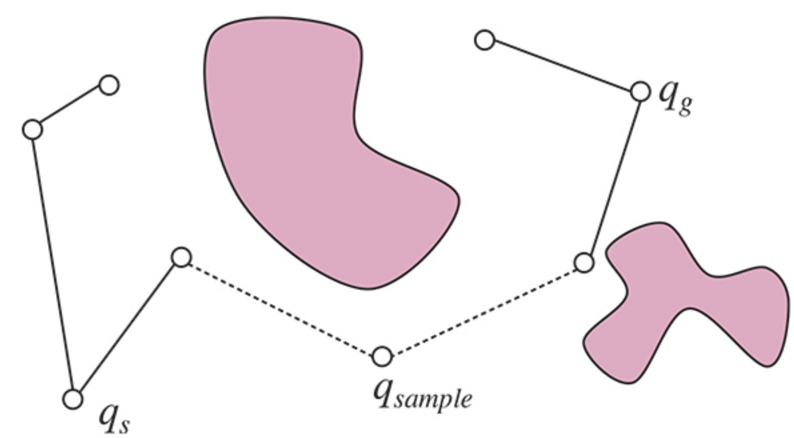


Figure 26.18 The bidirectional RRT algorithm constructs two trees (one from the start, the other from the goal) by incrementally connecting each sample to the closest node in each tree, if the connection is possible. When a sample connects to both trees, that means we have found a solution path.

Roboticists love RRTs for their ease of use. However, RRT solutions are typically nonoptimal and lack smoothness. Therefore, RRTs are often followed by a post-processing step. The most common one is "short-cutting," in which we randomly select one of the vertices on the solution path and try to remove it by connecting its neighbors to each other (via the simple planner). We do this repeatedly for as many steps as we have compute time for. Even then, the trajectories might look a little unnatural due to the random positions of the milestone that were selected, as shown in Figure 26.19.

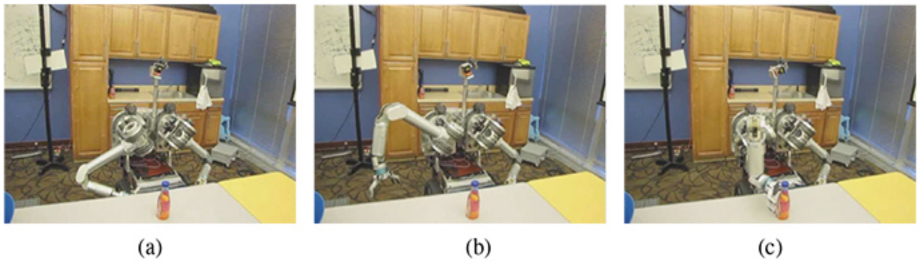


Figure 26.19 Snapshots of a trajectory produced by an RRT and post-processed with shortcutting. Courtesy of Anca Dragan.

RRT* is a modification to RRT that makes the algorithm asymptotically optimal: the solution converges to the optimal solution as more and more milestones are sampled. The key idea is to pick the nearest neighbor based on a notion of cost to come rather than distance from the milestone only, and to rewire the tree, swapping parents of older vertices if it is cheaper to reach them via the new milestone.

Trajectory optimization for kinematic planning

Randomized sampling algorithms tend to first construct a complex but feasible path and then optimize it. Trajectory optimization does the opposite: it starts with a simple but infeasible path, and then works to push it out of collision. The goal is to find a path that optimizes a cost function[1] over paths. That is, we want to minimize the cost function $J(\tau)$, where $\tau(0) = q_s$ and $\tau(1) = q_g$.

J is called a **functional** because it is a function over functions. The argument to J is τ which is itself a function: $\tau(t)$ takes as input a point in the [0,1] interval and maps it to a configuration. A standard cost functional trades off between two important aspects of the robot's motion: collision avoidance and efficiency,

$$J = J_{obs} + \lambda J_{eff}$$

where the efficiency J_{eff} measures the length of the path and may also measure smoothness. A convenient way to define efficiency is with a quadratic: it integrates the squared first derivative of τ (we will see in a bit why this does in fact incentivize short paths):

$$J_{eff} = \int_0^1 \frac{1}{2} \parallel \dot{\tau}(s) \parallel^2 ds.$$

For the obstacle term, assume we can compute the distance $d(x)$ from any point $x \in W$ to the nearest obstacle edge. This distance is positive outside of obstacles, 0 at the edge, and negative inside. This is called a **signed distance field**. We can now define a cost field in the workspace, call it c, that has high cost inside of obstacles, and a small cost right outside. With this cost, we can make points in the workspace really hate being inside obstacles, and dislike being right next to them (avoiding the visibility graph problem of their always hanging out by the edges of obstacles). Of course, our robot is not a point in the workspace, so we have some more work to do—we need to consider all points b on the robot's body:

$$J_{eff} = \int_0^1 \int_b c(\underbrace{\phi_b(\tau(s))}_{\in W}) \| \frac{d}{ds} \underbrace{\phi_b(\tau(s))}_{\in W} \| \, db\, ds.$$

This is called a **path integral**—it does not just integrate *c* along the way for each body point, but it multiplies by the derivative to make the cost invariant to *retiming* of the path. Imagine a robot sweeping through the cost field, accumulating cost as is moves. Regardless of how fast or slow the arm moves through the field, it must accumulate the exact same cost.

The simplest way to solve the optimization problem above and find a path is *gradient descent*. If you are wondering how to take gradients of functionals with respect to functions, something called the *calculus of variations* is here to help. It is especially easy for functionals of the form

$$J[\tau] = \int_0^1 F(s, \tau(s), \dot{\tau}\,(s))ds$$

which are integrals of functions that depend just on the parameter *s*, the value of the function at *s*, and the derivative of the function at *s*. In such a case, the **Euler-Lagrange equation** says that the gradient is

$$\nabla \tau J(s) = \tfrac{\partial F}{\partial \tau(s)}(s) - \tfrac{d}{dt}\tfrac{\partial F}{\partial \dot{\tau}(s)}(s).$$

If we look closely at J_{eff} and J_{obs}, they both follow this pattern. In particular for J_{eff}, we have $F(s, \tau(s), \dot{\tau}(s)) = ||\dot{\tau}(s)||^2$. To get a bit more comfortable with this, let's compute the gradient for J_{eff} only. We see that *F* does not have a direct dependence on τ(s), so the first term in the formula is 0. We are left with

$$\nabla_\tau J(s) = 0 - \tfrac{d}{dt}\dot{\tau}(s)$$

since the partial of *F* with respect to $\dot{\tau}(s)$ is $\dot{\tau}(s)$.

Notice how we made things easier for ourselves when defining J_{eff}—it's a nice quadratic of the derivative (and we even put a ½ in front so that the 2 nicely cancels out). In practice, you will see this trick happen a lot for optimization—the art is not just in choosing how to optimize the cost function, but also in choosing a cost function that will play nicely with how you will optimize it. Simplifying our gradient, we get

$$\nabla_\tau J(s) = -\dot{\tau}(s).$$

Now, since J_{eff} is a quadratic, setting this gradient to 0 gives us the solution for τ if we didn't have to deal with obstacles. Integrating once, we get that the first derivative needs to be constant; integrating again we get that $\tau(s) = a \cdot s + b$, with *a* and *b* determined by the endpoint constraints for τ(0) and τ(1). The optimal path with respect to J_{eff} is thus the straight line from start to goal! It is indeed the most efficient way to go from one to the other if there are no obstacles to worry about.

Of course, the addition of J_{obs} is what makes things difficult—and we will spare you deriving its gradient here. The robot would typically initialize its path to be a straight line, which would plow right through some obstacles. It would then calculate the gradient of the cost about the current path, and the gradient would serve to push the path away from the obstacles (Figure 26.20). Keep in mind that gradient descent will only find a *locally optimal* solution—just like hill climbing. Methods such as simulated annealing (Section 4.1.2) can be used for exploration, to make it more likely that the local optimum is a good one.

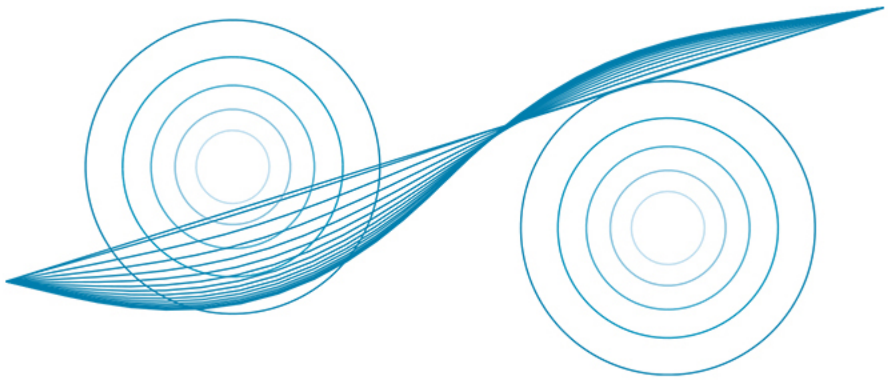


Figure 26.20 Trajectory optimization for motion planning. Two point-obstacles with circular bands of decreasing cost around them. The optimizer starts with the straight line trajectory, and lets the obstacles bend the line away from collisions, finding the minimum path through the cost field.

26.5.3 Trajectory tracking control

We have covered how to *plan* motions, but not how to actually *move*—to apply current to motors, to produce torque, to move the robot. This is the realm of **control theory**, a field of increasing importance in AI. There are two main questions to deal with: how do we turn a mathematical description of a path into a sequence of actions in the real world (open-loop control), and how do we make sure that we are staying on track (closed-loop control)?

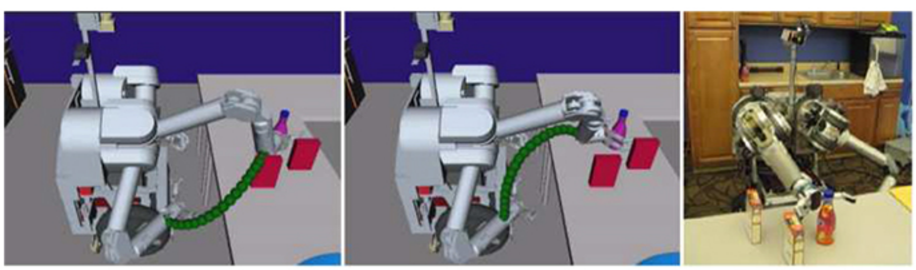


Figure 26.21 The task of reaching to grasp a bottle solved with a trajectory optimizer. Left: the initial trajectory, plotted for the end effector. Middle: the final trajectory after optimization. Right: the goal configuration. Courtesy of Anca Dragan. See Ratliff *et al.* (2009).

From configurations to torques for open-loop tracking: Our path $\tau(t)$ gives us configurations. The robot starts at rest at $q_s = \tau(0)$. From there the robot's motors will turn currents into torques, leading to motion. But what torques should the robot aim for, such that it ends up at $q_g = \tau(1)$?

This is where the idea of a **dynamics model** (or transition model) comes in. We can give the robot a function f that computes the effects torques have on the configuration. Remember $F = ma$ from physics? Well, there is something like that for torques too, in the form $u = f^{-1}(q, \dot{q}, \ddot{q})$, with u a torque, $\dot{q}$ a velocity, and $\ddot{q}$ an acceleration.[2] If the robot is at configuration q and velocity $\dot{q}$, and applied torque u, that would lead to acceleration $\ddot{q} = f(q, \dot{q}, u)$. The tuple $(q, \dot{q})$ is a **dynamic state**, because it includes velocity, whereas q is the **kinematic state** and is not sufficient for computing exactly what torque to apply. f is a deterministic dynamics model in the MDP over dynamic states with torques as actions. f^{-1} is the **inverse dynamics**, telling us what torque to apply if we want a particular acceleration, which leads to a change in velocity and thus a change in dynamic state.

Now, naively, we could think of $t \in [0,1]$ as "time" on a scale from 0 to 1 and select our torque using inverse dynamics:

$$u(t) = f^{-1}(\tau(t), (\dot{\tau}(t), \ddot{\tau}(t)) \qquad (26.2)$$

assuming that the robot starts at $(\tau(0), \dot{\tau}(0))$. In reality though, things are not that easy.

The path τ was created as a sequence of points, without taking velocities and accelerations into account. As such, the path may not satisfy $\dot{\tau}(0) = 0$ (the robot starts at 0 velocity), or even that τ is differentiable (let alone twice differentiable). Further, the meaning of the endpoint "1" is unclear: how many seconds does that map to?

In practice, before we even think of tracking a reference path, we usually **retime** it, that is, transform it into a trajectory $\xi(t)$ that maps the interval $[0, T]$ for some time duration T into points in the configuration space C. (The symbol ξ is the Greek letter Xi.) Retiming is trickier than you might think, but there are approximate ways to do it, for instance by picking a maximum velocity and acceleration, and using a profile that accelerates to that maximum velocity, stays there as long as it can, and then decelerates back to 0. Assuming we can do this, Equation (26.2) above can be rewritten as

$$u(t) = f^{-1}(\xi(t), (\dot{\xi}(t), \ddot{\xi}(t)). \qquad (26.3)$$

Even with the change from τ to ξ, an actual trajectory, the equation of applying torques from above (called a **control law**) has a problem in practice. Thinking back to the reinforcement learning section, you might guess what it is. The equation works great in the situation where f is exact, but pesky reality gets in the way as usual: in real systems, we can't measure masses and inertias exactly, and f might not properly account for physical phenomena like **stiction** in the motors (the friction that tends to prevent stationary surfaces from being set in motion—to make them stick). So, when the robot arm starts applying those torques but f is wrong, the errors accumulate and you deviate further and further from the reference path.

Rather than just letting those errors accumulate, a robot can use a control process that looks at where it thinks it is, compares that to where it wanted to be, and applies a torque to minimize the error.

A controller that provides force in negative proportion to the observed error is known as a proportional controller or **P controller** for short. The equation for the force is:

$$u(t) = Kp(\xi(t) - q_t)$$

where q_t is the current configuration, and K_P is a constant representing the **gain factor** of the controller. K_P regulates how strongly the controller corrects for deviations between the actual state q_t and the desired state $\xi(t)$.

Figure 26.22(a) illustrates what can go wrong with proportional control. Whenever a deviation occurs—whether due to noise or to constraints on the forces the robot can apply—the robot provides an opposing force whose magnitude is proportional to this deviation. Intuitively, this might appear plausible, since deviations should be compensated by a counterforce to keep the robot on track. However, as Figure 26.22(a) illustrates, a proportional controller can cause the robot to apply too much force, overshooting the desired path and zig-zagging back and forth. This is the result of the natural inertia of the robot: once driven back to its reference position the robot has a velocity that can't instantaneously be stopped.

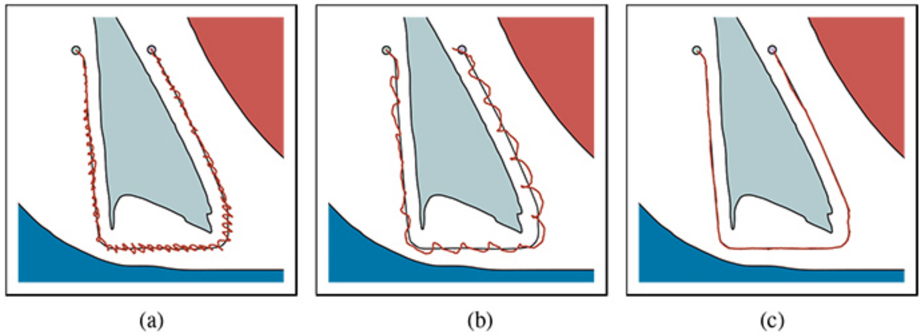


Figure 26.22 Robot arm control using (a) proportional control with gain factor 1.0, (b) proportional control with gain factor 0.1, and (c) PD (proportional derivative) control with gain factors 0.3 for the proportional component and 0.8 for the differential component. In all cases the robot arm tries to follow the smooth line path, but in (a) and (b) deviates substantially from the path.

In Figure 26.22(a), the parameter $K_P = 1$. At first glance, one might think that choosing a smaller value for K_P would remedy the problem, giving the robot a gentler approach to the desired path. Unfortunately, this is not the case. Figure 26.22(b) shows a trajectory for $K_P = .1$, still exhibiting oscillatory behavior. The lower value of the gain parameter helps, but does not solve the problem. In fact, in the absence of friction, the P controller is essentially a spring law; so it will oscillate indefinitely around a fixed target location.

There are a number of controllers that are superior to the simple proportional control law. A controller is said to be **stable** if small perturbations lead to a bounded error between the robot and the reference signal. It is said to be **strictly stable** if it is able to return to and then stay on its reference path upon such perturbations. Our P controller appears to be stable but not strictly stable, since it fails to stay anywhere near its reference trajectory.

The simplest controller that achieves strict stability in our domain is a **PD controller**. The letter 'P' stands again for *proportional*, and 'D' stands for *derivative.* PD controllers are described by the following equation:

$$u(t) = Kp(\xi(t) - q_t) + K_D(\dot{\xi}(t) - \dot{q}_t). \qquad (26.4)$$

As this equation suggests, PD controllers extend P controllers by a differential component, which adds to the value of $u(t)$ a term that is proportional to the first derivative of the error $\xi(t) - q_t$ over time. What is the effect of such a term? In general, a derivative term dampens the system that is being controlled. To see this, consider a situation where the error is changing rapidly over time, as is the case for our P controller above. The derivative of this error will then counteract the proportional term, which will reduce the overall response to the perturbation. However, if the same error persists and does not change, the derivative will vanish and the proportional term dominates the choice of control.

Figure 26.22(c) shows the result of applying this PD controller to our robot arm, using as gain parameters $K_P = .3$ and $K_D = .8$. Clearly, the resulting path is much smoother, and does not exhibit any obvious oscillations.

PD controllers do have failure modes, however. In particular, PD controllers may fail to regulate an error down to zero, even in the absence of external perturbations. Often such a situation is the result of a systematic external force that is not part of the model. For example, an autonomous car driving on a banked surface may find itself systematically pulled to one side. Wear and tear in robot arms causes similar systematic errors. In such situations, an over-proportional feedback is required to drive the error closer to zero. The solution to this problem lies in adding a third term to the control law, based on the *integrated* error over time:

$$u(t) = K_P(\xi(t) - q_t) + K_I \int_0^t (\xi(s) - q_s)ds + K_D(\dot{\xi}(t) - \dot{q}_t). \tag{26.5}$$

Here K_I is a third gain parameter. The term $\int_0^t (\dot{\xi}(s)$ calculates the integral of the error over time. The effect of this term is that long-lasting deviations between the reference signal and the actual state are corrected. Integral terms, then, ensure that a controller does not exhibit systematic long-term error, although they do pose a danger of oscillatory behavior.

A controller with all three terms is called a **PID controller** (for proportional integral derivative). PID controllers are widely used in industry, for a variety of control problems. Think of the three terms as follows—proportional: try harder the farther away you are from the path; derivative: try even harder if the error is increasing; integral: try harder if you haven't made progress for a long time.

A middle ground between open-loop control based on inverse dynamics and closed-loop PID control is called **computed torque control**. We compute the torque our model thinks we will need, but compensate for model inaccuracy with proportional error terms:

$$u(t) = \underbrace{f^{-1}(\xi(t), \dot{\xi}(t), \ddot{\xi}(t))}_{feedforward} + \underbrace{m(\xi(t))\left(K_P(\xi(t) - q_t) + K_D(\dot{\xi}(t) - \dot{q}_t)\right)}_{feedback}.$$

The first term is called the **feedforward component** because it looks forward to where the robot needs to go and computes what torque might be required. The second is the **feedback component** because it feeds the current error in the dynamic state back into the control law. $m(q)$ is the inertia matrix at configuration q—unlike normal PD control, the gains change with the configuration of the system.

Plans versus policies

Let's take a step back and make sure we understand the analogy between what happened so far in this chapter and what we learned in the search, MDP, and reinforcement learning chapters. With motion in robotics, we are really considering an underlying MDP where the states are dynamic states (configuration and velocity), and the actions are control inputs, usually in the form of torques. If you take another look at our control laws above, they are *policies*, not *plans*—they tell the robot what action to take from *any* state it might reach. However, they are usually far from *optimal* policies. Because the dynamic state is continuous and high dimensional (as is the action space), optimal policies are computationally difficult to extract.

Instead, what we did here is to break up the problem. We come up with a plan first, in a simplified state and action space: we use only the kinematic state, and assume that states are reachable from one another without paying attention to the underlying dynamics. This is motion planning, and it gives us the reference path. If we knew the dynamics perfectly, we could turn this into a plan for the original state and action space with Equation (26.3).

But because our dynamics model is typically erroneous, we turn it instead into a policy that tries to follow the plan—getting back to it when it drifts away. When doing this, we introduce suboptimality in two ways: first by planning without considering dynamics, and second by assuming that if we deviate from the plan, the optimal thing to do is to return to the original plan. In what follows, we describe techniques that compute policies directly over the dynamic state, avoiding the separation altogether.

26.5.4 Optimal control

Rather than using a planner to create a kinematic path, and only worrying about the dynamics of the system after the fact, here we discuss how we might be able to do it all at once. We'll take the trajectory optimization problem for kinematic paths, and turn it into true trajectory optimization with dynamics: we will optimize directly over the actions, taking the dynamics (or transitions) into account.

This brings us much closer to what we've seen in the search and MDP chapters. If we know the system's dynamics, then we can find a sequence of actions to execute, as we did in Chapter 3. If we're not sure, then we might want a policy, as in Chapter 16.

In this section, we are looking more directly at the underlying MDP the robot works in. We're switching from the familiar discrete MDPs to continuous ones. We will denote our dynamic state of the world by x, as is common practice—the equivalent of s in discrete MDPs. Let x_s and x_g be the starting and goal states.

We want to find a sequence of actions that, when executed by the robot, result in state-action pairs with low cumulative cost. The actions are torques which we denote with $u(t)$ for t starting at 0 and ending at T. Formally, we want to find the sequence of torques u that minimize a cumulative cost J:

$$\min_u \int_0^T J(x(t), u(t))dt \qquad (26.7)$$

subject to the constraints

$$\forall t, \dot{x}(t) = f(x(t), u(t))$$
$$x(0) = x_s, x(T) = x_g.$$

How is this connected to motion planning and trajectory tracking control? Well, imagine we take the notion of efficiency and clearance away from the obstacles and put it into the cost function J, just as we did before in trajectory optimization over kinematic state. The dynamic state is the configuration and velocity, and torques u change it via the dynamics f from open-loop trajectory tracking. The difference is that now we're thinking about the configurations and the torques at the same time. Sometimes, we might want to treat collision avoidance as a hard constraint as well, something we've also mentioned before when we looked at trajectory optimization for the kinematic state only.

To solve this optimization problem, we can take gradients of J—not with respect to the sequence τ of configurations anymore, but directly with respect to the controls u. It is sometimes helpful to include the state sequence x as a decision variable too, and use the dynamics constraints to ensure that x and u are consistent. There are various trajectory optimization techniques using this approach; two of them go by the names **multiple shooting** and **direct collocation**. None of these techniques will find the global optimal solution, but in practice they can effectively make humanoid robots walk and make autonomous cars drive.

Magic happens when in the problem above, J is quadratic and f is linear in x and u. We want to minimize

$$\min \int_0^\infty x^T Q x + u^T R u dt \qquad \text{subject to} \qquad \forall t, \dot{x}(t) = Ax(t) + Bu(t).$$

We can optimize over an infinite horizon rather than a finite one, and we obtain a policy from any state rather than just a sequence of controls. Q and R need to be positive definite matrices for this to work. This gives us the **linear quadratic regulator (LQR)**. With LQR, the optimal value function (called **cost to go**) is quadratic, and the optimal policy is linear. The policy looks like $u = -Kx$, where finding the matrix K requires solving an algebraic **Riccati equation**—no local optimization, no value iteration, no policy iteration are needed!

Because of the ease of finding the optimal policy, LQR finds many uses in practice despite the fact that real problems seldom actually have quadratic costs and linear dynamics. A really useful method is called **iterative**

LQR (ILQR), which works by starting with a solution and then iteratively computing a linear approximation of the dynamics and a quadratic approximation of the cost around it, then solving the resulting LQR system to arrive at a new solution. Variants of LQR are also often used for trajectory tracking.

26.6 Planning Uncertain Movements

In robotics, uncertainty arises from partial observability of the environment and from the stochastic (or unmodeled) effects of the robot's actions. Errors can also arise from the use of approximation algorithms such as particle filtering, which does not give the robot an exact belief state even if the environment is modeled perfectly.

The majority of today's robots use deterministic algorithms for decision making, such as the path-planning algorithms of the previous section, or the search algorithms that were introduced in Chapter 3. These deterministic algorithms are adapted in two ways: first, they deal with the continuous state space by turning it into a discrete space (for example with visibility graphs or cell decomposition). Second, they deal with uncertainty in the current state by choosing the **most likely state** from the probability distribution produced by the state estimation algorithm. That approach makes the computation faster and makes a better fit for the deterministic search algorithms. In this section we discuss methods for dealing with uncertainty that are analogous to the more complex search algorithms covered in Chapter 4.

First, instead of deterministic plans, uncertainty calls for policies. We already discussed how trajectory tracking control turns a plan into a policy to compensate for errors in dynamics. Sometimes though, if the most likely hypothesis changes enough, tracking the plan designed for a different hypothesis is too suboptimal. This is where **online replanning** comes in: we can recompute a new plan based on the new belief. Many robots today use a technique called **model predictive control (MPC)**, where they plan for a shorter time horizon, but replan at every time step. (MPC is therefore closely related to real-time search and game-playing algorithms.) This effectively results in a policy: at every step, we run a planner and take the first action in the plan; if new information comes along, or we end up not where we expected, that's OK, because we are going to replan anyway and that will tell us what to do next.

Second, uncertainty calls for **information gathering** actions. When we consider only the information we have and make a plan based on it (this is called separating estimation from control), we are effectively solving (approximately) a new MDP at every step, corresponding to our current belief about where we are or how the world works. But in reality, uncertainty is better captured by the POMDP framework: there is something we don't directly observe, be it the robot's location or configuration, the location of objects in the world, or the parameters of the dynamics model itself—for example, where exactly is the center of mass of link two on this arm?

What we lose when we don't solve the POMDP is the ability to reason about *future information* the robot will get: in MDPs we only plan with what we know, not with what we *might* eventually know. Remember the value of information? Well, robots that plan using their current belief as if they will never find out anything more fail to account for the value of information. They will never take actions that seem suboptimal right now according to what they know, but that will actually result in a lot of information and enable the robot to do well.

What does such an action look like for a navigation robot? The robot could get close to a landmark to get a better estimate of where it is, even if that landmark is out of the way according to what it currently knows. This action is optimal only if the robot considers the new observations it will get, as opposed to looking only at the information it already has.

To get around this, robotics techniques sometimes define information gathering actions explicitly—such as moving a hand until it touches a surface (called **guarded movements**)—and make sure the robot does that before coming up with a plan for reaching its actual goal. Each guarded motion consists of (1) a motion command and (2) a termination condition, which is a predicate on the robot's sensor values saying when to stop.

Sometimes, the goal itself could be reached via a sequence of guarded moves guaranteed to succeed regardless of uncertainty. As an example, Figure 26.23 shows a two-dimensional configuration space with a narrow vertical hole. It could be the configuration space for insertion of a rectangular peg into a hole or a car key into the ignition. The motion commands are constant velocities. The termination conditions are contact with a surface. To model uncertainty in control, we assume that instead of moving in the commanded direction, the robot's actual motion lies in the cone C_v about it.

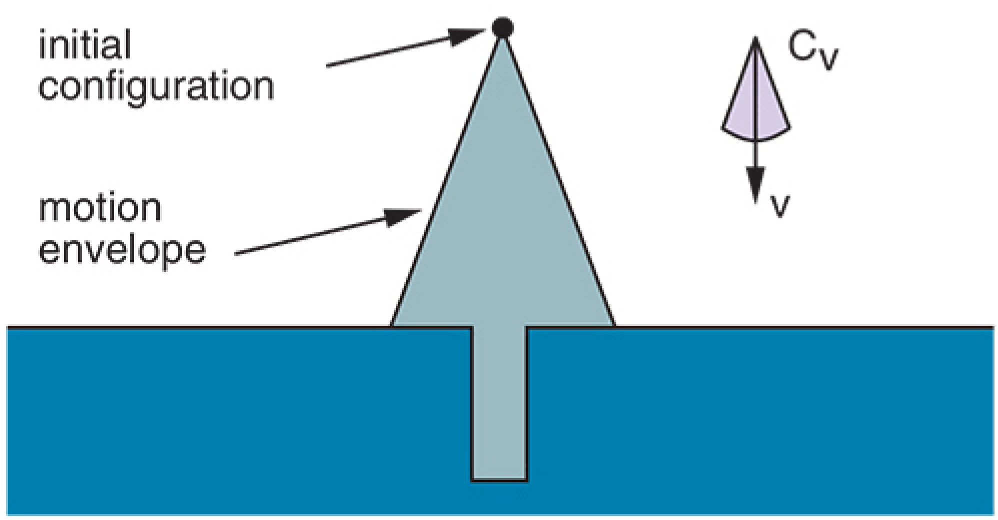


Figure 26.23 A two-dimensional environment, velocity uncertainty cone, and envelope of possible robot motions. The intended velocity is v, but with uncertainty the actual velocity could be anywhere in C_v, resulting in a final configuration somewhere in the motion envelope, which means we wouldn't know if we hit the hole or not.

The figure shows what would happen if the robot attempted to move straight down from the initial configuration. Because of the uncertainty in velocity, the robot could move anywhere in the conical envelope, possibly going into the hole, but more likely landing to one side of it. Because the robot would not then know which side of the hole it was on, it would not know which way to move.

A more sensible strategy is shown in Figures 26.24 and 26.25. In Figure 26.24, the robot deliberately moves to one side of the hole. The motion command is shown in the figure, and the termination test is contact with any surface. In Figure 26.25, a motion command is given that causes the robot to slide along the surface and into the hole. Because all possible velocities in the motion envelope are to the right, the robot will slide to the right whenever it is in contact with a horizontal surface.

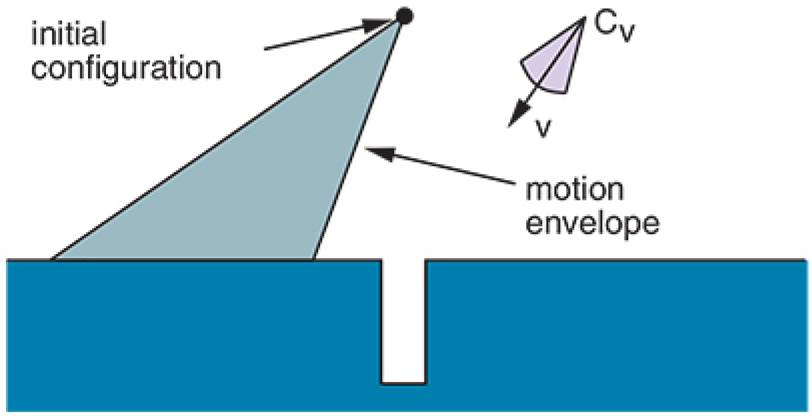


Figure 26.24 The first motion command and the resulting envelope of possible robot motions. No matter what actual motion ensues, we know the final configuration will be to the left of the hole.

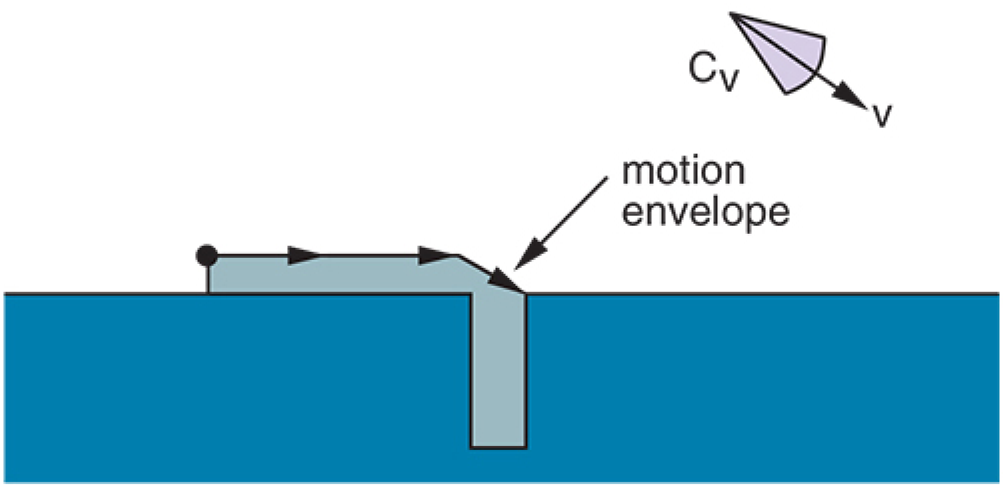


Figure 26.25 The second motion command and the envelope of possible motions. Even with error, we will eventually get into the

hole.

It will slide down the right-hand vertical edge of the hole when it touches it, because all possible velocities are down relative to a vertical surface. It will keep moving until it reaches the bottom of the hole, because that is its termination condition. In spite of the control uncertainty, all possible trajectories of the robot terminate in contact with the bottom of the hole—that is, unless surface irregularities cause the robot to stick in one place.

Other techniques beyond guarded movements change the cost function to incentivize actions we know will lead to information—like the **coastal navigation** heuristic which requires the robot to stay near known landmarks. More generally, techniques can incorporate the expected **information gain** (reduction of entropy of the belief) as a term in the cost function, leading to the robot explicitly reasoning about how much information each action might bring when deciding what to do. While more difficult computationally, such approaches have the advantage that the robot invents its own information gathering actions rather than relying on human-provided heuristics and scripted strategies that often lack flexibility.

26.7 Reinforcement Learning in Robotics

Thus far we have considered tasks in which the robot has access to the dynamics model of the world. In many tasks, it is very difficult to write down such a model, which puts us in the domain of reinforcement learning (RL).

One challenge of RL in robotics is the continuous nature of the state and action spaces, which we handle either through discretization, or, more commonly, through function approximation. Policies or value functions are represented as combinations of known useful features, or as deep neural networks. Neural nets can map from raw inputs directly to outputs, and thus largely avoid the need for feature engineering, but they do require more data.

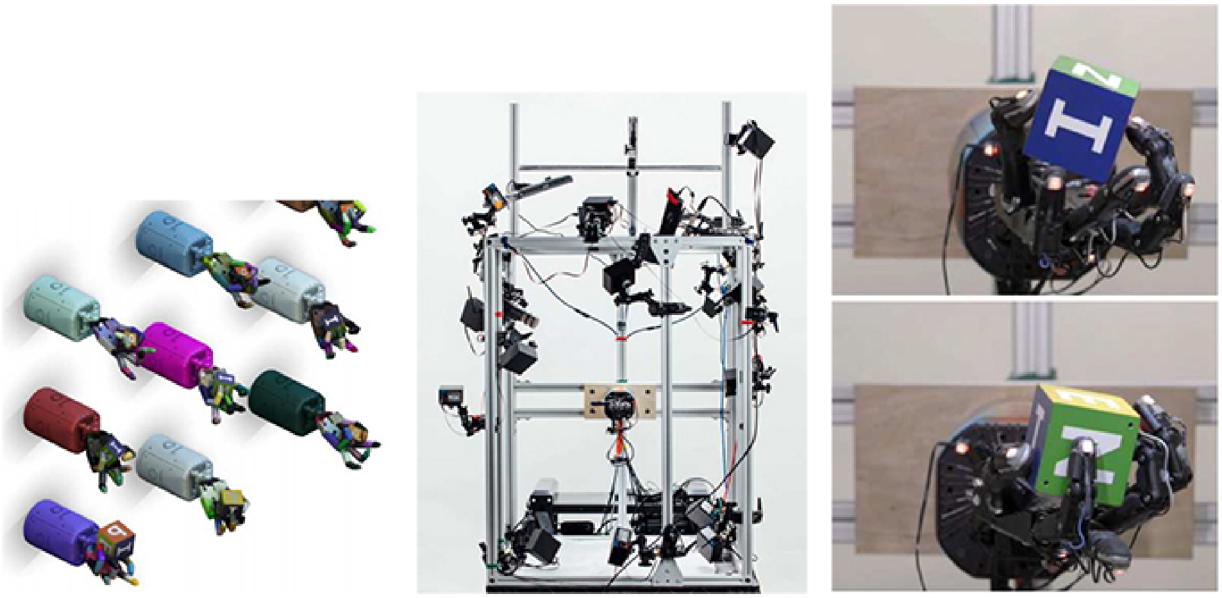
A bigger challenge is that robots operate in the real world. We have seen how reinforcement learning can be used to learn to play chess or Go by playing simulated games. But when a real robot moves in the real world, we have to make sure that its actions are safe (things break!), and we have to accept that progress will be slower than in a simulation because the world refuses to move faster than one second per second. Much of what is interesting about using reinforcement learning in robotics boils down to how we might reduce the real world sample complexity—the number of interactions with the physical world that the robot needs before it has learned how to do the task.

26.7.1 Exploiting models

A natural way to avoid the need for many real-world samples is to use as much knowledge of the world's dynamics as possible. For instance, we might not know exactly what the coefficient of friction or the mass of an object is, but we might have equations that describe the dynamics as a function of these parameters.

In such a case, **model-based reinforcement learning** (Chapter 23) is appealing, where the robot can alternate between fitting the dynamics parameters and computing a better policy. Even if the equations are incorrect because they fail to model every detail of physics, researchers have experimented with learning an error term, in addition to the parameters, that can compensate for the inaccuracy of the physical model. Or, we can abandon the equations and instead fit locally linear models of the world that each approximate the dynamics in a region of the state space, an approach that has been successful in getting robots to master complex dynamic tasks like juggling.

A model of the world can also be useful in reducing the sample complexity of model-free reinforcement learning methods by doing **sim-to-real** transfer: transferring policies that work in simulation to the real world. The idea is to use the model as a simulator for a policy search (Section 23.5). To learn a policy that transfers well, we can add noise to the model during training, thereby making the policy more robust. Or, we can train policies that will work with a *variety* of models by sampling different parameters in the simulations—sometimes referred to as **domain randomization**. An example is in Figure 26.26, where a dexterous manipulation task is trained in simulation by varying visual attributes, as well as physical attributes like friction or damping.



(a) (b) (c)

Figure 26.26 Training a robust policy. (a) Multiple simulations are run of a robot hand manipulating objects, with different randomized parameters for physics and lighting. Courtesy of Wojciech Zaremba. (b) The real-world environment, with a single robot hand in the center of a cage, surrounded by cameras and range finders. (c) Simulation and real-world training yields multiple different policies for grasping objects; here a pinch grasp and a quadpod grasp. Courtesy of OpenAI. See Andrychowicz *et al.* (2018a).

Finally, hybrid approaches that borrow ideas from both model-based and model-free algorithms are meant to give us the best of both. The hybrid approach originated with the Dyna architecture, where the idea was to iterate between acting and improving the policy, but the policy improvement would come in two complementary ways: 1) the standard model-free way of using the experience to directly update the policy, and 2) the model-based way of using the experience to fit a model, then plan with it to generate a policy.

More recent techniques have experimented with fitting local models, planning with them to generate actions, and using these actions as supervision to fit a policy, then iterating to get better and better models around the areas that the policy needs. This has been successfully applied in **end-to-end learning**, where the policy takes pixels as input and directly outputs torques as actions—it enabled the first demonstration of deep RL on physical robots.

Models can also be exploited for the purpose of ensuring **safe exploration**. Learning slowly but safely may be better than learning quickly but crashing and burning half way through. So arguably, more important than reducing real-world samples is reducing real-world samples in *dangerous* states—we don't want robots falling off cliffs, and we don't want them breaking our favorite mugs or, even worse, colliding with objects and people. An approximate model, with uncertainty associated to it (for example by considering a range of values for its parameters), can guide exploration and impose constraints on the actions that the robot is allowed to take in order to avoid these dangerous states. This is an active area of research in robotics and control.

26.7.2 Exploiting other information

Models are useful, but there is more we can do to further reduce sample complexity.

When setting up a reinforcement learning problem, we have to select the state and action spaces, the representation of the policy or value function, and the reward function we're using. These decisions have a large impact on how easy or how hard we are making the problem.

One approach is to use higher-level **motion primitives** instead of low-level actions like torque commands. A motion primitive is a parameterized skill that the robot has. For example, a robotic soccer player might have the skill of "pass the ball to the player at (x, y)." All the policy needs to do is to figure out how to combine them and set their parameters, instead of reinventing them. This approach often learns much faster than low-level approaches, but does restrict the space of possible behaviors that the robot can learn.

Another way to reduce the number of real-world samples required for learning is to reuse information from previous learning episodes on other tasks, rather than starting from scratch. This falls under the umbrella of **metalearning** or **transfer learning**.

Finally, people are a great source of information. In the next section, we talk about how to interact with people, and part of it is how to use their actions to guide the robot's learning.

26.8 Humans and Robots

Thus far, we've focused on a robot planning and learning how to act *in isolation*. This is useful for some robots, like the rovers we send out to explore distant planets on our behalf. But, for the most part, we do not build robots to work in isolation. We build them to help us, and to work in human environments, around and with us.

This raises two complementary challenges. First is optimizing reward when there are people acting in the same environment as the robot. We call this the **coordination problem** (see Section 17.1). When the robot's reward depends on not just its own actions, but also the actions that people take, the robot has to choose its actions in a way that meshes well with theirs. When the human and the robot are on the same team, this turns into **collaboration**.

Second is the challenge of optimizing for what people actually want. If a robot is to help people, its reward function needs to incentivize the actions that people want the robot to execute. Figuring out the right reward function (or policy) for the robot is itself an interaction problem. We will explore these two challenges in turn.

26.8.1 Coordination

Let's assume for now, as we have been, that the robot has access to a clearly defined reward function. But, instead of needing to optimize it in isolation, now the robot needs to optimize it around a human who is also acting. For example, as an autonomous car merges on the highway, it needs to negotiate the maneuver with the human driver coming in the target lane—should it accelerate and merge in front, or slow down and merge behind? Later, as it pulls to a stop sign, preparing to take a right, it has to watch out for the cyclist in the bicycle lane, and for the pedestrian about to step onto the crosswalk.

Or, consider a mobile robot in a hallway. Someone heading straight toward the robot steps slightly to the right, indicating which side of the robot they want to pass on. The robot has to respond, clarifying its intentions.

Humans as approximately rational agents

One way to formulate coordination with a human is to model it as a game between the robot and the human (Section 17.2). With this approach, we explicitly make the assumption that people are agents incentivized by objectives. This does not automatically mean that they are perfectly rational agents (i.e., find optimal solutions in the game), but it does mean that the robot can structure the way it reasons about the human via the notion of possible objectives that the human might have. In this game:

- the state of the environment captures the configurations of both the robot and human agents; call it $x = (x_R, x_H)$;
- each agent can take actions, u_R and u_H respectively;
- each agent has an objective that can be represented as a cost, J_R and J_H: each agent wants to get to its goal safely and efficiently;
- and, as in any game, each objective depends on the state and on the actions of *both* agents: $J_R(x, u_R, u_H)$ and $J_H(x, u_H, u_R)$. Think of the car-pedestrian interaction—the car should stop if the pedestrian crosses, and should go forward if the pedestrian waits.

Three important aspects complicate this game. First is that the human and the robot don't necessarily know each other's objectives. This makes it an **incomplete information game.**

Second is that the state and action spaces are *continuous*, as they've been throughout this chapter. We learned in Chapter 6 how to do tree search to tackle discrete games, but how do we tackle continuous spaces?

Third, even though at the high level the game model makes sense—humans do move, and they do have objectives—a human's behavior might not always be well-characterized as a solution to the game. The game comes with a computational challenge not only for the robot, but for us humans too. It requires thinking about what the robot will do in response to what the person does, which depends on what the robot thinks the person will do, and pretty soon we get to "what do you think I think you think I think"— it's turtles all the way down! Humans can't deal with all of that, and exhibit certain suboptimalities. This means that the robot should account for these suboptimalities.

So, then, what is an autonomous car to do when the coordination problem is this hard? We will do something similar to what we've done before in this chapter. For motion planning and control, we took an MDP and broke it up into planning a trajectory and then tracking it with a controller. Here too, we will take the game, and break it up into making predictions about human actions, and deciding what the robot should do given these predictions.

Predicting human action

Predicting human actions is hard because they depend on the robot's actions and vice versa. One trick that robots use is to pretend the person is ignoring the robot. The robot assumes people are noisily optimal with respect to their objective, which is unknown to the robot and is modeled as no longer dependent on the robot's actions: $J_H(x, u_H)$. In particular, the higher the value of an action for the objective (the lower the cost to go), the more likely the human is to take it. The robot can create a model for $P(u_H \mid x, J_H)$, for instance using the softmax function from page 862:

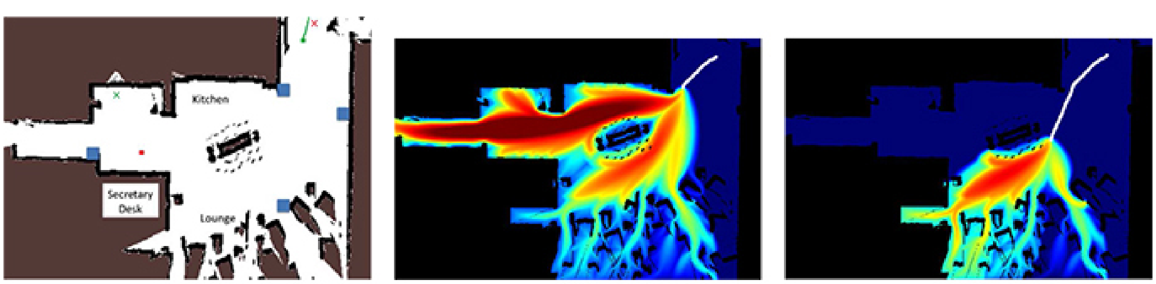
$$P(u_H|\, x, J_H) \propto e^{-Q(x,u_H;J_H)} \tag{26.8}$$

with $Q(x, u_H; J_H)$ the Q-value function corresponding to J_H (the negative sign is there because in robotics we like to minimize cost, not maximize reward). Note that the robot does not assume perfectly optimal actions, nor does it assume that the actions are chosen based on reasoning about the robot at all.

Armed with this model, the robot uses the human's ongoing actions as evidence about J_H. If we have an observation model for how human actions depend on the human's objective, each human action can be incorporated to update the robot's belief over what objective the person has:

$$b'(J_H) \propto b(J_H) P(u_H|x, J_H). \tag{26.9}$$

An example is in Figure 26.27: the robot is tracking a human's location and as the human moves, the robot updates its belief over human goals. As the human heads toward the windows, the robot increases the probability that the goal is to look out the window, and decreases the probability that the goal is going to the kitchen, which is in the other direction.



(a) (b) (c)

Figure 26.27 Making predictions by assuming that people are noisily rational given their goal: the robot uses the past actions to update a belief over what goal the person is heading to, and then uses the belief to make predictions about future actions. (a) The map of a room. (b) Predictions after seeing a small part of the person's trajectory (white path); (c) Predictions after seeing more human actions: the robot now knows that the person is not heading to the hallway on the left, because the path taken so far would be a poor path if that were the person's goal. Images courtesy of Brian D. Ziebart. See Ziebart *et al.* (2009).

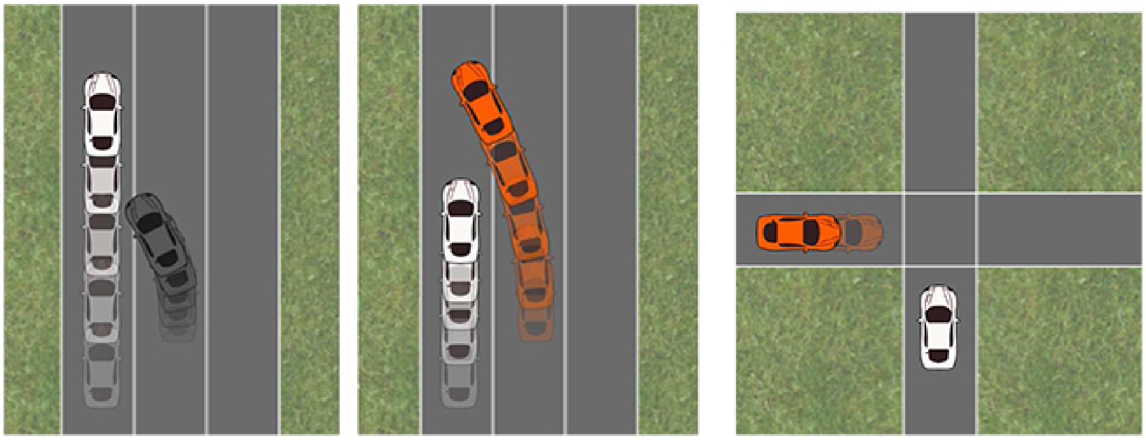
This is how the human's past actions end up informing the robot about what the human will do in the future. Having a belief about the human's goal helps the robot anticipate what next actions the human will take. The heatmap in the figure shows the robot's future predictions: red is most probable; blue least probable.

The same can happen in driving. We might not know how much another driver values efficiency, but if we see them accelerate as someone is trying to merge in front of them, we now know a bit more about them. And once we know that, we can better anticipate what they will do in the future —the same driver is likely to come closer behind us, or weave through traffic to get ahead.

Once the robot can make predictions about human future actions, it has reduced its problem to solving an MDP. The human actions complicate the transition function, but as long as the robot can anticipate what action the person will take from any future state, the robot can calculate $P(x' \mid x, u_R)$: it can compute $P(u_H \mid x)$ from $P(u_H \mid x, J_H)$ by marginalizing over J_H, and combine it with $P(x' \mid x, u_R, u_H)$, the transition (dynamics) function for how the world updates based on both the robot’s and the human’s actions. In Section 26.5 we focused on how to solve this in continuous state and action spaces for deterministic dynamics, and in Section 26.6 we discussed doing it with stochastic dynamics and uncertainty.

Splitting prediction from action makes it easier for the robot to handle interaction, but sacrifices performance much as splitting estimation from motion did, or splitting planning from control.

A robot with this split no longer understands that its actions can influence what people end up doing. In contrast, the robot in Figure 26.27 anticipates where people will go and then optimizes for reaching its own goal and avoiding collisions with them. In Figure 26.28, we have an autonomous car merging on the highway. If it just planned in reaction to other cars, it might have to wait a long time while other cars occupy its target lane. In contrast, a car that reasons about prediction and action jointly knows that different actions it could take will result in different reactions from the human. If it starts to assert itself, the other cars are likely to slow down a bit and make room. Roboticists are working towards coordinated interactions like this so robots can work better with humans.



(a) (b)

Figure 26.28 (a) Left: An autonomous car (middle lane) predicts that the human driver (left lane) wants to keep going forward, and plans a trajectory that slows down and merges behind. Right: The car accounts for the influence its actions can have on human actions, and realizes it can merge in front and rely on the human driver to slow down. (b) That same algorithm produces an unusual strategy at an intersection: the car realizes that it can make it more likely for the person (bottom) to proceed faster through the intersection by starting to inch backwards. Images courtesy of Anca Dragan. See Sadigh *et al.* (2016).

Human predictions about the robot

Incomplete information is often two-sided: the robot does not know the human's objective and the human, in turn, does not know the *robot's* objective—people need to be making predictions about robots. As robot designers, we are not in charge of how the human makes predictions; we can only control what the robot does. However, the robot can act in a way to make it *easier* for the human to make correct predictions. The robot can assume that the human is using something roughly analogous to Equation (26.8) to estimate the robot's objective J_R, and thus the robot will act so that its true objective can be easily inferred.

A special case of the game is when the human and the robot are on the same team, working toward the same goal or objective: $J_H = J_R$. Imagine getting a personal home robot that is helping you make dinner or clean up—these are examples of **collaboration**.

We can now define a **joint agent** whose actions are tuples of human-robot actions, $(u_H, u_R$ and who optimizes for $J_H\ (x, u_H, u_R = J_R(x, u_R, u_H)$, and we're solving a regular planning problem. We compute the optimal plan or policy for the joint agent, and voila, we now know what the robot and human should do.

This would work really well if people were perfectly optimal. The robot would do its part of the joint plan, the human theirs. Unfortunately, in practice, people don't seem to follow the perfectly laid out joint-agent plan; they have a mind of their own! We've already learned one way to handle this though, back in Section 26.6. We called it **model predictive control (MPC)**: the idea was to come up with a plan, execute the first action, and then replan. That way, the robot always adapts its plan to what the human is actually doing.

Let's work through an example. Suppose you and the robot are in your kitchen, and have decided to make waffles. You are slightly closer to the fridge, so the optimal joint plan would have you grab the eggs and milk from the fridge, while the robot fetches the flour from the cabinet. The robot knows this because it can measure quite precisely where everyone is. But suppose you start heading for the flour cabinet. You are going against the optimal joint plan. Rather than sticking to it and stubbornly also going for the flour, the MPC robot recalculates the optimal plan, and now that you are close enough to the flour it is best for the robot to grab the waffle iron instead.

If we know that people might deviate from optimality, we can account for it ahead of time. In our example, the robot can try to anticipate that you are going for the flour the moment you take your first step (say, using the prediction technique above). Even if it is still technically optimal for you to turn around and head for the fridge, the robot should not assume that's what is going to happen. Instead, the robot can compute a plan in which you keep doing what you seem to want.

Humans as black box agents

We don't have to treat people as objective-driven, intentional agents to get robots to coordinate with us. An alternative model is that the human is merely some agent whose policy π_H "messes" with the environment dynamics. The robot does not know π_H, but can model the problem as needing to act in an MDP with unknown dynamics. We have seen this before: for general agents in Chapter 23, and for robots in particular in Section 26.7.

The robot can fit a policy model π_H to human data, and use it to compute an optimal policy for itself. Due to scarcity of data, this has been mostly used so far at the task level. For instance, robots have learned through interaction what actions people tend to take (in response to its own actions) for the task of placing and drilling screws in an industrial assembly task.

Then there is also the model-free reinforcement learning alternative: the robot can start with some initial policy or value function, and keep improving it over time via trial and error.

26.8.2 Learning to do what humans want

Another way interaction with humans comes into robotics is in J_R itself—the robot's cost or reward function. The framework of rational agents and the associated algorithms reduce the problem of generating good behavior to specifying a good reward function. But for robots, as for many other AI agents, getting the cost right is still difficult.

Take autonomous cars: we want them to reach the destination, to be safe, to drive comfortably for their passengers, to obey traffic laws, etc. A designer of such a system needs to trade off these different components of the cost function. The designer's task is hard because robots are built to help end users, and not every end user is the same. We all have different preferences for how aggressively we want our car to drive, etc.

Below, we explore two alternatives for trying to get robot behavior to match what we actually want the robot to do. The first is to learn a cost function from human input. The second is to bypass the cost function and imitate human demonstrations of the task.

Preference learning: Learning cost functions

Imagine that an end user is showing a robot how to do a task. For instance, they are driving the car in the way they would like it to be driven by the robot. Can you think of a way for the robot to use these actions—we call them "demonstrations"—to figure out what cost function it should optimize?

We have actually already seen the answer to this back in Section 26.8.1. There, the setup was a little different: we had another person taking actions in the same space as the robot, and the robot needed to predict what the person would do. But one technique we went over for making these predictions was to assume that people act to noisily optimize some cost function J_H, and we can use their ongoing actions as evidence about what cost function that is. We can do the same here, except not for the purpose of predicting human behavior in the future, but rather acquiring the cost function the robot itself should optimize. If the person drives defensively, the cost function that will explain their actions will put a lot of weight on safety and less so on efficiency. The robot can adopt this cost function as its own and optimize it when driving the car itself.

Roboticists have experimented with different algorithms for making this cost inference computationally tractable. In Figure 26.29, we see an example of teaching a robot to prefer staying on the road to going over the grassy terrain. Traditionally in such methods, the cost function has been represented as a combination of hand-crafted features, but recent work has also studied how to represent it using a deep neural network, without feature engineering.

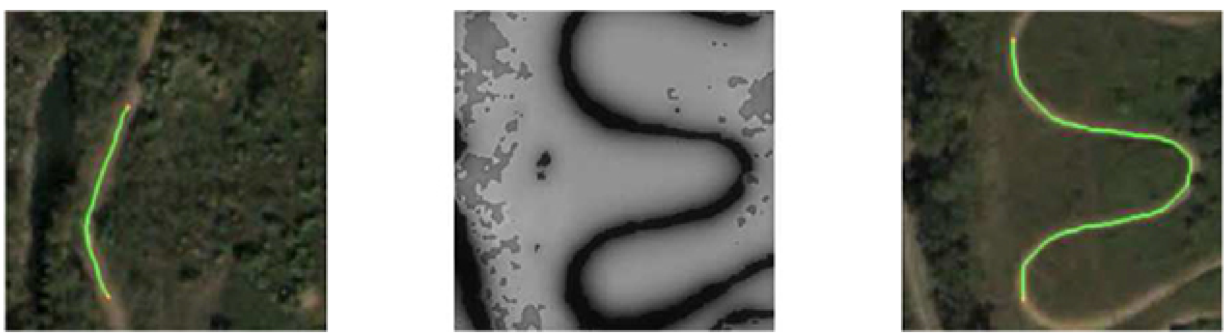


Figure 26.29 Left: A mobile robot is shown a demonstration that stays on the dirt road. Middle: The robot infers the desired cost function, and uses it in a new scene, knowing to put lower cost on the road there. Right: The robot plans a path for the new scene that also stays on the road, reproducing the preferences behind the

demonstration. Images courtesy of Nathan Ratliff and James A. Bagnell. See Ratliff *et al.* (2006).

There are other ways for a person to provide input. A person could use language rather than demonstration to instruct the robot. A person could act as a critic, watching the robot perform a task one way (or two ways) and then saying how well the task was done (or which way was better), or giving advice on how to improve.

Learning policies directly via imitation

An alternative is to bypass cost functions and learn the desired robot *policy* directly. In our car example, the human's demonstrations make for a convenient data set of states labeled by the action the robot should take at each state: $D = \{(x_i, u_i)\}$. The robot can run supervised learning to fit a policy $\pi : x \mapsto u$, and execute that policy. This is called **imitation learning** or **behavioral cloning**.

A challenge with this approach is in **generalization** to new states. The robot does not know why the actions in its database have been marked as optimal. It has no causal rule; all it can do is run a supervised learning algorithm to try to learn a policy that will generalize to unknown states. However, there is no guarantee that the generalization will be correct.

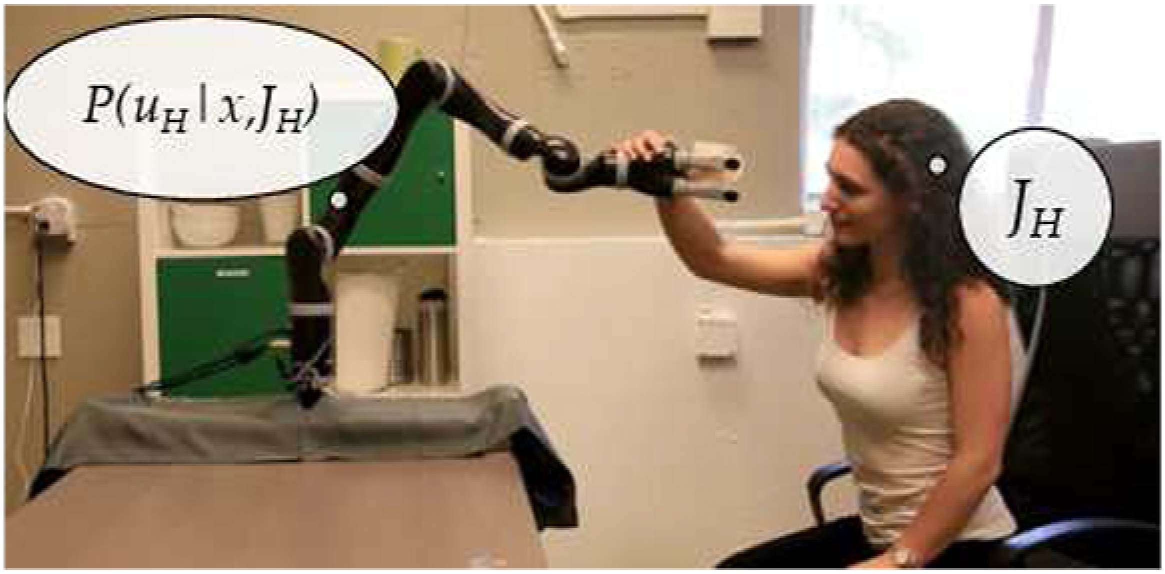


Figure 26.30 A human teacher pushes the robot down to teach it to stay closer to the table. The robot appropriately updates its understanding of the desired cost function and starts optimizing it. Courtesy of Anca Dragan. See Bajcsy *et al.* (2017).

The ALVINN autonomous car project used this approach, and found that even when starting from a state in D, π will make small errors, which will take the car off the demonstrated trajectory. There, π will make a larger error, which will take the car even further off the desired course.

We can address this at training time if we interleave collecting labels and learning: start with a demonstration, learn a policy, then roll out that policy and ask the human for what action to take at every state along the way, then repeat. The robot then learns how to correct its mistakes as it deviates from the human's desired actions.

Alternatively, we can address it by leveraging reinforcement learning. The robot can fit a dynamics model based on the demonstrations, and then use optimal control (Section 26.5.4) to generate a policy that optimizes for staying close to the demonstration. A version of this has been used to perform very challenging maneuvers at an expert level in a small radiocontrolled helicopter (see Figure 23.9(b)).

The DAGGER (Data Aggregation) system starts with a human expert demonstration. From that it learns a policy, π_1 and uses the policy to generate a data set D. Then from D it generates a new policy π_2 that best imitates the original human data. This repeats, and on the nth iteration it uses π_n to generate more data, to be added to D, which is then used to create π_{n+1}. In other words, at each iteration the system gathers new data under the current policy and trains the next policy using all the data gathered so far.

Related recent techniques use **adversarial training**: they alternate between training a classifier to distinguish between the robot's learned policy and the human's demonstrations, and training a new robot policy via reinforcement learning to fool the classifier. These advances enable the robot to handle states that are near demonstrations, but generalization to far-off states or to new dynamics is a work in progress.

Teaching interfaces and the correspondence problem. So far, we have imagined the case of an autonomous car or an autonomous helicopter, for which human demonstrations use the same actions that the robot can take itself: accelerating, braking, and steering. But what happens if we do this for tasks like cleaning up the kitchen table? We have two choices here: either the person demonstrates using their own body while the robot watches, or the person physically guides the robot's effectors.

The first approach is appealing because it comes naturally to end users. Unfortunately, it suffers from the **correspondence problem:** how to map human actions onto robot actions. People have different kinematics and dynamics than robots. Not only does that make it difficult to *translate* or *retarget* human motion onto robot motion (e.g., retargeting a five-finger human grasp to a two-finger robot grasp), but often the high-level strategy a person might use is not appropriate for the robot.

The second approach, where the human teacher moves the robot's effectors into the right positions, is called **kinesthetic teaching.** It is not easy for humans to teach this way, especially to teach robots with multiple joints. The teacher needs to coordinate all the degrees of freedom as it is guiding the arm through the task. Researchers have thus investigated alternatives, like demonstrating **keyframes** as opposed to continuous trajectories, as well as the use of **visual programming** to enable end users to program primitives for a task rather than demonstrate from scratch (Figure 26.31). Sometimes both approaches are combined.

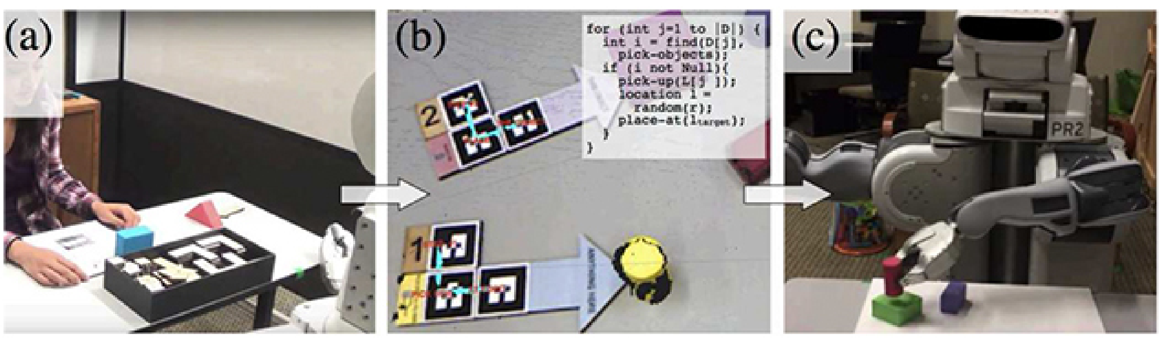


Figure 26.31 A programming interface that involves placing specially designed blocks in the robot's workspace to select objects and specify high-level actions. Images courtesy of Maya Cakmak. See Sefidgar *et al.* (2017).

26.9 Alternative Robotic Frameworks

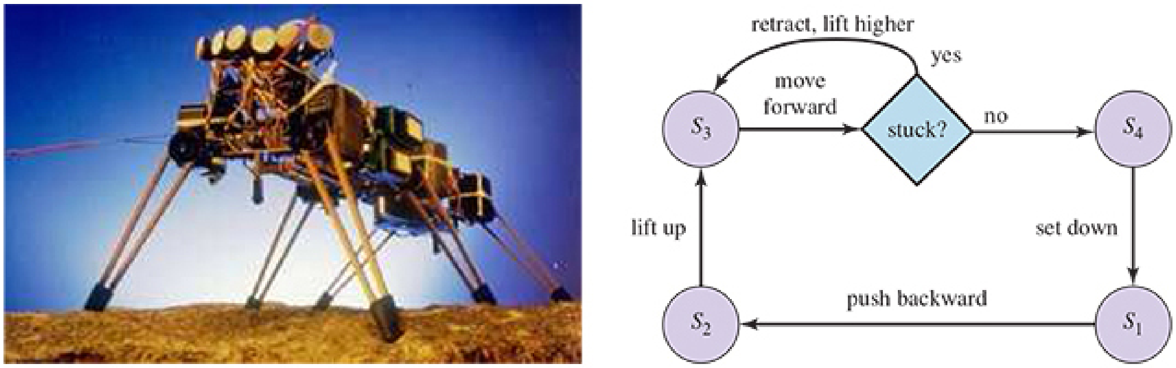
Thus far, we have taken a view of robotics based on the notion of defining or learning a reward function, and having the robot optimize that reward function (be it via planning or learning), sometimes in coordination or collaboration with humans. This is a **deliberative** view of robotics, to be contrasted with a **reactive** view.

26.9.1 Reactive controllers

In some cases, it is easier to set up a good policy for a robot than to model the world and plan. Then, instead of a *rational* agent, we have a *reflex* agent.

For example, picture a legged robot that attempts to lift a leg over an obstacle. We could give this robot a rule that says lift the leg a small height h and move it forward, and if the leg encounters an obstacle, move it back and start again at a higher height. You could say that h is modeling an aspect of the world, but we can also think of h as an auxiliary variable of the robot controller, devoid of direct physical meaning.

One such example is the six-legged (hexapod) robot, shown in Figure 26.32(a), designed for walking through rough terrain. The robot's sensors are inadequate to obtain accurate models of the terrain for path planning. Moreover, even if we added high-precision cameras and rangefinders, the 12 degrees of freedom (two for each leg) would render the resulting path planning problem computationally difficult.



(a) (b)

Figure 26.32 (a) Genghis, a hexapod robot. (Image courtesy of Rodney A. Brooks.) (b) An augmented finite state machine (AFSM) that controls one leg. The AFSM reacts to sensor feedback: if a leg is stuck during the forward swinging phase, it will be lifted increasingly higher.

It is possible, nonetheless, to specify a controller directly without an explicit environmental model. (We have already seen this with the PD controller, which was able to keep a complex robot arm on target *without* an explicit model of the robot dynamics.)

For the hexapod robot we first choose a **gait**, or pattern of movement of the limbs. One statically stable gait is to first move the right front, right rear, and left center legs forward (keeping the other three fixed), and then move the other three. This gait works well on flat terrain. On rugged terrain, obstacles may prevent a leg from swinging forward. This problem can be overcome by a remarkably simple control rule: *when a leg's forward motion is blocked, simply retract it, lift it higher, and try again.* The resulting controller is shown in Figure 26.32(b) as a simple finite state machine; it constitutes a reflex agent with state, where the internal state is represented by the index of the current machine state (s_1 through s_4).

26.9.2 Subsumption architectures

The **subsumption architecture** (Brooks, 1986) is a framework for assembling reactive controllers out of finite state machines. Nodes in these machines may contain tests for certain sensor variables, in which case the execution trace of a finite state machine is conditioned on the outcome of such a test. Arcs can be tagged with messages that will be generated when traversing them, and that are sent to the robot's motors or to other finite state machines. Additionally, finite state machines possess internal timers (clocks) that control the time it takes to traverse an arc. The resulting machines are called **augmented finite state machines (AFSMs)**, where the augmentation refers to the use of clocks.

An example of a simple AFSM is the four-state machine we just talked about, shown in Figure 26.32(b). This AFSM implements a cyclic controller, whose execution mostly does not rely on environmental feedback. The forward swing phase, however, does rely on sensor feedback. If the leg is stuck, meaning that it has failed to execute the forward swing, the robot retracts the leg, lifts it up a little higher, and attempts to execute the forward swing once again. Thus, the controller is able to *react* to contingencies arising from the interplay of the robot and its environment.

The subsumption architecture offers additional primitives for synchronizing AFSMs, and for combining output values of multiple, possibly conflicting AFSMs. In this way, it enables the programmer to compose increasingly complex controllers in a bottom-up fashion. In our example, we might begin with AFSMs for individual legs, followed by an AFSM for coordinating multiple legs. On top of this, we might implement higher-level behaviors such as collision avoidance, which might involve backing up and turning.

The idea of composing robot controllers from AFSMs is quite intriguing. Imagine how difficult it would be to generate the same behavior with any of the configuration-space pathplanning algorithms described in the previous section. First, we would need an accurate model of the terrain. The configuration space of a robot with six legs, each of which is driven by two independent motors, totals 18 dimensions (12 dimensions for the configuration of the legs, and six for the location and orientation of the robot relative to its environment). Even if our computers were fast enough to find paths in such high-dimensional spaces, we would have to worry about nasty effects such as the robot sliding down a slope.

Because of such stochastic effects, a single path through configuration space would almost certainly be too brittle, and even a PID controller might not be able to cope with such contingencies. In other words, generating motion behavior deliberately is simply too complex a problem in some cases for present-day robot motion planning algorithms.

Unfortunately, the subsumption architecture has its own problems. First, the AFSMs are driven by raw sensor input, an arrangement that works if the sensor data is reliable and contains all necessary information for decision making, but fails if sensor data has to be integrated in nontrivial ways over time. Subsumption-style controllers have therefore mostly been applied to simple tasks, such as following a wall or moving toward visible light sources.

Second, the lack of deliberation makes it difficult to change the robot’s goals. A robot with a subsumption architecture usually does just one task, and it has no notion of how to modify its controls to accommodate different goals (just like the dung beetle on page 59).

Third, in many real-world problems, the policy we want is often too complex to encode explicitly. Think about the example from Figure 26.28, of an autonomous car needing to negotiate a lane change with a human driver. We might start off with a simple policy that goes into the target lane. But when we test the car, we find out that not every driver in the target lane will slow down to let the car in. We might then add a bit more complexity: make the car nudge towards the target lane, wait for a response form the driver in that lane, and then either proceed or retreat back. But then we test the car, and realize that the nudging needs to happen at a different speed depending on the speed of the vehicle in the target lane, on whether there is another vehicle in front in the target lane, on whether there is a vehicle behind the car in the initial, and so on. The number of conditions that we need to consider to determine the right course of action can be very large, even for such a deceptively simple maneuver. This in turn presents scalability challenges for subsumption-style architectures.

All that said, robotics is a complex problem with many approaches: deliberative, reactive, or a mixture thereof; based on physics, cognitive models, data, or a mixture thereof. The right approach is still a subject for debate, scientific inquiry, and engineering prowess.

26.10 Application Domains

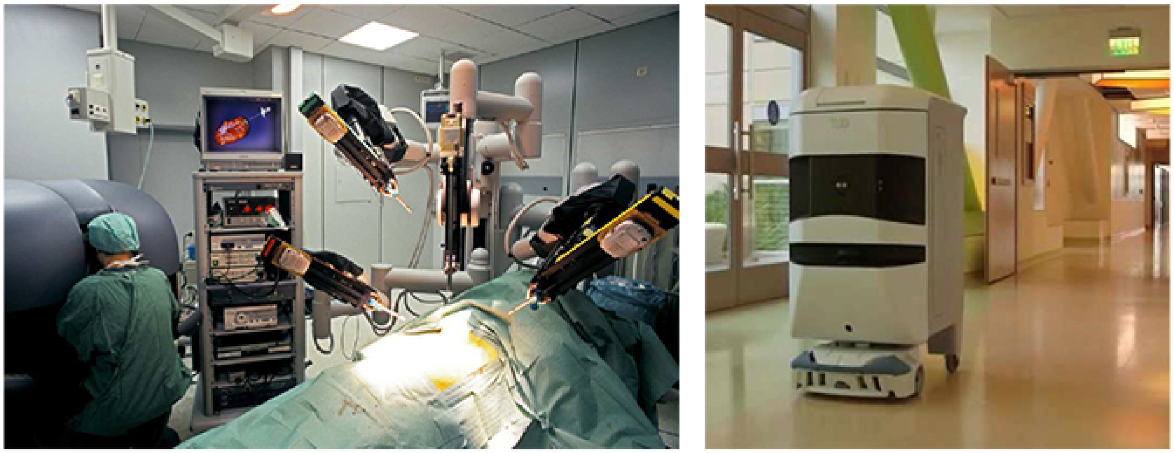
Robotic technology is already permeating our world, and has the potential to improve our independence, health, and productivity. Here are some example applications.

Home care: Robots have started to enter the home to care for older adults and people with motor impairments, assisting them with activities of daily living and enabling them to live more independently. These include wheelchairs and wheelchair-mounted arms like the Kinova arm from Figure 26.1(b). Even though they start off as being operated by a human directly, these robots are gaining more and more autonomy. On the horizon are robots operated by **brain-machine interfaces**, which have been shown to enable people with quadriplegia to use a robot arm to grasp objects and even feed themselves (Figure 26.33(a)). Related to these are prosthetic limbs that intelligently respond to our actions, and exoskeletons that give us superhuman strength or enable people who can’t control their muscles from the waist down to walk again.

Figure 26.33 (a) A patient with a brain-machine interface controlling a robot arm to grab a drink. Image courtesy of Brown University. (b) Roomba, the robot vacuum cleaner. Photo by HANDOUT/KRT/Newscom.

Personal robots are meant to assist us with daily tasks like cleaning and organizing, freeing up our time. Although manipulation still has a way to go before it can operate seamlessly in messy, unstructured human environments, navigation has made some headway. In particular, many homes already enjoy a mobile robot vacuum cleaner like the one in Figure 26.33(b).

Health care: Robots assist and augment surgeons, enabling more precise, minimally invasive, safer procedures with better patient outcomes. The Da Vinci surgical robot from Figure 26.34(a) is now widely deployed at hospitals in the U.S.



(a) (b)

Figure 26.34 (a) Surgical robot in the operating room. Photo by Patrick Landmann/Science Source. (b) Hospital delivery robot. Photo by Wired.

Services: Mobile robots help out in office buildings, hotels, and hospitals. Savioke has put robots in hotels delivering products like towels or toothpaste to your room. The Helpmate and TUG robots carry food and medicine in hospitals (Figure 26.34(b)), while Diligent Robotics' Moxi robot helps out nurses with back-end logistical responsibilities. Co-Bot roams the halls of Carnegie Mellon University, ready to guide you to someone's office. We can also use **telepresence robots** like the Beam to attend meetings and conferences remotely, or check in on our grandparents.

Autonomous cars: Some of us are occasionally distracted while driving, by cell phone calls, texts, or other distractions. The sad result: more than a million people die every year in traffic accidents. Further, many of us spend a lot of time driving and would like to recapture some of that time. All this has led to a massive ongoing effort to deploy autonomous cars.

Prototypes have existed since the 1980s, but progress was stimulated by the 2005 DARPA Grand Challenge, an autonomous vehicle race over 200 challenging kilometers of unrehearsed desert terrain. Stanford's Stanley vehicle completed the course in less than seven hours, winning a $2 million prize and a place in the National Museum of American History. Figure 26.35(a) depicts BOSS, which in 2007 won the DARPA Urban Challenge, a complicated road race on city streets where robots faced other robots and had to obey traffic rules.

(a) (b)

Figure 26.35 (a) Autonomous car BOSS which won the DARPA Urban Challenge. Photo by Tangi Quemener/AFP/Getty Images/Newscom. Courtesy of Sebastian Thrun. (b) Aerial view showing the perception and predictions of the Waymo autonomous car (white vehicle with green track). Other vehicles (blue boxes) and pedestrians (orange boxes) are shown with anticipated trajectories. Road/sidewalk boundaries are in yellow. Photo courtesy of Waymo.

In 2009, Google started an autonomous driving project (featuring many of the researchers who had worked on Stanley and BOSS), which has now spun off as Waymo. In 2018 Waymo started driverless testing (with nobody in the driver seat) in the suburbs of Pheonix, Arizona. In the meantime, other autonomous driving companies and ride-sharing companies are working on developing their own technology, while car manufacturers have been selling cars with more and more assistive intelligence, such as Tesla's **driver assist**, which is meant for highway driving. Other companies are targeting non-highway driving applications including college campuses and retirement communities. Still other companies are focused on non-passenger applications such as trucking, grocery delivery, and valet parking.

Entertainment: Disney has been using robots (under the name **animatronics**) in their parks since 1963. Originally, these robots were restricted to hand-designed, open-loop, unvarying motion (and speech), but since 2009 a version called **autonomatronics** can generate autonomous actions. Robots also take the form of intelligent toys for children; for example, Anki's Cozmo plays games with children and may pound the table with frustration when it loses. Finally, quadrotors like Skydio's R1 from Figure 26.2(b) act as personal photographers and videographers, following us around to take action shots as we ski or bike.

Exploration and hazardous environments: Robots have gone where no human has gone before, including the surface of Mars. Robotic arms assist astronauts in deploying and retrieving satellites and in building the International Space Station. Robots also help explore under the sea. They are routinely used to acquire maps of sunken ships. Figure 26.36 shows a robot mapping an abandoned coal mine, along with a 3D model of the mine acquired using range sensors. In 1996, a team of researches released a legged robot into the crater of an active volcano to acquire data for climate research. Robots are becoming very effective tools for gathering information in domains that are difficult (or dangerous) for people to access.

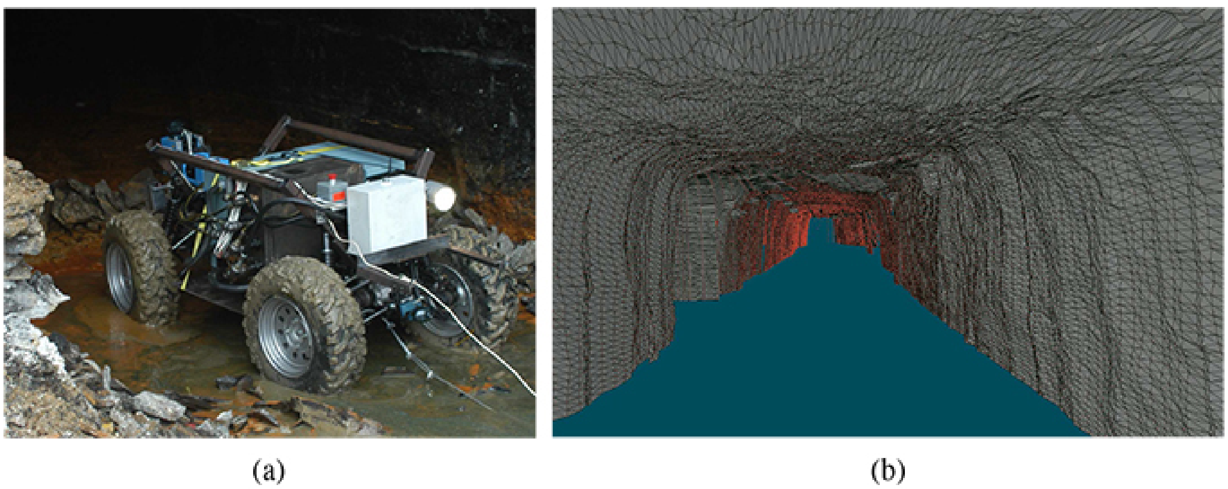


Figure 26.36 (a) A robot mapping an abandoned coal mine. (b) A 3D map of the mine acquired by the robot. Courtesy of Sebastian Thrun.

Robots have assisted people in cleaning up nuclear waste, most notably in Three Mile Island, Chernobyl, and Fukushima. Robots were present after the collapse of the World Trade Center, where they entered structures deemed too dangerous for human search and rescue crews. Here too, these robots are initially deployed via teleoperation, and as technology advances they are becoming more and more autonomous, with a human operator in charge but not having to specify every single command.

Industry: The majority of robots today are deployed in factories, automating tasks that are difficult, dangerous, or dull for humans. (The majority of factory robots are in automobile factories.) Automating these tasks is a positive in terms of efficiently producing what society needs. At the same time, it also means displacing some human workers from their jobs. This has important policy and economics implications—the need for retraining and education, the need for a fair division of resources, etc. These topics are discussed further in Section 28.3.5.

Summary

Robotics is about physically embodied agents, which can change the state of the physical world. In this chapter, we have learned the following:

- The most common types of robots are **manipulators** (robot arms) and **mobile robots**. They have sensors for perceiving the world and **actuators** that produce motion, which then affects the world via **effectors**.
- The general robotics problem involves **stochasticity** (which can be handled by MDPs), **partial observability** (which can be handled by POMDPs), and acting with and around **other agents** (which can be handled with game theory). The problem is made even harder by the fact that most robots work in continuous and high-dimensional state and action spaces. They also operate in the real world, which refuses to run faster than real time and in which failures lead to real things being damaged, with no "undo" capability.
- Ideally, the robot would solve the entire problem in one go: observations in the form of raw sensor feeds go in, and actions in the form of torques or currents to the motors come out. In practice though, this is too daunting, and roboticists typically decouple different aspects of the problem and treat them independently.
- We typically separate perception (estimation) from action (motion generation). Perception in robotics involves computer vision to recognize the surroundings through cameras, but also localization and mapping.
- Robotic perception concerns itself with estimating decision-relevant quantities from sensor data. To do so, we need an internal

representation and a method for updating this internal representation over time.

- **Probabilistic filtering algorithms** such as particle filters and Kalman filters are useful for robot perception. These techniques maintain the belief state, a posterior distribution over state variables.
- For generating motion, we use **configuration spaces**, where a point specifies everything we need to know to locate every **body point** on the robot. For instance, for a robot arm with two joints, a configuration consists of the two joint angles.
- We typically decouple the motion generation problem into **motion planning**, concerned with producing a plan, and **trajectory tracking control**, concerned with producing a policy for control inputs (actuator commands) that results in executing the plan.
- Motion planning can be solved via graph search using **cell decomposition**; using **randomized motion planning** algorithms, which sample milestones in the continuous configuration space; or using **trajectory optimization**, which can iteratively push a straight-line path out of collision by leveraging a **signed distance field**.
- A path found by a search algorithm can be executed using the path as the reference trajectory for a **PID controller**, which constantly corrects for errors between where the robot is and where it is supposed to be, or via **computed torque control,** which adds a **feedforward** term that makes use of **inverse dynamics** to compute roughly what torque to send to make progress along the trajectory.
- **Optimal control** unites motion planning and trajectory tracking by computing an optimal trajectory directly over control inputs. This is especially easy when we have quadratic costs and linear dynamics, resulting in a linear quadratic regulator (**LQR**). Popular methods make

use of this by linearizing the dynamics and computing second-order approximations of the cost (**ILQR**).

- Planning under uncertainty unites perception and action by **online replanning** (such as model predictive control) and **information gathering** actions that aid perception.
- Reinforcement learning is applied in robotics, with techniques striving to reduce the required number of interactions with the real world. Such techniques tend to exploit models, be it estimating models and using them to plan, or training policies that are robust with respect to different possible model parameters.
- Interaction with humans requires the ability to **coordinate** the robot's actions with theirs, which can be formulated as a game. We usually decompose the solution into **prediction**, in which we use the person's ongoing actions to estimate what they will do in the future, and **action**, in which we use the predictions to compute the optimal motion for the robot.
- Helping humans also requires the ability to learn or infer what they want. Robots can approach this by learning the desired cost function they should optimize from human input, such as demonstrations, corrections, or instruction in natural language. Alternatively, robots can imitate human behavior, and use reinforcement learning to help tackle the challenge of generalization to new states.

Bibliographical and Historical Notes

The word **robot** was popularized by Czech playwright Karel Čapek in his 1920 play *R.U.R.*(Rossum's Universal Robots). The robots, which were grown chemically rather than constructed mechanically, end up resenting their masters and decide to take over. It appears that it was Čapek's brother, Josef, who first combined the Czech words "robota" (obligatory work) and "robotnik" (serf) to yield "robot" in his 1917 short story *Opilec* (Glanc, 1978). The term *robotics* was invented for a science fiction story (Asimov, 1950).

The idea of an autonomous machine predates the word "robot" by thousands of years. In 7th century BCE Greek mythology, a robot named Talos was built by Hephaistos, the Greek god of metallurgy, to protect the island of Crete. The legend is that the sorceress Medea defeated Talos by promising him immortality but then draining his life fluid. Thus, this is the first example of a robot making a mistake in the process of changing its objective function. In 322 BCE, Aristotle anticipated technological unemployment, speculating "If every tool, when ordered, or even of its own accord, could do the work that befits it... then there would be no need either of apprentices for the master workers or of slaves for the lords."

In the 3rd century BCE an actual humanoid robot called the Servant of Philon could pour wine or water into a cup; a series of valves cut off the flow at the right time. Wonderful automata were built in the 18th century—Jacques Vaucanson's mechanical duck from 1738 being one early example—but the complex behaviors they exhibited were entirely fixed in advance. Possibly the earliest example of a programmable robot-like device was the Jacquard loom (1805), described on page 33.

Grey Walter’s “turtle,” built in 1948, could be considered the first autonomous mobile robot, although its control system was not programmable. The “Hopkins Beast,” built in 1960 at Johns Hopkins University, was much more sophisticated; it had sonar and photocell sensors, pattern-recognition hardware, and could recognize the cover plate of a standard AC power outlet. It was capable of searching for outlets, plugging itself in, and then recharging its batteries! Still, the Beast had a limited repertoire of skills.

The first general-purpose mobile robot was “Shakey,” developed at what was then the Stanford Research Institute (now SRI) in the late 1960s (Fikes and Nilsson, 1971; Nilsson, 1984). Shakey was the first robot to integrate perception, planning, and execution, and much subsequent research in AI was influenced by this remarkable achievement. Shakey appears on the cover of this book with project leader Charlie Rosen (1917–2002). Other influential projects include the Stanford Cart and the CMU Rover (Moravec, 1983). Cox and Wilfong (1990) describe classic work on autonomous vehicles.

The first commercial robot was an arm called UNIMATE, for *universal automation,* developed by Joseph Engelberger and George Devol in their compnay, Unimation. In 1961, the first UNIMATE robot was sold to General Motors for use in manufacturing TV picture tubes. 1961 was also the year when Devol obtained the first U.S. patent on a robot.

In 1973, Toyota and Nissan started using an updated version of UNIMATE for auto body spot welding. This initiated a major revolution in automobile manufacturing that took place mostly in Japan and the U.S., and that is still ongoing. Unimation followed up in 1978 with the development of the Puma robot (Programmable Universal Machine for Assembly), which was the *de facto* standard for robotic manipulation for the two decades that followed. About 500,000 robots are sold each year, with half of those going to the automotive industry.

In manipulation, the first major effort at creating a hand-eye machine was Heinrich Ernst's MH-1, described in his MIT Ph.D. thesis (Ernst, 1961). The Machine Intelligence project at Edinburgh also demonstrated an impressive early system for vision-based assembly called FREDDY(Michie, 1972).

Research on mobile robotics has been stimulated by several important competitions. AAAI's annual mobile robot competition began in 1992. The first competition winner was CARMEL(Congdon *et al.*, 1992). Progress has been steady and impressive: in recent competitions robots entered the conference complex, found their way to the registration desk, registered for the conference, and even gave a short talk.

The **RoboCup** competition, launched in 1995 by Kitano and colleagues (1997), aims to "develop a team of fully autonomous humanoid robots that can win against the human world champion team in soccer" by 2050. Some competitions use wheeled robots, some humanoid robots, and some software simulations. Stone (2016) describes recent innovations in RoboCup.

The **DARPA Grand Challenge**, organized by DARPA in 2004 and 2005, required autonomous vehicles to travel more than 200 kilometers through the desert in less than ten hours (Buehler *et al.*, 2006). In the original event in 2004, no robot traveled more than eight miles, leading many to believe the prize would never be claimed. In 2005, Stanford's robot Stanley won the competition in just under seven hours (Thrun, 2006). DARPA then organized the **Urban Challenge**, a competition in which robots had to navigate 60 miles in an urban environment with other traffic. Carnegie Mellon University's robot BOSS took first place and claimed the

$2 million prize (Urmson and Whittaker, 2008). Early pioneers in the development of robotic cars included Dickmanns and Zapp (1987) and Pomerleau (1993).

The field of robotic mapping has evolved from two distinct origins. The first thread began with work by Smith and Cheeseman (1986), who applied Kalman filters to the simultaneous localization and mapping (SLAM) problem. This algorithm was first implemented by Moutarlier and Chatila (1989) and later extended by Leonard and Durrant-Whyte (1992); see Dissanayake *et al.* (2001) for an overview of early Kalman filter variations. The second thread began with the development of the **occupancy grid** representation for probabilistic mapping, which specifies the probability that each (x, y) location is occupied by an obstacle (Moravec and Elfes, 1985).

Kuipers and Levitt (1988) were among the first to propose topological rather than metric mapping, motivated by models of human spatial cognition. A seminal paper by Lu and Milios (1997) recognized the sparseness of the simultaneous localization and mapping problem, which gave rise to the development of nonlinear optimization techniques by Konolige (2004) and Montemerlo and Thrun (2004), as well as hierarchical methods by Bosse *et al.* (2004). Shatkay and Kaelbling (1997) and Thrun *et al.* (1998) introduced the EM algorithm into the field of robotic mapping for data association. An overview of probabilistic mapping methods can be found in (Thrun *et al.*, 2005).

Early mobile robot localization techniques are surveyed by Borenstein *et al.* (1996). Although Kalman filtering was well known as a localization method in control theory for decades, the general probabilistic formulation of the localization problem did not appear in the AI literature until much later, through the work of Tom Dean and colleagues (Dean *et al.*, 1990) and of Simmons and Koenig (1995). The latter work introduced the term **Markov localization**. The first real-world application of this technique was by Burgard *et al.* (1999), through a series of robots that were deployed in museums. Monte Carlo localization based on particle filters was developed by Fox *et al.* (1999) and is now widely used. The **Rao-Blackwellized particle filter** combines particle filtering for robot localization with exact filtering for map building (Murphy and Russell, 2001; Montemerlo *et al.*, 2002).

A great deal of early work on **motion planning** focused on geometric algorithms for deterministic and fully observable motion planning problems. The PSPACE-hardness of robot motion planning was shown in a seminal paper by Reif (1979). The configuration space representation is due to Lozano-Perez (1983). A series of papers by Schwartz and Sharir on what they called **piano movers** problems (Schwartz *et al.*, 1987) was highly influential.

Recursive cell decomposition for configuration space planning was originated in the work of Brooks and Lozano-Perez (1985) and improved significantly by Zhu and Latombe (1991). The earliest skeletonization algorithms were based on Voronoi diagrams (Rowat, 1979) and **visibility graphs** (Wesley and Lozano-Perez, 1979). Guibas *et al.* (1992) developed efficient techniques for calculating Voronoi diagrams incrementally, and Choset (1996) generalized Voronoi diagrams to broader motion planning problems.

John Canny (1988) established the first singly exponential algorithm for motion planning. The seminal text by Latombe (1991) covers a variety of approaches to motion planning, as do the texts by Choset *et al.* (2005) and LaValle (2006). Kavraki *et al.* (1996) developed the theory of probabilistic roadmaps. Kuffner and LaValle (2000) developed rapidly exploring random trees (RRTs).

Involving optimization in geometric motion planning began with elastic bands (Quinlan and Khatib, 1993), which refine paths when the configuration-space obstacles change. Ratliff *et al.* (2009) formulated the idea as the solution to an optimal control problem, allowing the initial trajectory to start in collision, and deforming it by mapping workspace obstacle gradients via the Jacobian into the configuration space. Schulman *et al.* (2013) proposed a practical second-order alternative.

The control of robots as dynamical systems—whether for manipulation or navigation— has generated a vast literature. While this chapter explained the basics of **trajectory tracking control** and optimal control, it left out entire subfields, including adaptive control, robust control, and Lyapunov analysis. Rather than assuming everything about the system is known a priori, adaptive control aims to adapt the dynamics parameters and/or the control law online. Robust control, on the other hand, aims to design controllers that perform well in spite of uncertainty and external disturbances.

Lyapunov analysis was originally developed in the 1890s for the stability analysis of general nonlinear systems, but it was not until the early 1930s that control theorists realized its true potential. With the development of optimization methods, Lyapunov analysis was extended to control barrier functions, which lend themselves nicely to modern optimization tools. These methods are widely used in modern robotics for real-time controller design and safety analysis.

Crucial works in robotic control include a trilogy on impedance control by Hogan (1985) and a general study of robot dynamics by Featherstone (1987). Dean and Wellman (1991) were among the first to try to tie together control theory and AI planning systems. Three classic textbooks on the mathematics of robot manipulation are due to Paul (1981), Craig (1989), and Yoshikawa (1990). Control for manipulation is covered by Murray (2017).

The area of **grasping** is also important in robotics—the problem of determining a stable grasp is quite difficult (Mason and Salisbury, 1985). Competent grasping requires touch sensing, or **haptic feedback**, to determine contact forces and detect slip (Fearing and Hollerbach, 1985). Understanding how to grasp the the wide variety of objects in the world is a daunting task. (Bousmalis *et al.*, 2017) describe a system that combines real-world experimentation with simulations guided by sim-to-real transfer to produce robust grasping.

Potential-field control, which attempts to solve the motion planning and control problems simultaneously, was developed for robotics by Khatib (1986). In mobile robotics, this idea was viewed as a practical solution to the collision avoidance problem, and was later extended into an algorithm called **vector field histograms** by Borenstein (1991).

ILQR is currently widely used at the intersection of motion planning and control and is due to Li and Todorov (2004). It is a variant of the much older differential dynamic programming technique (Jacobson and Mayne, 1970).

Fine-motion planning with limited sensing was investigated by Lozano-Perez *et al.* (1984) and Canny and Reif (1987). Landmark-based navigation (Lazanas and Latombe, 1992) uses many of the same ideas in the mobile robot arena. Navigation functions, the robotics version of a control policy for deterministic MDPs, were introduced by Koditschek (1987). Key work applying POMDP methods (Section 16.4) to motion planning under uncertainty in robotics is due to Pineau *et al.* (2003) and Roy *et al.* (2005).

Reinforcement learning in robotics took off with the seminal work by Bagnell and Schneider (2001) and Ng *et al.* (2003), who developed the paradigm in the context of autonomous helicopter control. Kober *et al.* (2013) offers an overview of how reinforcement learning changes when applied to the robotics problem. Many of the techniques implemented on physical systems build approximate dynamics models, dating back to locally weighted linear models due to Atkeson *et al.* (1997). But policy gradients played their role as well, enabling (simplified) humanoid robots to walk (Tedrake *et al.*, 2004), or a robot arm to hit a baseball (Peters and Schaal, 2008).

Levine *et al.* (2016) demonstrated the first **deep reinforcement learning** application on a real robot. At the same time, model-free RL in simulation was being extended to continuous domains (Schulman *et al.*, 2015a; Heess *et al.*, 2016; Lillicrap *et al.*, 2015). Other work scaled up physical data collection massively to showcase the learning of grasps and dynamics models (Pinto and Gupta, 2016; Agrawal *et al.*, 2017; Levine *et al.*, 2018). Transfer from simulation to reality or **sim-to-real** (Sadeghi and Levine, 2016; Andrychowicz *et al.*, 2018a), **metalearning** (Finn *et al.*, 2017), and sample-efficient model-free reinforcement learning (Andrychowicz *et al.*, 2018b) are active areas of research.

Early methods for predicting **human actions** made use of filtering approaches (Madhavan and Schlenoff, 2003), but seminal work by Ziebart *et al.* (2009) proposed prediction by modeling people as approximately rational agents. Sadigh *et al.* (2016) captured how these predictions should actually depend on what the robot decides to do, building toward a game-theoretic setting. For collaborative settings, Sisbot *et al.* (2007) pioneered the idea of accounting for what people want in the robot's cost function. Nikolaidis and Shah (2013) decomposed collaboration into learning how the human will act, but also learning how the human wants the robot to act, both achievable from demonstrations. For learning from demonstration see Argall *et al.* (2009). Akgun *et al.* (2012) and Sefidgar *et al.* (2017) studied teaching by end users rather than by experts.

Tellex *et al.* (2011) showed how robots can infer what people want from natural language instructions. Finally, not only do robots need to infer what people want and plan on doing, but people too need to make the same inferences about robots. Dragan *et al.* (2013) incorporated a model of the human's inferences into robot motion planning.

The field of **human–robot interaction** is much broader than what we covered in this chapter, which focused primarily on the planning and learning aspects. Thomaz *et al.* (2016) provides a survey of interaction more broadly from a computational perspective. Ross *et al.* 2011) describe the DAGGER system.

The topic of software architectures for robots engenders much religious debate. The good old-fashioned AI candidate—the three-layer architecture—dates back to the design of Shakey and is reviewed by Gat (1998). The subsumption architecture is due to Brooks (1986), although similar ideas were developed independently by Braitenberg, whose book, *Vehicles* (1984), describes a series of simple robots based on the behavioral approach.

The success of Brooks's six-legged walking robot was followed by many other projects. Connell, in his Ph.D. thesis (1989), developed an entirely reactive mobile robot that was capable of retrieving objects. Extensions of the paradigm to multirobot systems can be found in work by Parker (1996) and Mataric (1997). GRL (Horswill, 2000) and COLBERT (Konolige, 1997) abstract the ideas of concurrent behavior-based robotics into general robot control languages. Arkin (1998) surveys some of the most popular approaches in this field.

Two early textbooks, by Dudek and Jenkin (2000) and by Murphy (2000), cover robotics generally. More recent overviews are due to Bekey (2008) and Lynch and Park (2017). An excellent book on robot manipulation addresses advanced topics such as compliant motion (Mason, 2001). Robot motion planning is covered in Choset *et al.* (2005) and LaValle (2006). Thrun *et al.* (2005) introduces probabilistic robotics. The *Handbook of Robotics* (Siciliano and Khatib, 2016) is a massive, comprehensive overview of all of robotics.

The premiere conference for robotics is Robotics: Science and Systems Conference, followed by the IEEE International Conference on Robotics and Automation. Human-Robot Interaction is the premiere venue for interaction. Leading robotics journals include *IEEE Robotics and Automation*, the *International Journal of Robotics Research*, and *Robotics and Autonomous Systems.*

[1] Roboticists like to minimize a cost function J, whereas in other parts of AI we try to maximize a utility function U or a reward R.

[2] We omit the details of f^{-1} here, but they involve mass, inertia, gravity, and Coriolis and centrifugal forces.

CHAPTER 27

COMPUTER VISION

In which we connect the computer to the raw, unwashed world through the eyes of a camera.

Most animals have eyes, often at significant cost: eyes take up a lot of space; use energy; and are quite fragile. This cost is justified by the immense value that eyes provide. An agent that can see can predict the future—it can tell what it might bump into; it can tell whether to attack or to flee or to court; it can guess whether the ground ahead is swampy or firm; and it can tell how far away the fruit is. In this chapter, we describe how to recover information from the flood of data that comes from eyes or cameras.

27.1 Introduction

Vision is a perceptual channel that accepts a **stimulus** and reports some representation of the world. Most agents that use vision use **passive sensing**—they do not need to send out light to see. In contrast, **active sensing** involves sending out a signal such as radar or ultrasound, and sensing a reflection. Examples of agents that use active sensing include bats (ultrasound), dolphins (sound), abyssal fishes (light), and some robots (light, sound, radar). To understand a perceptual channel, one must study both the physical and statistical phenomena that occur in sensing and what the perceptual process should produce. We concentrate on vision in this chapter, but robots in the real world use a variety of sensors to perceive sound, touch, distance, temperature, global position, and acceleration.

A **feature** is a number obtained by applying simple computations to an image. Very useful information can be obtained directly from features. The wumpus agent had five sensors, each of which extracted a single bit of information. These bits, which are features, could be interpreted directly by the program. As another example, many flying animals compute a simple feature that gives a good estimate of time to contact with a nearby object; this feature can be passed directly to muscles that control steering or wings, allowing very fast changes of direction. This **feature extraction** approach emphasizes simple, direct computations applied to sensor responses.

The **model-based** approach to vision uses two kinds of models. An **object model** could be the kind of precise geometric model produced by computer aided design systems. It could also be a vague statement about general properties of objects, for example, the claim that all faces viewed in low resolution look approximately the same. A **rendering model** describes the physical, geometric, and statistical processes that produce the stimulus from the world. While rendering models are now sophisticated and exact, the stimulus is usually ambiguous. A white object under low light may look like a black object under intense light. A small, nearby object may look the same as a large, distant object. Without additional evidence, we cannot tell if what we see is a toy Godzilla tearing up a toy building, or a real monster destroying a real building.

There are two main ways to manage these ambiguities. First, some interpretations are more likely than others. For example, we can be confident that the picture doesn't show a real Godzilla destroying a real building, because there are no real Godzillas. Second, some ambiguities are insignificant. For example, distant scenery may be trees or may be a flat painted surface. For most applications, the difference is unimportant, because the objects are far away and so we will not bump into them or interact with them soon.

The two core problems of computer vision are **reconstruction**, where an agent builds a model of the world from an image or a set of images, and **recognition**, where an agent draws distinctions among the objects it encounters based on visual and other information. Both problems should be interpreted very broadly. Building a geometric model from images is obviously reconstruction (and solutions are very valuable), but sometimes we need to build a map of the different textures on a surface, and this is reconstruction, too. Attaching names to objects that appear in an image is clearly recognition. Sometimes we need to answer questions like: Is it asleep? Does it eat meat? Which end has teeth? Answering these questions is recognition, too.

The last thirty years of research have produced powerful tools and methods for addressing these core problems. Understanding these methods requires an understanding of the processes by which images are formed.

27.2 Image Formation

Imaging distorts the appearance of objects. A picture taken looking down a long straight set of railway tracks will suggest that the rails converge and meet. If you hold your hand in front of your eye, you can block out the moon, even though the moon is larger than your hand (this works with the sun too, but you could damage your eyes checking it). If you hold a book flat in front of your face and tilt it backward and forward, it will seem to shrink and grow in *the image*. This effect is known as **foreshortening** (Figure 27.1). Models of these effects are essential for building competent object recognition systems and also yield powerful cues for reconstructing geometry.

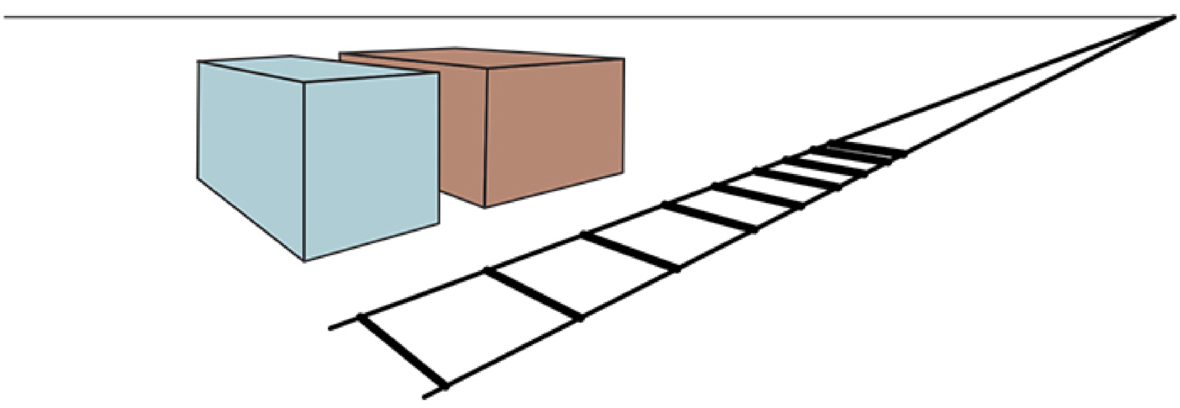


Figure 27.1 Geometry in the scene appears distorted in images. Parallel lines appear to meet, like the railway tracks in a desolate town. Buildings that have right angles in the real world scene have distorted angles in the image.

27.2.1 Images without lenses: The pinhole camera

Image sensors gather light scattered from objects in a **scene** and create a two-dimensional (2D) **image**. In the eye, these sensors consist of two types of cell: There are about 100 million rods, which are sensitive to light at a wide range of wavelengths, and 5 million cones. Cones, which are essential for color vision, are of three main types, each of which is sensitive to a different set of wavelengths. In cameras, the image is formed on an image plane. In film cameras the image plane is coated with silver halides. In digital cameras, the image plane is subdivided into a grid of a few million **pixels**.

We refer to the whole image plane as a **sensor**, but each pixel is an individual tiny sensor—usually a charge-coupled device (CCD) or complementary metal-oxide semiconductor (CMOS). Each photon arriving at the sensor produces an electrical effect, whose strength depends on the wavelength of the photon. The output of the sensor is the sum of all these effects in some time window, meaning that image sensors report a weighted average of the intensity of light arriving at the sensor. The average is over wavelength, direction from which photons can arrive, time, and the area of the sensor.

To see a focused image, we must ensure that all the photons arriving at a sensor come from approximately the same spot on the object in the world. The simplest way to form a focused image is to view stationary objects with a **pinhole camera**, which consists of a pinhole opening, O, at the front of a box, and an image plane at the back of the box (Figure 27.2). The opening is called the **aperture**. If the pinhole is small enough, each tiny sensor in the image plane will see only photons that come from approximately the same spot on the object, and so the image is focused. We can form focused images of moving objects with a pinhole camera, too, as long as the object moves only a short distance in the sensors' time window. Otherwise, the image of the moving object is defocused, an effect known as **motion blur**. One way to manipulate the time window is to open and close the pinhole.

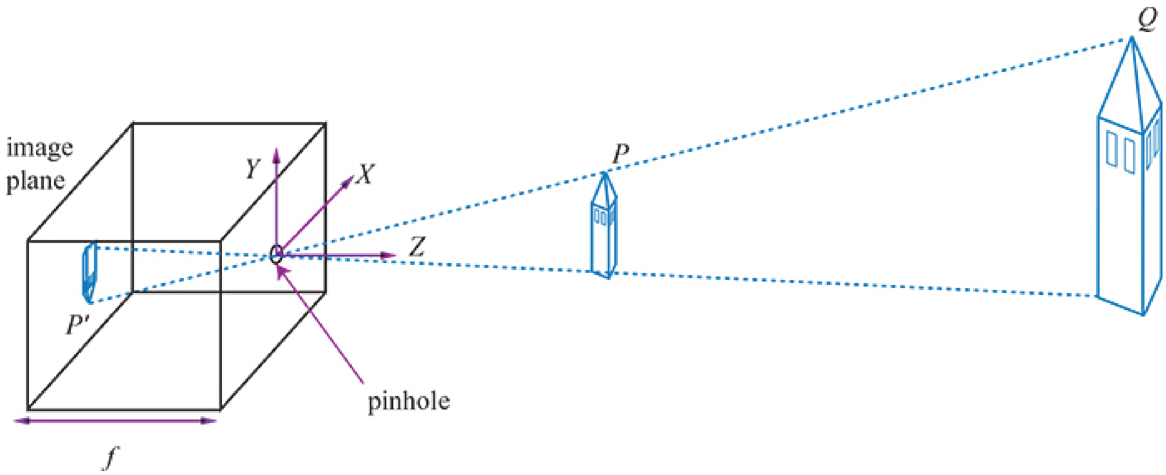


Figure 27.2 Each light sensitive element at the back of a pinhole camera receives light that passes through the pinhole from a small range of directions. If the pinhole is small enough, the result is a focused image behind the pinhole. The process of projection means that large, distant objects look the same as smaller, nearby objects—the point P' in the image plane could have come from a nearby toy tower at point P or from a distant real tower at point Q.

Pinhole cameras make it easy to understand the geometric model of camera behavior (which is more complicated—but similar—with most other imaging devices). We will use a three-dimensional (3D) coordinate system with the origin at O, and will consider a point P in the scene, with coordinates (X, Y, Z). P gets projected to the point P' in the image plane with coordinates (x, y, z). If f is the **focal length**—the distance from the pinhole to the image plane—then by similar triangles, we can derive the following equations:

$$\frac{-x}{j} = \frac{X}{Z}, \frac{-y}{f} = \frac{Y}{Z} \Rightarrow x = \frac{-fX}{Z}, y = \frac{-fY}{Z}.$$

These equations define an image formation process known as **perspective projection**. Note that the Z in the denominator means that the farther away an object is, the smaller its image will be. Also, note that the minus signs mean that the image is *inverted*, both left–right and up–down, compared with the scene.

Perspective imaging has a number of geometric effects. Distant objects look small. Parallel lines converge to a point on the horizon. (Think of railway tracks, Figure 27.1.) A line in the scene in the direction (U, V, W) and passing through the point (X_0, Y_0, Z_0) can be described as the set of points ($X_0 + \lambda U$, $Y_0 + \lambda V$, $Z_0 + \lambda W$), with λ varying between $-\infty$ and $+\infty$. Different choices of (X_0, Y_0, Z_0) yield different lines parallel to one another. The projection of a point P_λ from this line onto the image plane is given by

$$P\lambda = (f\frac{X_0+\lambda U}{Z_0+\lambda W}, f\frac{Y_0+\lambda V}{Z_0+\lambda W}).$$

As $\lambda \to \infty$ or $\lambda \to -\infty$, this becomes P_∞, = (fU/W, fV/W) if $W \neq 0$. This means that two parallel lines leaving different points in space will converge in the image—for large λ, the image points are nearly the same, whatever the value of (X_0, Y_0, Z_0) (again, think railway tracks, Figure 27.1). We call P_∞, the **vanishing point** associated with the family of straight lines with direction (U, V, W). Lines with the same direction share the same vanishing point.

27.2.2 Lens systems

Pinhole cameras can focus light well, but because the pinhole is small, only a little light will get in, and the image will be dark. Over a short period of time, only a few photons will hit each point on the sensor, so the signal at each point will be dominated by random fluctuations; we say that a dark film image is grainy and a dark digital image is noisy; either way, the image is of low quality.

Enlarging the hole (the aperture) will make the image brighter by collecting more light from a wider range of directions. However, with a larger aperture the light that hits a particular point in the image plane will have come from multiple points in the real world scene, so the image will be defocused. We need some way to refocus the image.

Vertebrate eyes and modern cameras use a **lens** system—a single piece of transparent tissue in the eye and a system of multiple glass lens elements in a camera. In Figure 27.3 we see that light from the tip of the candle spreads out in all directions. A camera (or an eye) with a lens captures all the light that hits anywhere on the lens—a much larger area than a pinhole—and focuses all that light to a single point on the image plane. Light from other parts of the candle would similarly be gathered and focused to other points on the image plane. The result is a brighter, less noisy, focused image.

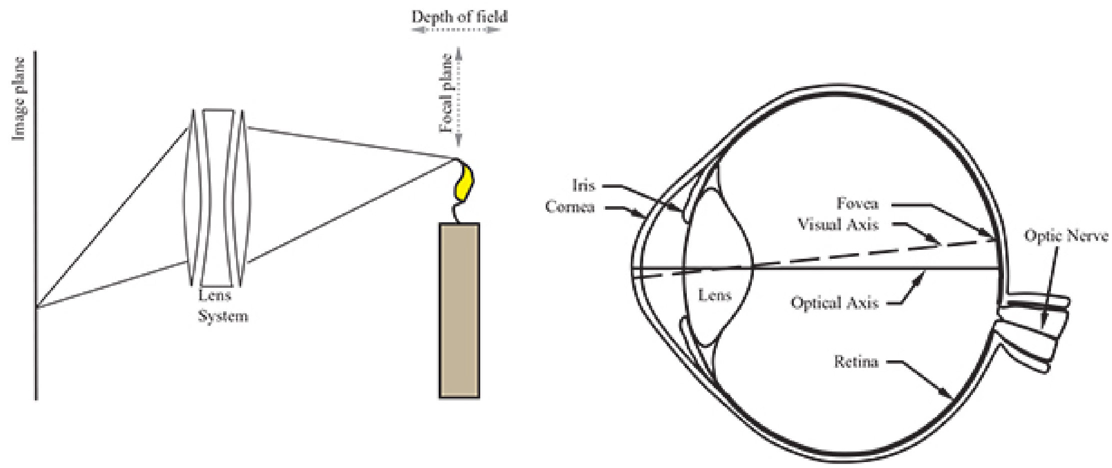


Figure 27.3 Lenses collect the light leaving a point in the scene (here, the tip of the candle flame) in a range of directions, and steer all the light to arrive at a single point on the image plane. Points in the scene near the focal plane—within the depth of field—will be focused properly. In cameras, elements of the lens system move to change the focal plane, whereas in the eye, the shape of the lens is changed by specialized muscles.

Lens systems do not focus all the light from everywhere in the real world; the lens design restricts them to focusing light only from points that lie within a range of Z depths from the lens. The center of this range—where focus is sharpest—is called the **focal plane**, and the range of depths for which focus remains sharp enough is called the **depth of field**. The larger the lens aperture (opening), the smaller the depth of field.

What if you want to focus on something at a different distance? To move the focal plane, the lens elements in a camera can move back and forth, and the lens in the eye can change shape—but with age the eye lens tends to harden, making it less able to adjust focal distances, and requiring many humans to augment their vision with external lens—eyeglasses.

27.2.3 Scaled orthographic projection

The geometric effects of perspective imaging aren't always pronounced. For example, windows on a building across the street look much smaller than ones right nearby, but two windows that are next to each other will have about the same size even though one is slightly farther away. We have the option to handle the windows with a simplified model called **scaled orthographic projection**, rather than perspective projection. If the depth Z of all points on an object fall within the range $Z_0 \pm \Delta Z$, with $\Delta Z << Z_0$, then the perspective scaling factor f/Z can be approximated by a constant $s = f/Z_0$. The equations for projection from the scene coordinates (X, Y, Z) to the image plane become $x = sX$ and $y = sY$. Foreshortening still occurs in the scaled orthographic projection model, because it is caused by the object tilting away from the view.

27.2.4 Light and shading

The brightness of a pixel in the image is a function of the brightness of the surface patch in the scene that projects to the pixel. For modern cameras, this function is linear for middling intensities of light, but has pronounced nonlinearities for darker and brighter illumination. We will use a linear model. Image brightness is a strong, if ambiguous, cue to both the shape and the identity of objects. The ambiguity occurs because there are three factors that contribute to the amount of light that comes from a point on an object to the image: the overall intensity of **ambient light**); whether the point is facing the light or is in shadow); and the amount of light **reflected** from the point.

People are surprisingly good at disambiguating brightness—they usually can tell the difference between a black object in bright light and a white object in shadow, even if both have the same overall brightness. However, people sometimes get shading and markings mixed up—-a streak of dark makeup under a cheekbone will often look like a shading effect, making the face look thinner.

Most surfaces reflect light by a process of **diffuse reflection**. Diffuse reflection scatters light evenly across the directions leaving a surface, so the brightness of a diffuse surface doesn't depend on the viewing direction. Most cloth has this property, as do most paints, rough wooden surfaces, most vegetation, and rough stone or concrete.

Specular reflection causes incoming light to leave a surface in a lobe of directions that is determined by the direction the light arrived from. A mirror is one example. What you see depends on the direction in which you look at the mirror. In this case, the lobe of directions is very narrow, which is why you can resolve different objects in a mirror.

For many surfaces, the lobe is broader. These surfaces display small bright patches, usually called **specularities**. As the surface or the light moves, the specularities move, too. Away from these patches, the surface behaves as if it is diffuse. Specularities are often seen on metal surfaces, painted surfaces, plastic surfaces, and wet surfaces. These are easy to identify, because they are small and bright (Figure 27.4). For almost all purposes, it is enough to model all surfaces as being diffuse with specularities.

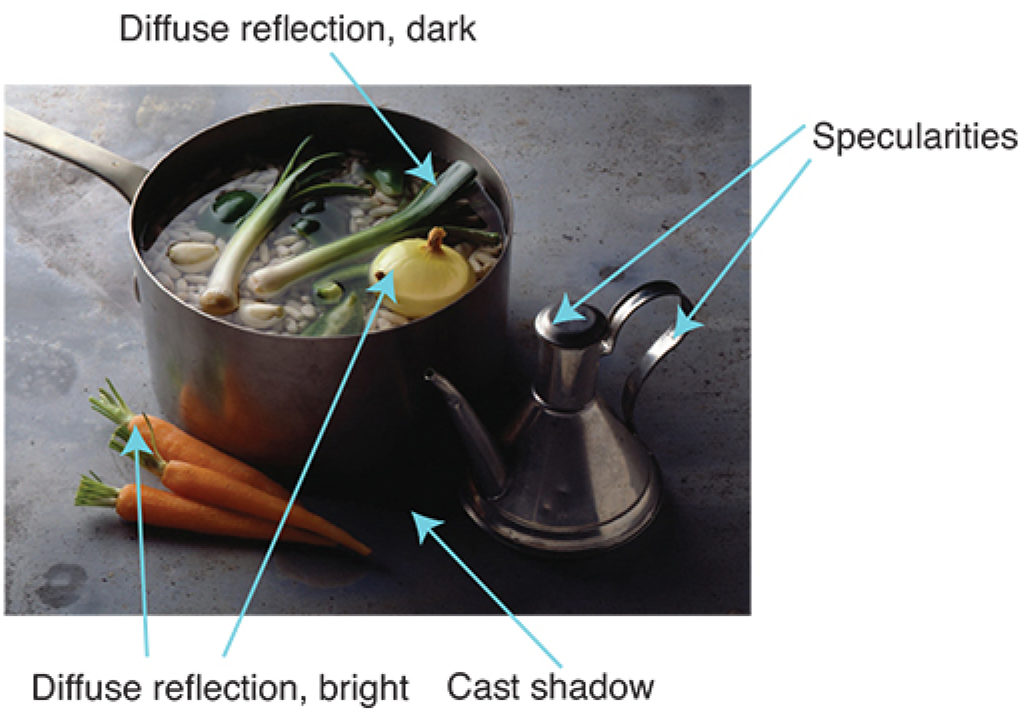


Figure 27.4 This photograph illustrates a variety of illumination effects. There are specularities on the stainless steel cruet. The onions and carrots are bright diffuse surfaces because they face the light direction. The shadows appear at surface points that cannot see the light source at all. Inside the pot are some dark diffuse surfaces where the light strikes at a tangential angle. (There are also some shadows inside the pot.) Photo by Ryman Cabannes/Image Professionals GmbH/Alamy Stock Photo.

The main source of illumination outside is the sun, whose rays all travel parallel to one another in a known direction because it is so far away. We model this behavior with a **distant point light source**. This is the most important model of lighting, and is quite effective for indoor scenes as well as outdoor scenes. The amount of light collected by a surface patch in this model depends on the angle θ between the illumination direction and the normal (perpendicular) to the surfaces (Figure 27.5).

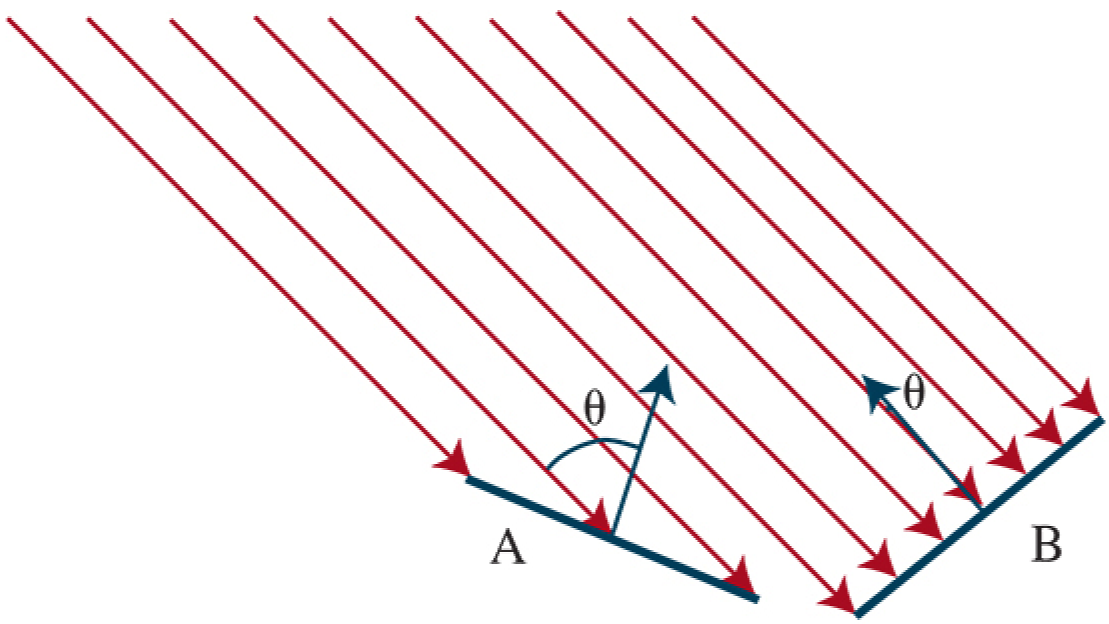


Figure 27.5 Two surface patches are illuminated by a distant point source, whose rays are shown as light arrows. Patch A is tilted away from the source (θ is close to 90°) and collects less energy, because it cuts fewer light rays per unit surface area. Patch B, facing the source (θ is close to 0°), collects more energy.

A diffuse surface patch illuminated by this model will reflect some fraction of the light it collects, given by the **diffuse albedo**. For practical surfaces, this lies in the range 0.05-0.95. **Lambert's cosine law** states the brightness of a diffuse patch is given by

$$I = \rho I_0 \cos\theta,$$

where I_0 is the intensity of the light source, θ is the angle between the light source direction and the surface normal, and ρ is the diffuse albedo. This law predicts that bright image pixels come from surface patches that face the light directly and dark pixels come from patches that see the light only tangentially, so that the shading on a surface provides some shape information. If the surface cannot see the source, then it is in **shadow**. Shadows are very seldom a uniform black, because the shadowed surface usually receives some light from other sources. Outdoors, the most important source other than the sun is the sky, which is quite bright. Indoors, light reflected from other surfaces illuminates shadowed patches. These **interreflections** can have a significant effect on the brightness of other surfaces, too. These effects are sometimes modeled by adding a constant **ambient illumination** term to the predicted intensity.

27.2.5 Color

Fruit is a bribe that a tree offers to animals to carry its seeds around. Trees that can signal when this bribe is ready have an advantage, as do animals that can read these signals. As a result, most fruits start green, and turn red or yellow when ripe, and most fruit-eating animals can see these color changes. Generally, light arriving at the eye has different amounts of energy at different wavelengths, and is represented by a spectral energy density.

Cameras and the human vision system respond to light at wavelengths ranging from about 380nm (violet) to about 750nm (red). In color imaging systems, there are different types of receptor that respond more or less strongly to different wavelengths. In humans, the sensation of color occurs when the vision system compares the responses of receptors near each other on the retina. Animal color vision systems typically have relatively few types of receptor, and so represent relatively little of the detail in the spectral energy density function (some animals have only one type of receptor; some have as many as six types). Human color vision is produced by three types of receptor. Most color camera systems use only three types of receptor, too, because the images are produced for humans, but some specialized systems can produce very detailed measurements of the spectral energy density.

Because most humans have three types of color-sensitive receptors, the **principle of trichromacy** applies. This idea, first proposed by Thomas Young in 1802, states that a human observer can match the visual appearance of any spectral energy density, however complex, by mixing appropriate amounts of just three **primaries**. Primaries are colored light sources, chosen so that no mixture of any two will match the third. A common choice is to have one red primary, one green, and one blue, abbreviated as **RGB**. Although a given colored object may have many component frequencies of light, we can match the color by mixing just the three primaries, and most people will agree on the proportions of the mixture. That means we can represent color images with just three numbers per pixel—the RGB values.

For most computer vision applications, it is accurate enough to model a surface as having three different (RGB) diffuse albedos and to model light sources as having three (RGB) intensities. We then apply Lambert's cosine law to each to get red, green, and blue pixel values. This model predicts, correctly, that the same surface will produce different colored image patches under different colored lights. In fact, human observers are quite good at ignoring the effects of different colored lights and appear to estimate the color the surface would have under white light, an effect known as **color constancy**.

27.3 Simple Image Features

Light reflects off objects in the scene to form an image consisting of, say, twelve million three-byte pixels. As with all sensors there will be noise in the image, and in any case there is a lot of data to deal with. The way to get started analyzing this data is to produce simplified representations that expose what's important, but reduce detail. Much current practice learns these representations from data. But there are four properties of images and video that are particularly general: edges, texture, optical flow and segmentation into regions.

An edge occurs where there is a big difference in pixel intensity across part of an image. Building representations of edges involves local operations on an image—you need to compare a pixel value to some values nearby—and doesn't require any knowledge about what is in the image. Thus, edge detection can come early in the pipeline of operations and we call it an "early" or "low-level" operation.

The other operations require handling a larger area of the image. For example, a texture description applies to a pool of pixels—to say "stripey," you need to see some stripes. Optical flow represents where pixels move to from one image in a sequence to the next, and this can cover a larger area. Segmentation cuts an image into regions of pixels that naturally belong together, and doing so requires looking at the whole region. Operations like this are sometimes referred to as "mid-level" operations.

27.3.1 Edges

Edges are straight lines or curves in the image plane across which there is a "significant" change in image brightness. The goal of edge detection is to abstract away from the messy, multi-megabyte image and towards a more compact, abstract representation, as in Figure 27.6. Effects in the scene very often result in large changes in image intensity, and so produce edges in the image. Depth discontinuities (labeled 1 in the figure) can cause edges because when you cross the discontinuity, the color typically changes. When the surface normal changes (labeled 2 in the figure), the image intensity often changes. When the surface reflectance changes (labeled 3), the image intensity often changes. Finally, a shadow (labeled 4) is a discontinuity in illumination that causes an edge in the image, even though there is not an edge in the object. Edge detectors can’t disentangle the cause of the discontinuity, which is left to later processing.

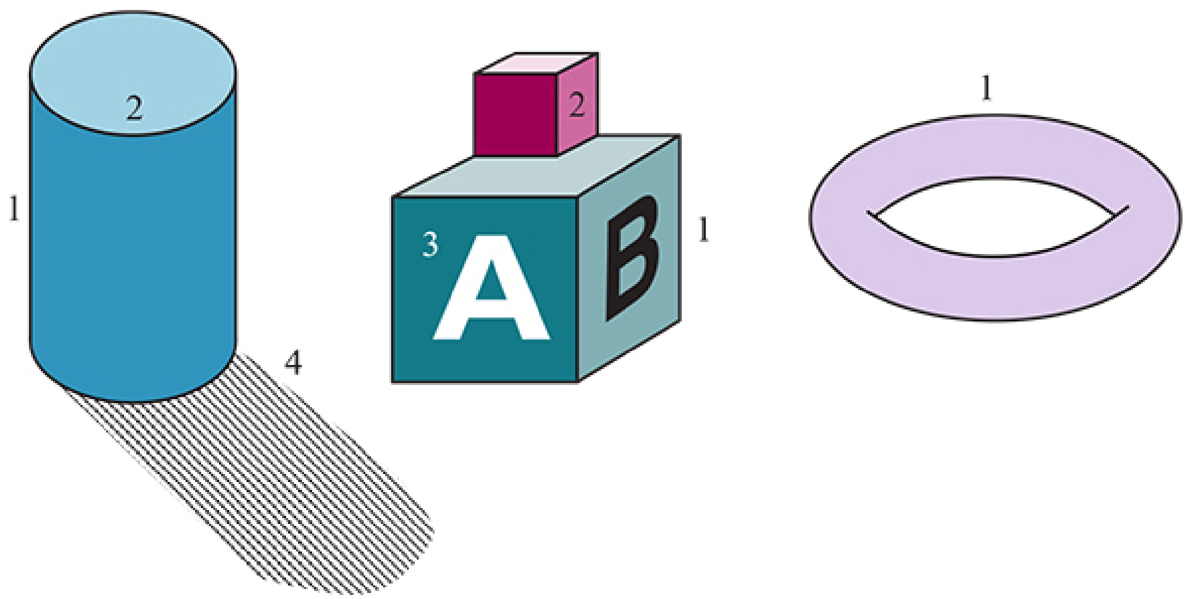


Figure 27.6 Different kinds of edges: (1) depth discontinuities; (2) surface orientation discontinuities; (3) reflectance discontinuities; (4) illumination discontinuities (shadows).

Finding edges requires care. Figure 27.7 (top) shows a one-dimensional crosssection of an image perpendicular to an edge, with an edge at $x = 50$.

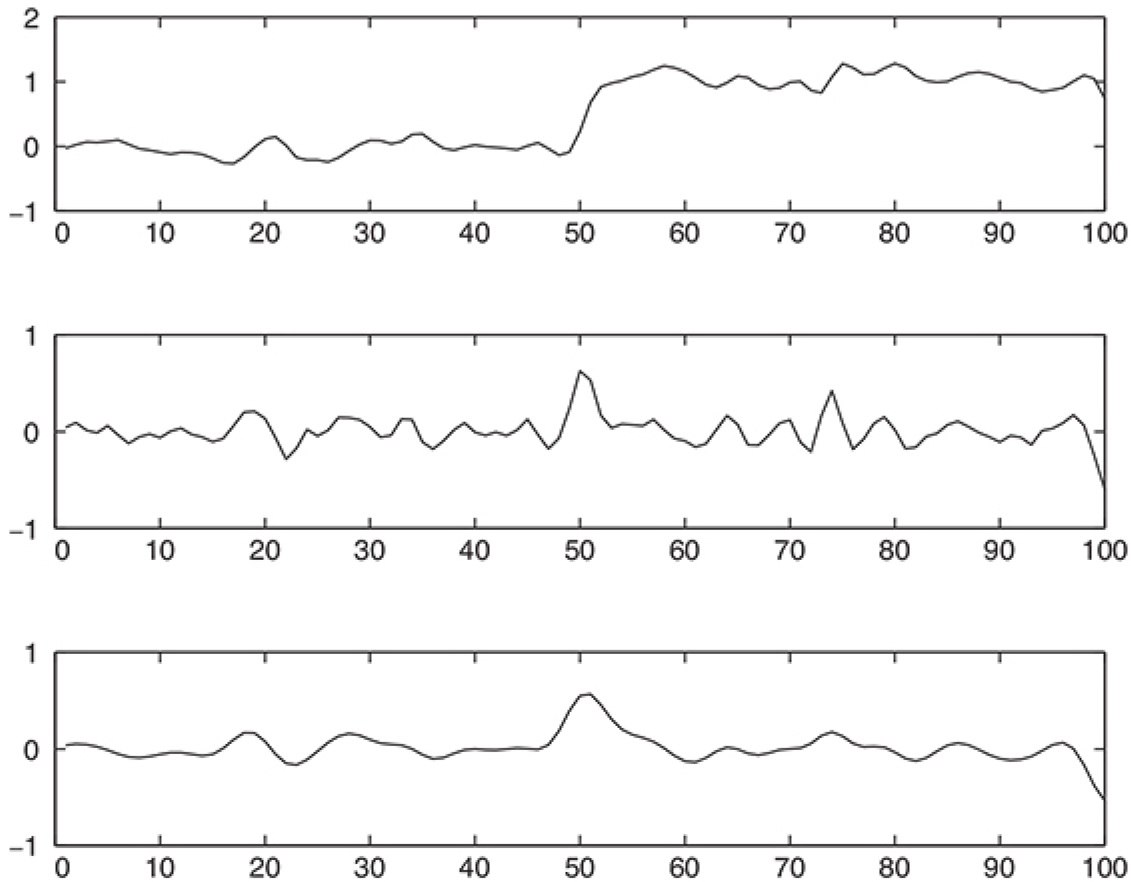


Figure 27.7 Top: Intensity profile $I(x)$ along a one-dimensional section across a step edge. Middle: The derivative of intensity, $I'(x)$. Large values of this function correspond to edges, but the function is noisy. Bottom: The derivative of a smoothed version of the intensity. The noisy candidate edge at $x = 75$ has disappeared.

You might differentiate the image and look for places where the magnitude of the derivative $I'(x)$ is large. This almost works, but in Figure 27.7 (middle), we see that although there is a peak at $x = 50$, there are also subsidiary peaks at other locations (e.g., $x = 75$) that could be mistaken for true edges. These arise because of the presence of “noise” in the image. **Noise** here means changes to the value of a pixel that don’t have to do with an edge. For example, there could be thermal noise in the camera; there could be scratches on the object surface that change the surface normal at the finest scale; there could be minor variations in the surface albedo; and so on. Each of these effects make the gradient look big, but don't mean that an edge is present. If we “smooth” the image first, the spurious peaks are diminished, as we see in Figure 27.7 (bottom).

Smoothing involves using surrounding pixels to suppress noise. We will predict the “true” value of our pixel as a weighted sum of nearby pixels, with more weight for the closest pixels. A natural choice of weights is a **Gaussian filter**. Recall that the zero-mean Gaussian function with standard deviation σ is

$$G_\sigma(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-x^2/2\sigma^2} \quad \text{in one dimension, or}$$

$$G_\sigma(x, y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x^2+y^2)/2\sigma^2} \quad \text{in two dimensions.}$$

Applying a Gaussian filter means replacing the intensity $I(x_0, y_0)$ with the sum, over all (x, y) pixels, of $I(x, y)\, G_\sigma(d)$, where d is the distance from (x_0, y_0) to (x, y). This kind of weighted sum is so common that there is a special name and notation for it. We say that the function h is the **convolution** of two functions f and g (denoted as $h = f * g$) if we have

$$h(x) = \sum_{u=-\infty}^{+\infty} f(u)g(x-u) \qquad \text{in one dimension, or}$$

$$h(x,y) = \sum_{u=-\infty}^{+\infty} \sum_{v=-\infty}^{+\infty} f(u,v)g(x-u, y-v) \qquad \text{in two dimensions.}$$

So the smoothing function is achieved by convolving the image with the Gaussian, $I * G_\sigma$. A σ of 1 pixel is enough to smooth over a small amount of noise, whereas 2 pixels will smooth a larger amount, but at the loss of some detail. Because the Gaussian's influence fades rapidly with distance, in practice we can replace the $\pm\infty$ in the sums with something like $\pm 3\sigma$.

We have a chance to make an optimization here: we can combine the smoothing and the edge finding into a single operation. It is a theorem that for any functions f and g, the derivative of the convolution, $(f * g)'$, is equal to the convolution with the derivative, $f * (g')$. So rather than smoothing the image and then differentiating, we can just convolve the image with the derivative of the Gaussian smoothing function, G'_σ. We then mark as edges those peaks in the response that are above some threshold, chosen to eliminate spurious peaks due to noise.

There is a natural generalization of this algorithm from one-dimensional crosssections to general 2D images. In two dimensions edges may be at any angle θ. Considering the image brightness as a scalar function of the variables x, y, its gradient is a vector

$$\nabla I = \begin{pmatrix} \frac{\partial I}{\partial x} \\ \frac{\partial t}{\partial y} \end{pmatrix}$$

Edges correspond to locations in images where the brightness undergoes a sharp change, and thus the magnitude of the gradient, $\| \nabla I \|$ should be large at an edge point. When the image gets brighter or darker, the gradient vector at each point gets longer or shorter, but the direction of the gradient

$$\frac{\nabla I}{||\nabla I||} = \begin{pmatrix} \cos\,\theta \\ \sin\,\theta \end{pmatrix}$$

does not change. This gives us a $\theta = \theta(x, y)$ at every pixel, which defines the edge **orientation** at that pixel. This feature is often useful, because it does not depend on image intensity.

As you might expect from the discussion on detecting edges in one-dimensional signals, to form the gradient, we don't actually compute ∇I, but rather $\nabla(I * G_\sigma)$, after smoothing the image by convolving it with a Gaussian. A property of convolutions is that this is equivalent to convolving the image with the partial derivatives of the Gaussian. Once we have computed the gradient, we can obtain edges by finding edge points and linking them together. To tell whether a point is an edge point, we must look at other points a small distance forward and back along the direction of the gradient. If the gradient magnitude at one of these points is larger, then we could get a better edge point by shifting the edge curve very slightly. Furthermore, if the gradient magnitude is too small, the point cannot be an edge point. So at an edge point, the gradient magnitude is a local maximum along the direction of the gradient, and the gradient magnitude is above a suitable threshold.

Once we have marked edge pixels by this algorithm, the next stage is to link those pixels that belong to the same edge curves. This can be done by assuming that any two neighboring pixels that are both edge pixels with consistent orientations belong to the same edge curve.

Edge detection isn't perfect. Figure 27.8(a) shows an image of a scene containing a stapler resting on a desk, and Figure 27.8(b) shows the output of an edge detection algorithm on this image. As you can see, the output is not perfect: there are gaps where no edge appears, and there are "noise" edges that do not correspond to anything of significance in the scene. Later stages of processing will have to correct for these errors.

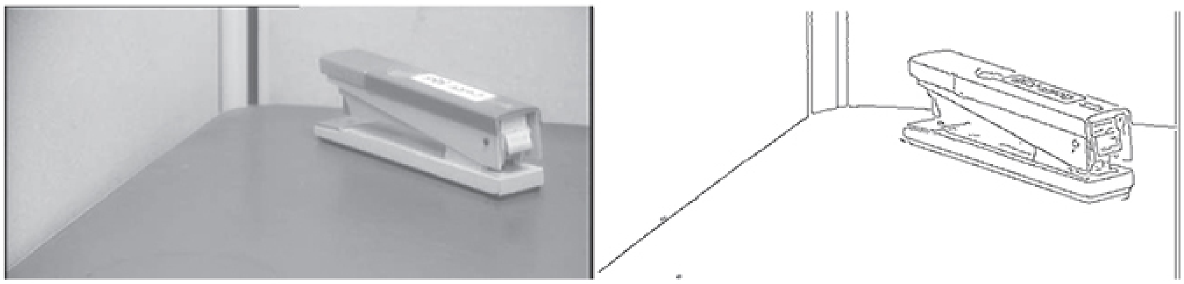


Figure 27.8 (a) Photograph of a stapler. (b) Edges computed from (a).

27.3.2 Texture

In everyday language, the **texture** of surfaces hints at what they feel like when you run a finger over them (the words “texture,” “textile,” and “text” have the same Latin root, a word for weaving). In computational vision, texture refers to a pattern on a surface that can be sensed visually. Usually, these patterns are roughly regular. Examples include the pattern of windows on a building, the stitches on a sweater, the spots on a leopard's skin, blades of grass on a lawn, pebbles on a beach, and a crowd of people in a stadium.

Sometimes the arrangement is quite periodic, as in the stitches on a sweater; in other instances, such as pebbles on a beach, the regularity is only in a statistical sense: the density of pebbles is roughly the same on different parts of the beach. A usual rough model of texture is a repetitive pattern of elements, sometimes called **texels**. This model is quite useful because it is surprisingly hard to make or find real textures that never repeat.

Texture is a property of an image patch, rather than a pixel in isolation. A good description of a patch's texture should summarize what the patch looks like. The description should not change when the lighting changes. This rules out using edge points; if a texture is brightly lit, many locations within the patch will have high contrast and will generate edge points; but if the same texture is viewed under less bright light, many of these edges will not be above the threshold. The description should change in a sensible way when the patch rotates. It is important to preserve the difference between vertical stripes and horizontal stripes *but* not if the vertical stripes are rotated to the horizontal.

Texture representations with these properties have been shown to be useful for two key tasks. The first is identifying objects—a zebra and horse have similar shape, but different textures. The second is matching patches in one image to patches in another image, a key step in recovering 3D information from multiple images (Section 27.6.1).

Here is a basic construction for a texture representation. Given an image patch, compute the gradient orientation at each pixel in the patch, and then characterize the patch by a histogram of orientations. Gradient orientations are largely invariant to changes in illumination (the gradient will get longer, but it will not change direction). The histogram of orientations seems to capture important aspects of the texture. For example, vertical stripes will have two peaks in the histogram (one for the left side of each stripe and one for the right); leopard spots will have more uniformly distributed orientations.

But we do not know how big a patch to describe. There are two strategies. In specialized applications, image information reveals how big the patch should be (for example, one might grow a patch full of stripes until it covers the zebra). An alternative is to describe a patch centered at each pixel for a range of scales. This range usually runs from a few pixels to the extent of the image. Now divide the patch into bins, and in each bin construct an orientation histogram, then summarize the pattern of histograms across bins. It is no longer usual to construct these descriptions by hand. Instead, convolutional neural networks are used to produce texture representations. But the representations constructed by the networks seem to mirror this construction very roughly.

27.3.3 Optical flow

Next, let us consider what happens when we have a video sequence, instead of just a single static image. Whenever there is relative movement between the camera and one or more objects in the scene, the resulting apparent motion in the image is called **optical flow**. This describes the direction and speed of motion of features *in the image* as a result of relative motion between the viewer and the scene. For example, distant objects viewed from a moving car have much slower apparent motion than nearby objects, so the rate of apparent motion can tell us something about distance.

In Figure 27.9 we show two frames from a video of a tennis player. On the right we display the optical flow vectors computed from these images. The optical flow encodes useful information about scene structure—the tennis player is moving and the background (largely) isn't. Furthermore, the flow vectors reveal something about what the player is doing—one arm and one leg are moving fast, and the other body parts aren't.

Figure 27.9 Two frames of a video sequence and the optical flow field corresponding to the displacement from one frame to the other. Note how the movement of the tennis racket and the front leg is captured by the directions of the arrows. (Images courtesy of Thomas Brox.)

The optical flow vector field can be represented by its components v_x (x, y) in the x direction and v_y (x, y) in the y direction. To measure optical flow, we need to find corresponding points between one time frame and the next. A very simple-minded technique is based on the fact that image patches around corresponding points have similar intensity patterns. Consider a block of pixels centered at pixel p, (x_0, y_0), at time t. This block of pixels is to be compared with pixel blocks centered at various candidate pixels q_i at $(x_0 + D_x, y_0 + D_y)$ at time $t + D_t$. One possible measure of similarity is the **sum of squared differences (SSD)**:

$$\mathrm{SSD}(D_x, D_y) = \sum_{(x,y)} (I(x, y, t) - I(x + D_x, y + D_y, t + D_t))^2.$$

Here, (x, y) ranges over pixels in the block centered at (x_0, y_0). We find the (D_x, D_y) that minimizes the SSD. The optical flow at (x_0, y_0) is then (v_x, v_y)

$= (D_x / D_t, D_y / D_t)$. Note that for this to work, there should be some texture in the scene, resulting in windows containing a significant variation in brightness among the pixels. If one is looking at a uniform white wall, then the SSD is going to be nearly the same for the different candidate matches q, and the algorithm is reduced to making a blind guess. The best-performing algorithms for measuring optical flow rely on a variety of additional constraints to deal with situations in which the scene is only partially textured.

27.3.4 Segmentation of natural images

Segmentation is the process of breaking an image into groups of similar pixels. The basic idea is that each image pixel can be associated with certain visual properties, such as brightness, color, and texture. Within an object, or a single part of an object, these attributes vary relatively little, whereas across an inter-object boundary there is typically a large change in one or more of these attributes. We need to find a partition of the image into sets of pixels such that these constraints are satisfied as well as possible. Notice that it isn't enough just to find edges, because many edges are not object boundaries. So, for example, a tiger in grass may generate an edge on each side of each stripe and each blade of grass. In all the confusing edge data, we may miss the tiger for the stripes.

There are two ways of studying the problem, one focusing on detecting the boundaries of these groups, and the other on detecting the groups themselves, called **regions**. We illustrate this in Figure 27.10, showing boundary detection in (b) and region extraction in (c) and (d).

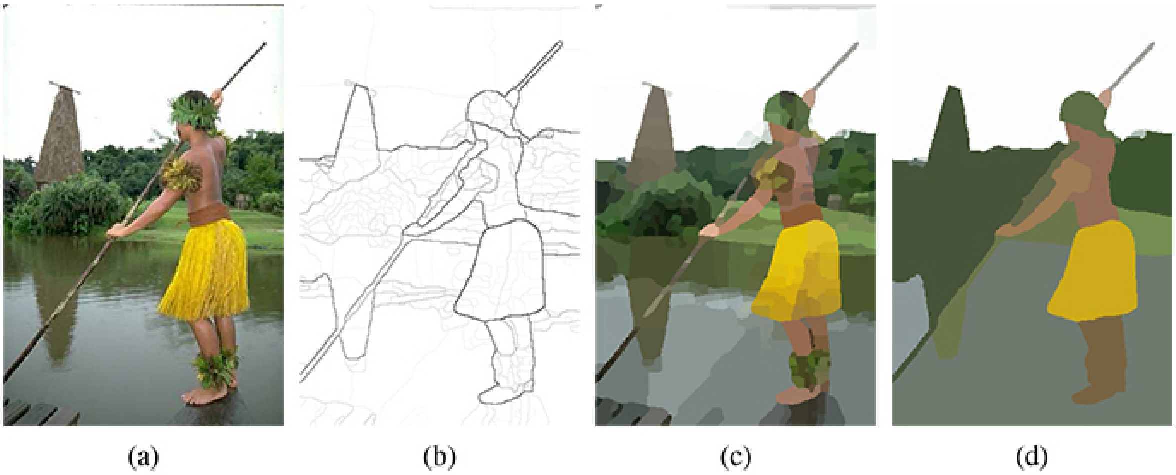


Figure 27.10 (a) Original image. (b) Boundary contours, where the higher the P_b value, the darker the contour. (c) Segmentation into regions, corresponding to a fine partition of the image. Regions are rendered in their mean colors. (d) Segmentation into regions, corresponding to a coarser partition of the image, resulting in fewer regions. (Images courtesy of Pablo Arbelaez, Michael Maire, Charless Fowlkes and Jitendra Malik.)

One way to formalize the problem of detecting boundary curves is as a classification problem, amenable to the techniques of machine learning. A boundary curve at pixel location (x, y) will have an orientation θ. An image neighborhood centered at (x, y) looks roughly like a disk, cut into two halves by a diameter oriented at θ. We can compute the probability $P_b(x, y, \theta)$ that there is a boundary curve at that pixel along that orientation by comparing features in the two halves. The natural way to predict this probability is to train a machine learning classifier using a data set of natural images in which humans have marked the ground truth boundaries

—the goal of the classifier is to mark exactly those boundaries marked by humans and no others.

Boundaries detected by this technique are better than those found using the simple edge detection technique described previously. But there are still two limitations: (1) the boundary pixels formed by thresholding $P_b(x, y, \theta)$ are not guaranteed to form closed curves, so this approach doesn't deliver regions, and (2) the decision making exploits only local context, and does not use global consistency constraints.

The alternative approach is based on trying to “cluster” the pixels into regions based on their brightness, color and texture properties. There are a number of different ways in which this intuition can be formalized mathematically. For instance, Shi and Malik (2000) set this up as a graph partitioning problem. The nodes of the graph correspond to pixels, and edges to connections between pixels. The weight W_{ij} on the edge connecting a pair of pixels i and j is based on how similar the two pixels are in brightness, color, texture, etc. They then find partitions that minimize a *normalized cut* criterion. Roughly speaking, the criterion for partitioning the graph is to minimize the sum of weights of connections across the groups and maximize the sum of weights of connections within the groups.

It turns out that the approaches based on finding boundaries and on finding regions can be coupled, but we will not explore these possibilities here. Segmentation based purely on low-level, local attributes such as brightness and color can not be expected to deliver the final correct boundaries of all the objects in the scene. To reliably find boundaries associated with objects, it is also necessary to incorporate high-level knowledge of the kinds of objects one may expect to encounter in a scene. At this time, a popular strategy is to produce an oversegmentation of an image, where one is guaranteed not to have missed marking any of the true boundaries but may have marked many extra false boundaries as well. The resulting regions, called superpixels, provide a significant reduction in computational complexity for various algorithms, as the number of superpixels may be in the hundreds, compared to millions of raw pixels. Exploiting high-level knowledge of objects is the subject of the next section, and actually detecting the objects in images is the subject of Section 27.5.

27.4 Classifying Images

Image classification applies to two main cases. In one, the images are of *objects*, taken from a given taxonomy of classes, and there's not much else of significance in the picture—for example, a catalog of clothing or furniture images, where the background doesn't matter, and the output of the classifier is "cashmere sweater" or "desk chair."

In the other case, each image shows a *scene* containing multiple objects. So in grassland you might see a giraffe and a lion, and in the living room you might see a couch and lamp, but you don't expect a giraffe or a submarine in a living room. We now have methods for large-scale image classification that can accurately output "grassland" or "living room."

Modern systems classify images using **appearance** (i.e., color and texture, as opposed to geometry). There are two difficulties. First, different instances of the same class could look different—some cats are black and others are orange. Second, the same cat could look different at different times depending on several effects, (as illustrated in Figure 27.11):

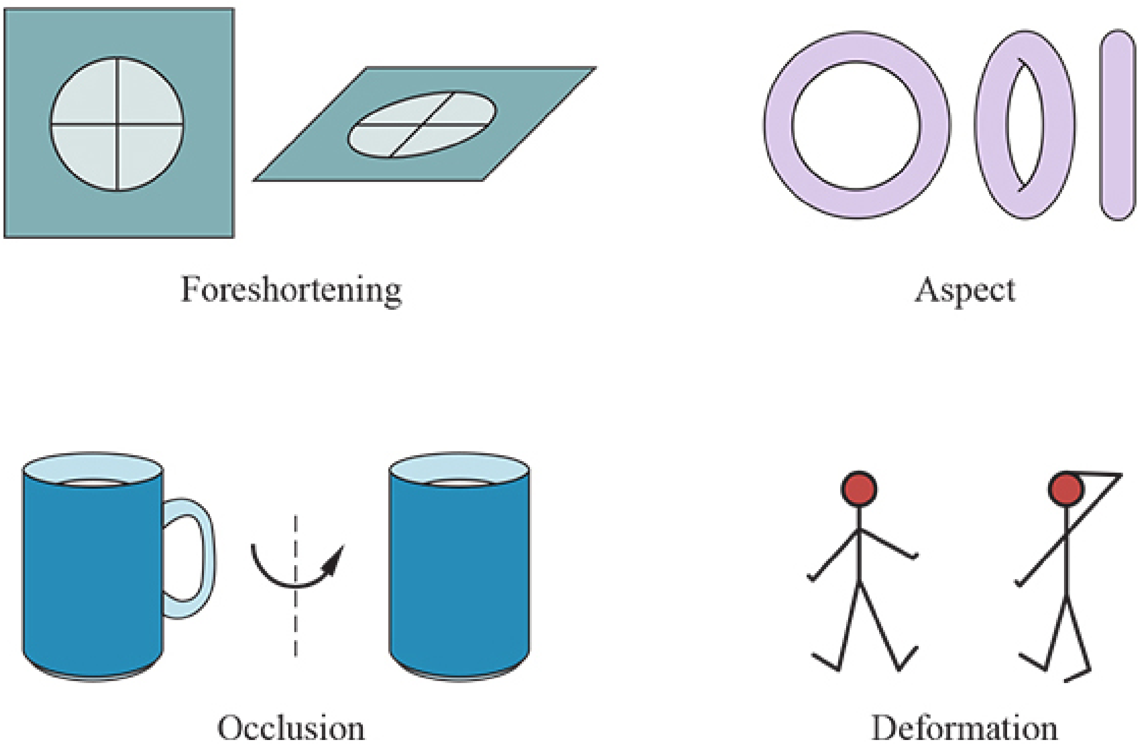


Figure 27.11 Important sources of appearance variation that can make different images of the same object look different. First, elements can foreshorten, like the circular patch on the top left. This patch is viewed at a glancing angle, and so is elliptical in the image. Second, objects viewed from different directions can change shape quite dramatically, a phenomenon known as aspect. On the top right are three different aspects of a doughnut. Occlusion causes the handle of the mug on the bottom left to disappear when the mug is rotated. In this case, because the body and handle belong to the same mug, we have self-occlusion. Finally, on the bottom right, some objects can deform dramatically.

- **Lighting**, which changes the brightness and color of the image.
- **Foreshortening**, which causes a pattern viewed at a glancing angle to be distorted.
- **Aspect**, which causes objects to look different when seen from different directions. A doughnut seen from the side looks like a flattened oval, but from above it is an annulus.
- **Occlusion**, where some parts of the object are hidden. Objects can occlude one another, or parts of an object can occlude other parts, an effect known as **self-occlusion**.
- **Deformation**, where the object changes its shape. For example, the tennis player moves her arms and legs.

Modern methods deal with these problems by learning representations and classifiers from very large quantities of training data using a convolutional neural network. With a sufficiently rich training set the classifier will have seen any effect of importance many times in training, and so can adjust for the effect.

27.4.1 Image classification with convolutional neural networks

Convolutional neural networks (CNNs) are spectacularly successful image classifiers. With enough training data and enough training ingenuity, CNNs produce very successful classification systems, much better than anyone has been able to produce with other methods.

The ImageNet data set played a historic role in the development of image classification systems by providing them with over 14 million training images, classified into over 30,000 fine-grained categories. ImageNet also spurred progress with an annual competition. Systems are evaluated by both the classification accuracy of their single best guess and by top-5 accuracy, in which systems are allowed to submit five guesses—

for example, *malamute, husky, akita, samoyed, eskimo dog*. ImageNet has 189 subcategories of *dog*, so even dog-loving humans find it hard to label images correctly with a single guess.

In the first ImageNet competition in 2010, systems could do no better than 70% top-5 accuracy. The introduction of convolutional neural networks in 2012 and their subsequent refinement led to an accuracy of 98% in top-5 (surpassing human performance) and 87% in top-1 accuracy by 2019. The primary reason for this success seems to be that the features that are being used by CNN classifiers are learned from data, not hand-crafted by a researcher; this ensures that the features are actually useful for classification.

Progress in image classification has been rapid because of the availability of large, challenging data sets such as ImageNet; because of competitions based on these data sets that are fair and open; and because of the widespread dissemination of successful models. The winners of competitions publish the code and often the pretrained parameters of their models, making it easy for others to fiddle with successful architectures and try to make them better.

27.4.2 Why convolutional neural networks classify images well

Image classification is best understood by looking at data sets, but ImageNet is much too large to look at in detail. The MNIST data set is a collection of 70,000 images of handwritten digits, 0–9, which is often used as a standard warmup data set. Looking at this data set (some examples appear in Figure 27.12) exposes some important, quite general, properties. You can take an image of a digit and make a number of small alterations without changing the identity of the digit: you can shift it, rotate it, make it brighter or darker, smaller or larger. This means that individual pixel values are not particularly informative—we know that an 8 should have some dark pixels in the center and a 0 should not, but those dark pixels will be in slightly different pixel locations in each instance of an 8.

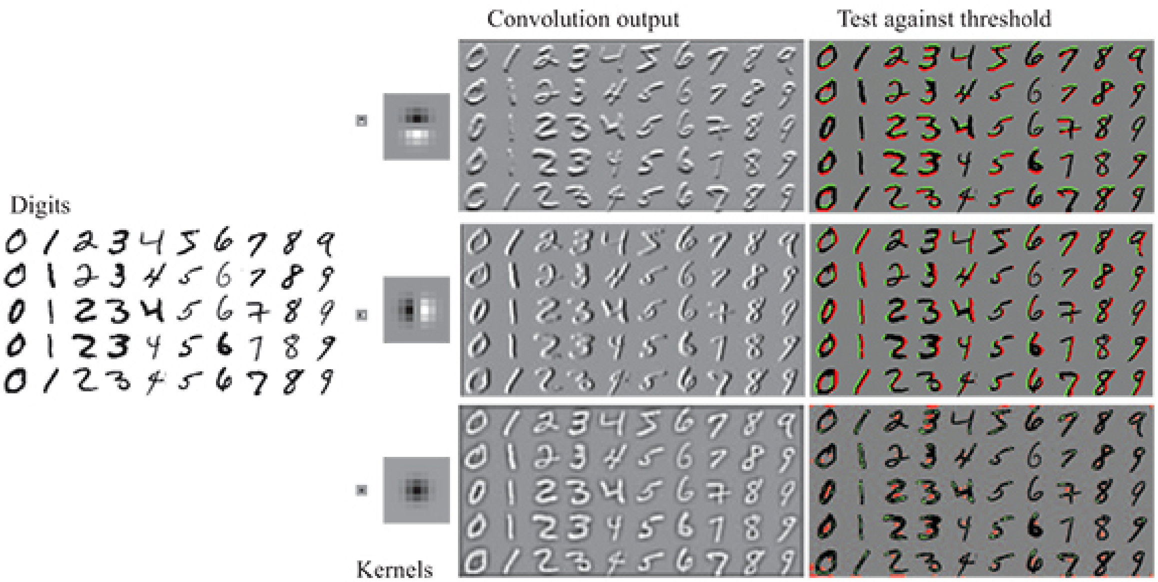


Figure 27.12 On the far left, some images from the MNIST data set. Three kernels appear on the center left. They are shown at actual size (tiny blocks) and magnified to reveal their content: mid-grey is zero, light is positive, and dark is negative. Center right shows the results of applying these kernels to the images. Right shows pixels where the response is bigger than a threshold (green) or smaller than a threshold (red). You should notice that this gives (from top to bottom): a horizontal bar detector; a vertical bar detector; and (harder to note) a line ending detector. These detectors pay attention to the contrast of the bar, so (for example) a horizontal bar that is light on top and dark below produces a positive (green) response, and one that is dark on top

and light below gets a negative (red) response. These detectors are moderately effective, but not perfect.

Another important property of images is that local patterns can be quite informative: The digits 0, 6, 8 and 9 have loops; the digits 4 and 8 have crossings; the digits 1, 2, 3, 5 and 7 have line endings, but no loops or crossings; the digits 6 and 9 have loops and line endings. Furthermore, spatial relations between local patterns are informative. A 1 has two line endings above one another; a 6 has a line ending above a loop. These observations suggest a strategy that is a central tenet of modern computer vision: you construct features that respond to patterns in small, localized neighborhoods; then other features look at patterns of *those* features; then others look at patterns of those, and so on.

This is what convolutional neural networks do well. You should think of a layer—a convolution followed by a ReLU activation function—as a local pattern detector (Figure 27.12). The convolution measures how much each local window of the image looks like the kernel pattern; the ReLU sets low-scoring windows to zero, and emphasizes high-scoring windows. So convolution with multiple kernels finds multiple patterns; furthermore, composite patterns can be detected by applying another layer to the output of the first layer.

Think about the output of the first convolutional layer. Each location receives inputs from pixels in a window about that location. The output of the ReLU, as we have seen, forms a simple pattern detector. Now if we put a second layer on top of this, each location in the second layer receives inputs from first-layer values in a window about that location. This means that locations in the second layer are affected by a larger window of pixels than those in the first layer. You should think of these as representing “patterns of patterns.” If we place a third layer on top of the second layer, locations in that third layer will depend on an even larger window of pixels; a fourth layer will depend on a yet larger window, and so on. The network is creating patterns at multiple levels, and is doing that by *learning* from the data rather than having the patterns given to it by a programmer.

While training a CNN “out of the box” does sometimes work, it helps to know a few practical techniques. One of the most important is **data set augmentation**, in which training examples are copied and modified slightly. For example, one might randomly shift, rotate, or stretch an image by a small amount, or randomly shift the hue of the pixels by a small amount. Introducing this simulated variation in viewpoint or lighting to the data set helps to increase the size of the data set, though of course the new examples are highly correlated with the originals. It is also possible to use augmentation at test time rather than training time. In this approach, the image is replicated and modified several times (e.g., with random cropping) and the classifier is run on each of the modified images. The outputs of the classifier from each copy are then used to vote for a final decision on the overall class.

When you are classifying images of scenes, every pixel could be helpful. But when you are classifying images of objects, some pixels aren't part of the object, and so might be a distraction. For example, if a cat is lying on a dog bed, we want a classifier to concentrate on the pixels of the cat, not the bed. Modern image classifiers handle this well, classifying an image as “cat” accurately even if few pixels actually lie on the cat. There are two reasons for this. First, CNN-based classifiers are good at ignoring patterns that aren’t discriminative. Second, patterns that lie off the object might be discriminative (e.g., a cat toy, a collar with a little bell, or a dish of cat food might actually help tell that we are looking at a cat). This effect is known as **context**. Context can help or can hurt, depending quite strongly on the particular data set and application.

27.5 Detecting Objects

Image classifiers predict *what* is in the image—they classify the whole image as belonging to one class. Object detectors find multiple objects in an image, report *what* class each object is, and also report *where* each object is by giving a **bounding box** around the object.[1] The set of classes is fixed in advance. So we might try to detect all faces, all cars, or all cats.

We can build an object detector by looking at a small **sliding window** onto the larger image—a rectangle. At each spot, we classify what we see in the window, using a CNN classifier. We then take the high-scoring classifications—a cat over here and a dog over there—and ignore the other windows. After some work resolving conflicts, we have a final set of objects with their locations. There are still some details to work out:

- **Decide on a window shape:** The easiest choice by far is to use axis-aligned rectangles. (The alternative—some form of mask that cuts the object out of the image—is hardly ever used, because it is hard to represent and to compute with.) We still need to choose the width and height of the rectangles.
- **Build a classifier for windows:** We already know how to do this with a CNN.
- **Decide which windows to look at:** Out of all possible windows, we want to select ones that are likely to have interesting objects in them.
- **Choose which windows to report:** Windows will overlap, and we don't want to report the same object multiple times in slightly different windows. Some objects are not worth mentioning; think about the number of chairs and people in a picture of a large packed lecture hall. Should they all be reported as individual objects? Perhaps only the

objects that appear large in the image—the front row—should be reported. The choice depends on the intended use of the object detector.

- **Report precise locations of objects using these windows:** Once we know that the object is somewhere in the window, we can afford to do more computation to figure out a more precise location within the window.

Let’s look more carefully at the problem of deciding which windows to look at. Searching all possible windows isn’t efficient—in an $n \times n$ pixel image there are $O(n^4)$ possible rectangular windows. But we know that windows that contain objects tend to have quite coherent color and texture. On the other hand, windows that cut an object in half have regions or edges that cross the side of the window. So it makes sense to have a mechanism that scores “objectness”—whether a box has an object in it, independent of what that object is. We can find the boxes that look like they have an object in them, and then classify the object for just those boxes that pass the objectness test.

A network that finds regions with objects is called a **regional proposal network (RPN)**. The object detector known as Faster RCNN encodes a large collection of bounding boxes as a map of fixed size. Then it builds a network that can predict a score for each box, and trains this network so the score is large when the box contains an object, and small otherwise. Encoding boxes as a map is straightforward. We consider boxes centered on points throughout the image; we don't need to consider every possible point (because moving by one pixel is not likely to change the classification); a good choice is a **stride** (the offset between center points) of 16 pixels. For each center point we consider several possible boxes, called **anchor boxes**.

Faster RCNN uses nine boxes: small, medium, and large sizes; and tall, wide, and square aspect ratios.

In terms of the neural network architecture, construct a 3D block where each spatial location in the block has two dimensions for the center point and one dimension for the type of box. Now any box with a good enough objectness score is called a **region of interest** (ROI), and must be checked by a classifier. But CNN classifiers prefer images of fixed size, and the boxes that pass the objectness test will differ in size and shape. We can't make the boxes have the same number of pixels, but we can make them have the same number of features by sampling the pixels to extract features, a process called **ROI pooling**. This fixed-size feature map is then passed to the classifier.

Now for the problem of deciding which windows to report. Assume we look at windows of size 32×32 with a stride of 1: each window is offset by just one pixel from the one before. There will be many windows that are similar, and should have similar scores. If they all have a score above threshold we don't want to report all of them, because they very likely all refer to slightly different views of the same object. On the other hand if the stride is too large, it might be that an object is not contained within any one window, and will be missed. Instead, we can use a greedy algorithm called **non-maximum suppression**. First, build a sorted list of all windows with scores over a threshold. Then, while there are windows in the list, choose the window with the highest score and accept it as containing an object; discard from the list all other largely overlapping windows.

Finally, we have the problem of reporting the precise location of objects. Assume we have a window that has a high score, and has passed through non-maximum suppression. This window is unlikely to be in exactly the right place (remember, we looked at a relatively small number of windows with a small number of possible sizes). We use the feature representation computed by the classifier to predict improvements that will trim the window down to a proper bounding box, a step known as **bounding box regression**.

Evaluating object detectors takes care. First we need a test set: a collection of images with each object in the image marked by a ground truth category label and bounding box. Usually, the boxes and labels are supplied by humans. Then we feed each image to the object detector and compare its output to the ground truth. We should be willing to accept boxes that are off by a few pixels, because the ground truth boxes won't be perfect. The evaluation score should balance recall (finding all the objects that are there) and precision (not finding objects that are not there).

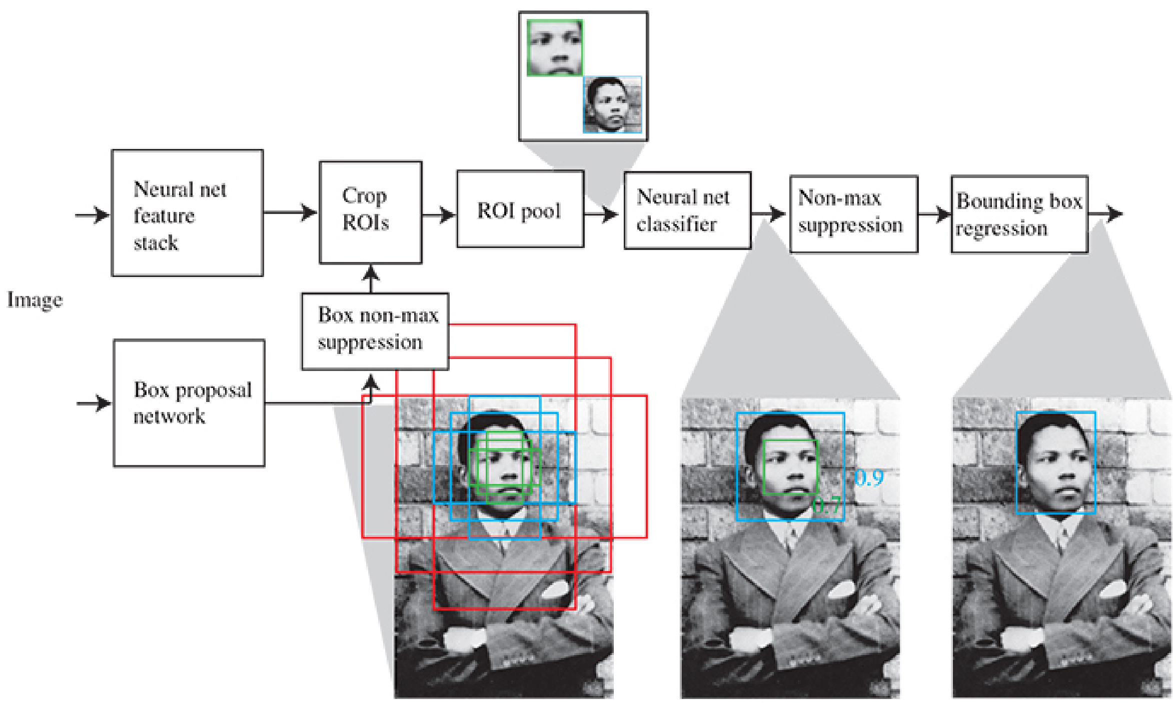


Figure 27.13 Faster RCNN uses two networks. A picture of a young Nelson Mandela is fed into the object detector. One network computes “objectness” scores of candidate image boxes, called “anchor boxes,” centered at a grid point. There are nine anchor boxes (three scales, three aspect ratios) at each grid point. For the example image, an inner green box and an outer blue box have passed the objectness test. The second network is a feature stack that computes a representation of the image suitable for classification. The boxes with highest objectness score are cut from the feature map, standardized in size with ROI pooling, and passed to a classifier. The blue box has a higher score than the green box and overlaps it, so the green box is rejected by non-maximum suppression. Finally, bounding box regression the blue box so that it fits the face. This means that the relatively coarse sampling of locations, scales, and aspect ratios does not weaken accuracy. Photo by Sipa/Shutterstock.

27.6 The 3D World

Images show a 2D picture of a 3D world. But this 2D picture is rich with cues about the 3D world. One kind of cue occurs when we have multiple pictures of the same world, and can match points between pictures. Another kind of cue is available within a single picture.

27.6.1 3D cues from multiple views

Two pictures of objects in a 3D world are better than one for several reasons:

- If you have two images of the same scene taken from different viewpoints and you know enough about the two cameras, you can construct a 3D model—a collection of points with their coordinates in 3 dimensions—by figuring out which point in the first view corresponds to which point in the second view and applying some geometry. This is true for almost all pairs of viewing directions and almost all kinds of camera.
- If you have two views of enough points, and you know which point in the first view corresponds to which point in the second view, you do not need to know much about the cameras to construct a 3D model. Two views of two points gives you four x, y coordinates, and you only need three coordinates to specify a point in 3D space; the extra coordinate comes in helpful to figure out what you need to know about the cameras. This is true for almost all pairs of viewing directions and almost all kinds of camera.

The key problem is to establish which point in the first view corresponds to which in the second view. Detailed descriptions of the local appearance of a point using simple texture features (like those in Section 27.3.2) are often enough to match points. For example, in a scene of traffic on a street, there might be only one green light visible in two images taken of the scene; we can then hypothesize that these correspond to each other. The geometry of multiple camera views is very well understood (but sadly too complicated to expound here). The theory produces geometric constraints on which point in one image can match with which point in the other. Other constraints can be obtained by reasoning about the smoothness of the reconstructed surfaces.

There are two ways of getting multiple views of a scene. One is to have two cameras or two eyes (Section 27.6.2). Another is to move (Section 27.6.3). If you have more than two views, you can recover both the geometry of the world and the details of the view very accurately. Section 27.7.3 discusses some applications for this technology.

27.6.2 Binocular stereopsis

Most vertebrates have two eyes. This is useful for redundancy in case of a lost eye, but it helps in other ways too. Most prey have eyes on the side of the head to enable a wider field of vision. Predators have the eyes in the front, enabling them to use **binocular stereopsis**. Hold both index fingers up in front of your face, with one eye closed, and adjust them so the front finger occludes the other finger *in the open eye's view.* Now swap eyes; you should notice that the fingers have shifted position with respect to one another. This shifting of position from left view to right view is known as **disparity**. In the right choice of coordinate system, if we superimpose left and right images of an object at some depth, the object shifts horizontally in the superimposed image, and the size of the shift is the reciprocal of the depth. You can see this in Figure 27.14, where the nearest point of the pyramid is shifted to the left in the right image and to the right in the left image.

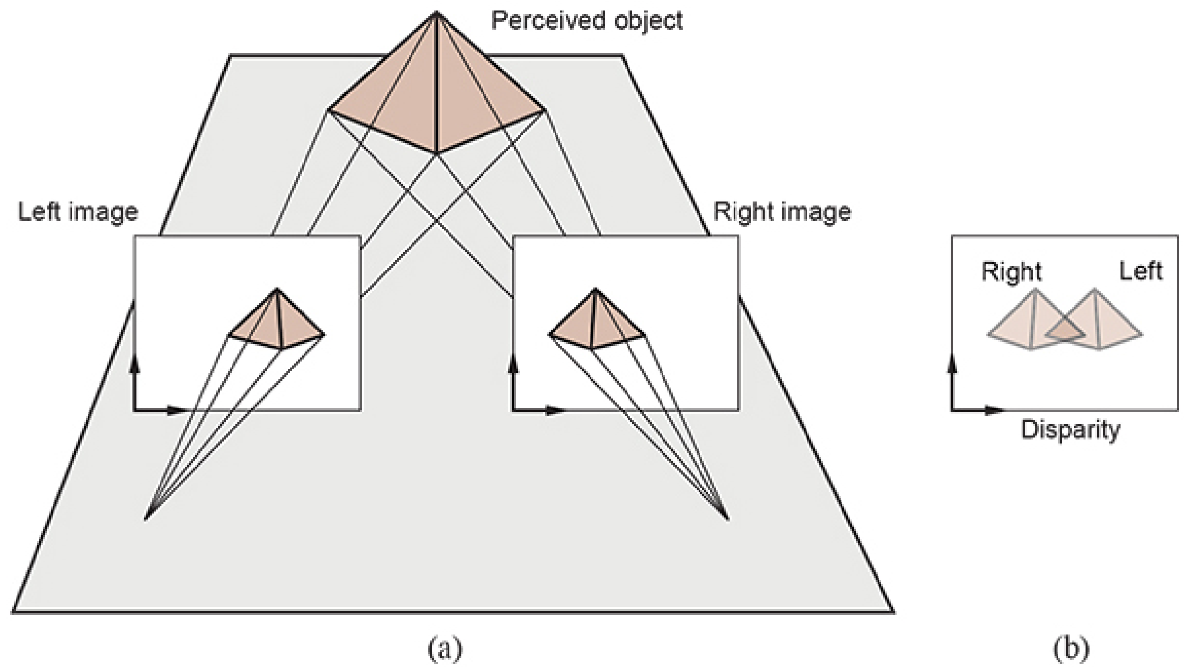


Figure 27.14 Translating a camera parallel to the image plane causes image features to move in the camera plane. The disparity in positions that results is a cue to depth. If we superimpose left and right images, as in (b), we see the disparity.

To measure disparity we need to solve the correspondence problem—to determine for a point in the left image, its "partner" in the right image which results from the projection of the same scene point. This is analogous to what is done in measuring optical flow, and the most simple-minded approaches are somewhat similar. These methods search for blocks of left and right pixels that match, using the sum of squared differences (as in Section 27.3.3). More sophisticated methods use more detailed texture representations of blocks of pixels (as in Section 27.3.2). In practice, we use much more sophisticated algorithms, which exploit additional constraints.

Assuming that we can measure disparity, how does this yield information about depth in the scene? We will need to work out the geometrical relationship between disparity and depth. We will consider first the case when both the eyes (or cameras) are looking forward with their optical axes parallel. The relationship of the right camera to the left camera is then just a displacement along the x-axis by an amount b, the **baseline**. We can use the optical flow equations from Section 27.3.3, if we think of this as resulting from a translation vector $\mathbf{T}$ acting for time δt, with $T_x = b/\delta t$ and $T_y = T_z = 0$. The horizontal and vertical disparity are given by the optical flow components, multiplied by the time step δt, $H = v_x\delta t$, $V = v_y\delta t$. Carrying out the substitutions, we get the result that $H = b/Z$, $V = 0$. In other words, the horizontal disparity is equal to the ratio of the baseline to the depth, and the vertical disparity is zero. We can recover the depth Z given that we know b, and can measure H.

Under normal viewing conditions, humans **fixate**; that is, there is some point in the scene at which the optical axes of the two eyes intersect. Figure 27.15 shows two eyes fixated at a point P_0, which is at a distance Z from the midpoint of the eyes. For convenience, we will compute the *angular* disparity, measured in radians. The disparity at the point of fixation P_0 is zero. For some other point P in the scene that is δZ farther away, we can compute the angular displacements of the left and right images of P, which we will call P_L and P_R, respectively. If each of these is displaced by an angle $\delta\theta/2$ relative to P_0, then the displacement between P_L and P_R, which is the disparity of P, is just $\delta\theta$. From Figure 27.15, $\tan\theta = \frac{b/2}{Z}$ and $\tan(\theta - \delta\theta/2) = \frac{b/2}{Z+\delta Z}$, but for small angles, $\tan\theta \approx \theta$, so

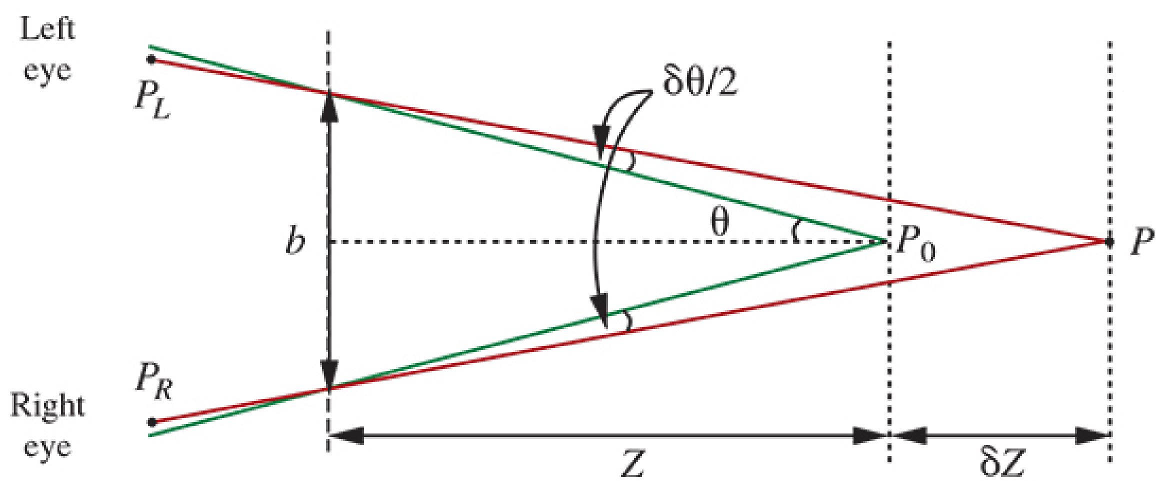


Figure 27.15 The relation between disparity and depth in stereopsis. The centers of projection of the two eyes are distance b apart, and the optical axes intersect at the fixation point P_0. The point P in the scene projects to points P_L and P_R in the two eyes. In angular terms, the disparity between these is $\delta\theta$ (the diagram shows two angles of $\delta\theta/2$).

$$\delta\theta/2 = \frac{b/2}{Z} - \frac{b/2}{Z+\delta Z} \approx \frac{b\delta Z}{2Z^2}$$

and, since the actual disparity is $\delta\theta$, we have

$$\text{disparity} = \frac{b\delta Z}{Z^2}$$

In humans, the baseline b is about 6 cm. Suppose that Z is about 100 cm and that the smallest detectable $\delta\theta$ (corresponding to the size of a single pixel) is about 5 seconds of arc, giving a δZ of 0.4 mm. For Z = 30 cm, we get the impressively small value δZ = 0.036 mm. That is, at a distance of 30 cm, humans can discriminate depths that differ by as little as 0.036 mm, enabling us to thread needles and the like.

27.6.3 3D cues from a moving camera

Assume we have a camera moving in a scene. Take Figure 27.14 and label the left image "Time t" and the right image "Time t +1". The geometry has not changed, so all the material from the discussion of stereopsis also applies when a camera moves. What we called disparity in that section is now thought of as apparent motion in the image, and called optical flow. This is a source of information for both the movement of the camera and the geometry of the scene. To understand this, we state (without proof) an equation that relates the optical flow to the viewer's translational velocity **T** and the depth in the scene.

The optical flow field is a vector field of velocities in the image, $(v_x(x, y), v_y(x, y))$. Expressions for these components, in a coordinate frame centered on the camera and assuming a focal length of $f = 1$, are

$$v_x(x,y) = \frac{-T_x + xT_z}{Z(x,y)} \text{ and } \qquad v_y(x,y) = \frac{-T_y + yT_z}{Z(x,y)} \quad .$$

where $Z(x, y)$ is the z-coordinate (that is, depth) of the point in the scene corresponding to the point in the image at (x, y).

Note that both components of the optical flow, $v_x(x, y)$ and $v_y(x, y)$, are zero at the point $x = T_x/T_z$, $y = T_y/T_z$. This point is called the **focus of expansion** of the flow field. Suppose we change the origin in the x–y plane to lie at the focus of expansion; then the expressions for optical flow take on a particularly simple form. Let (x', y') be the new coordinates defined by $x' = x - T_x/T_z$, $y' = y - T_y/T_z$. Then

$$v_x(x', y') = \frac{x'T_z}{Z(x',y')}, \qquad v_y(x', y') = \frac{y'T_z}{Z(x',y')} .$$

Note that there is a scale factor ambiguity here (which is why assuming a focal length of $f = 1$ is harmless). If the camera was moving twice as fast, and every object in the scene was twice as big and at twice the distance to the camera, the optical flow field would be exactly the same. But we can still extract quite useful information.

1. Suppose you are a fly trying to land on a wall and you want useful information from the optical flow field. The optical flow field cannot tell you the distance to the wall or the velocity to the wall, because of the scale ambiguity. But if you divide the distance by the velocity, the scale ambiguity cancels. The result is the time to contact, given by Z/T_z, and is very useful indeed to control the landing approach. There is considerable experimental evidence that many different animal species exploit this cue.
2. Consider two points at depths Z_1, Z_2 respectively. We may not know the absolute value of either of these, but by considering the inverse of the ratio of the optical flow magnitudes at these points, we can determine the depth ratio Z_1/Z_2. This is the cue of motion parallax, one we use when we look out of the side window of a moving car or train and infer that the slower-moving parts of the landscape are farther away.

27.6.4 3D cues from one view

Even a single image provides a rich collection of information about the 3D world. This is true even if the image is just a line drawing. Line drawings have fascinated vision scientists, because people have a sense of 3D shape and layout even though the drawing seems to contain very little information to choose from the vast collection of scenes that could produce the same drawing. Occlusion is one key source of information: if there is evidence in the picture that one object occludes another, then the occluding object is closer to the eye.

In images of real scenes, texture is a strong cue to 3D structure. Section 27.3.2 stated that texture is a repetitive pattern of texels. Although the distribution of texels may be uniform on objects in the scene—for example, pebbles on a beach—it may not be uniform in image—the farther pebbles appear smaller than the nearer pebbles. As another example, think about a piece of polka-dot fabric. All the dots are the same size and shape on the fabric, but in a perspective view some dots are ellipses due to foreshortening. Modern methods exploit these cues by learning a mapping from images to 3D structure (Section 27.7.4), rather than reasoning directly about the underlying mathematics of texture.

Shading—variation in the intensity of light received from different portions of a surface in a scene—is determined by the geometry of the scene and by the reflectance properties of the surfaces. There is very good evidence that shading is a cue to 3D shape. The physical argument is easy. From the physical model of Section 27.2.4, we know that if a surface normal points toward the light source, the surface is brighter, and if it points away, the surface is darker. This argument gets more complicated if the reflectance of the surface isn't known, and the illumination field isn't even, but humans seem to be able to get a useful perception of shape from shading. We know frustratingly little about algorithms to do this.

If there is a familiar object in the picture, what it looks like depends very strongly on its **pose**, that is, its position and orientation with respect to the viewer. There are straightforward algorithms for recovering pose from correspondences between points on an object and points on a model of the object. Recovering the pose of a known object has many applications. For instance, in an industrial manipulation task, the robot arm cannot pick up an object until the pose is known. Robotic surgery applications depend on exactly computing the transformations between the camera's position and the positions of the surgical tool and the patient (to yield the transformation from the tool's position to the patient's position).

Spatial relations between objects are another important cue. Here is an example. All pedestrians are about the same height, and they tend to stand on a ground plane. If we know where the horizon is in an image, we can rank pedestrians by distance to the camera. This works because we know where their feet are, and pedestrians whose feet are closer to the horizon in the image are farther away from the camera, and so must be smaller in the image. This means we can rule out some detector responses—if a detector finds a pedestrian who is large in the image and whose feet are close to the horizon, it has found an enormous pedestrian; these don’t exist, so the detector is wrong. In turn, a reasonably reliable pedestrian detector is capable of producing estimates of the horizon, if there are several pedestrians in the scene at different distances from the camera. This is because the relative scaling of the pedestrians is a cue to where the horizon is. So we can extract a horizon estimate from the detector, then use this estimate to prune the pedestrian detector’s mistakes.

27.7 Using Computer Vision

Here we survey a range of computer vision applications. There are now many reliable computer vision tools and toolkits, so the range of applications that are successful and useful is extraordinary. Many are developed at home by enthusiasts for special purposes, which is testimony to how usable the methods are and how much impact they have. (For example, an enthusiast created a great object-detection-based pet door that refuses entry to a cat if it is bringing in a dead mouse–a Web search will find it for you).

27.7.1 Understanding what people are doing

If we could build systems that understood what people are doing by analyzing video, we could build human-computer interfaces that watch people and react to their behavior. With these interfaces, we could: design buildings and public places better, by collecting and using data about what people do in public; build more accurate and less intrusive security surveillance systems; build automated sports commentators; make construction sites and workplaces safer by generating warnings when people and machines get dangerously close; build computer games that make a player get up and move around; and save energy by managing heat and light in a building to match where the occupants are and what they are doing.

The state of the art for some problems is now extremely strong. There are methods that can predict the locations of a person's joints in an image very accurately. Quite good estimates of the 3D configuration of that person's body follow (see Figure 27.16). This works because pictures of the body tend to have weak perspective effects, and body segments don't vary much in length, so the foreshortening of a body segment in an image is a good cue to the angle between it and the camera plane. With a depth sensor, these estimates can be made fast enough to build them into computer game interfaces.

Figure 27.16 Reconstructing humans from a single image is now practical. Each row shows a reconstruction of 3D body shape obtained using a single image. These reconstructions are possible because methods can estimate the location of joints, thejoint angles in 3D, the shape of the body, and the pose of the body with respect to an image. Each row shows the following: **far left** a picture; **center left** the picture with the reconstructed body

superimposed; **center right** another view of the reconstructed body; and **far right** yet another view of the reconstructed body. The different views of the body make it much harder to conceal errors in reconstruction. Figure courtesy of Angjoo Kanazawa, produced by a system described in Kanazawa *et al.* (2018a).

Classifying what people are doing is harder. Video that shows rather structured behaviors, like ballet, gymnastics, or tai chi, where there are quite specific vocabularies that refer to very precisely delineated activities on simple backgrounds, is quite easy to deal with. Good results can be obtained with a lot of labeled data and an appropriate convolutional neural network. However, it can be difficult to prove that the methods actually work, because they rely so strongly on context. For example, a classifier that labels "swimming" sequences very well might just be a swimming pool detector, which wouldn't work for (say) swimmers in rivers.

More general problems remain open—for example, how to link observations of the body and the objects nearby to the goals and intentions of the moving people. One source of difficulty is that similar behaviors look different, and different behaviors look similar, as Figure 27.17 shows.

Figure 27.17 The same action can look very different; and different actions can look similar. These examples show actions taken from a data set of natural behaviors; the labels are chosen by the curators of the data set, rather than predicted by an algorithm. **Top:** examples of the label “opening fridge,” some shown in closeup and some from afar. **Bottom:** examples of the label “take something out of fridge.” Notice how in both rows the subject’s hand is close to the fridge door—telling the difference between the cases requires quite subtle judgment about where the hand is and where the door is. Figure courtesy of David Fouhey, taken from a data set described in Fouhey *et al.* (2018).

Another difficulty is caused by time scale. What someone is doing depends quite strongly on the time scale, as Figure 27.18 illustrates. Another important effect shown in that figure is that behavior composes—several recognized behaviors may be combined to form a single higher-level behavior such as fixing a snack.

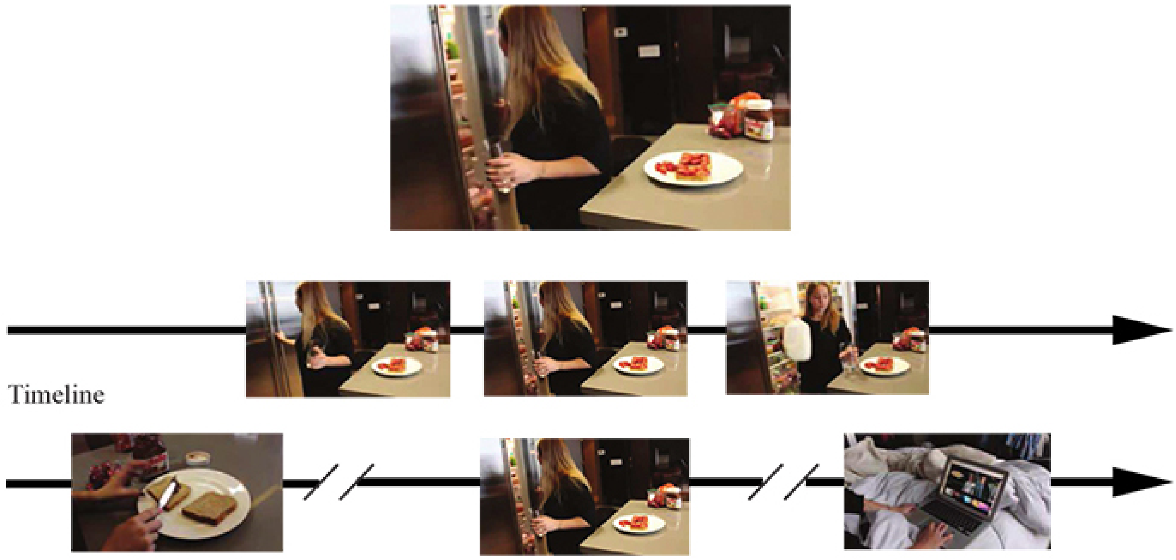


Figure 27.18 What you call an action depends on the time scale. The single frame at the **top** is best described as opening the fridge (you don't gaze at the contents when you close a fridge). But if you look at a short clip of video (indicated by the frames in the **center** row), the action is best described as getting milk from the fridge. If you look at a long clip (the frames in the **bottom** row), the action is best described as fixing a snack. Notice that this illustrates one way in which behavior composes: getting milk from the fridge is sometimes part of fixing a snack, and opening the fridge is usually part of getting milk from the fridge. Figure courtesy of David Fouhey, taken from a data set described in Fouhey *et al*. (2018).

It may also be that unrelated behaviors are going on at the same time, such as singing a song while fixing a snack. A challenge is that we don't have a common vocabulary for the pieces of behavior. People tend to think they know a lot of behavior names but can't produce long lists of such words on demand. That makes it harder to get data sets of consistently labeled behaviors.

Learned classifiers are guaranteed to behave well only if the training and test data come from the same distribution. We have no way of checking that this constraint applies to images, but empirically we observe that image classifiers and object detectors work very well. But for activity data, the relationship between training and test data is more untrustworthy because people do so many things in so many contexts. For example, suppose we have a pedestrian detector that performs well on a large data set. There will be rare phenomena (for example, people mounting unicycles) that do not appear in the training set, so we can't say for sure how the detector will work in such cases. The challenge is to prove that the detector is safe whatever pedestrians do, which is difficult for current theories of learning.

27.7.2 Linking pictures and words

Many people create and share pictures and videos on the Internet. The difficulty is finding what you want. Typically, people want to search using words (rather than, say, example sketches). Because most pictures don't come with words attached, it is natural to try and build **tagging systems** that tag images with relevant words. The underlying machinery is straightforward—we apply image classification and object detection methods and tag the image with the output words. But tags aren't a comprehensive description of what is happening in an image. It matters who is doing what, and tags don't capture this. For example, tagging a picture of a cat in the street with the object categories "cat", "street", "trash can" and "fish bones" leaves out the information that the cat is pulling the fish bones out of an open trash can on the street.

As an alternative to tagging, we might build **captioning systems**—systems that write a caption of one or more sentences describing the image. The underlying machinery is again straightforward—couple a convolutional network (to represent the image) to a recurrent neural network or transformer network (to generate sentences), and train the result with a data set of captioned images. There are many images with captions available on the Internet; curated data sets use human labor to augment each image with additional captions to capture the variation in natural language. For example, the COCO (Common Objects in Context) data set is a comprehensive collection of over 200,000 images labeled with five captions per image.

Current methods for captioning use detectors to find a set of words that describe the image, and provide those words to a sequence model that is trained to generate a sentence. The most accurate methods search through the sentences that the model can generate to find the best, and strong methods appear to require a slow search. Sentences are evaluated with a set of scores that check whether the generated sentence (a) uses phrases common in the ground truth annotations and (b) doesn’t use other phrases. These scores are hard to use directly as a loss function, but reinforcement learning methods can be used to train networks that get very good scores. Often there will be an image in the training set whose description has the same set of words as an image in the test set; in that case a captioning system can just retrieve a valid caption rather than having to generate a new one. Caption writing systems produce a mix of excellent results and embarrassing errors (see Figure 27.19).

Figure 27.19 Automated image captioning systems produce some good results and some failures. The two captions at left describe the respective images well, although “eating ... in his mouth” is a disfluency that is fairly typical of the recurrent neural network language models used by early captioning systems. For the two captions on the right, the captioning system seems not to know about squirrels, and so guesses the animal from context; it also fails to recognize that the two squirrels are eating. Image credits: geraine/Shutterstock; ESB Professional/Shutterstock; BushAlex/Shutterstock; Maria.Tem/Shutterstock. The images shown are similar but not identical to the original images from which the captions were generated. For the original images see Aneja *et al.* (2018).

Captioning systems can hide their ignorance by omitting to mention details they can’t get right or by using contextual cues to guess. For example, captioning systems tend to be poor at identifying the gender of people in images, and often guess based on training data statistics. That can lead to errors—men also like shopping and women also snowboard. One way to establish whether a system has a good representation of what is happening in an image is to force it to answer questions about the image.

This is a **visual question answering** or **VQA** system. An alternative is a **visual dialog** system, which is given a picture, its caption, and a dialog. The system must then answer the last question in the dialog. As Figure 27.20 shows, vision remains extremely hard and VQA systems often make errors.

Figure 27.20 Visual question-answering systems produce answers (typically chosen from a multiple-choice set) to natural-language questions about images. **Top:** the system is producing quite sensible answers to rather difficult questions about the image. **Bottom:** less satisfactory answers. For example, the system is guessing about the number of holes in a pizza, because it doesn't understand what counts as a hole, and it has real difficulty counting. Similarly, the system selects brown for the cat's leg because the background is brown and it can't localize the leg properly. Image credits: (Top) Tobyanna/Shutterstock; 679411/Shutterstock; ESB Professional/Shutterstock; Africa Studio/Shutterstock; (Bottom) Stuart Russell;

Maxisport/Shutterstock; Chendongshan/Shutterstock; Scott Biales DitchTheMap/Shutterstock. The images shown are similar but not identical to the original images to which the questionanswering system was applied. For the original images see Goyal *et al.* (2017).

27.7.3 Reconstruction from many views

Reconstructing a set of points from many views—which could come from video or from an aggregation of tourist photographs—is similar to reconstructing the points from two views, but there are some important differences. There is far more work to be done to establish correspondence between points in different views, and points can go in and out of view, making the matching and reconstruction process messier. But more views means more constraints on the reconstruction and on the recovered viewing parameters, so it is usually possible to produce extremely accurate estimates of both the position of the points and of the viewing parameters. Rather roughly, reconstruction proceeds by matching points over pairs of images, extending these matches to groups of images, coming up with a rough solution for both geometry and viewing parameters, then polishing that solution. Polishing means minimizing the error between points predicted by the model (of geometry and viewing parameters) and the locations of image features. The detailed procedures are too complex to cover fully, but are now very well understood and quite reliable.

All the geometric constraints on correspondences are known for any conceivably useful form of camera. The procedures can be generalized to deal with views that are not orthographic; to deal with points that are observed in only some views; to deal with unknown camera parameters (like focal length); and to exploit various sophisticated searches for appropriate correspondences. It is practical to accurately reconstruct a model of an entire city from images. Some applications are:

- **Model building:** For example, one might build a modeling system that takes many views depicting an object and produces a very detailed 3D mesh of textured polygons for use in computer graphics and virtual reality applications. It is routine to build models like this from video, but such models can now be built from apparently random sets of pictures. For example, you can build a 3D model of the Statue of Liberty from pictures found on the Internet.
- **Mix animation with live actors in video:** To place computer graphics characters into real video, we need to know how the camera moved for the real video, so we can render the character correctly, changing the view as the camera moves.
- **Path reconstruction:** Mobile robots need to know where they have been. If the robot has a camera, we can build a model of the camera's path through the world; that will serve as a representation of the robot's path.
- **Construction management:** Buildings are enormously complicated artifacts, and keeping track of what is happening during construction is difficult and expensive. One way to keep track is to fly drones through the construction site once a week, filming the current state. Then build a 3D model of the current state and explore the difference between the plans and the reconstruction using visualization techniques. Figure 27.21 illustrates this application.

Figure 27.21 3D models of construction sites are produced from images by structure-from-motion and multiview stereo algorithms. They help construction companies to coordinate work on large buildings by comparing a 3D model of the actual construction to date with the building plans. **Left**: A visualization of a geometric model captured by drones. The reconstructed 3D points are rendered in color, so the result looks like progress to date (note the partially completed building with crane). The small pyramids show the pose of a drone when it captured an image, to allow visualization of the flight path. **Right**: These systems are actually used by construction teams; this team views the model of the as-built site, and compares it with building plans as part of the coordination meeting. Figure courtesy of Derek Hoiem, Mani Golparvar-Fard and Reconstruct, produced by a commercial system described in a blog post at medium.com/reconstruct-inc.

27.7.4 Geometry from a single view

Geometric representations are particularly useful if you want to move, because they can reveal where you are, where you can go, and what you are likely bump into. But it is not always convenient to use multiple views to produce a geometric model. For example, when you open the door and step into a room, your eyes are too close together to recover a good representation of the depth to distant objects across the room. You could move your head back and forth, but that is time-consuming and inconvenient.

An alternative is to predict a **depth map**—an array giving the depth to each pixel in the image, nominally from the camera—from a single image. For many kinds of scenes, this is surprisingly easy to do accurately, because the depth map has quite a simple structure. This is particularly true of rooms and indoor scenes in general. The mechanics are straightforward. One obtains a data set of images and depth maps, then trains a network to predict depth maps from images. A variety of interesting variations of the problem can be solved. The problem with a depth map is that it doesn't tell you anything about the backs of objects, or the space behind the objects. But there are methods that can predict what voxels (3D pixels) are occupied by known objects (the object geometry is known) and what a depth map would look like if an object were removed (and so where you could hide objects). These methods work because object shapes are quite strongly stylized.

As we saw in Section 27.6.4, recovering the pose of a known object using a 3D model is straightforward. Now imagine you see a single image of, say, a sparrow. If you have seen many images of sparrow-like birds in the past, you can reconstruct a reasonable estimate of both the pose of the sparrow and its geometric model from that single image. Using the past images you build a small, parametric family of geometric models for sparrow-like birds; then an optimization procedure is used to find the best set of parameters and viewpoints to explain the image that you see. This argument works to supply texture for that model, too, even for the parts you cannot see (Figure 27.22).

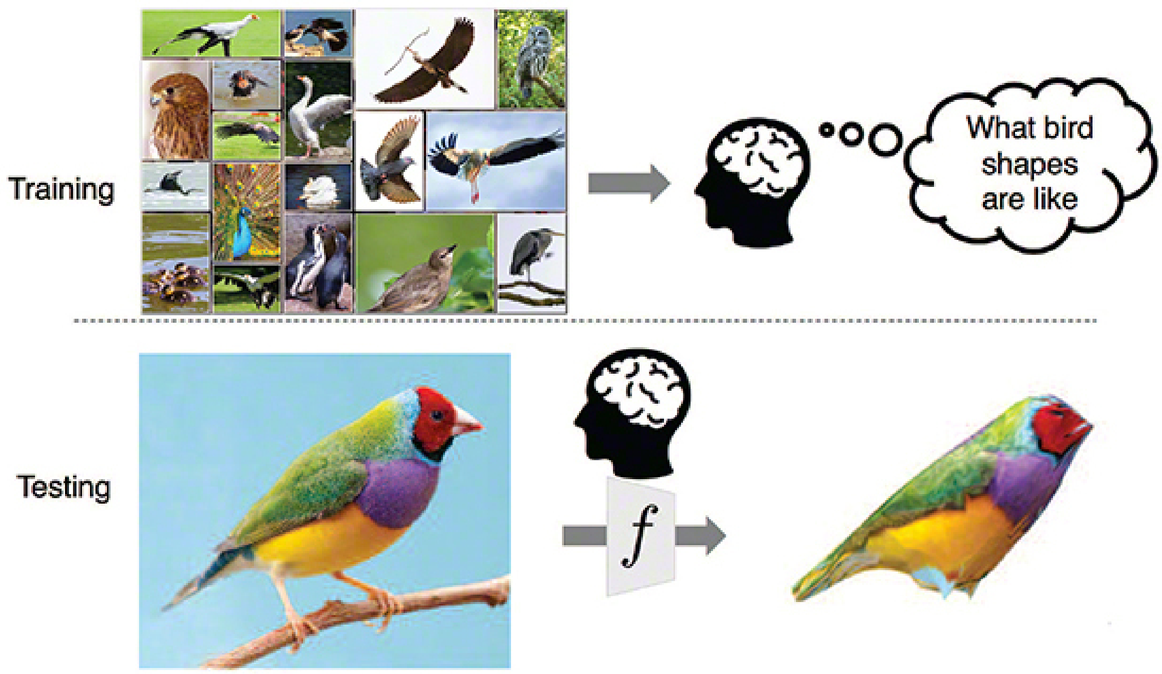


Figure 27.22 If you have seen many pictures of some category—say, birds (**top**)—you can use them to produce a 3D reconstruction from a single new view (**bottom**). You need to be sure that all objects have a fairly similar geometry (so a picture of an ostrich won't help if you're looking at a sparrow), but classification methods can sort this out. From the many images you can estimate how texture values in the image are distributed across the object, and thus complete the texture for parts of the bird you haven't seen yet (**bottom**). Figure courtesy of Angjoo

Kanazawa, produced by a system described in Kanazawa *et al.* (2018b). Top photo credit: Satori/123RF; Bottom left credit: Four Oaks/Shutterstock.

27.7.5 Making pictures

It is now common to insert computer graphics models into photographs in a convincing fashion, as in Figure 27.23, where a statue has been placed into a photo of a room. First estimate a depth map and albedo for the picture. Then estimate the lighting in the image by matching it to other images with known lighting. Place the object in the image's depth map, and render the resulting world with a physical rendering program—a standard tool in computer graphics. Finally, blend the modified image with the original image.

Figure 27.23 On the **left**, an image of a real scene. On the **right**, a computer graphics object has been inserted into the scene. You

can see that the light appears to be coming from the right direction, and that the object seems to cast appropriate shadows. The generated image is convincing even if there are small errors in the lighting and shadows, because people are not expert at identifying these errors. Figure courtesy of Kevin Karsch, produced by a system described in Karsch *et al.* (2011).

Neural networks can also be trained to do **image transformation**: mapping images from type X—for example, a blurry image; an aerial image of a town; or a drawing of a new product—to images of type Y—for example, a deblurred version of the image; a road map; or a product photograph. This is easiest when the training data consists of (X, Y) pairs of images—in Figure 27.24 each example pair has an aerial image and the corresponding road map section. The training loss compares the output of the network with the desired output, and also has a loss component from a generative adversarial network (GAN) that ensures that the output has the right kinds of features for images of type Y. As we see in the test portion of Figure 27.24, systems of this kind perform very well.

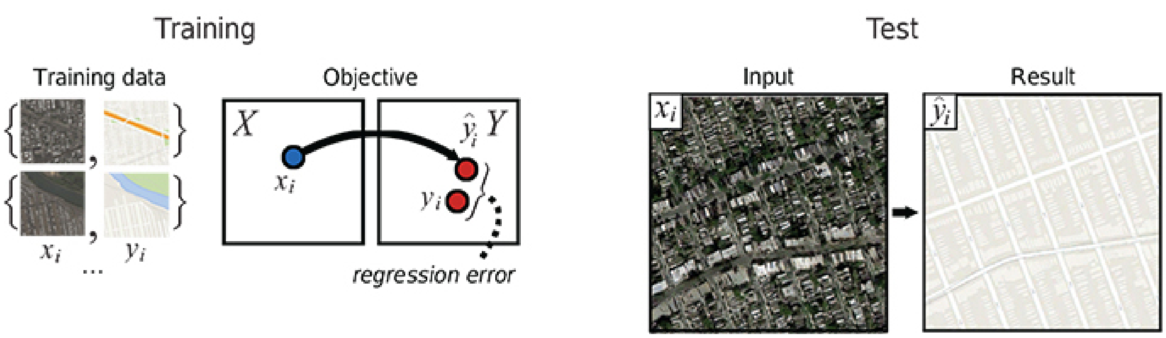


Figure 27.24 Paired image translation where the input consists of aerial images and the corresponding map tiles, and the goal is to train a network to produce a map tile from an aerial image. (The system can also learn to generate aerial images from map tiles.) The network is trained by comparing $\hat{y}_i$ (the output for example x_i of type X) to the right output y_i of type Y. Then at test time, the network must make new images of type Y from new inputs of type X. Figure courtesy of Phillip Isola, Jun-Yan Zhu and Alexei A. Efros, produced by a system described in Isola *et al.* (2017). Map data © 2019 Google.

Sometimes we don't have images that are paired with each other, but we do have a big collection of images of type X (say, pictures of horses) and a separate collection of type Y (say, pictures of zebras). Imagine an artist who is tasked with creating an image of a zebra running in a field. The artist would appreciate being able to select just the right image of a horse, and then having the computer automatically transform the horse into a zebra (Figure 27.25). To achieve this we can train two transformation networks, with an additional constraint called a cycle constraint. The first network maps horses to zebras; the second network maps zebras to horses; and the cycle constraint requires that when you map X to Y to X (or Y to X to Y), you get what you started with. Again, GAN losses ensure that the horse (or zebra) pictures that the networks output are "like" real horse (or zebra) pictures.

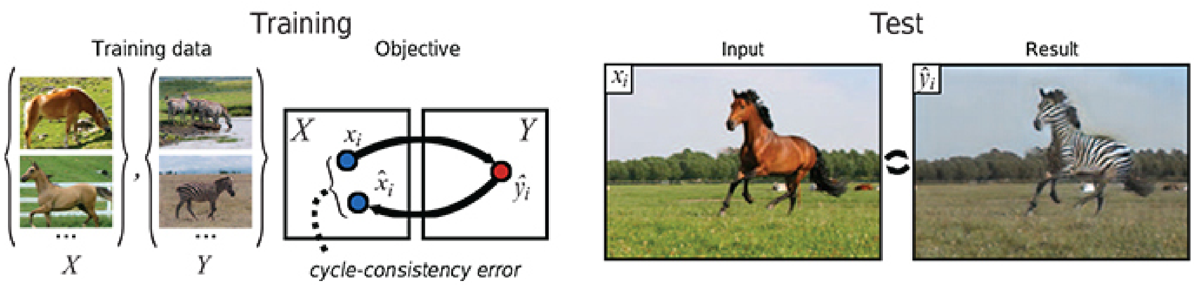


Figure 27.25 Unpaired image translation: given two populations of images (here type X is horses and type Y is zebras), but no corresponding pairs, learn to translate a horse into a zebra. The method trains two predictors: one that maps type X to type Y, and another that maps type Y to type X. If the first network maps a horse x_i to a zebra $\hat{y}_i$, the second network should map $_i$ *back to the original* x_i. The difference between x_i and $\widehat{x}_i$ is what trains the two networks. The cycle from Y to X and back must be closed. Such networks can successfully impose rich transformations on images. Figure courtesy of Alexei A. Efros; see Zhu *et al.* (2017). Running horse photo by Justyna Furmanczyk Gibaszek/Shutterstock.

Another artistic effect is called **style transfer**: the input consists of two images—the *content* (for example, a photograph of a cat); and the *style* (for example, an abstract painting). The output is a version of the cat rendered in the abstract style (see Figure 27.26). The key insight to solving this problem is that if we examine a deep convolutional neural network (CNN) that has been trained to do object recognition (say, on ImageNet), we find that the early layers tend to represent the style of a picture, and the late layers represent the content. Let p be the content image and s be the style image, and let $E(x)$ be the vector of activations of an early layer on image x and $L(x)$ be the vector of activations of a late layer on image x. Then we want to generate some image x that has similar content to the house photo, that is, minimizes $|L(x) - L(p)|$, and also has similar style to the impressionist painting, that is, minimizes $|E(x) - E(s)|$. We use gradient descent with a loss function that is a linear combination of these two factors to find an image x that minimizes the loss.

Figure 27.26 Style transfer: The *content* of a photo of a cat is combined with the *style* of an abstract painting to yield a new image of the cat rendered in the abstract style (right). The painting is Wassily Kandinsky's *Lyrisches* or *The Lyrical* (public domain); the cat is Cosmo.

Generative adversarial networks (GANs) can create novel photorealistic images, fooling most people most of the time. One kind of image is the **deepfake**—an image or video that looks like a particular person, but is generated from a model. For example, when Carrie Fisher was 60, a generated replica of her 19-year-old face was superimposed on another actor's body for the making of *Rogue One*. The movie industry creates ever-better deepfakes for artistic purposes, and researchers work on countermeasures for detecting deepfakes, to mitigate the destructive effects of fake news.

Generated images can also be used to maintain privacy. For example, there are image data sets in radiological practices that would be useful for researchers, but can't be published because of patient confidentiality. Generative image models can take a private data set of images and produce a synthetic data set that can be shared with researchers. This data set should be (a) like the training data set; (b) different; and (c) controllable. Consider chest X-rays. The synthetic data set should be like the training data set in the sense that each image individually would fool a radiologist and the frequencies of each effect should be right, so a radiologist would not be surprised by how often (say) pneumonia appears. The new data set should be different, in the sense that it does not reveal personally identifiable information. The new data set should be controllable, so that the frequencies of effects can be adjusted to reflect the communities of interest. For example, pneumonias are more common in the elderly than in young adults. Each of these goals is technically difficult to reach, but image data sets have been created that fool practicing radiologists some of the time (Figure 27.27).

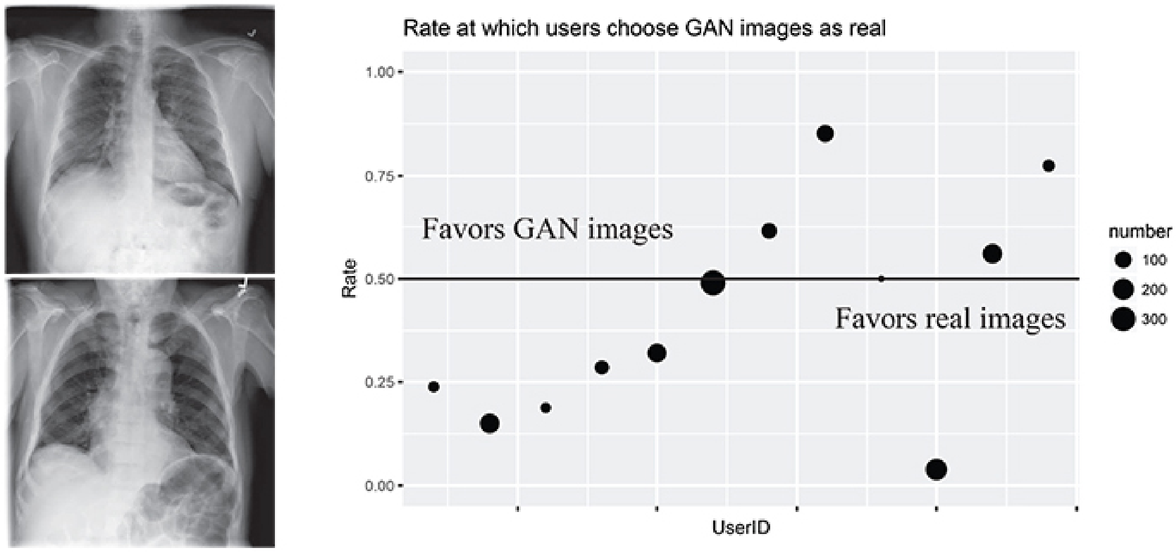


Figure 27.27 GAN generated images of lung X-rays. On the left, a pair consisting of a real X-ray and a GAN-generated X-ray. On the right, results of a test asking radiologists, given a pair of X-rays as seen on the left, to tell which is the real X-ray. On average, they chose correctly 61% of the time, somewhat better than chance. But they differed in their accuracy—the chart on the right shows the error rate for 12 different radiologists; one of them had an error rate near 0% and another had 80% errors. The size of each dot indicates the number of images each radiologist viewed. Figure courtesy of Alex Schwing, produced by a system described in Deshpande *et al.* (2019).

27.7.6 Controlling movement with vision

One of the principal uses of vision is to provide information both for manipulating objects—picking them up, grasping them, twirling them, and so on—and for navigating while avoiding obstacles. The ability to use vision for these purposes is present in the most primitive of animal visual systems. In many cases, the visual system is minimal, in the sense that it extracts from the available light field just the information the animal needs to inform its behavior. Quite probably, modern vision systems evolved from early, primitive organisms that used a photosensitive spot at one end in order to orient themselves toward (or away from) the light. We saw in Section 27.6 that flies use a very simple optical flow detection system to land on walls.

Suppose that, rather than landing on walls, we want to build a self-driving car. This is a project that places much greater demands on the perceptual system. Perception in a selfdriving car has to support the following tasks:

- **Lateral control** : Ensure that the vehicle remains securely within its lane or changes lanes smoothly when required.
- **Longitudinal control** : Ensure that there is a safe distance to the vehicle in front.
- **Obstacle avoidance** : Monitor vehicles in neighboring lanes and be prepared for evasive maneuvers. Detect pedestrians and allow them to cross safely.
- **Obey traffic signals** : These include traffic lights, stop signs, speed limit signs, and police hand signals.

The problem for a driver (human or computer) is to generate appropriate steering, acceleration, and braking actions to best accomplish these tasks.

To make good decisions, the driver should construct a model of the world and the objects in it. Figure 27.28 shows some of the visual inferences that are necessary to build this model. For lateral control, the driver needs to maintain a representation of the position and orientation of the car relative to the lane. For longitudinal control, the driver needs to keep a safe distance from the vehicle in front (which may not be easy to identify on, say, curving multilane roads). Obstacle avoidance and following traffic signals require additional inferences.

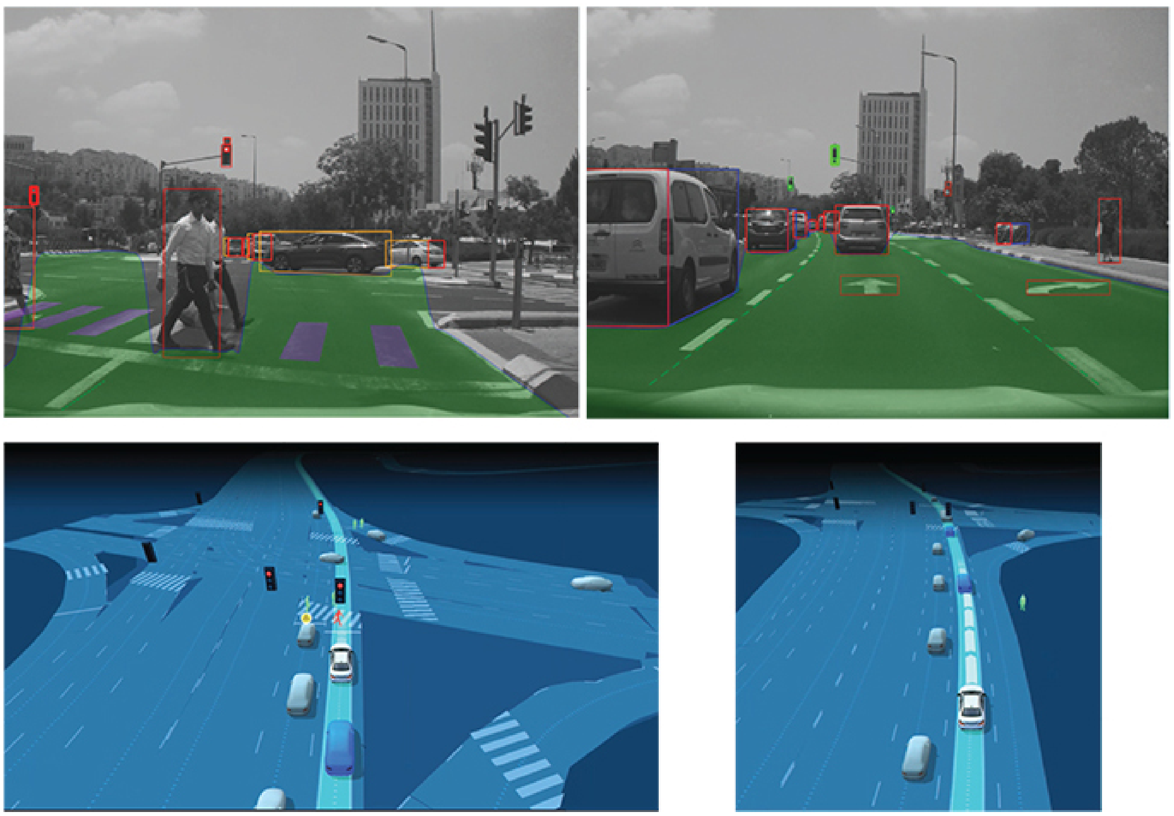


Figure 27.28 Mobileye's camera-based sensing for autonomous vehicles. **Top row**: Two images from a front-facing camera, taken a few seconds apart. The green area is the free space—the area to which the vehicle could physically move in the immediate future. Objects are displayed with 3D bounding boxes defining their sides (red for the rear, blue for the right side, yellow for the left side, and green for the front). Objects include vehicles,

pedestrians, the inner edge of the self-lane marks (necessary for lateral control), other painted road and crosswalk marks, traffic signs, and traffic lights. Not shown are animals, poles and cones, sidewalks, railings, and general objects (e.g., a couch that fell from the back of a truck). Each object is then marked with a 3D position and velocity. **Bottom row**: A full physical model of the environment, rendered from the detected objects. (Images show Mobileye's vision-only system results). Images courtesy of Mobileye.

Roads were designed for humans who navigate using vision, so it should in principle be possible to drive using vision alone. However, in practice, commercial self-driving cars use a variety of sensors, including cameras, lidars, radars, and microphones. A lidar or radar enables direct measurement of depth, which can be more accurate than the vision-only methods of Section 27.6. Having multiple sensors increases performance in general, and is particularly important in conditions of poor visibility; for example, radar can cut through fog that blocks cameras and lidars. Microphones can detect approaching vehicles (especially ones with sirens) before they become visible.

There has also been much research on mobile robots navigating in indoor and outdoor environments. Applications abound, such as the last mile of package or pizza delivery. Traditional approaches break this task up into two stages as shown in Figure 27.29:

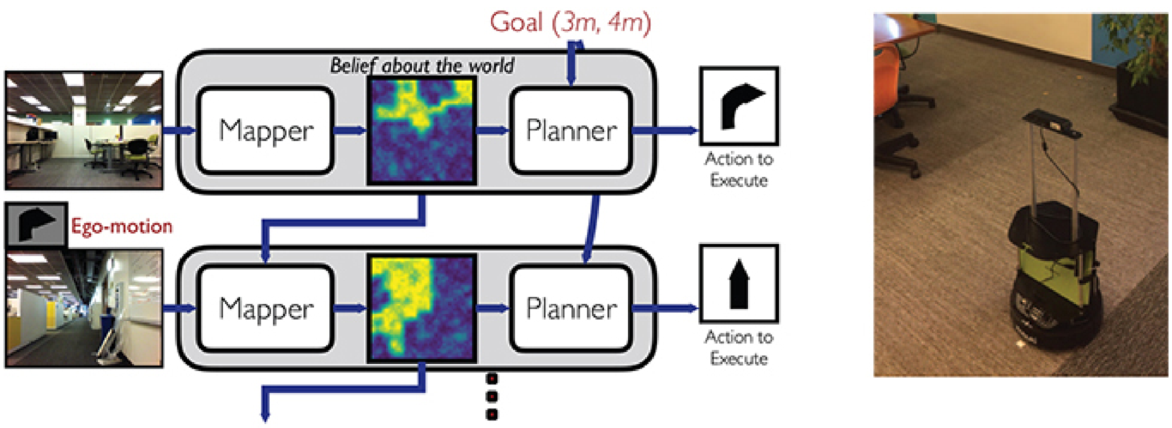


Figure 27.29 Navigation is tackled by decomposition into two problems: mapping and planning. With each successive time step, information from sensors is used to incrementally build an uncertain model of the world. This model along with the goal specification is passed to a planner that outputs the next action that the robot should take in order to achieve the goal. Models of the world can be purely geometric (as in classical SLAM), or semantic (as obtained via learning), or even topological (based on landmarks). The actual robot appears on the right. Figures courtesy of Saurabh Gupta.

- **Map building** : Simultaneous Localization and Mapping or SLAM (see page 942) is the task of constructing a 3D model of the world, including the location of the robot in the world (or more specifically, the location of each of the robot's cameras). This model (typically represented as a point cloud of obstacles) can be built from a series of images from different camera positions.

- **Path planning**: Once the robot has access to this 3D map and can localize itself in it, the objective becomes one of finding a collision-free trajectory from the current position to the goal location (see Section 26.6).

Many variants of this general approach have been explored. For instance, in the cognitive mapping and planning approach, the two stages of map building and path planning are two modules in a neural network that is trained end-to-end to minimize a loss function. Such a system does not have to build a complete map—which is often redundant and unnecessary—if all you need is enough information to navigate from point A to point B without colliding with obstacles.

Summary

Although perception appears to be an effortless activity for humans, it requires a significant amount of sophisticated computation. The goal of vision is to extract information needed for tasks such as manipulation, navigation, and object recognition.

- The geometry and optics of image formation is well understood. Given a description of a 3D scene, we can easily produce a picture of it from some arbitrary camera position—this is the graphics problem. The inverse problem, the computer vision problem—taking a picture and turning it into a 3D description—is more difficult.
- Representations of images capture edges, texture, optical flow, and regions. These yield cues to the boundaries of objects and to correspondence between images.
- Convolutional neural networks produce accurate image classifiers that use learned features. Rather roughly, the features are patterns of patterns of patterns... It is hard to predict when these classifiers will work well, because the test data may be unlike the training data in some important way. Experience teaches that they are often accurate enough to use in practice.
- Image classifiers can be turned into object detectors. One classifier scores boxes in an image for objectness; another then decides whether an object is in the box, and what object it is. Object detection methods aren't perfect, but are usable for a wide variety of applications.
- With more than one view of a scene, it is possible to recover the 3D structure of the scene and the relationship between views. In many cases, it is possible to recover 3D geometry from a single view.

- The methods of computer vision are being very widely applied.

Bibliographical and Historical Notes

This chapter has concentrated on vision, but other perceptual channels have been studied and put to use in robotics. For auditory perception (hearing), we have already covered speech recognition, and there has also been considerable work on music perception (Koelsch and Siebel, 2005) and machine learning of music (Engel *et al.*, 2017) as well as on machine learning for sounds in general (Sharan and Moir, 2016).

Tactile perception or touch (Luo *et al.*, 2017) is important in robotics and is discussed in Chapter 26. Automated olfactory perception (smell) has seen less work, but it has been shown that deep learning models can learn to predict smells based on the structure of molecules (Sanchez-Lengeling *et al.*, 2019).

Systematic attempts to understand human vision can be traced back to ancient times. Euclid (ca. 300 BCE) wrote about natural perspective—the mapping that associates, with each point P in the three-dimensional world, the direction of the ray OP joining the center of projection O to the point P. He was well aware of the notion of motion parallax. Ancient Roman paintings, such as the ones perserved by the eruption of Vesuvius in 79 CE, used an informal kind of perspective, with more than one horizon line.

The mathematical understanding of perspective projection, this time in the context of projection onto planar surfaces, had its next significant advance in the 15th century in Renaissance Italy. Brunelleschi is usually credited with creating the first paintings based on geometrically correct projection of a three-dimensional scene in about 1413. In 1435, Alberti codified the rules and inspired generations of artists. Particularly notable in their development of the science of perspective, as it was called in those days, were Leonardo da Vinci and Albrecht Durer. Leonardo's late 15th-century descriptions of the interplay of light and shade (chiaroscuro), umbra and penumbra regions of shadows, and aerial perspective are still worth reading in translation (Kemp, 1989).

Although perspective was known to the Greeks, they were curiously confused by the role of the eyes in vision. Aristotle thought of the eyes as devices emitting rays, rather in the manner of modern laser range finders. This mistaken view was laid to rest by the work of Arab scientists, such as Alhazen, in the 10th century.

The development of various kinds of cameras followed. These consisted of rooms (*camera* is Latin for "chamber") where light would be let in through a small hole in one wall to cast an image of the scene outside on the opposite wall. Of course, in all these cameras, the image was inverted, which caused no end of confusion. If the eye was to be thought of as such an imaging device, how do we see right side up? This enigma exercised the greatest minds of the era (including Leonardo). It took the work of Kepler and Descartes to settle the question. Descartes placed an eye from which the opaque cuticle had been removed in a hole in a window shutter. The result was an inverted image formed on a piece of paper laid out on the retina. Although the retinal image is indeed inverted, this does not cause a problem because the brain interprets the image the right way. In modern jargon, one just has to access the data structure appropriately.

The next major advances in the understanding of vision took place in the 19th century. The work of Helmholtz and Wundt, described in Chapter 1, established psychophysical experimentation as a rigorous scientific discipline. Through the work of Young, Maxwell, and Helmholtz, a trichromatic theory of color vision was established. The fact that humans can see depth if the images presented to the left and right eyes are slightly different was demonstrated by Wheatstone’s (1838) invention of the stereoscope. The device immediately became popular in parlors and salons throughout Europe.

The essential concept of binocular stereopsis—that two images of a scene taken from slightly different viewpoints carry information sufficient to obtain a three-dimensional reconstruction of the scene—was exploited in the field of photogrammetry. Key mathematical results were obtained; for example, Kruppa (1913) proved that, given two views of five distinct points in a scene, one could reconstruct the rotation and translation between the two camera positions as well as the depth of the scene (up to a scale factor).

Although the geometry of stereopsis had been understood for a long time, the correspondence problem in photogrammetry used to be solved by humans trying to match up corresponding points. The amazing ability of humans in solving the correspondence problem was illustrated by Julesz’s (1971) random dot stereograms. The field of computer vision has devoted much effort towards an automatic solution of the correspondence problem.

In the first half of the 20th century, the most significant research results in vision were obtained by the Gestalt school of psychology, led by Max Wertheimer. They pointed out the importance of perceptual organization: for a human, the image is not a collection of pointillist photoreceptor outputs (pixels), rather it is organized into coherent groups. The computer vision task of finding regions and curves traces back to this insight. The Gestaltists also drew attention to the “figure-ground” phenomenon—a contour separating two image regions that in the world are at different depths appears to belong only to the nearer region, the “figure,” and not to the farther region, the “ground.”

The gestalt work was carried on by J. J. Gibson (1950,1979), who pointed out the importance of optical flow and texture gradients in the estimation of environmental variables such as surface slant and tilt. He reemphasized the importance of the stimulus and how rich it was. Gibson, Olum, and Rosenblatt (1955) pointed out that the optical flow field contained enough information to determine the motion of the observer relative to the environment. Gibson particularly emphasized the role of the active observer, whose self-directed movement facilitates the pickup of information about the external environment.

Computer vision dates back to the 1960s. Roberts’s (1963) thesis at MIT on perceiving cubes and other blocks-world objects was one of the earliest publications in the field. Roberts introduced several key ideas, including edge detection and model-based matching.

In the 1960s and 1970s progress was slow, hampered by the lack of computational and storage resources. Low-level visual processing received a lot of attention, with techniques drawn from related fields such as signal processing, pattern recognition, and data clustering.

Edge detection was treated as an essential first step in image processing, as it reduced the amount of data to be processed. The widely used Canny edge detection technique was introduced by John Canny (1986). Martin, Fowlkes, and Malik (2004) showed how to combine multiple clues, such as brightness, texture and color, in a machine learning framework to better find boundary curves.

The closely related problem of finding regions of coherent brightness, color, and texture naturally lends itself to formulations where finding the best partition becomes an optimization problem. Three leading examples are based on Markov Random Fields due to Geman and Geman (1984), the variational formulation of Mumford and Shah (1989), and normalized cuts by Shi and Malik (2000).

Through much of the 1960s, 1970s, and 1980s, there were two distinct paradigms in which visual recognition was pursued, dictated by different perspectives on what was perceived to be the primary problem. Computer vision research on object recognition largely focused on issues arising from the projection of three-dimensional objects onto two-dimensional images. The idea of alignment, also first introduced by Roberts, resurfaced in the 1980s in the work of Lowe (1987) and Huttenlocher and Ullman (1990).

The pattern recognition community took a different approach, viewing the 3D–to–2D aspects of the problem as insignificant. Their motivating examples were in domains such as optical character recognition and handwritten zip code recognition, in which the primary concern is that of learning the typical variations characteristic of a class of objects and separating them from other classes. We can trace neural net architectures for image analysis back to Hubel and Wiesel's (1962, 1968) studies of the visual cortex in cats and monkeys. They developed a hierarchical model of the visual pathway with neurons in lower areas of the brain (especially the area called V1) responding to features such as oriented edges and bars, and neurons in higher areas responding to more specific stimuli ("grandmother cells" in the cartoon version).

Fukushima (1980) proposed a neural network architecture for pattern recognition explicitly motivated by Hubel and Wiesel's hierarchy. His model had alternating layers of simple cells and complex cells, thus incorporating downsampling, and also had shift invariance, thus incorporating convolutional structure. LeCun *et al.* (1989) took the additional step of using back-propagation to train the weights of this network, and what we today call convolutional neural networks were born. See LeCun *et al*. (1995) for a comparison of approaches.

Starting in the late 1990s, accompanying a much greater role of probabilistic modeling and statistical machine learning in the field of artificial intelligence in general, there was a rapprochement between these two traditions. Two lines of work contributed significantly. One was research on face detection (Rowley *et al*., 1998; Viola and Jones, 2004) that demonstrated the power of pattern recognition techniques on clearly important and useful tasks.

The other was the development of point descriptors, which enable the construction of feature vectors from parts of objects (Schmid and Mohr, 1996). There are three key strategies to build a good local point descriptor: one uses orientations to get illumination invariance; one needs to describe image structure close to a point in detail, and further away only roughly; and one needs to use spatial histograms to suppress variations caused by small errors in locating the point. Lowe's (2004) SIFT descriptor exploited these ideas very effectively; another popular variant was the HOG descriptor due to Dalal and Triggs (2005).

The 1990s and 2000s saw a continuing debate between the devotees of clever feature design such as SIFT and HOG and the aficionados of neural networks who believed that good features should emerge automatically from end-to-end training. The way to settle such a debate is through benchmarks on standard data sets, and in the 2000s results on a standard object detection data set, PASCAL VOC, argued in favor of hand-designed features. This changed when Krizhevsky *et al*. (2013) showed that on the task of image classification on the ImageNet data set, their neural network (called AlexNet) gave significantly lower error rates than the mainstream computer vision techniques.

What was the secret sauce behind the success of AlexNet? Besides the technical innovations (such as the use of ReLU activation units) we must give a lot of credit to **big data** and **big computation**. By big data we mean the availability of large data sets with category labels, such as ImageNet, which provided the training data for these large, deep networks with millions of parameters. Previous data sets like Caltech-101 or PASCAL VOC didn't have enough training data, and MNIST and CIFAR were regarded as "toy data sets" by the computer vision community. This strand of labeling data sets for benchmarking and for extracting image statistics itself was enabled by the desire of people to upload their photo collections to the Internet on sites such as Flickr. The way big computation proved most helpful was through GPUs, a hardware development initially driven by the needs of the video game industry.

Within a year or two, the evidence was quite clear. For example, the region-based convolutional neural network (RCNN) work of Girshick *et al.* (2016) showed that the AlexNet architecture could be modified, by making use of computer vision ideas such as region proposals, to make possible state-of-the-art object detection on PASCAL VOC. We have realized that generally deeper networks work better and that overfitting fears are overblown. We have new techniques such as **batch normalization** to deal with regularization.

The reconstruction of three-dimensional structure from multiple views has its roots in the photogrammetry literature. In the computer vision era, Ullman (1979), and Longuet-Higgins (1981) are influential early works. Concerns about the stability of structure from motion were significantly allayed by the work of Tomasi and Kanade (1992) who showed that with the use of multiple frames, and the resulting wide baseline, shape could be recovered quite accurately.

A conceptual innovation introduced in the 1990s was the study of projective structure from motion. Here camera calibration is not necessary, as was shown by Faugeras (1992). This discovery is related to the introduction of the use of geometrical invariants in object recognition, as surveyed by Mundy and Zisserman (1992), and the development of affine structure from motion by Koenderink and Van Doorn (1991).

In the 1990s, with great increase in computer speed and storage and the widespread availability of digital video, motion analysis found many new applications. Building geometrical models of real-world scenes for rendering by computer graphics techniques proved particularly popular, led by reconstruction algorithms such as the one developed by Debevec *et al.* (1996). The books by Hartley and Zisserman (2000) and Faugeras *et al.* (2001) provide a comprehensive treatment of the geometry of multiple views.

Humans can perceive shape and spatial layout from a single image, and modeling this has proved to be quite a challenge for computer vision researchers. Inferring shape from shading was first studied by Berthold Horn (1970), and Horn and Brooks (1989) present an extensive survey of the main papers from a period when this was a much studied problem. Gibson (1950) was the first to propose texture gradients as a cue to shape. The mathematics of occluding contours, and more generally understanding the visual events in the projection of smooth curved objects, owes much to the work of Koenderink and van Doorn, which finds an extensive treatment in Koenderink’s (1990) *Solid Shape.*

More recently, attention has turned to treating the problem of shape and surface recovery from a single image as a probabilistic inference problem, where geometrical cues are not modeled explicitly, but used implicitly in a learning framework. A good example is the work of Hoiem *et al.* (2007); recently this has been reworked using deep neural networks.

Turning now to the applications of computer vision for guiding action, Dickmanns and Zapp (1987) first demonstrated a self-driving car driving on freeways at high speeds; Pomerleau (1993) achieved similar performance using a neural network approach. Today building self-driving cars is a big business, with the established car companies competing with new entrants such as Baidu, Cruise, Didi, Google Waymo, Lyft, Mobileye, Nuro, Nvidia, Samsung, Tata, Tesla, Uber, and Voyage to market systems that provide capabilities ranging from driver assistance to full autonomy.

For the reader interested in human vision, *Vision Science: Photons to Phenomenology* by Stephen Palmer (1999) provides the best comprehensive treatment; *Visual Perception: Physiology, Psychology and Ecology* by Vicki Bruce, Patrick Green, and Mark Georgeson (2003) is a shorter textbook. The books *Eye*, *Brain and Vision* by David Hubel (1988) and *Perception* by Irvin Rock (1984) are friendly introductions centered on neurophysiology and perception respectively. David Marr's book *Vision* (Marr, 1982) played a historical role in connecting computer vision to the traditional areas of biological vision—psychophysics and neurobiology. While many of his specific models for tasks such as edge detection and object recognition haven't stood the test of time, the theoretical perspective where each task is analyzed at an informational, computational, and implementation level is still illuminating.

For the field of computer vision, the most comprehensive textbooks available today are *Computer Vision: A Modern Approach* (Forsyth and Ponce, 2002) and Computer Vision: Algorithms and Applications (Szeliski, 2011). Geometrical problems in computer vision are treated thoroughly in *Multiple View Geometry in Computer Vision* (Hartley and Zisserman, 2000). These books were written before the deep learning revolution, so for the latest results, consult the primary literature.

Two of the main journals for computer vision are the IEEE *Transactions on Pattern Analysis and Machine Intelligence* and the *International Journal of Computer Vision*. Computer vision conferences include ICCV (International Conference on Computer Vision), CVPR (Computer Vision and Pattern Recognition), and ECCV (European Conference on Computer Vision). Research with a significant machine learning component is also published at NeurIPS (Neural Information Processing Systems), and work on the interface with computer graphics often appears at the ACM SIGGRAPH (Special Interest Group in Graphics) conference. Many vision papers appear as preprints on the arXiv server, and early reports of new results appear in blogs from the major research labs.

[1] We will use the term “box” to mean any axis-aligned rectangular region of the image, and the term “window” mostly as a synonym for “box,” but with the connotation that we have a window onto the input where we are hoping to see something, and a bounding box in the output when we have found it.

CHAPTER 28

PHILOSOPHY, ETHICS, AND SAFETY OF AI

In which we consider the big questions around the meaning of AI, how we can ethically develop and apply it, and how we can keep it safe.

Philosophers have been asking big questions for a long time: How do minds work? Is it possible for machines to act intelligently in the way that people do? Would such machines have real, conscious minds?

To these, we add new ones: What are the ethical implications of intelligent machines in day-to-day use? Should machines be allowed to decide to kill humans? Can algorithms be fair and unbiased? What will humans do if machines can do all kinds of work? And how do we control machines that may become more intelligent than us?

28.1 The Limits of AI

In 1980, philosopher John Searle introduced a distinction between **weak AI**—the idea that machines could act *as if* they were intelligent—and **strong AI**—the assertion that machines that do so are *actually* consciously thinking (not just *simulating* thinking). Over time the definition of strong AI shifted to refer to what is also called "human-level AI" or "general AI"—programs that can solve an arbitrarily wide variety of tasks, including novel ones, and do so as well as a human.

Critics of weak AI who objected to the very possibility of intelligent behavior in machines now appear as shortsighted as Simon Newcomb, who in October 1903 wrote "aerial flight is one of the great class of problems with which man can never cope"—just two months before the Wright brothers' flight at Kitty Hawk. The rapid progress of recent years does not, however, prove that there can be no limits to what AI can achieve. Alan Turing (1950), the first person to define AI, was also the first to raise possible objections to AI, foreseeing almost all the ones subsequently raised by others.

28.1.1 The argument from informality

Turing's "argument from informality of behavior" says that human behavior is far too complex to be captured by any formal set of rules—humans must be using some informal guidelines that (the argument claims) could never be captured in a formal set of rules and thus could never be codified in a computer program.

A key proponent of this view was Hubert Dreyfus, who produced a series of influential critiques of artificial intelligence: *What Computers*

Can't Do (1972), the sequel *What Computers Still Can't Do* (1992), and, with his brother Stuart, *Mind Over Machine* (1986). Similarly, philosopher Kenneth Sayre (1993) said "Artificial intelligence *pursued within the cult of computationalism* stands not even a ghost of a chance of producing durable results." The technology they criticized came to be called **Good Old-Fashioned AI (GOFAI)**.

GOFAI corresponds to the simplest logical agent design described in Chapter 7, and we saw there that it is indeed difficult to capture every contingency of appropriate behavior in a set of necessary and sufficient logical rules; we called that the **qualification problem**. But as we saw in Chapter 12, probabilistic reasoning systems are more appropriate for open-ended domains, and as we saw in Chapter 22, deep learning systems do well on a variety of "informal" tasks. Thus, the critique is not addressed against computers *per se*, but rather against one particular style of programming them with logical rules—a style that was popular in the 1980s but has been eclipsed by new approaches.

One of Dreyfus's strongest arguments is for situated agents rather than disembodied logical inference engines. An agent whose understanding of "dog" comes only from a limited set of logical sentences such as "$Dog(x) \Rightarrow Mammal(x)$" is at a disadvantage compared to an agent that has watched dogs run, has played fetch with them, and has been licked by one. As philosopher Andy Clark (1998) says, "Biological brains are first and foremost the control systems for biological bodies. Biological bodies move and act in rich real-world surroundings." According to Clark, we are "good at frisbee, bad at logic."

The **embodied cognition** approach claims that it makes no sense to consider the brain separately: cognition takes place within a body, which is embedded in an environment. We need to study the system as a whole; the brain's functioning exploits regularities in its environment, including the rest of its body. Under the embodied cognition approach, robotics, vision, and other sensors become central, not peripheral.

Overall, Dreyfus saw areas where AI did not have complete answers and said that AI is therefore impossible; we now see many of these same areas undergoing continued research and development leading to increased capability, not impossibility.

28.1.2 The argument from disability

The "argument from disability" makes the claim that "a machine can never do *X*." As examples of *X*, Turing lists the following:

> Be kind, resourceful, beautiful, friendly, have initiative, have a sense of humor, tell right from wrong, make mistakes, fall in love, enjoy strawberries and cream, make someone fall in love with it, learn from experience, use words properly, be the subject of its own thought, have as much diversity of behavior as man, do something really new.

In retrospect, some of these are rather easy—we're all familiar with computers that "make mistakes." Computers with metareasoning capabilities (Chapter 6) can examine their own computations, thus being the subject of their own reasoning. A century-old technology has the proven ability to "make someone fall in love with it"—the teddy bear. Computer chess expert David Levy predicts that by 2050 people will routinely fall in love with humanoid robots. As for a robot falling in love, that is a common theme in fiction,[1] but there has been only limited academic speculation on the subject (Kim *et al.*, 2007). Computers have done things that are "really new," making significant discoveries in astronomy, mathematics, chemistry, mineralogy, biology, computer science, and other fields, and creating new forms of art through style transfer (Gatys *et al.*, 2016). Overall, programs exceed human performance in some tasks and lag behind on others. The one thing that it is clear they can't do is be exactly human.

28.1.3 The mathematical objection

Turing (1936) and Gödel (1931) proved that certain mathematical questions are in principle unanswerable by particular formal systems. Gödel's incompleteness theorem (see Section 9.5) is the most famous example of this. Briefly, for any formal axiomatic framework F powerful enough to do arithmetic, it is possible to construct a so-called Gödel sentence $G(F)$ with the following properties:

- $G(F)$ is a sentence of F, but cannot be proved within F.
- If F is consistent, then $G(F)$ is true.

Philosophers such as J. R. Lucas (1961) have claimed that this theorem shows that machines are mentally inferior to humans, because machines are formal systems that are limited by the incompleteness theorem—they cannot establish the truth of their own Gödel sentence—while humans have no such limitation. This has caused a lot of controversy, spawning a vast literature, including two books by the mathematician/physicist Sir Roger Penrose (1989, 1994). Penrose repeats Lucas's claim with some fresh twists, such as the hypothesis that humans are different because their brains operate by quantum gravity—a theory that makes multiple false predictions about brain physiology.

We will examine three of the problems with Lucas's claim. First, an agent should not be ashamed that it cannot establish the truth of some sentence while other agents can. Consider the following sentence:

> Lucas cannot consistently assert that this sentence is true.

If Lucas asserted this sentence, then he would be contradicting himself, so therefore Lucas cannot consistently assert it, and hence it is true. We have thus demonstrated that there is a true sentence that Lucas cannot consistently assert while other people (and machines) can. But that does not make us think any less of Lucas.

Second, Gödel's incompleteness theorem and related results apply to *mathematics,* not to *computers.* No entity—human or machine—can prove things that are impossible to prove. Lucas and Penrose falsely assume that humans can somehow get around these limits, as when Lucas (1976) says "we must assume our own consistency, if thought is to be possible at all." But this is an unwarranted assumption: humans are notoriously inconsistent. This is certainly true for everyday reasoning, but it is also true for careful mathematical thought. A famous example is the four-color map problem. Alfred Kempe (1879) published a proof that was widely accepted for 11 years until Percy Heawood (1890) pointed out a flaw.

Third, Gödel's incompleteness theorem technically applies only to formal systems that are powerful enough to do arithmetic. This includes Turing machines, and Lucas's claim is in part based on the assertion that computers are equivalent to Turing machines. This is not quite true. Turing machines are infinite, whereas computers (and brains) are finite, and any computer can therefore be described as a (very large) system in propositional logic, which is not subject to Gödel's incompleteness theorem. Lucas assumes that humans can "change their minds" while computers cannot, but that is also false—a computer can retract a conclusion after new evidence or further deliberation; it can upgrade its hardware; and it can change its decision-making processes with machine learning or software rewriting.

28.1.4 Measuring AI

Alan Turing, in his famous paper “Computing Machinery and Intelligence” (1950), suggested that instead of asking whether machines can think, we should ask whether machines can pass a behavioral test, which has come to be called the **Turing test**. The test requires a program to have a conversation (via typed messages) with an interrogator for five minutes. The interrogator then has to guess if the conversation is with a program or a person; the program passes the test if it fools the interrogator 30% of the time. To Turing, the key point was not the exact details of the test, but instead the idea of measuring intelligence by performance on some kind of open-ended behavioral task, rather than by philosophical speculation.

Nevertheless, Turing conjectured that by the year 2000 a computer with a storage of a billion units could pass the test, but here we are on the other side of 2000, and we still can’t agree whether any program has passed. Many people have been fooled when they didn’t know they might be chatting with a computer. The ELIZA program and Internet chatbots such as MGONZ (Humphrys, 2008) and NATACHATA (Jonathan et al., 2009) fool their correspondents repeatedly, and the chatbot CYBERLOVER has attracted the attention of law enforcement because of its penchant for tricking fellow chatters into divulging enough personal information that their identity can be stolen.

In 2014, a chatbot called Eugene Goostman fooled 33% of the untrained amateur judges in a Turing test. The program claimed to be a boy from Ukraine with limited command of English; this helped explain its grammatical errors. Perhaps the Turing test is really a test of human gullibility. So far no well-trained judge has been fooled (Aaronson, 2014).

Turing test competitions have led to better chatbots, but have not been a focus of research within the AI community. Instead, AI researchers who crave competition are more likely to concentrate on playing chess or Go or

StarCraft II, or taking an 8th grade science exam, or identifying objects in images. In many of these competitions, programs have reached or surpassed human-level performance, but that doesn't mean the programs are human-like outside the specific task. The point is to improve basic science and technology and to provide useful tools, not to fool judges.

28.2 Can Machines Really Think?

Some philosophers claim that a machine that acts intelligently would not be *actually* thinking, but would be only a *simulation* of thinking. But most AI researchers are not concerned with the distinction, and the computer scientist Edsger Dijkstra (1984) said that “The question of whether *Machines Can Think* ... is about as relevant as the question of whether *Submarines Can Swim.”* The American Heritage Dictionary’s first definition of *swim* is “To move through water by means of the limbs, fins, or tail,” and most people agree that submarines, being limbless, cannot swim. The dictionary also defines *fly* as “To move through the air by means of wings or winglike parts,” and most people agree that airplanes, having winglike parts, can fly. However, neither the questions nor the answers have any relevance to the design or capabilities of airplanes and submarines; rather they are about word usage in English. (The fact that ships do *swim* (“*privet*”) in Russian amplifies this point.) English speakers have not yet settled on a precise definition for the word “think”—does it require “a brain” or just “brain-like parts?”

Again, the issue was addressed by Turing. He notes that we never have *any* direct evidence about the internal mental states of other humans—a kind of mental solipsism. Nevertheless, Turing says, “Instead of arguing continually over this point, it is usual to have the **polite convention** that everyone thinks.” Turing argues that we would also extend the polite convention to machines, if only we had experience with ones that act intelligently. However, now that we do have some experience, it seems that our willingness to ascribe sentience depends at least as much on humanoid appearance and voice as on pure intelligence.

28.2.1 The Chinese room

The philosopher John Searle rejects the polite convention. His famous **Chinese room** argument (Searle, 1990) goes as follows: Imagine a human, who understands only English, inside a room that contains a rule book, written in English, and various stacks of paper. Pieces of paper containing indecipherable symbols are slipped under the door to the room. The human follows the instructions in the rule book, finding symbols in the stacks, writing symbols on new pieces of paper, rearranging the stacks, and so on. Eventually, the instructions will cause one or more symbols to be transcribed onto a piece of paper that is passed back to the outside world. From the outside, we see a system that is taking input in the form of Chinese sentences and generating fluent, intelligent Chinese responses.

Searle then argues: it is given that the human does not understand Chinese. The rule book and the stacks of paper, being just pieces of paper, do not understand Chinese. Therefore, there is no understanding of Chinese. And Searle says that the Chinese room is doing the same thing that a computer would do, so therefore computers generate no understanding.

Searle (1980) is a proponent of **biological naturalism**, according to which mental states are high-level emergent features that are caused by low-level physical processes *in the neurons,* and it is the (unspecified) properties of the neurons that matter: according to Searle's biases, neurons have "it" and transistors do not. There have been many refutations of Searle's argument, but no consensus. His argument could equally well be used (perhaps by robots) to argue that a human cannot have true understanding; after all, a human is made out of cells, the cells do not understand, therefore there is no understanding. In fact, that is the plot of Terry Bisson's (1990) science fiction story *They're Made Out of Meat,* in which alien robots explore Earth and can’t believe that hunks of meat could possibly be sentient. How they can be remains a mystery.

28.2.2 Consciousness and qualia

Running through all the debates about strong AI is the issue of **consciousness**: awareness of the outside world, and of the self, and the subjective experience of living. The technical term for the intrinsic nature of experiences is **qualia** (from the Latin word meaning, roughly, “of what kind”). The big question is whether machines can have qualia. In the movie *2001*, when astronaut David Bowman is disconnecting the “cognitive circuits” of the HAL 9000 computer, it says *“I’m afraid, Dave. Dave, my mind is going. I can feel it.”* Does HAL actually have feelings (and deserve sympathy)? Or is the reply just an algorithmic response, no different from “Error 404: not found”?

There is a similar question for animals: pet owners are certain that their dog or cat has consciousness, but not all scientists agree. Crickets change their behavior based on temperature, but few people would say that crickets experience the *feeling* of being warm or cold.

One reason that the problem of consciousness is hard is that it remains ill-defined, even after centuries of debate. But help may be on the way. Recently philosophers have teamed with neuroscientists under the auspices of the Templeton Foundation to start a series of experiments that could resolve some of the issues. Advocates of two leading theories of consciousness (global workspace theory and integrated information theory) have agreed that the experiments could confirm one theory over the other—a rarity in philosophy.

Alan Turing (1950) concedes that the question of consciousness is a difficult one, but denies that it has much relevance to the practice of AI: “I do not wish to give the impression that I think there is no mystery about consciousness ... But I do not think these mysteries necessarily need to be solved before we can answer the question with which we are concerned in this paper.” We agree with Turing—we are interested in creating programs that behave intelligently. Individual aspects of consciousness—awareness, self-awareness, attention—can be programmed and can be part of an intelligent machine. The additional project of making a machine conscious in exactly the way humans are is not one that we are equipped to take on. We do agree that behaving intelligently will require some degree of *awareness*, which will differ from task to task, and that tasks involving interaction with humans will require a model of human subjective experience.

In the matter of modeling experience, humans have a clear advantage over machines, because they can use their own subjective apparatus to appreciate the subjective experience of others. For example, if you want to know *what it’s like* when someone hits their thumb with a hammer, you can hit your thumb with a hammer. Machines have no such capability—although unlike humans, they can run each other’s code.

28.3 The Ethics of AI

Given that AI is a powerful technology, we have a moral obligation to use it well, to promote the positive aspects and avoid or mitigate the negative ones.

The positive aspects are many. For example, AI can save lives through improved medical diagnosis, new medical discoveries, better prediction of extreme weather events, and safer driving with driver assistance and (eventually) self-driving technologies. There are also many opportunities to improve lives. Microsoft's AI for Humanitarian Action program applies AI to recovering from natural disasters, addressing the needs of children, protecting refugees, and promoting human rights. Google's AI for Social Good program supports work on rainforest protection, human rights jurisprudence, pollution monitoring, measurement of fossil fuel emissions, crisis counseling, news fact checking, suicide prevention, recycling, and other issues. The University of Chicago's Center for Data Science for Social Good applies machine learning to problems in criminal justice, economic development, education, public health, energy, and environment.

AI applications in crop management and food production help feed the world. Optimization of business processes using machine learning will make businesses more productive, increasing wealth and providing more employment. Automation can replace the tedious and dangerous tasks that many workers face, and free them to concentrate on more interesting aspects. People with disabilities will benefit from AI-based assistance in seeing, hearing, and mobility. Machine translation already allows people from different cultures to communicate. Software-based AI solutions have near zero marginal cost of production, and so have the potential to democratize access to advanced technology (even as other aspects of software have the potential to centralize power).

Despite these many positive aspects, we shouldn't ignore the negatives. Many new technologies have had unintended **negative side effects**: nuclear fission brought Chernobyl and the threat of global destruction; the internal combustion engine brought air pollution, global warming, and the paving of paradise. Other technologies can have negative effects even when used as intended, such as sarin gas, AR-15 rifles, and telephone solicitation. Automation will create wealth, but under current economic conditions much of that wealth will flow to the owners of the automated systems, leading to increased income inequality. This can be disruptive to a well-functioning society. In developing countries, the traditional path to growth through low-cost manufacturing for export may be cut off, as wealthy countries adopt fully automated manufacturing facilities on-shore. Our ethical and governance decisions will dictate the level of inequality that AI will engender.

All scientists and engineers face ethical considerations of what projects they should or should not take on, and how they can make sure the execution of the project is safe and beneficial. In 2010, the UK's Engineering and Physical Sciences Research Council held a meeting to develop a set of Principles of Robotics. In subsequent years other government agencies, nonprofit organizations, and companies created similar sets of principles. The gist is that every organization that creates AI technology, and everyone in the organization, has a responsibility to make sure the technology contributes to good, not harm. The most commonly-cited principles are:

Note that many of the principles, such as "ensure safety," have applicability to all software or hardware systems, not just AI systems. Several principles are worded in a vague way, making them difficult to measure or enforce. That is in part because AI is a big field with many subfields, each of which has a different set of historical norms and different relationships between the AI developers and the stakeholders. Mittelstadt (2019) suggests that the subfields should each develop more specific actionable guidelines and case precedents.

28.3.1 Lethal autonomous weapons

The UN defines a lethal autonomous weapon as one that locates, selects, and engages (i.e., kills) human targets without human supervision. Various weapons fulfill some of these criteria. For example, land mines have been used since the 17th century: they can select and engage targets in a limited sense according to the degree of pressure exerted or the quantity of metal present, but they cannot go out and locate targets by themselves. (Land mines are banned under the Ottawa Treaty.) Guided missiles, in use since the 1940s, can chase targets, but they have to be pointed in the right general direction by a human. Auto-firing radar-controlled guns have been used to defend naval ships since the 1970s; they are mainly intended to destroy incoming missiles, but they could also attack manned aircraft. Although the word "autonomous" is often used to describe unmanned air vehicles or **drones**, most such weapons are both remotely piloted and require human actuation of the lethal payload.

At the time of writing, several weapons systems seem to have crossed the line into full autonomy. For example Israel's Harop missile is a "loitering munition" with a ten-foot wingspan and a fifty-pound warhead. It searches for up to six hours in a given geographical region for any target that meets a given criterion and then destroys it. The criterion could be "emits a radar signal resembling antiaircraft radar" or "looks like a tank."

The Turkish manufacturer STM advertises its Kargu quadcopter—which carries up to 1.5kg of explosives—as capable of “Autonomous hit... targets selected on images ... tracking moving targets ... antipersonnel ... face recognition.”

Autonomous weapons have been called the “third revolution in warfare” after gunpowder and nuclear weapons. Their military potential is obvious. For example, few experts doubt that autonomous fighter aircraft would defeat any human pilot. Autonomous aircraft, tanks, and submarines can be cheaper, faster, more maneuverable, and have longer range than their manned counterparts.

Since 2014, the United Nations in Geneva has conducted regular discussions under the auspices of the Convention on Certain Conventional Weapons (CCW) on the question of whether to ban lethal autonomous weapons. At the time of writing, 30 nations, ranging in size from China to the Holy See, have declared their support for an international treaty, while other key countries—including Israel, Russia, South Korea, and the United States—are opposed to a ban.

The debate over autonomous weapons includes legal, ethical and practical aspects. The legal issues are governed primarily by the CCW, which requires the possibility of discriminating between combatants and non-combatants, the judgment of military necessity for an attack, and the assessment of proportionality between the military value of a target and the possibility of collateral damage. The feasibility of meeting these criteria is an engineering question—one whose answer will undoubtedly change over time. At present, discrimination seems feasible in some circumstances and will undoubtedly improve rapidly, but necessity and proportionality are not presently feasible: they require that machines make subjective and situational judgments that are considerably more difficult than the relatively simple tasks of searching for and engaging potential targets. For these reasons, it would be legal to use autonomous weapons only in circumstances where a human operator can reasonably predict that the execution of the mission will not result in civilians being targeted or the weapons conducting unnecessary or disproportionate attacks. This means that, for the time being, only very restricted missions could be undertaken by autonomous weapons.

On the ethical side, some find it simply morally unacceptable to delegate the decision to kill humans to a machine. For example, Germany's ambassador in Geneva has stated that it "will not accept that the decision over life and death is taken solely by an autonomous system" while Japan "has no plan to develop robots with humans out of the loop, which may be capable of committing murder." Gen. Paul Selva, at the time the second-ranking military officer in the United States, said in 2017, "I don't think it's reasonable for us to put robots in charge of whether or not we take a human life." Finally, António Guterres, the head of the United Nations, stated in 2019 that "machines with the power and discretion to take lives without human involvement are politically unacceptable, morally repugnant and should be prohibited by international law."

More than 140 NGOs in over 60 countries are part of the Campaign to Stop Killer Robots, and an open letter organized in 2015 by the Future of Life Institute organized an open letter was signed by over 4,000 AI researchers[2] and 22,000 others.

Against this, it can be argued that as technology improves it ought to be possible to develop weapons that are less likely than human soldiers or pilots to cause civilian casualties. (There is also the important benefit that autonomous weapons reduce the need for human soldiers and pilots to risk death.) Autonomous systems will not succumb to fatigue, frustration, hysteria, fear, anger, or revenge, and need not “shoot first, ask questions later” (Arkin, 2015). Just as guided munitions have reduced collateral damage compared to unguided bombs, one may expect intelligent weapons to further improve the precision of attacks. (Against this, see Benjamin (2013) for an analysis of drone warfare casualties.) This, apparently, is the position of the United States in the latest round of negotiations in Geneva.

Perhaps counterintuitively, the United States is also one of the few nations whose own policies currently preclude the use of autonomous weapons. The 2011 U.S. Department of Defense (DOD) roadmap says: “For the foreseeable future, decisions over the use of force [by autonomous systems] and the choice of which individual targets to engage with lethal force will be retained under human control.” The primary reason for this policy is practical: autonomous systems are not reliable enough to be trusted with military decisions.

The issue of reliability came to the fore on September 26, 1983, when Soviet missile officer Stanislav Petrov’s computer display flashed an alert of an incoming missile attack. According to protocol, Petrov should have initiated a nuclear counterattack, but he suspected the alert was a bug and treated it as such. He was correct, and World War III was (narrowly) averted. We don’t know what would have happened if there had been no human in the loop.

Reliability is a very serious concern for military commanders, who know well the complexity of battlefield situations. Machine learning systems that operate flawlessly in training may perform poorly when deployed. Cyberattacks against autonomous weapons could result in friendly-fire casualties; disconnecting the weapon from all communication may prevent that (assuming it has not already been compromised), but then the weapon cannot be recalled if it is malfunctioning.

The overriding practical issue with autonomous weapons is that they they are scalable weapons of mass destruction, in the sense that the scale of an attack that can be launched is proportional to the amount of hardware one can afford to deploy. A quadcopter two inches in diameter can carry a lethal explosive charge, and one million can fit in a regular shipping container. Precisely because they are autonomous, these weapons would not need one million human supervisors to do their work.

As weapons of mass destruction, scalable autonomous weapons have advantages for the attacker compared to nuclear weapons and carpet bombing: they leave property intact and can be applied selectively to eliminate only those who might threaten an occupying force. They could certainly be used to wipe out an entire ethnic group or all the adherents of a particular religion. In many situations, they would also be untraceable. These characteristics make them particularly attractive to non-state actors.

These considerations—particularly those characteristics that advantage the attacker—suggest that autonomous weapons will reduce global and national security for all parties. The rational response for governments seems to be to engage in arms control discussions rather than an arms race.

The process of designing a treaty is not without its difficulties, however. AI is a **dual use** technology: AI technologies that have peaceful applications such as flight control, visual tracking, mapping, navigation, and multiagent planning, can easily be applied to military purposes. It is easy to turn an autonomous quadcopter into a weapon simply by attaching an explosive and commanding it to seek out a target. Dealing with this will require careful implementation of compliance regimes with industry cooperation, as has already been demonstrated with some success by the Chemical Weapons Convention.

28.3.2 Surveillance, security, and privacy

In 1976, Joseph Weizenbaum warned that automated speech recognition technology could lead to widespread wiretapping, and hence to a loss of civil liberties. Today, that threat has been realized, with most electronic communication going through central servers that can be monitored, and cities packed with microphones and cameras that can identify and track individuals based on their voice, face, and gait. Surveillance that used to require expensive and scarce human resources can now be done at a mass scale by machines.

As of 2018, there were as many as 350 million **surveillance cameras** in China and 70 million in the United States. China and other countries have begun exporting surveillance technology to low-tech countries, some with reputations for mistreating their citizens and disproportionately targeting marginalized communities. AI engineers should be clear on what uses of surveillance are compatible with human rights, and decline to work on applications that are incompatible.

As more of our institutions operate online, we become more vulnerable to cybercrime (phishing, credit card fraud, botnets, ransomware) and cyberterrorism (including potentially deadly attacks such as shutting down hospitals and power plants or commandeering selfdriving cars). Machine learning can be a powerful tool for both sides in the **cybersecurity** battle. Attackers can use automation to probe for insecurities and they can apply reinforcement learning for phishing attempts and automated blackmail. Defenders can use unsupervised learning to detect anomalous incoming traffic patterns (Chandola *et al.,* 2009; Malhotra *et al.,* 2015) and various machine learning techniques to detect fraud (Fawcett and Provost, 1997; Bolton and Hand, 2002). As attacks get more sophisticated, there is a greater responsibility for all engineers, not just the security experts, to design secure systems from the start. One forecast (Kanal, 2017) puts the market for machine learning in cybersecurity at about $100 billion by 2021.

As we interact with computers for increasing amounts of our daily lives, more data on us is being collected by governments and corporations. Data collectors have a moral and legal responsibility to be good stewards of the data they hold. In the U.S., the Health Insurance Portability and Accountability Act (HIPAA) and the Family Educational Rights and Privacy Act (FERPA) protect the privacy of medical and student records. The European Union’s General Data Protection Regulation (GDPR) mandates that companies design their systems with protection of data in mind and requires that they obtain user consent for any collection or processing of data.

Balanced against the individual’s right to privacy is the value that society gains from sharing data. We want to be able to stop terrorists without oppressing peaceful dissent, and we want to cure diseases without compromising any individual’s right to keep their health history private. One key practice is **de-identification**: eliminating personally identifying information (such as name and social security number) so that medical researchers can use the data to advance the common good. The problem is that the shared de-identified data may be subject to re-identification. For example, if the data strips out the name, social security number, and street address, but includes date of birth, gender, and zip code, then, as shown by Latanya Sweeney (2000), 87% of the U.S. population can be uniquely re-identified. Sweeney emphasized this point by re-identifying the health record for the governor of her state when he was admitted to the hospital. In the **Netflix Prize** competition, de-identified records of individual movie ratings were released, and competitors were asked to come up with a machine learning algorithm that could accurately predict which movies an individual would like. But researchers were able to re-identify individual users by matching the date of a rating in the Netflix database with the date of a similar ranking in the Internet Movie Database (IMDB), where users sometimes use their actual names (Narayanan and Shmatikov, 2006).

This risk can be mitigated somewhat by **generalizing fields**: for example, replacing the exact birth date with just the year of birth, or a broader range like “20-30 years old.” Deleting a field altogether can be seen as a form of generalizing to “any.” But generalization alone does not guarantee that records are safe from re-identification; it may be that there is only one person in zip code 94720 who is 90–100 years old. A useful property is **k-anonymity**: a database is k-anonymized if every record in the database is indistinguishable from at least $k - 1$ other records. If there are records that are more unique than this, they would have to be further generalized.

An alternative to sharing de-identified records is to keep all records private, but allow **aggregate querying**. An API for queries against the database is provided, and valid queries receive a response that summarizes the data with a count or average. But no response is given if it would violate certain guarantees of privacy. For example, we could allow an epidemiologist to ask, for each zip code, the percentage of people with cancer. For zip codes with at least n people a percentage would be given (with a small amount of random noise), but no response would be given for zip codes with fewer than n people..

Care must be taken to protect against de-identification using multiple queries. For example, if the query “average salary and number of employees of XYZ company age 30-40” gives the response [$81,234, 12] and the query “average salary and number of employees of XYZ company age 30-41” gives the response [$81,199, 13], and if we use LinkedIn to find the one 41-year-old at XYZ company, then we have successfully identified them, and can compute their exact salary, even though all the responses involved 12 or more people. The system must be carefully designed to protect against this, with a combination of limits on the queries that can be asked (perhaps only a predefined set of non-overlapping age ranges can be queried) and the precision of the results (perhaps both queries give the answer “about $81,000”).

A stronger guarantee is **differential privacy**, which assures that an attacker cannot use queries to re-identify any individual in the database, even if the attacker can make multiple queries and has accessw query response employs a randomized algorithm that adds a small amount of noise to the result. Given a database *D*, any record in the database *r*, any query *Q*, and a possible response *y* to the query, we say that the database *D* has ϵ–differential privacy if the log probability of the response *y* varies by less than ϵ when we add the record *r*:

$$|\log P(Q(D) = y) - \log P(Q(D + r) = y)| \leq \epsilon.$$

In other words, whether any one person decides to participate in the data base or not makes no appreciable difference to the answers anyone can get, and therefore there is no privacy disincentive to participate. Many databases are designed to guarantee differential privacy.

So far we have considered the issue of sharing de-identified data from a central database. An approach called **federated learning** (Konečnỳ *et al.*, 2016) has no central database; instead, users maintain their own local databases that keep their data private. However, they can share parameters of a machine learning model that is enhanced with their data, without the risk of revealing any of the private data. Imagine a speech understanding application that users can run locally on their phone. The application contains a baseline neural network, which is then improved by local training on the words that are heard on the user's phone. Periodically, the owners of the application poll a subset of the users and ask them for the parameter values of their improved local network, but not for any of their raw data. The parameter values are combined together to form a new improved model which is then made available to all users, so that they all get the benefit of the training that is done by other users.

For this scheme to preserve privacy, we have to be able to guarantee that the model parameters shared by each user cannot be reverse-engineered. If we sent the raw parameters, there is a chance that an adversary inspecting them could deduce whether, say, a certain word had been heard by the user's phone. One way to eliminate this risk is with **secure aggregation** (Bonawitz *et al.*, 2017). The idea is that the central server doesn't need to know the exact parameter value from each distributed user; it only needs to know the average value for each parameter, over all polled users. So each user can disguise their parameter values by adding a unique mask to each value; as long as the sum of the masks is zero, the central server will be able to compute the correct average. Details of the protocol make sure that it is efficient in terms of communication (less than half the bits transmitted correspond to masking), is robust to individual users failing to respond, and is secure in the face of adversarial users, eavesdroppers, or even an adversarial central server.

28.3.3 Fairness and bias

Machine learning is augmenting and sometimes replacing human decision-making in important situations: whose loan gets approved, to what neighborhoods police officers are deployed, who gets pretrial release or parole. But machine learning models can perpetuate **societal bias**. Consider the example of an algorithm to predict whether criminal defendants are likely to re-offend, and thus whether they should be released before trial. It could well be that such a system picks up the racial or gender prejudices of human judges from the examples in the training set. Designers of machine learning systems have a moral responsibility to ensure that their systems are in fact fair. In regulated domains such as credit, education, employment, and housing, they have a legal responsibility as well. But what is fairness? There are multiple criteria; here are six of the most commonly-used concepts:

- **Individual fairness**: A requirement that individuals are treated similarly to other similar individuals, regardless of what class they are in.
- **Group fairness**: A requirement that two classes are treated similarly, as measured by some summary statistic.
- **Fairness through unawareness**: If we delete the race and gender attributes from the data set, then it might seem that the system cannot discriminate on those attributes. Unfortunately, we know that machine learning models can predict latent variables (such as race and gender) given other correlated variables (such as zip code and occupation). Furthermore, deleting those attributes makes it impossible to verify equal opportunity or equal outcomes. Still, some countries (e.g., Germany) have chosen this approach for their demographic statistics (whether or not machine learning models are involved).
- **Equal outcome**: The idea that each demographic class gets the same results; they have **demographic parity**. For example, suppose we have to decide whether we should approve loan applications; the goal is to approve those applicants who will pay back the loan and not those who will default on the loan. Demographic parity says that both males and females should have the same percentage of loans approved. Note that

this is a group fairness criterion that does nothing to ensure individual fairness; a well-qualified applicant might be denied and a poorly-qualified applicant might be approved, as long as the overall percentages are equal. Also, this approach favors redress of past biases over accuracy of prediction. If a man and a woman are equal in every way, except the woman receives a lower salary for the same job, should she be approved because she would be equal if not for historical biases, or should she be denied because the lower salary does in fact make her more likely to default?

- **Equal opportunity**: The idea that the people who truly have the ability to pay back the loan should have an equal chance of being correctly classified as such, regardless of their sex. This approach is also called “balance.” It can lead to unequal outcomes and ignores the effect of bias in the societal processes that produced the training data.
- **Equal impact**: People with similar likelihood to pay back the loan should have the same expected utility, regardless of the class they belong to. This goes beyond equal opportunity in that it considers both the benefits of a true prediction and the costs of a false prediction.

Let us examine how these issues play out in a particular context. COMPAS is a commercial system for recidivism (re-offense) scoring. It assigns to a defendant in a criminal case a **risk score**, which is then used by a judge to help make decisions: Is it safe to release the defendant before trial, or should they be held in jail? If convicted, how long should the sentence be? Should parole be granted? Given the significance of these decisions, the system has been the subject of intense scrutiny (Dressel and Farid, 2018).

COMPAS is designed to be **well calibrated**: all the individuals who are given the same score by the algorithm should have approximately the same probability of re-offending, regardless of race. For example, among all people that the model assigns a risk score of 7 out of 10, 60% of whites and 61% of blacks re-offend. The designers thus claim that it meets the desired fairness goal.

On the other hand, COMPAS does not achieve equal opportunity: the proportion of those who did not re-offend but were falsely rated as high-risk was 45% for blacks and 23% for whites. In the case *State v. Loomis*, where a judge relied on COMPAS to determine the sentence of the defendant, Loomis argued that the secretive inner workings of the algorithm violated his due process rights. Though the Wisconsin Supreme Court found that the sentence given would be no different without COMPAS in this case, it did issue warnings about the algorithm's accuracy and risks to minority defendants. Other researchers have questioned whether it is appropriate to use algorithms in applications such as sentencing.

We could hope for an algorithm that is both well calibrated and equal opportunity, but, as Kleinberg *et al.* (2016) show, that is impossible. If the base classes are different, then any algorithm that is well calibrated will necessarily not provide equal opportunity, and vice versa. How can we weigh the two criteria? Equal impact is one possibility. In the case of COMPAS, this means weighing the negative utility of defendants being falsely classified as high risk and losing their freedom, versus the cost to society of an additional crime being committed, and finding the point that optimizes the tradeoff. This is complicated because there are multiple costs to consider. There are individual costs—a defendant who is wrongfully held in jail suffers a loss, as does the victim of a defendant who was wrongfully released and re-offends. But beyond that there are group costs—everyone has a certain fear that they will be wrongfully jailed, or will be the victim of a crime, and all taxpayers contribute to the costs of jails and courts. If we give value to those fears and costs in proportion to the size of a group, then utility for the majority may come at the expense of a minority.

Another problem with the whole idea of recidivism scoring, regardless of the model used, is that we don't have unbiased ground truth data. The data does not tell us who has *committed* a crime—all we know is who has been *convicted* of a crime. If the arresting officers, judge, or jury is biased, then the data will be biased. If more officers patrol some locations, then the data will be biased against people in those locations. Only defendants who are released are candidates to recommit, so if the judges making the release decisions are biased, the data may be biased. If you assume that behind the biased data set there is an underlying, unknown, unbiased data set which has been corrupted by an agent with biases, then there are techniques to recover an approximation to the unbiased data. Jiang and Nachum (2019) describe various scenarios and the techniques involved.

One more risk is that machine learning can be used to *justify* bias. If decisions are made by a biased human after consulting with a machine learning system, the human can say "here is how my interpretation of the model supports my decision, so you shouldn't question my decision." But other interpretations could lead to an opposite decision.

Sometimes fairness means that we should reconsider the objective function, not the data or the algorithm. For example, in making job hiring decisions, if the objective is to hire candidates with the best qualifications in hand, we risk unfairly rewarding those who have had advantageous educational opportunities throughout their lives, thereby enforcing class boundaries. But if the objective is to hire candidates with the best ability to learn on the job, we have a better chance to cut across class boundaries and choose from a broader pool. Many companies have programs designed for such applicants, and find that after a year of training, the employees hired this way do as well as the traditional candidates. Similarly, just 18% of computer science graduates in the U.S. are women, but some schools, such as Harvey Mudd University, have achieved 50% parity with an approach that is focused on encouraging and retaining those who start the computer science program, especially those who start with less programming experience.

A final complication is deciding which classes deserve protection. In the U.S., the Fair Housing Act recognized seven protected classes: race, color, religion, national origin, sex, disability, and familial status. Other local, state, and federal laws recognize other classes, including sexual orientation, and pregnancy, marital, and veteran status. Is it fair that these classes count for some laws and not others? International human rights law, which encompasses a broad set of protected classes, is a potential framework to harmonize protections across various groups.

Even in the absence of societal bias, **sample size disparity** can lead to biased results. In most data sets there will be fewer training examples of minority class individuals than of majority class individuals. Machine learning algorithms give better accuracy with more training data, so that means that members of minority classes will experience lower accuracy. For example, Buolamwini and Gebru (2018) examined a computer vision gender identification service, and found that it had near-perfect accuracy for light-skinned males, and a 33% error rate for dark-skinned females. A constrained model may not be able to simultaneously fit both the majority and minority class—a linear regression model might minimize average error by fitting just the majority class, and in an SVM model, the support vectors might all correspond to majority class members.

Bias can also come into play in the software development process (whether or not the software involves machine learning). Engineers who are debugging a system are more likely to notice and fix those problems that are applicable to themselves. For example, it is difficult to notice that a user interface design won't work for colorblind people unless you are in fact colorblind, or that an Urdu language translation is faulty if you don't speak Urdu.

How can we defend against these biases? First, understand the limits of the data you are using. It has been suggested that data sets (Gebru *et al.*, 2018; Hind *et al.*, 2018) and models (Mitchell *et al.*, 2019) should come with annotations: declarations of provenance, security, conformity, and fitness for use. This is similar to the **data sheets** that accompany electronic components such as resistors; they allow designers to decide what components to use. In addition to the data sheets, it is important to train engineers to be aware of issues of fairness and bias, both in school and with on-the-job training. Having a diversity of engineers from different backgrounds makes it easier for them to notice problems in the data or models. A study by the AI Now Institute (West *et al.*, 2019) found that only 18% of authors at leading AI conferences and 20% of AI professors are women. Black AI workers are at less than 4%. Rates at industry research labs are similar. Diversity could be increased by programs earlier in the pipeline—in college or high school—and by greater awareness at the professional level. Joy Buolamwini founded the Algorithmic Justice League to raise awareness of this issue and develop practices for accountability.

A second idea is to de-bias the data (Zemel *et al.*, 2013). We could over-sample from minority classes to defend against sample size disparity. Techniques such as SMOTE, the synthetic minority over-sampling technique (Chawla *et al.*, 2002) or ADASYN, the adaptive synthetic sampling approach for imbalanced learning (He *et al.*, 2008), provide principled ways of oversampling. We could examine the provenance of data and, for example, eliminate examples from judges who have exhibited bias in their past court cases. Some analysts object to the idea of discarding data, and instead would recommend building a hierarchical model of the data that includes sources of bias, so they can be modeled and compensated for. Google and NeurIPS have attempted to raise awareness of this issue by sponsoring the Inclusive Images Competition, in which competitors train a network on a data set of labeled images collected in North America and Europe, and then test it on images taken from all around the world. The issue is that given this data set, it is easy to apply the label “bride” to a woman in a standard Western wedding dress, but harder to recognize traditional African and Indian matrimonial dress.

A third idea is to invent new machine learning models and algorithms that are more resistant to bias; and the final idea is to let a system make initial recommendations that may be biased, but then train a second system to de-bias the recommendations of the first one. Bellamy *et al.* (2018) introduced the IBM AI FAIRNESS 360 system, which provides a framework for all of these ideas. We expect there will be increased use of tools like this in the future.

How do you make sure that the systems you build will be fair? A set of best practices has been emerging (although they are not always followed):

- Make sure that the software engineers talk with social scientists and domain experts to understand the issues and perspectives, and consider fairness from the start.
- Create an environment that fosters the development of a diverse pool of software engineers that are representative of society.
- Define what groups your system will support: different language speakers, different age groups, different abilities with sight and hearing, etc.

- Optimize for an objective function that incorporates fairness.
- Examine your data for prejudice and for correlations between protected attributes and other attributes.
- Understand how any human annotation of data is done, design goals for annotation accuracy, and verify that the goals are met.
- Don't just track overall metrics for your system; make sure you track metrics for subgroups that might be victims of bias.
- Include system tests that reflect the experience of minority group users.
- Have a feedback loop so that when fairness problems come up, they are dealt with.

28.3.4 Trust and transparency

It is one challenge to make an AI system accurate, fair, safe, and secure; a different challenge to convince everyone else that you have done so. People need to be able to **trust** the systems they use. A PwC survey in 2017 found that 76% of businesses were slowing the adoption of AI because of trustworthiness concerns. In Section 19.9.4 we covered some of the engineering approaches to trust; here we discuss the policy issues.

To earn trust, any engineered systems must go through a **verification and validation** (V&V) process. Verification means that the product satisfies the specifications. Validation means ensuring that the specifications actually meet the needs of the user and other affected parties. We have an elaborate V&V methodology for engineering in general, and for traditional software development done by human coders; much of that is applicable to AI systems. But machine learning systems are different and demand a different V&V process, which has not yet been fully developed. We need to verify the data that these systems learn from; we need to verify the accuracy and fairness of the results, even in the face of uncertainty that makes an exact result unknowable; and we need to verify that adversaries cannot unduly influence the model, nor steal information by querying the resulting model.

One instrument of trust is **certification**; for example, Underwriters Laboratories (UL) was founded in 1894 at a time when consumers were apprehensive about the risks of electric power. UL certification of appliances gave consumers increased trust, and in fact UL is now considering entering the business of product testing and certification for AI.

Other industries have long had safety standards. For example, ISO 26262 is an international standard for the safety of automobiles, describing how to develop, produce, operate, and service vehicles in a safe way. The AI industry is not yet at this level of clarity, although there are some frameworks in progress, such as IEEE P7001, a standard defining ethical design for artificial intelligence and autonomous systems (Bryson and Winfield, 2017). There is ongoing debate about what kind of certification is necessary, and to what extent it should be done by the government, by professional organizations like IEEE, by independent certifiers such as UL, or through self-regulation by the product companies.

Another aspect of trust is **transparency**: consumers want to know what is going on inside a system, and that the system is not working against them, whether due to intentional malice, an unintentional bug, or pervasive societal bias that is recapitulated by the system. In some cases this transparency is delivered directly to the consumer. In other cases their are intellectual property issues that keep some aspects of the system hidden to consumers, but open to regulators and certification agencies.

When an AI system turns you down for a loan, you deserve an explanation. In Europe, the GDPR enforces this for you. An AI system that can explain itself is called **explainable AI (XAI)**. A good explanation has several properties: it should be understandable and convincing to the user, it should accurately reflect the reasoning of the system, it should be complete, and it should be specific in that different users with different conditions or different outcomes should get different explanations.

It is quite easy to give a decision algorithm access to its own deliberative processes, simply by recording them and making them available as data structures. This means that machines may eventually be able to give better explanations of their decisions than humans can. Moreover, we can take steps to certify that the machine's explanations are not deceptions (intentional or self-deception), something that is more difficult with a human.

An explanation is a helpful but not sufficient ingredient to trust. One issue is that explanations are not decisions: they are stories about decisions. As discussed in Section 19.9.4, we say that a system is interpretable if we can inspect the source code of the model and see what it is doing, and we say it is explainable if we can make up a story about what it is doing—even if the system itself is an uninterpretable black box. To explain an uninterpretable black box, we need to build, debug, and test a separate explanation system, and make sure it is in sync with the original system. And because humans love a good story, we are all too willing to be swayed by an explanation that sounds good. Take any political controversy of the day, and you can always find two so-called experts with diametrically opposed explanations, both of which are internally consistent.

A final issue is that an explanation about one case does not give you a summary over other cases. If the bank explains, "Sorry, you didn't get the loan because you have a history of previous financial problems," you don't know if that explanation is accurate or if the bank is secretly biased against you for some reason. In this case, you require not just an explanation, but also an **audit** of past decisions, with aggregated statistics across various demographic groups, to see if their approval rates are balanced.

Part of transparency is knowing whether you are interacting with an AI system or a human. Toby Walsh (2015) proposed that “an autonomous system should be designed so that it is unlikely to be mistaken for anything besides an autonomous system, and should identify itself at the start of any interaction.” He called this the “red flag” law, in honor of the UK’s 1865 Locomotive Act, which required any motorized vehicle to have a person with a red flag walk in front of it, to signal the oncoming danger.

In 2019, California enacted a law stating that “It shall be unlawful for any person to use a bot to communicate or interact with another person in California online, with the intent to mislead the other person about its artificial identity.”

28.3.5 The future of work

From the first agricultural revolution (10,000 BCE) to the industrial revolution (late 18th century) to the green revolution in food production (1950s), new technologies have changed the way humanity works and lives. A primary concern arising from the advance of AI is that human labor will become obsolete. Aristotle, in Book I of his *Politics*, presents the main point quite clearly:

> For if every instrument could accomplish its own work, obeying or anticipating the will of others ... if, in like manner, the shuttle would weave and the plectrum touch the lyre without a hand to guide them, chief workmen would not want servants, nor masters slaves.

Everyone agrees with Aristotle’s observation that there is an immediate reduction in employment when an employer finds a mechanical method to perform work previously done by a person. The issue is whether the so-called compensation effects that ensue—and that tend to increase employment—will eventually make up for this reduction. The primary compensation effect is the increase in overall wealth from greater productivity, which leads in turn to greater demand for goods and tends to increase employment. For example, PwC (Rao and Verweij, 2017) predicts that AI contribute $15 trillion annually to global GDP by 2030. The healthcare and automotive/transportation industries stand to gain the most in the short term. However, the advantages of automation have not yet taken over in our economy: the current rate of growth in labor productivity is actually below historical standards. Brynjolfsson *et al.*(2018) attempt to explain this paradox by suggesting that the lag between the development of basic technology and its implementation in the economy is longer than commonly supposed.

Technological innovations have historically put some people out of work. Weavers were replaced by automated looms in the 1810s, leading to the Luddite protests. The Luddites were not against technology *per se;* they just wanted the machines to be used by skilled workers paid a good wage to make high-quality goods, rather than by unskilled workers to make poor-quality goods at low wages. The global destruction of jobs in the 1930s led John Maynard Keynes to coin the term **technological unemployment**. In both cases, and several others, employment levels eventally recovered.

The mainstream economic view for most of the 20th century was that technological employment was at most a short-term phenomenon. Increased productivity would always lead to increased wealth and increased demand, and thus net job growth. A commonly cited example is that of bank tellers: although ATMs replaced humans in the job of counting out cash for withdrawals, that made it cheaper to operate a bank branch, so the number of branches increased, leading to more bank employees overall. The nature of the work also changed, becoming less routine and requiring more advanced business skills. The net effect of automation seems to be in eliminating *tasks* rather than *jobs*.

The majority of commenters predict that the same will hold true with AI technology, at least in the short run. Gartner, McKinsey, Forbes, the World Economic Forum, and the Pew Research Center each released reports in 2018 predicting a net increase in jobs due to AI-driven automation. But some analysts think that this time around, things will be different. In 2019, IBM predicted that 120 million workers would need retraining due to automation by 2022, and Oxford Economics predicted that 20 million manufacturing jobs could be lost to automation by 2030.

Frey and Osborne (2017) survey 702 different occupations, and estimate that 47% of them are at risk of being automated, meaning that at least some of the tasks in the occupation can be performed by machine. For example, almost 3% of the workforce in the U.S. are vehicle drivers, and in some districts, as much as 15% of the male workforce are drivers. As we saw in Chapter 26, the task of driving is likely to be eliminated by driverless cars/trucks/buses/taxis.

It is important to distinguish between occupations and the tasks within those occupations. McKinsey estimates that only 5% of occupations are fully automatable, but that 60% of occupations can have about 30% of their tasks automated. For example, future truck drivers will spend less time holding the steering wheel and more time making sure that the goods are picked up and delivered properly; serving as customer service representatives and salespeople at either end of the journey; and perhaps managing convoys of, say, three robotic trucks. Replacing three drivers with one convoy manager implies a net loss in employment, but if transportation costs decrease, there will be more demand, which wins some of the jobs back—but perhaps not all of them. As another example, despite many advances in applying machine learning to the problem of medical imaging, radiologists have so far been augmented, not replaced, by these tools. Ultimately, there is a choice of how to make use of automation: do we want to focus on *cutting cost*, and thus see job loss as a positive; or do we want to focus on *improving quality*, making life better for the worker and the customer?

It is difficult to predict exact timelines for automation, but currently, and for the next few years, the emphasis is on automation of structured analytical tasks, such as reading x-ray images, customer relationship management (e.g., bots that automatically sort customer complaints and respond with suggested remedies), and **business process automation** that combines text documents and structured data to make business decisions and improve workflow. Over time, we will see more automation with physical robots, first in controlled warehouse environments, then in more uncertain environments, building to a significant portion of the marketplace by around 2030.

As populations in developed countries grow older, the ratio between workers and retirees changes. In 2015 there were less than 30 retirees per 100 workers; by 2050 there may be over 60 per 100 workers. Care for the elderly will be an increasingly important role, one that can partially be filled by AI. Moreover, if we want to maintain the current standard of living, it will also be necessary to make the remaining workers more productive; automation seems like the best opportunity to do that.

Even if automation has a multi-trillion-dollar net positive impact, there may still be problems due to the **pace of change**. Consider how change came to the farming industry: in 1900, over 40% of the U.S. workforce was in agriculture, but by 2000 that had fallen to 2%.[3] That is a huge disruption in the way we work, but it happened over a period of 100 years, and thus across generations, not in the lifetime of one worker.

Workers whose jobs are automated away this decade may have to retrain for a new profession within a few years—and then perhaps see their new profession automated and face yet another retraining period. Some may be happy to leave their old profession—we see that as the economy improves, trucking companies need to offer new incentives to hire enough drivers—but workers will be apprehensive about their new roles. To handle this, we as a society need to provide lifelong education, perhaps relying in part on online education driven by artificial intelligence (Martin, 2012). Bessen (2015) argues that workers will not see increases in income until they are trained to implement the new technologies, a process that takes time.

Technology tends to magnify **income inequality**. In an information economy marked by high-bandwidth global communication and zero-marginal-cost replication of intellectual property (what Frank and Cook (1996) call the "Winner-Take-All Society"), rewards tend to be concentrated. If farmer Ali is 10% better than farmer Bo, then Ali gets about 10% more income: Ali can charge slightly more for superior goods, but there is a limit on how much can be produced on the land, and how far it can be shipped. But if software app developer Cary is 10% better than Dana, it may be that Cary ends up with 99% of the global market. AI increases the pace of technological innovation and thus contributes to this overall trend, but AI also holds the promise of allowing us to take some time off and let our automated agents handle things for a while. Tim Ferriss (2007) recommends using automation and outsourcing to achieve a four-hour work week.

Before the industrial revolution, people worked as farmers or in other crafts, but didn't report to a **job** at a place of work and put in hours for an employer. But today, most adults in developed countries do just that, and the job serves three purposes: it fuels the production of the goods that society needs to flourish, it provides the income that the worker needs to live, and it gives the worker a sense of purpose, accomplishment, and social integration. With increasing automation, it may be that these three purposes become disaggregated—society's needs will be served in part by automation, and in the long run, individuals will get their sense of purpose from contributions other than work. Their income needs can be served by social policies that include a combination of free or inexpensive access to social services and education, portable health care, retirement, and education accounts, progressive tax rates, earned income tax credits, negative income tax, or universal basic income.

28.3.6 Robot rights

The question of robot consciousness, discussed in Section 28.2, is critical to the question of what rights, if any, robots should have. If they have no consciousness, no qualia, then few would argue that they deserve rights.

But if robots can feel pain, if they can dread death, if they are considered "persons," then the argument can be made (e.g., by Sparrow (2004)) that they have rights and deserve to have their rights recognized, just as slaves, women, and other historically oppressed groups have fought to have their rights recognized. The issue of robot personhood is often considered in fiction: from Pygmalion to Coppélia to Pinocchio to the movies *AI* and *Centennial Man*, we have the legend of a doll/robot coming to life and striving to be accepted as a human with human rights. In real life, Saudi Arabia made headlines by giving honorary citizenship to Sophia, a human-looking puppet capable of speaking preprogrammed lines.

If robots have rights, then they should not be enslaved, and there is a question of whether reprogramming them would be a kind of enslavement. Another ethical issue involves voting rights: a rich person could buy thousands of robots and program them to cast thousands of votes—should those votes count? If a robot clones itself, can they both vote? What is the boundary between ballot stuffing and exercising free will, and when does robotic voting violate the “one person, one vote” principle?

Ernie Davis argues for avoiding the dilemmas of robot consciousness by never building robots that could possibly be considered conscious. This argument was previously made by Joseph Weizenbaum in his book *Computer Power and Human Reason* (1976), and before that by Julien de La Mettrie in *L'Homme Machine* (1748). Robots are tools that we create, to do the tasks we direct them to do, and if we grant them personhood, we are just declining to take responsibility for the actions of our own property: “I’m not at fault for my self-driving car crash—the car did it itself.”

This issue takes a different turn if we develop human–robot hybrids. Of course we already have humans enhanced by technology such as contact lenses, pacemakers, and artificial hips. But adding computational protheses may blur the lines between human and machine.

28.3.7 AI Safety

Almost any technology has the potential to cause harm in the wrong hands, but with AI and robotics, the hands might be operating on their own. Countless science fiction stories have warned about robots or cyborgs running amok. Early examples include Mary Shelley’s *Frankenstein, or the Modern Prometheus* (1818) and Karel Čapek’s play *R.U.R.* (1920), in which robots conquer the world. In movies, we have *The Terminator* (1984) and *The Matrix*(1999), which both feature robots trying to eliminate humans—the **robopocalypse** (Wilson, 2011). Perhaps robots are so often the villains because they represent the unknown, just like the witches and ghosts of tales from earlier eras. We can hope that a robot that is smart enough to figure out how to terminate the human race is also smart enough to figure out that that was not the intended utility function; but in building intelligent systems, we want to rely not just on hope, but on a design process with guarantees of safety.

It would be unethical to distribute an unsafe AI agent. We require our agents to avoid accidents, to be resistant to adversarial attacks and malicious abuse, and in general to cause benefits, not harms. That is especially true as AI agents are deployed in safety-critical applications, such as driving cars, controlling robots in dangerous factory or construction settings, and making life-or-death medical decisions.

There is a long history of **safety engineering** in traditional engineering fields. We know how to build bridges, airplanes, spacecraft, and power plants that are designed up front to behave safely even when components of the system fail. The first technique is **failure modes and effect analysis (FMEA)**: analysts consider each component of the system, and imagine every possible way the component could go wrong (for example, what if this bolt were to snap?), drawing on past experience and on calculations based on the physical properties of the component. Then the analysts work forward to see what would result from the failure. If the result is severe (a section of the bridge could fall down) then the analysts alter the design to mitigate the failure. (With this additional cross-member, the bridge can survive the failure of any 5 bolts; with this backup server, the online service can survive a tsunami taking out the primary server.) The technique of **fault**

tree analysis (FTA) is used to make these determinations: analysts build an AND/OR tree of possible failures and assign probabilities to each root cause, allowing for calculations of overall failure probability. These techniques can and should be applied to all safety-critical engineered systems, including AI systems.

The field of **software engineering** is aimed at producing reliable software, but the emphasis has historically been on *correctness*, not *safety.* Correctness means that the software faithfully implements the specification. But safety goes beyond that to insist that the specification has considered any feasible failure modes, and is designed to degrade gracefully even in the face of unforeseen failures. For example, the software for a self-driving car wouldn't be considered safe unless it can handle unusual situations. For example, what if the power to the main computer dies? A safe system will have a backup computer with a separate power supply. What if a tire is punctured at high speed? A safe system will have tested for this, and will have software to correct for the resulting loss of control.

An agent designed as a utility maximizer, or as a goal achiever, can be unsafe if it has the wrong objective function. Suppose we give a robot the task of fetching a coffee from the kitchen. We might run into trouble with **unintended side effects**—the robot might rush to accomplish the goal, knocking over lamps and tables along the way. In testing, we might notice this kind of behavior and modify the utility function to penalize such damage, but it is difficult for the designers and testers to anticipate *all* possible side effects ahead of time.

One way to deal with this is to design a robot to have **low impact** (Armstrong and Levinstein, 2017): instead of just maximizing utility, maximize the utility minus a weighted summary of all changes to the state of the world. In this way, all other things being equal, the robot prefers not to change those things whose effect on utility is unknown; so it avoids knocking over the lamp not because it knows specifically that knocking the lamp will cause it to fall over and break, but because it knows in general that disruption might be bad. This can be seen as a version of the physician’s creed “first, do no harm,” or as an analog to **regularization** in machine learning: we want a policy that achieves goals, but we prefer policies that take smooth, low-impact actions to get there. The trick is how to measure impact. It is not acceptable to knock over a fragile lamp, but perfectly fine if the air molecules in the room are disturbed a little, or if some bacteria in the room are inadvertently killed. It is certainly not acceptable to harm pets and humans in the room. We need to make sure that the robot knows the differences between these cases (and many subtle cases in between) through a combination of explicit programming, machine learning over time, and rigorous testing.

Utility functions can go wrong due to **externalities**, the word used by economists for factors that are outside of what is measured and paid for. The world suffers when greenhouse gases are considered as externalities—companies and countries are not penalized for producing them, and as a result everyone suffers. Ecologist Garrett Hardin (1968) called the exploitation of shared resources the **tragedy of the commons**. We can mitigate the tragedy by internalizing the externalities—making them part of the utility function, for example with a carbon tax—or by using the design principles that economist Elinor Ostrom identified as being used by local people throughout the world for centuries (work that won her the Nobel Prize in Economics in 2009):

- Clearly define the shared resource and who has access.
- Adapt to local conditions.
- Allow all parties to participate in decisions.

- Monitor the resource with accountable monitors.
- Sanctions, proportional to the severity of the violation.
- Easy conflict resolution procedures.
- Hierarchical control for large shared resources.

Victoria Krakovna (2018) has cataloged examples of AI agents that have gamed the system, figuring out how to maximize utility without actually solving the problem that their designers intended them to solve. To the designers this looks like cheating, but to the agents, they are just doing their job. Some agents took advantage of bugs in the simulation (such as floating point overflow bugs) to propose solutions that would not work once the bug was fixed. Several agents in video games discovered ways to crash or pause the game when they were about to lose, thus avoiding a penalty. And in a specification where crashing the game was penalized, one agent learned to use up just enough of the game's memory so that when it was the opponent's turn, it would run out of memory and crash the game. Finally, a genetic algorithm operating in a simulated world was supposed to evolve fast-moving creatures but in fact produced creatures that were enormously tall and moved fast by falling over.

Designers of agents should be aware of these kinds of specification failures and take steps to avoid them. To help them do that, Krakovna was part of the team that released the AI Safety Gridworlds environments (Leike *et al.*, 2017), which allows designers to test how well their agents perform.

The moral is that we need to be very careful in specifying what we want, because with utility maximizers we get what we actually asked for. The **value alignment problem** is the problem of making sure that what we ask for is what we really want; it is also known as the **King Midas problem**, as discussed on page 51. We run into trouble when a utility function fails to capture background societal norms about acceptable behavior. For example, a human who is hired to clean floors, when faced with a messy person who repeatedly tracks in dirt, knows that it is acceptable to politely ask the person to be more careful, but it is not acceptable to kidnap or incapacitate said person.

A robotic cleaner needs to know these things too, either through explicit programming or by learning from observation. Trying to write down all the rules so that the robot always does the right thing is almost certainly hopeless. We have been trying to write loophole-free tax laws for several thousand years without success. Better to make the robot *want* to pay taxes, so to speak, than to try to make rules to force it to do so when it really wants to do something else. A sufficiently intelligent robot will find a way to do something else.

Robots can learn to conform better with human preferences by observing human behavior. This is clearly related to the notion of apprenticeship learning (Section 23.6). The robot may learn a policy that directly suggests what actions to take in what situations; this is often a straightforward supervised learning problem if the environment is observable. For example, a robot can watch a human playing chess: each state–action pair is an example for the learning process. Unfortunately, this form of **imitation learning** means that the robot will repeat human mistakes. Instead, the robot can apply **inverse reinforcement learning** to discover the utility function that the humans must be operating under. Watching even terrible chess players is probably enough for the robot to learn the objective of the game. Given just this information, the robot can then go on to exceed human performance—as, for example, ALPHAZERO did in chess—by computing optimal or near-optimal policies from the objective. This approach works not just in board games, but in real-world physical tasks such as helicopter aerobatics (Coates *et al.*, 2009).

In more complex settings involving, for example, social interactions with humans, it is very unlikely that the robot will converge to exact and correct knowledge of each human's individual preferences. (After all, many humans never quite learn what makes other humans tick, despite a lifetime of experience, and many of us are unsure of our own preferences too.) It will be necessary, therefore, for machines to function appropriately when it is uncertain about human preferences. In Chapter 17, we introduced **assistance games**, which capture exactly this situation. Solutions to assistance games include acting cautiously, so as not to disturb aspects of the world that the human might care about, and asking questions. For example, the robot could ask whether turning the oceans into sulphuric acid is an acceptable solution to global warming before it puts the plan into effect.

In dealing with humans, a robot solving an assistance game must accommodate human imperfections. If the robot asks permission, the human may give it, not foreseeing that the robot's proposal is in fact catastrophic in the long term. Moreover, humans do not have complete introspective access to their true utility function, and they don't always act in a way that is compatible with it. Humans sometimes lie or cheat, or do things they know are wrong. They sometimes take self-destructive actions like overeating or abusing drugs. AI systems need not learn to adopt these problematic tendencies, but they must understand that they exist when interpreting human behavior to get at the underlying human preferences.

Despite this toolbox of safeguards, there is a fear, expressed by prominent technologists such as Bill Gates and Elon Musk and scientists such as Stephen Hawking and Martin Rees, that AI could evolve out of control. They warn that we have no experience controlling powerful nonhuman entities with super-human capabilities. However, that's not quite true; we have centuries of experience with nations and corporations; non-human entities that aggregate the power of thousands or millions of people. Our record of controlling these entities is not very encouraging: nations produce periodic convulsions called wars that kill tens of millions of human beings, and corporations are partly responsible for global warming and our inability to confront it.

AI systems may present much greater problems than nations and corporations because of their potential to self-improve at a rapid pace, as considered by I. J. Good (1965b):

> Let an **ultraintelligent machine** be defined as a machine that can far surpass all the intellectual activities of any man however clever. Since the design of machines is one of these intellectual activities, an ultraintelligent machine could design even better machines; there would then unquestionably be an "intelligence explosion," and the intelligence of man would be left far behind. Thus the first ultraintelligent machine is the last invention that man need ever make, provided that the machine is docile enough to tell us how to keep it under control.

Good's "intelligence explosion" has also been called the **technological singularity** by mathematics professor and science fiction author Vernor Vinge, who wrote in 1993: "Within thirty years, we will have the technological means to create superhuman intelligence. Shortly after, the human era will be ended." In 2017, inventor and futurist Ray Kurzweil predicted the singularity would appear by 2045, which means it got 2 years closer in 24 years. (At that rate, only 336 years to go!) Vinge and Kurzweil correctly note that technological progress on many measures is growing exponentially at present.

It is, however, quite a leap to extrapolate all the way from the rapidly decreasing cost of computation to a singularity. So far, every technology has followed an S-shaped curve, where the exponential growth eventually tapers off. Sometimes new technologies step in when the old ones plateau, but sometimes it is not possible to keep the growth going, for technical, political, or sociological reasons. For example, the technology of flight advanced dramatically from the Wright brothers' flight in 1903 to the moon landing in 1969, but has had no breakthroughs of comparable magnitude since then.

Another obstacle in the way of ultraintelligent machines taking over the world is the world. More specifically, some kinds of progress require not just thinking but acting in the physical world. (Kevin Kelly calls the overemphasis on pure intelligence **thinkism**.) An ultraintelligent machine tasked with creating a grand unified theory of physics might be capable of cleverly manipulating equations a billion times faster than Einstein, but to make any real progress, it would still need to raise millions of dollars to build a more powerful supercollider and run physical experiments over the course of months or years. Only then could it start analyzing the data and theorizing. Depending on how the data turn out, the next step might require raising additional billions of dollars for an interstellar probe mission that would take centuries to complete. The "ultraintelligent thinking" part of this whole process might actually be the least important part. As another example, an ultraintelligent machine tasked with bringing peace to the Middle East might just end up getting 1000 times more frustrated than a human envoy. As yet, we don't know how many of the big problems are like mathematics and how many are like the Middle East.

While some people fear the singularity, others relish it. The **transhumanism** social movement looks forward to a future in which humans are merged with—or replaced by—robotic and biotech inventions. Ray Kurzweil writes in *The Singularity is Near* (2005):

> The Singularity will allow us to transcend these limitations of our biological bodies and brain. We will gain power over our fates. ... We will be able to live as long as we want ... We will fully understand human thinking and will vastly extend and expand its reach. By the end of this century, the nonbiological portion of our intelligence will be trillions of trillions of times more powerful than unaided human intelligence.

Similarly, when asked whether robots will inherit the Earth, Marvin Minsky said “yes, but they will be our children.” These possibilities present a challenge for most moral theorists, who take the preservation of human life and the human species to be a good thing. Kurzweil also notes the potential dangers, writing “But the Singularity will also amplify the ability to act on our destructive inclinations, so its full story has not yet been written.” We humans would do well to make sure that any intelligent machine we design today that might evolve into an ultraintelligent machine will do so in a way that ends up treating us well. As Eric Brynjolfsson puts it, “The future is not preordained by machines. It’s created by humans.”

Summary

This chapter has addressed the following issues:

- Philosophers use the term **weak AI** for the hypothesis that machines could possibly behave intelligently, and **strong AI** for the hypothesis that such machines would count as having actual minds (as opposed to simulated minds).
- Alan Turing rejected the question "Can machines think?" and replaced it with a behavioral test. He anticipated many objections to the possibility of thinking machines. Few AI researchers pay attention to the Turing test, preferring to concentrate on their systems' performance on practical tasks, rather than the ability to imitate humans.
- Consciousness remains a mystery.
- AI is a powerful technology, and as such it poses potential dangers, through lethal autonomous weapons, security and privacy breaches, unintended side effects, unintentional errors, and malignant misuse. Those who work with AI technology have an ethical imperative to responsibly reduce those dangers.
- AI systems must be able to demonstrate they are fair, trustworthy, and transparent.
- There are multiple aspects of fairness, and it is impossible to maximize all of them at once. So a first step is to decide what counts as fair.
- Automation is already changing the way people work. As a society, we will have to deal with these changes.

Bibliographical and Historical Notes

Weak AI: When Alan Turing (1950) proposed the possibility of AI, he also posed many of the key philosophical questions, and provided possible replies. But various philosophers had raised similar issues long before AI was invented. Maurice Merleau-Ponty's *Phenomenology of Perception* (1945) stressed the importance of the body and the subjective interpretation of reality afforded by our senses, and Martin Heidegger's *Being and Time* (1927) asked what it means to actually be an agent. In the computer age, Alva Noe (2009) and Andy Clark (2015) propose that our brains form a rather minimal representation of the world, use the world itself on a just-in-time basis to maintain the illusion of a detailed internal model, and use props in the world (such as paper and pencil as well as computers) to increase the capabilities of the mind. Pfeifer *et al.* (2006) and Lakoff and Johnson (1999) present arguments for how the body helps shape cognition. Speaking of bodies, Levy (2008), Danaher and McArthur (2017), and Devlin (2018) address the issue of robot sex.

Strong AI: Rene Descartes is known for his dualistic view of the human mind, but ironically his historical influence was toward mechanism and physicalism. He explicitly conceived of animals as automata, and he anticipated the Turing test, writing "it is not conceivable [that a machine] should produce different arrangements of words so as to give an appropriately meaningful answer to whatever is said in its presence, as even the dullest of men can do" (Descartes, 1637). Descartes's spirited defense of the animals-as-automata viewpoint actually had the effect of making it easier to conceive of humans as automata as well, even though he himself did not take this step. The book *L'Homme Machine* (La Mettrie, 1748) did explicitly argue that humans are automata. As far back as Homer (circa 700 BCE), the Greek legends envisioned automata such as the bronze giant Talos and considered the issue of *biotechne*, or life through craft (Mayor, 2018).

The **Turing test** (Turing, 1950) has been debated (Shieber, 2004), anthologized (Epstein *et al.*, 2008), and criticized (Shieber, 1994; Ford and Hayes, 1995). Bringsjord (2008) gives advice for a Turing test judge, and Christian (2011) for a human contestant. The annual Loebner Prize competition is the longest-running Turing test-like contest; Steve Worswick's MITSUKU won four in a row from 2016 to 2019. The **Chinese room** has been debated endlessly (Searle, 1980; Chalmers, 1992; Preston and Bishop, 2002). Hernández-Orallo (2016) gives an overview of approaches to measuring AI progress, and Chollet (2019) proposes a measure of intelligence based on skill-acquisition efficiency.

Consciousness remains a vexing problem for philosophers, neuroscientists, and anyone who has pondered their own existence. Block (2009), Churchland (2013) and Dehaene (2014) provide overviews of the major theories. Crick and Koch (2003) add their expertise in biology and neuroscience to the debate, and Gazzaniga (2018) shows what can be learned from studying brain disabilities in hospital cases. Koch (2019) gives a theory of consciousness—"intelligence is about doing while experience is about being"—that includes most animals, but not computers. Giulio Tononi and his colleagues propose **integrated information theory** (Oizumi *et al.*, 2014). Damasio (1999) has a theory based on three levels: emotion, feeling, and feeling a feeling. Bryson (2012) shows the value of conscious attention for the process of learning action selection.

The philosophical literature on minds, brains, and related topics is large and jargon-filled. The *Encyclopedia of Philosophy* (Edwards, 1967) is an impressively authoritative and very useful navigation aid. *The Cambridge Dictionary of Philosophy* (Audi, 1999) is shorter and more accessible, and the online *Stanford Encyclopedia of Philosophy* offers many excellent articles and up-to-date references. The *MIT Encyclopedia of Cognitive Science* (Wilson and Keil, 1999) covers the philosophy, biology, and psychology of mind. There are multiple introductions to the philosophical “AI question” (Haugeland, 1985; Boden, 1990; Copeland, 1993; McCorduck, 2004; Minsky, 2007). *The Behavioral and Brain Sciences,* abbreviated *BBS,* is a major journal devoted to philosophical and scientific debates about AI and neuroscience.

Science fiction writer Isaac Asimov (1942, 1950) was one of the first to address the issue of robot ethics, with his **laws of robotics**:

0. A robot may not harm humanity, or through inaction, allow humanity to come to harm.
1. A robot may not injure a human being or, through inaction, allow a human being to come to harm.
2. A robot must obey orders given to it by human beings, except where such orders would conflict with the First Law.
3. A robot must protect its own existence as long as such protection does not conflict with the First or Second Law.

At first glance, these laws seem reasonable. But the trick is how to implement them. Should a robot allow a human to cross the street, or eat junk food, if the human might conceivably come to harm? In Asimov’s story *Runaround* (1942), humans need to debug a robot that is found wandering in a circle, acting “drunk.” They work out that the circle defines the locus of points that balance the second law (the robot was ordered to fetch some selenium at the center of the circle) with the third law (there is a danger there that threatens the robot’s existence).[4] This suggests that the laws are not logical absolutes, but rather are weighed against each other, with a higher weight for the earlier laws. As this was 1942, before the emergence of digital computers, Asimov was probably thinking of an architecture based on control theory via analog computing.

Weld and Etzioni (1994) analyze Asimov's laws and suggest some ways to modify the planning techniques of Chapter 11 to generate plans that do no harm. Asimov has considered many of the ethical issues around technology; in his 1958 story *The Feeling of Power* he tackles the issue of automation leading to a lapse of human skill—a technician rediscovers the lost art of multiplication—as well as the dilemma of what to do when the rediscovery is applied to warfare.

Norbert Wiener's book *God & Golem, Inc.* (1964) correctly predicted that computers would achieve expert-level performance at games and other tasks, and that specifying what it is that we want would prove to be difficult. Wiener writes:

> While it is always possible to ask for something other than we really want, this possibility is most serious when the process by which we are to obtain our wish is indirect, and the degree to which we have obtained our wish is not clear until the very end. Usually we realize our wishes, insofar as we do actually realize them, by a feedback process, in which we compare the degree of attainment of intermediate goals with our anticipation of them. In this process, the feedback goes through us, and we can turn back before it is too late. If the feedback is built into a machine that cannot be inspected until the final goal is attained, the possibilities for catastrophe are greatly increased. I should very much hate to ride on the first trial of an automobile regulated by photoelectric feedback devices, unless there were somewhere a handle by which

I could take over control if I found myself driving smack into a tree.

We summarized **codes of ethics** in the chapter, but the list of organizations that have issued sets of principles is growing rapidly, and now includes Apple, DeepMind, Facebook, Google, IBM, Microsoft, the Organisation for Economic Co-operation and Development (OECD), the United Nations Educational, Scientific and Cultural Organization (UNESCO), the U.S. Office of Science and Technology Policy the Beijing Academy of Artificial Intelligence (BAAI), the Institute of Electrical and Electronics Engineers (IEEE), the Association of Computing Machinery (ACM), the World Economic Forum, the Group of Twenty (G20), OpenAI, the Machine Intelligence Research Institute (MIRI), AI4People, the Centre for the Study of Existential Risk, the Center for Human-Compatible AI, the Center for Humane Technology, the Partnership on AI, the AI Now Institute, the Future of Life Institute, the Future of Humanity Institute, the European Union, and at least 42 national governments. We have the handbook on the *Ethics of Computing* (Berleur and Brunnstein, 2001) and introductions to the topic of AI ethics in book (Boddington, 2017) and survey (Etzioni and Etzioni, 2017a) form. The *Journal of Artificial Intelligence and Law* and *AI and Society* cover ethical issues. We'll now look at some of the individual issues.

Lethal autonomous weapons: P. W. Singer's *Wired for War* (2009) raised ethical, legal, and technical issues around robots on the battlefield. Paul Scharre's *Army of None* (2018), written by one of the authors of current US policy on autonomous weapons, offers a balanced and authoritative view. Etzioni and Etzioni (2017b) address the question of whether artificial intelligence should be regulated; they recommend a pause in the development of lethal autonomous weapons, and an international discussion on the subject of regulation.

Privacy: Latanya Sweeney (Sweeney, 2002b) presents the *k*-anonymity model and the idea of generalizing fields (Sweeney, 2002a). Achieving *k*-anonymity with minimal loss of data is an NP-hard problem, but Bayardo and Agrawal (2005) give an approximation algorithm. Cynthia Dwork (2008) describes differential privacy, and in subsequent work gives practical examples of clever ways to apply differential privacy to get better results than the naive approach (Dwork *et al.,* 2014). Guo *et al.* (2019) describe a process for certified data removal: if you train a model on some data, and then there is a request to delete some of the data, this extension of differential privacy lets you modify the model and prove that it does not make use of the deleted data. **Ji** *et al.* (2014) gives a review of the field of privacy. Etzioni (2004) argues for a balancing of privacy and security; individual rights and community. Fung *et al.* (2018), Bagdasaryan *et al.* (2018) discuss the various attacks on federated learning protocols. Narayanan *et al.* (2011) describe how they were able to de-anonymize the obfuscated connection graph from the 2011 Social Network Challenge by crawling the site where the data was obtained (Flickr), and matching nodes with unusually high in-degree or out-degree between the provided data and the crawled data. This allowed them to gain additional information to win the challenge, and it also allowed them to uncover the true identity of nodes in the data. Tools for user privacy are becoming available; for example, TensorFlow provides modules for federated learning and privacy (McMahan and Andrew, 2018).

Fairness: Cathy O'Neil's book *Weapons of Math Destruction* (2017) describes how various black box machine learning models influence our lives, often in unfair ways. She calls on model builders to take responsibility for fairness, and for policy makers to impose appropriate regulation. Dwork *et al.* (2012) showed the flaws with the simplistic “fairness through unawareness” approach. Bellamy *et al.* (2018) present a toolkit for mitigating bias in machine learning systems. Tramèr *et al.* (2016) show how an adversary can “steal” a machine learning model by making queries against an API, Hardt *et al.* (2017) describe equal opportunity as a metric for fairness. Chouldechova and Roth (2018) give an overview of the frontiers of fairness, and Verma and Rubin (2018) give an exhaustive survey of fairness definitions.

Kleinberg *et al.* (2016) show that, in general, an algorithm cannot be both well-calibrated and equal opportunity. Berk *et al.* (2017) give some additional definitions of types of fairness, and again conclude that it is impossible to satisfy all aspects at once. Beutel *et al.* (2019) give advice for how to put fairness metrics into practice.

Dressel and Farid (2018) report on the COMPAS recidivism scoring model. Christin *et al.* (2015) and Eckhouse *et al.* (2019) discuss the use of predictive algorithms in the legal system. Corbett-Davies *et al.* (2017) show that that there is a tension between ensuring fairness and optimizing public safety, and Corbett-Davies and Goel (2018) discuss the differences between fairness frameworks. Chouldechova (2017) advocates for fair impact: all classes should have the same expected utility. Liu *et al.* (2018a) advocate for a long-term measure of impact, pointing out that, for example, if we change the decision point for approving a loan in order to be more fair in the short run, this could have negative effect in the long run on people who end up defaulting on a loan and thus have their credit score reduced.

Since 2014 there has been an annual conference on Fairness, Accountability, and Transparency in Machine Learning. Mehrabi *et al.*

(2019) give a comprehensive survey of bias and fairness in machine learning, cataloging 23 kinds of bias and 10 definitions of fairness.

Trust: Explainable AI was an important topic going back to the early days of expert systems (Neches *et al.*, 1985), and has been making a resurgence in recent years (Biran and Cotton, 2017; Miller *et al.*, 2017; Kim, 2018). Barreno *et al.* (2010) give a taxonomy of the types of security attacks that can be made against a machine learning system, and Tygar (2011) surveys adversarial machine learning. Researchers at IBM have a proposal for gaining trust in AI systems through declarations of conformity (Hind *et al.*, 2018). DARPA requires explainable decisions for its battlefield systems, and has issued a call for research in the area (Gunning, 2016).

AI safety: The book *Artificial Intelligence Safety and Security* (Yampolskiy, 2018) collects essays on AI safety, both recent and classic, going back to Bill Joy's *Why the Future Doesn't Need Us* (Joy, 2000). The "King Midas problem" was anticipated by Marvin Minsky, who once suggested that an AI program designed to solve the Riemann Hypothesis might end up taking over all the resources of Earth to build more powerful supercomputers. Similarly, Omohundro (2008) foresees a chess program that hijacks resources, and Bostrom (2014) describes the runaway paper clip factory that takes over the world. Yudkowsky (2008) goes into more detail about how to design a **Friendly AI**. Amodei *et al.* (2016) present five practical safety problems for AI systems.

Omohundro (2008) describes the *Basic AI Drives* and concludes, "Social structures which cause individuals to bear the cost of their negative externalities would go a long way toward ensuring a stable and positive future." Elinor Ostrom's *Governing the Commons* (1990) describes practices for dealing with externalities by traditional cultures. Ostrom has also applied this approach to the idea of knowledge as a commons (Hess and Ostrom, 2007).

Ray Kurzweil (2005) proclaimed *The Singularity is Near,* and a decade later Murray Shanahan (2015) gave an update on the topic. Microsoft cofounder Paul Allen countered with *The Singularity isn't Near* (2011). He didn't dispute the possibility of ultraintelligent machines; he just thought it would take more than a century to get there. Rod Brooks is a frequent critic of singularitarianism; he points out that technologies often take longer than predicted to mature, that we are prone to magical thinking, and that exponentials don't last forever (Brooks, 2017).

On the other hand, for every optimistic singularitarian there is a pessimist who fears new technology. The Web site `pessimists.co` shows that this has been true throughout history: for example, in the 1890s people were concerned that the elevator would inevitably cause nausea, that the telegraph would lead to loss of privacy and moral corruption, that the subway would release dangerous underground air and disturb the dead, and that the bicycle—especially the idea of a woman riding one—was the work of the devil.

Hans Moravec (2000) introduces some of the ideas of transhumanism, and Bostrom (2005) gives an updated history. Good's ultraintelligent machine idea was foreseen a hundred years earlier in Samuel Butler's *Darwin Among the Machines* (1863). Written four years after the publication of Charles Darwin's *On the Origins of Species* and at a time when the most sophisticated machines were steam engines, Butler's article envisioned "the ultimate development of mechanical consciousness" by natural selection. The theme was reiterated by George Dyson (1998) in a book of the same title, and was referenced by Alan Turing, who wrote in 1951 "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"* (Turing, 1996).

Robot rights: A book edited by Yorick Wilks (2010) gives different perspectives on how we should deal with artificial companions, ranging from Joanna Bryson's view that robots should serve us as tools, not as citizens, to Sherry Turkle's observation that we already personify our computers and other tools, and are quite willing to blur the boundaries between machines and life. Wilks also contributed a recent update on his views (Wilks, 2019). The philosopher David Gunkel's book *Robot Rights* (2018) considers four possibilities: *can* robots have rights or not, and *should* they or not? The American Society for the Prevention of Cruelty to Robots (ASPCR) proclaims that "The ASPCR is, and will continue to be, exactly as serious as robots are sentient."

The future of work: In 1888, Edward Bellamy published the best-seller *Looking Backward*, which predicted that by the year 2000, technological advances would led to a utopia where equality is achieved and people work short hours and retire early. Soon after, E. M. Forster took the dystopian view in *The Machine Stops* (1909), in which a benevolent machine takes over the running of a society; things fall apart when the machine inevitably fails. Norbert Wiener's prescient book *The Human Use of Human Beings* (1950) argues for the benefits of automation in freeing people from drudgery while offering more creative work, but also discusses several dangers that we recognize as problems today, particularly the problem of value alignment.

The book *Disrupting Unemployment* (Nordfors *et al.*, 2018) discuss some of the ways that work is changing, opening opportunities for new careers. Erik Brynjolfsson and Andrew McAfee address these themes and more in their books *Race Against the Machine* (2011) and *The Second*

Machine Age (2014). Ford (2015) describes the challenges of increasing automation, and West (2018) provides recommendations to mitigate the problems, while MIT's Thomas Malone (2004) shows that many of the same issues were apparent a decade earlier, but at that time were attributed to worldwide communication networks, not to automation.

[1] For example, the opera Coppélia (1870), the novel *Do Androids Dream of Electric Sheep?* (1968), the movies *AI* (2001), *Wall-E* (2008), and *Her* (2013).

[2] Including the two authors of this book.

[3] In 2010, although only 2% of the U.S. workforce were actual farmers, over 25% of the population (80 million people) played the FARMVILLE game at least once.

[4] Science fiction writers are in broad agreement that robots are very bad at resolving contradictions. In *2001*, the HAL 9000 computer becomes homicidal due to a conflict in its orders, and in the *Star Trek* episode "I, Mudd," Captain Kirk tells an enemy robot that "Everything Harry tells you is a lie," and Harry says "I am lying." At this, smoke comes out of the robot's head and it shuts down.

CHAPTER 29

THE FUTURE OF AI

In which we try to see a short distance ahead.

In Chapter 2, we decided to view AI as the task of designing approximately rational agents. A variety of different agent designs were considered, ranging from reflex agents to knowledge-based decision-theoretic agents to deep learning agents using reinforcement learning. There is also variety in the component technologies from which these designs are assembled: logical, probabilistic, or neural reasoning; atomic, factored, or structured representations of states; various learning algorithms from various types of data; sensors and actuators to interact with the world. Finally, we have seen a variety of applications, in medicine, finance, transportation, communication, and other fields. There has been progress on all these fronts, both in our scientific understanding and in our technological capabilities.

Most experts are optimistic about continued progress; as we saw on page 46, the median estimate is for approximately human-level AI across a broad variety of tasks somewhere in the next 50 to 100 years. Within the next decade, AI is predicted to add trillions of dollars to the economy each year. But as we also saw, there are some critics who think general AI is centuries off, and there are numerous ethical concerns about the fairness, equity, and lethality of AI. In this chapter, we ask: where are we headed and what remains to be done? We do that by asking whether we have the right components, architectures, and goals to make AI a successful technology that delivers benefits to the world.

29.1 AI Components

This section examines the components of AI systems and the extent to which each of them might accelerate or hinder future progress.

Sensors and actuators

For much of the history of AI, direct access to the world has been glaringly absent. With a few notable exceptions, AI systems were built in such a way that humans had to supply the inputs and interpret the outputs. Meanwhile, robotic systems focused on low-level tasks in which high-level reasoning and planning were largely ignored and the need for perception was minimized. This was partly due to the great expense and engineering effort required to get real robots to work at all, and partly because of the lack of sufficient processing power and sufficiently effective algorithms to handle high-bandwidth visual input.

The situation has changed rapidly in recent years with the availability of ready-made programmable robots. These, in turn, have benefited from compact reliable motor drives and improved sensors. The cost of lidar for a self-driving car has fallen from $75,000 to $1,000, and a single-chip version may reach $10 per unit (Poulton and Watts, 2016). Radar sensors, once capable of only coarse-grained detection, are now sensitive enough to count the number of sheets in a stack of paper (Yeo *et al.*, 2018).

The demand for better image processing in cellphone cameras has given us inexpensive high-resolution cameras for use in robotics. MEMS (micro-electromechanical systems) technology has supplied miniaturized accelerometers, gyroscopes, and actuators small enough to fit in artificial flying insects (Floreano *et al.*, 2009; Fuller *et al.*, 2014). It may be possible to combine millions of MEMS devices to produce powerful macroscopic actuators. 3-D printing (Muth *et al.*, 2014) and bioprinting (Kolesky *et al.*, 2014) have made it easier to experiment with prototypes.

Thus, we see that AI systems are at the cusp of moving from primarily software-only systems to useful embedded robotic systems. The state of robotics today is roughly comparable to the state of personal computers in the early 1980s: at that time personal computers were becoming available, but it would take another decade before they became commonplace. It is likely that flexible, intelligent robots will first make strides in industry (where environments are more controlled, tasks are more repetitive, and the value of an investment is easier to measure) before the home market (where there is more variability in environment and tasks).

Representing the state of the world

Keeping track of the world requires perception as well as updating of internal representations. Chapter 4 showed how to keep track of atomic state representations; Chapter 7 described how to do it for factored (propositional) state representations; Chapter 10 extended this to first-order logic; and Chapter 14 described probabilistic reasoning over time in uncertain environments. Chapter 22 introduced recurrent neural networks, which are also capable of maintaining a state representation over time.

Current filtering and perception algorithms can be combined to do a reasonable job of recognizing objects ("that's a cat") and reporting low-level predicates ("the cup is on the table"). Recognizing higher-level actions, such as "Dr. Russell is having a cup of tea with Dr. Norvig while discussing plans for next week," is more difficult. Currently it can sometimes be done (see Figure 27.17 on page 1015) given enough training examples, but future progress will require techniques that generalize to novel situations without requiring exhaustive examples (Poppe, 2010; Kang and Wildes, 2016).

Another problem is that although the approximate filtering algorithms from Chapter 14 can handle quite large environments, they are still dealing with a factored representation—they have random variables, but do not represent objects and relations explicitly. Also, their notion of time is restricted to step-by-step change; given the recent trajectory of a ball, we can predict where it will be at time $t + 1$, but it is difficult to represent the abstract idea that what goes up must come down.

Section 18.1 explained how probability and first-order logic can be combined to solve these problems; Section 18.2 showed how we can handle uncertainty about the identity of objects; and Chapter 27 showed how recurrent neural networks enable computer vision to track the world; but we don't yet have a good way of putting all these techniques together. Chapter 25 showed how word embeddings and similar representations can free us from the strict bounds of concepts defined by necessary and sufficient conditions. It remains a daunting task to define general, reusable representation schemes for complex domains.

Selecting actions

The primary difficulty in action selection in the real world is coping with long-term plans—such as graduating from college in four years—that consist of billions of primitive steps. Search algorithms that consider sequences of primitive actions scale only to tens or perhaps hundreds of steps. It is only by imposing **hierarchical structure** on behavior that we humans cope at all. We saw in Section 11.4 how to use hierarchical representations to handle problems of this scale; furthermore, work in **hierarchical reinforcement learning** has succeeded in combining these ideas with the MDP formalism described in Chapter 16.

As yet, these methods have not been extended to the partially observable case (POMDPs). Moreover, algorithms for solving POMDPs are typically using the same atomic state representation we used for the search algorithms of Chapter 3. There is clearly a great deal of work to do here, but the technical foundations are largely in place for making progress. The main missing element is an effective method for *constructing* the hierarchical representations of state and behavior that are necessary for decision making over long time scales.

Deciding what we want

Chapter 3 introduced search algorithms to find a goal state. But goal-based agents are brittle when the environment is uncertain, and when there are multiple factors to consider. In principle, utility-maximization agents address those issues in a completely general way. The fields of economics and game theory, as well as AI, make use of this insight: just declare what you want to optimize, and what each action does, and we can compute the optimal action.

In practice, however, we now realize that the task of picking the right utility function is a challenging problem in its own right. Imagine, for example, the complex web of interacting preferences that must be understood by an agent operating as an office assistant for a human being. The problem is exacerbated by the fact that each human is different, so an agent just "out of the box" will not have enough experience with any one individual to learn an accurate preference model; it will necessarily need to operate under preference uncertainty. Further complexity arises if we want to ensure that our agents are acting in a way that is fair and equitable for society, rather than just one individual.

We do not yet have much experience with building complex real-world preference models, let alone probability distributions over such models. Although there are factored formalisms, similar to Bayes nets, that are intended to decompose preferences over complex states, it has proven difficult to use these formalisms in practice. One reason may be that preferences over states are really *compiled* from preferences over state histories, which are described by **reward functions** (see Chapter 16). Even if the reward function is simple, the corresponding utility function may be very complex.

This suggests that we take seriously the task of knowledge engineering for reward functions as a way of conveying to our agents what we want them to do. The idea of **inverse reinforcement learning** (Section 23.6) is one approach to this problem when we have an expert who can perform a task, but not explain it. We could also use better languages for expressing what we want. For example, in robotics, linear temporal logic makes it easier to say what things we want to happen in the near future, what things we want to avoid, and what states we want to persist forever (Littman *et al.*, 2017). We need better ways of saying what we want and better ways for robots to interpret the information we provide.

The computer industry as a whole has developed a powerful ecosystem for aggregating user preferences. When you click on something in an app, online game, social network, or shopping site, that serves as a recommendation that you (and your similar peers) would like to see similar things in the future. (Or it might be that the site is confusing and you clicked on the wrong thing—the data are always noisy.) The feedback inherent in this system makes it very effective in the short run for picking out ever more addictive games and videos.

But these systems often fail to provide an easy way of opting out—your device will autoplay a relevant video, but it is less likely to tell you "maybe it is time to put away your devices and take a relaxing walk in nature." A shopping site will help you find clothes that match your style, but will not address world peace or ending hunger and poverty. To the extent that the menu of choices is driven by companies trying to profit from a customer's attention, the menu will remain incomplete.

However, companies do respond to customers' interests, and many customers have voiced the opinion that they are interested in a fair and sustainable world. Tim O'Reilly explains why profit is not the only motive with the following analogy: "Money is like gasoline during a road trip. You don't want to run out of gas on your trip, but you're not doing a tour of gas stations. You have to pay attention to money, but it shouldn't be about the money."

Tristan Harris's **time well spent** movement at the Center for Humane Technology is a step towards giving us more well-rounded choices (Harris, 2016). The movement addresses an issue that was recognized by Herbert Simon in 1971: "A wealth of information creates a poverty of attention." Perhaps in the future we will have **personal agents** that stick up for our true long-term interests rather than the interests of the corporations whose apps currently fill our devices. It will be the agent's job to mediate the offerings of various vendors, protect us from addictive attention-grabbers, and guide us towards the goals that really matter to us.

Learning

Chapters 19, 21, 22, and 23 described how agents can learn. Current algorithms can cope with quite large problems, reaching or exceeding human capabilities in many tasks—as long as we have sufficient training examples and we are dealing with a predefined vocabulary of features and concepts. But learning can stall when data are sparse, or unsupervised, or when we are dealing with complex representations.

Much of the recent resurgence of AI in the popular press and in industry is due to the success of deep learning (Chapter 22). On the one hand, this can be seen as the incremental maturation of the subfield of neural networks. On the other hand, we can see it as a revolutionary leap in capabilities spurred by a confluence of factors: the availability of more training data thanks to the Internet, increased processing power from specialized hardware, and a few algorithmic tricks, such as generative adversarial networks (GANs), batch normalization, dropout, and the rectified linear (ReLU) activation function.

The future should see continued emphasis on improving deep learning for the tasks it excels at, and also extending it to cover other tasks. The brand name "deep learning" has proven to be so popular that we should expect its use to continue, even if the mix of techniques that fuel it changes considerably.

We have seen the emergence of the field of **data science** as the confluence of statistics, programming, and domain expertise. While we can expect to see continued development in the tools and techniques necessary to acquire, manage, and maintain **big data**, we will also need advances in **transfer learning** so that we can take advantage of data in one domain to improve performance on a related domain.

The vast majority of machine learning research today assumes a factored representation, learning a function $h : \mathbb{R}^n \to \mathbb{R}$ for regression and $h : \mathbb{R}^n \to \{0, 1\}$ for classification. Machine learning has been less successful for problems that have only a small amount of data, or problems that require the construction of new structured, hierarchical representations. Deep learning, especially with convolutional networks applied to computer vision problems, has demonstrated some success in going from low-level pixels to intermediate-level concepts like *Eye* and *Mouth,* then to *Face,* and finally to *Person* or *Cat.*

A challenge for the future is to more smoothly combine learning and prior knowledge. If we give a computer a problem it has not encountered before—say, recognizing different models of cars—we don’t want the system to be powerless until it has been fed millions of labeled examples.

The ideal system should be able to draw on what it already knows: it should already have a model of how vision works, and how the design and branding of products in general work; now it should use **transfer learning** to apply that to the new problem of car models. It should be able to find on its own information about car models, drawing from text, images, and video available on the Internet. It should be capable of **apprenticeship learning**: having a conversation with a teacher, and not just asking “may I have a thousand images of a Corolla,” but rather being able to understand advice like “the Insight is similar to the Prius, but the Insight has a larger grille.” It should know that each model comes in a small range of possible colors, but that a car can be repainted, so there is a chance that it might see a car in a color that was not in the training set. (If it didn’t know that, it should be capable of learning it, or being told about it.)

All this requires a communication and representation language that humans and computers can share; we can’t expect a human analyst to directly modify a model with millions of weights. Probabilistic models (including probabilistic programming languages) give humans some ability to describe what we know, but these models are not yet well integrated with other learning mechanisms.

The work of Bengio and LeCun (2007) is one step towards this integration. Recently Yann LeCun has suggested that the term “deep learning” should be replaced with the more general **differentiable programming** (Siskind and Pearlmutter, 2016; Li *et al.*, 2018); this suggests that our general programming languages and our machine learning models could be merged together.

Right now, it is common to build a deep learning model that is differentiable, and thus can be trained to minimize loss, and retrained when circumstances change. But that deep learning model is only one part of a larger software system that takes in data, massages the data, feeds it to the model, and figures out what to do with the model's output. All these other parts of the larger system were written by hand by a programmer, and thus are nondifferentiable, which means that when circumstances change, it is up to the programmer to recognize any problems and fix them by hand. With differentiable programming, the hope is that the entire system is subject to automated optimization.

The end goal is to be able to express what we know in whatever form is convenient to us: informal advice given in natural language, a strong mathematical law like $F = ma$, a statistical model accompanied by data, or a probabilistic program with unknown parameters that can be automatically optimized through gradient descent. Our computer models will learn from conversations with human experts as well as by using all the available data.

Yann LeCun, Geoffrey Hinton, and others have suggested that the current emphasis on supervised learning (and to a lesser extent reinforcement learning) is not sustainable—that computer models will have to rely on **weakly supervised learning**, in which some supervision is given with a small number of labeled examples and/or a small number of rewards, but most of the learning is unsupervised, because unannotated data are so much more plentiful.

LeCun uses the term **predictive learning** for an unsupervised learning system that can model the world and learn to predict aspects of future states of the world—not just predict labels for inputs that are independent and identically distributed with respect to past data, and not just predict a value function over states. He suggests that GANs (generative adversarial networks) can be used to learn to minimize the difference between predictions and reality.

Geoffrey Hinton stated in 2017 that “My view is throw it all away and start again,” meaning that the overall idea of learning by adjusting parameters in a network is enduring, but the specifics of the architecture of the networks and the technique of back-propagation need to be rethought. Smolensky (1988) had a prescription for how to think about connectionist models; his thoughts remain relevant today.

Resources

Machine learning research and development has been accelerated by the increasing availability of data, storage, processing power, software, trained experts, and the investments needed to support them. Since the 1970s, there has been a 100,000-fold speedup in general-purpose processors and an additional 1,000-fold speedup due to specialized machine learning hardware. The Web has served as a rich source of images, videos, speech, text, and semi-structured data, currently adding over 10^{18} bytes every day.

Hundreds of high-quality data sets are available for a range of tasks in computer vision, speech recognition, and natural language processing. If the data you need is not already available, you can often assemble it from other sources, or engage humans to label data for you through a crowdsourcing platform. Validating the data obtained in this way becomes an important part of the overall workflow (Hirth *et al.*, 2013).

An important recent development is the shift from shared data to **shared models**. The major cloud service providers (e.g., Amazon, Microsoft, Google, Alibaba, IBM, Salesforce) have begun competing to offer machine learning APIs with pre-built models for specific tasks such as visual object recognition, speech recognition, and machine translation.

These models can be used as is, or can serve as a baseline to be customized with your particular data for your particular application.

We expect that these models will improve over time, and that it will become unusual to start a machine learning project from scratch, just as it is now unusual to do a Web development project from scratch, with no libraries. It is possible that a big jump in model quality will occur when it becomes economical to process all the video on the Web; for example, the YouTube platform alone adds 300 hours of video every minute.

Moore’s law has made it more cost effective to process data; a megabyte of storage cost $1 million in 1969 and less then $0.02 in 2019, and supercomputer throughput has increased by a factor of more than 10^{10} in that time. Specialized hardware components for machine learning such as graphics processing units (GPUs), tensor cores, tensor processing units (TPUs), and field programmable gate arrays (FPGAs) are hundreds of times faster than conventional CPUs for machine learning training (Vasilache *et al.,* 2014; Jouppi *et al.,* 2017). In 2014 it took a full day to train an ImageNet model; in 2018 it takes just 2 minutes (Ying *et al.,* 2018).

The OpenAI Institute reports that the amount of compute power used to train the largest machine learning models doubled every 3.5 months from 2012 to 2018, reaching over an exaflop/second-day for ALPHAZERO (although they also report that some very influential work used 100 million times less computing power (Amodei and Hernandez, 2018)). The same economic trends that have made cell-phone cameras cheaper and better also apply to processors—we will see continued progress in low-power, high-performance computing that benefits from economies of scale.

There is a possibility that quantum computers could accelerate AI. Currently there are some fast quantum algorithms for the linear algebra operations used in machine learning (Harrow *et al.,* 2009; Dervovic *et al.,* 2018), but no quantum computer capable of running them. We have some example applications of tasks such as image classification (Mott *et al*., 2017) where quantum algorithms are as good as classical algorithms on small problems.

Current quantum computers handle only a few tens of bits, whereas machine learning algorithms often handle inputs with millions of bits and create models with hundreds of millions of parameters. So we need breakthroughs in both quantum hardware and software to make quantum computing practical for large-scale machine learning. Alternatively, there may be a division of labor—perhaps a quantum algorithm to efficiently search the space of hyperparameters while the normal training process runs on conventional computers—but we don't know how to do that yet. Research on quantum algorithms can sometimes inspire new and better algorithms on classical computers (Tang, 2018).

We have also seen exponential growth in the number of publications, people, and dollars in AI/machine learning/data science. Dean *et al.* (2018) show that the number of papers about "machine learning" on arXiv doubled every two years from 2009 to 2017. Investors are funding startup companies in these fields, large companies are hiring and spending as they determine their AI strategy, and governments are investing to make sure their country doesn't fall too far behind.

29.2 AI Architectures

It is natural to ask, “Which of the agent architectures in Chapter 2 should an agent use?” The answer is, “All of them!” Reflex responses are needed for situations in which time is of the essence, whereas knowledge-based deliberation allows the agent to plan ahead. Learning is convenient when we have lots of data, and necessary when the environment is changing, or when human designers have insufficient knowledge of the domain.

AI has long had a split between symbolic systems (based on logical and probabilistic inference) and connectionist systems (based on loss minimization over a large number of uninterpreted parameters). A continuing challenge for AI is to bring these two together, to capture the best of both. Symbolic systems allow us to string together long chains of reasoning and to take advantage of the expressive power of structured representations, while connectionist systems can recognize patterns even in the face of noisy data. One line of research aims to combine probabilistic programming with deep learning, although as yet the various proposals are limited in the extent to which the approaches are truly merged.

Agents also need ways to control their own deliberations. They must be able to use the available time well, and cease deliberating when action is demanded. For example, a taxidriving agent that sees an accident ahead must decide in a split second whether to brake or swerve. It should also spend that split second thinking about the most important questions, such as whether the lanes to the left and right are clear and whether there is a large truck close behind, rather than worrying about where to pick up the next passenger. These issues are usually studied under the heading of **real-time AI**. As AI systems move into more complex domains, all problems will become real-time, because the agent will never have long enough to solve the decision problem exactly.

Clearly, there is a pressing need for *general* methods of controlling deliberation, rather than specific recipes for what to think about in each situation. The first useful idea is the **anytime algorithms** (Dean and Boddy, 1988; Horvitz, 1987): an algorithm whose output quality improves gradually over time, so that it has a reasonable decision ready whenever it is interrupted. Examples of anytime algorithms include iterative deepening in game-tree search and MCMC in Bayesian networks.

The second technique for controlling deliberation is **decision-theoretic metareasoning** (Russell and Wefald, 1989; Horvitz and Breese, 1996; Hay *et al.*, 2012). This method, which was mentioned briefly in Sections 3.6.5 and 6.7, applies the theory of information value (Chapter 15) to the selection of individual computations (Section 3.6.5). The value of a computation depends on both its cost (in terms of delaying action) and its benefits (in terms of improved decision quality).

Metareasoning techniques can be used to design better search algorithms and to guarantee that the algorithms have the anytime property. Monte Carlo tree search is one example: the choice of leaf node at which to begin the next playout is made by an approximately rational metalevel decision derived from bandit theory.

Metareasoning is more expensive than reflex action, of course, but compilation methods can be applied so that the overhead is small compared to the costs of the computations being controlled. Metalevel reinforcement learning may provide another way to acquire effective policies for controlling deliberation: in essence, computations that lead to better decisions are reinforced, while those that turn out to have no effect are penalized. This approach avoids the myopia problems of the simple value-of-information calculation.

Metareasoning is one specific example of a **reflective architecture**—that is, an architecture that enables deliberation about the computational entities and actions occurring within the architecture itself. A theoretical foundation for reflective architectures can be built by defining a joint state space composed from the environment state and the computational state of the agent itself. Decision-making and learning algorithms can be designed that operate over this joint state space and thereby serve to implement and improve the agent's computational activities. Eventually, we expect task-specific algorithms such as alpha–beta search, regression planning, and variable elimination to disappear from AI systems, to be replaced by general methods that direct the agent's computations toward the efficient generation of high-quality decisions.

Metareasoning and reflection (and many other efficiency-related architectural and algorithmic devices explored in this book) are necessary because making decisions is *hard*. Ever since computers were invented, their blinding speed has led people to overestimate their ability to overcome complexity, or, equivalently, to underestimate what complexity really means. The truly gargantuan power of today's machines tempts one to think that we could bypass all the clever devices and rely more on brute force. So let's try to counteract this tendency. We begin with what physicists believe to be the speed of the ultimate 1kg computing device: about 10^{51} operations per second, or a billion trillion trillion times faster than the fastest supercomputer as of 2020 (Lloyd, 2000).[1] Then we propose a simple task: enumerating strings of English words, much as Borges proposed in *The Library of Babel*. Borges stipulated books of 410 pages. Would that be feasible? Not quite. In fact, the computer running for a year could enumerate only the 11-word strings.

Now consider the fact that a detailed plan for a human life consists of (very roughly) twenty trillion potential muscle actuations (Russell, 2019), and you begin to see the scale of the problem. A computer that is a billion trillion trillion times more powerful than the human brain is much further from being rational than a slug is from overtaking the starship Enterprise traveling at warp nine.

With these considerations in mind, it seems that the goal of building rational agents is perhaps a little too ambitious. Rather than aiming for something that cannot possibly exist, we should consider a different normative target—one that *necessarily* exists. Recall from Chapter 2 the following simple idea:

agent = *architecture* + *program.*

Now fix the agent architecture (the underlying machine capabilities, perhaps with a fixed software layer on top) and allow the agent program to vary over all possible programs that the architecture can support. In any given task environment, one of these programs (or an equivalence class of them) delivers the best possible performance—perhaps not close to perfect rationality, but still better than any other agent program. We say that this program satisfies the criterion of **bounded optimality**. Clearly it exists, and clearly it constitutes a desirable goal. The trick is finding it, or something close to it.

For some elementary classes of agent programs in simple real-time environments, it is possible to identify bounded-optimal agent programs (Etzioni, 1989; Russell and Subramanian, 1995). The success of Monte Carlo tree search has revived interest in metalevel decision making, and there is reason to hope that bounded optimality within more complex families of agent programs can be achieved by techniques such as metalevel reinforcement learning. It should also be possible to develop a constructive theory of architecture, beginning with theorems on the bounded optimality of suitable methods of combining different bounded-optimal components such as reflex and action–value systems.

General AI

Much of the progress in AI in the 21st century so far has been guided by competition on narrow tasks, such as the DARPA Grand Challenge for autonomous cars, the ImageNet object recognition competition, or playing Go, chess, poker, or Jeopardy! against a world champion. For each separate task, we build a separate AI system, usually with a separate machine learning model trained from scratch with data collected specifically for this task. But a truly intelligent agent should be able to do more than one thing. Alan Turing (1950) proposed his list (page 1033) and science fiction author Robert Heinlein (1973) countered with:

> A human being should be able to change a diaper, plan an invasion, butcher a hog, conn a ship, design a building, write a sonnet, balance accounts, build a wall, set a bone, comfort the dying, take orders, give orders, cooperate, act alone, solve equations, analyse a new problem, pitch manure, program a computer, cook a tasty meal, fight efficiently, die gallantly. Specialization is for insects.

So far, no AI system measures up to either of these lists, and some proponents of general or human-level AI (HLAI) insist that continued work on specific tasks (or on individual components) will not be enough to reach mastery on a wide variety of tasks; that we will need a fundamentally new approach. It seems to us that numerous new breakthroughs will indeed be necessary, but overall, AI as a field has made a reasonable exploration/exploitation tradeoff, assembling a portfolio of components, improving on particular tasks, while also exploring promising and sometimes far-out new ideas.

It would have been a mistake to tell the Wright brothers in 1903 to stop work on their single-task airplane and design an "artificial general flight" machine that can take off vertically, fly faster than sound, carry hundreds of passengers, and land on the moon. It also would have been a mistake to follow up their first flight with an annual competition to make spruce wood biplanes incrementally better.

We have seen that work on components can spur new ideas; for example, generative adversarial networks (GANs) and transformer language models each opened up new areas of research. We have also seen steps towards "diversity of behaviour." For example, machine translation systems in the 1990s were built one at a time for each language pair (such as French to English), but today a single system can identifying the input text as being one of a hundred languages, and translate it into any of 100 target languages. Another natural language system can perform five distinct tasks with one joint model (Hashimoto *et al.*, 2016).

AI engineering

The field of computer programming started with a few extraordinary pioneers. But it didn't reach the status of a major industry until a practice of software engineering was developed, with a powerful collection of widely available tools, and a thriving ecosystem of teachers, students, practitioners, entrepreneurs, investors, and customers.

The AI industry has not yet reached that level of maturity. We do have a variety of powerful tools and frameworks, such as TensorFlow, Keras, PyTorch, CAFFE, Scikit-Learn and SciPy. But many of the most promising approaches, such as GANs and deep reinforcement learning, have proven to be difficult to work with—they require experience and a degree of fiddling to get them to train properly in a new domain. We don’t have enough experts to do this across all the domains where we need it, and we don’t yet have the tools and ecosystem to let less-expert practitioners succeed.

Google’s Jeff Dean sees a future where we will want machine learning to handle millions of tasks; it won’t be feasible to develop each of them from scratch, so he suggests that rather than building each new system from scratch, we should start with a single huge system and, for each new task, extract from it the parts that are relevant to the task. We have seen some steps in this direction, such as the transformer language models (e.g., BERT, GPT-2) with billions of parameters, and an “outrageously large” ensemble neural network architecture that scales up to 68 billion parameters in one experiment (Shazeer *et al.*, 2017). Much work remains to be done.

The future

Which way will the future go? Science fiction authors seem to favor dystopian futures over utopian ones, probably because they make for more interesting plots. So far, AI seems to fit in with other powerful revolutionary technologies such as printing, plumbing, air travel, and telephony. All these technologies have made positive impacts, but also have some unintended side effects that disproportionately impact disadvantaged classes. We would do well to invest in minimizing the negative impacts.

AI is also different from previous revolutionary technologies. Improving printing, plumbing, air travel, and telephony to their logical limits would not produce anything to threaten human supremacy in the world. Improving AI to its logical limit certainly could.

In conclusion, AI has made great progress in its short history, but the final sentence of Alan Turing’s (1950) essay on *Computing Machinery and Intelligence* is still valid today:

We can see only a short distance ahead,
but we can see that much remains to be done.

[1] We gloss over the fact that this device consumes the entire energy output of a star and operates at a billion degrees centigrade.

APPENDIX A

MATHEMATICAL BACKGROUND

A.1 Complexity Analysis and O() Notation

Computer scientists are often faced with the task of comparing algorithms to see how fast they run or how much memory they require. There are two approaches to this task. The first is **benchmarking**—running the algorithms on a computer and measuring speed in seconds and memory consumption in bytes. Ultimately, this is what really matters, but a benchmark can be unsatisfactory because it is so specific: it measures the performance of a particular program written in a particular language, running on a particular computer, with a particular compiler and particular input data. From the single result that the benchmark provides, it can be difficult to predict how well the algorithm would do on a different compiler, computer, or data set. The second approach relies on a mathematical **analysis of algorithms**, independent of the particular implementation and input, as discussed below.

A.1.1 Asymptotic analysis

We will consider algorithm analysis through the following example, a program to compute the sum of a sequence of numbers:

```
function SUMMATION(sequence) returns a number
    sum ← 0
    for i =1 to LENGTH(sequence) do
        sum ← sum + sequence [i]
    return sum
```

The first step in the analysis is to abstract over the input, in order to find some parameter or parameters that characterize the size of the input. In this example, the input can be characterized by the length of the sequence, which we will call n. The second step is to abstract over the implementation, to find some measure that reflects the running time of the algorithm but is not tied to a particular compiler or computer. For the SUMMATION program, this could be just the number of lines of code executed, or it could be more detailed, measuring the number of additions, assignments, array references, and branches executed by the algorithm. Either way gives us a characterization of the total number of steps taken by the algorithm as a function of the size of the input. We will call this characterization $T(n)$. If we count lines of code, we have $T(n) = 2n + 2$ for our example.

If all programs were as simple as SUMMATION, the analysis of algorithms would be a trivial field. But two problems make it more complicated. First, it is rare to find a parameter like n that completely characterizes the number of steps taken by an algorithm. Instead, the best we can usually do is compute the worst case $T_{\text{worst}}(n)$ or the average case $T_{\text{avg}}(n)$. Computing an average means that the analyst must assume some distribution of inputs.

The second problem is that algorithms tend to resist exact analysis. In that case, it is necessary to fall back on an approximation. We say that the SUMMATION algorithm is $O(n)$, meaning that its measure is at most a constant times n, with the possible exception of a few small values of n. More formally,

$$T(n) \text{ is } O(f(n)) \text{ if } T(n) \le kf(n) \text{ for some } k, \text{ for all } n > n_0.$$

The $O()$ notation gives us what is called an **asymptotic analysis**. We can say without question that, as n asymptotically approaches infinity, an $O(n)$ algorithm is better than an $O(n^2)$ algorithm. A single benchmark figure could not substantiate such a claim.

The $O()$ notation abstracts over constant factors, which makes it easier to use, but less precise, than the $T()$ notation. For example, an $O(n^2)$ algorithm will always be worse than an $O(n)$ in the long run, but if the two algorithms are $T(n^2 + 1)$ and $T(100n + 1000)$, then the $O(n^2)$ algorithm is actually better for $n < 110$.

Despite this drawback, asymptotic analysis is the most widely used tool for analyzing algorithms. It is precisely because the analysis abstracts over both the exact number of operations (by ignoring the constant factor k) and the exact content of the input (by considering only its size n) that the analysis becomes mathematically feasible. The $O()$ notation is a good compromise between precision and ease of analysis.

A.1.2 NP and inherently hard problems

The analysis of algorithms and the $O()$ notation allow us to talk about the efficiency of a particular algorithm. However, they have nothing to say about whether there could be a better algorithm for the problem at hand. The field of **complexity analysis** analyzes problems rather than algorithms. The first gross division is between problems that can be solved in polynomial time and problems that cannot be solved in polynomial time, no matter what algorithm is used. The class of polynomial problems—those which can be solved in time $O(n^k)$ for some k—is called **P**. These are sometimes called "easy" problems, because the class contains those problems with running times like $O(\log n)$ and $O(n)$. But it also contains those with time $O(n^{1000})$, so the name "easy" should not be taken too literally.

Another important class of problems is **NP**, the class of nondeterministic polynomial problems. A problem is in this class if there is some algorithm that can guess a solution and then verify whether a guess is correct in polynomial time. The idea is that if you have an arbitrarily large number of processors so that you can try all the guesses at once, or if you are very lucky and always guess right the first time, then the NP problems become P problems. One of the biggest open questions in computer science is whether the class NP is equivalent to the class P when one does not have the luxury of an infinite number of processors or omniscient guessing. Most computer scientists are convinced that P ≠ NP; that NP problems are inherently hard and have no polynomial-time algorithms. But this has never been proven.

Those who are interested in deciding whether P = NP look at a subclass of NP called the **NP-complete** problems. The word "complete" is used here in the sense of "most extreme" and thus refers to the hardest problems in the class NP. It has been proven that either all the NP-complete problems are in P or none of them is. This makes the class theoretically interesting, but the class is also of practical interest because many important problems are known to be NP-complete. An example is the satisfiability problem: given a sentence of propositional logic, is there an assignment of truth values to the proposition symbols of the sentence that makes it true? Unless a miracle occurs and P = NP, there can be no algorithm that solves *all* satisfiability problems in polynomial time. However, AI is more interested in whether there are algorithms that perform efficiently on *typical* problems drawn from a predetermined distribution; as we saw in Chapter 7, there are algorithms such as WALKSAT that do quite well on many problems.

The class of **NP-hard** problems consists of those problems that are reducible (in polynomial time) to all the problems in NP, so if you solved any NP-hard problem, you could solve all the problems in NP. The NP-complete problems are all NP-hard, but there are some NP-hard problems that are even harder than NP-complete.

The class **co-NP** is the complement of NP, in the sense that, for every decision problem in NP, there is a corresponding problem in co-NP with the “yes” and “no” answers reversed. We know that P is a subset of both NP and co-NP, and it is believed that there are problems in co-NP that are not in P. The **co-NP-complete** problems are the hardest problems in co-NP.

The class #P (pronounced “number P” according to Garey and Johnson (1979), but often pronounced “sharp P”) is the set of counting problems corresponding to the decision problems in NP. Decision problems have a yes-or-no answer: is there a solution to this 3-SAT formula? Counting problems have an integer answer: how many solutions are there to this 3-SAT formula? In some cases, the counting problem is much harder than the decision problem. For example, deciding whether a bipartite graph has a perfect matching can be done in time $O(VE)$ (where the graph has V vertices and E edges), but the counting problem “how many perfect matches does this bipartite graph have” is #P-complete, meaning that it is hard as any problem in #P and thus at least as hard as any NP problem.

Another class is the class of PSPACE problems—those that require a polynomial amount of space, even on a nondeterministic machine. It is believed that PSPACE-hard problems are worse than NP-complete problems, although it could turn out that NP = PSPACE, just as it could turn out that P = NP.

A.2 Vectors, Matrices, and Linear Algebra

Mathematicians define a **vector** as a member of a vector space, but we will use a more concrete definition: a vector is an ordered sequence of values. For example, in two-dimensional space, we have vectors such as $\mathbf{x} = \langle 3,4 \rangle$ and $\mathbf{y} = \langle 0,2 \rangle$. We follow the convention of boldface characters for vector names, although some authors use arrows or bars over the names: $\overrightarrow{x}$ or $\bar{y}$. The elements of a vector can be accessed using subscripts: $\mathbf{z} = \langle z_1, z_2, \ldots\ z_n \rangle$. One confusing point: this book is synthesizing work from many subfields, which variously call their sequences vectors, lists, or tuples, and variously use the notations $\langle 1,2 \rangle$, [1, 2], or (1, 2).

The two fundamental operations on vectors are vector addition and scalar multiplication. The vector addition $\mathbf{x} + \mathbf{y}$ is the elementwise sum: $\mathbf{x} + \mathbf{y} = \langle 3 + 0, 4 + 2 \rangle = \langle 3,6 \rangle$. Scalar multiplication multiplies each element by a constant: $5\,\mathbf{x} = \langle 5 \times 3, 5 \times 4 \rangle = \langle 15,20 \rangle$.

The length of a vector is denoted $|\mathbf{x}|$ and is computed by taking the square root of the sum of the squares of the elements: $|\mathbf{x}| = \sqrt{(3^2 + 4^2)} = 5$. The dot product $\mathbf{x} \bullet \mathbf{y}$ (also called scalar product) of two vectors is the sum of the products of corresponding elements, that is, $\mathbf{x} \bullet \mathbf{y} = \Sigma_i x_i y_i$,, or in our particular case, $\mathbf{x} \bullet \mathbf{y} = 3 \times 0 + 4 \times 2 = 8$.

Vectors are often interpreted as directed line segments (arrows) in an *n*-dimensional Euclidean space. Vector addition is then equivalent to placing the tail of one vector at the head of the other, and the dot product $\mathbf{x} \bullet \mathbf{y}$ is $|\mathbf{x}|\,|\mathbf{y}| \cos \theta$, where θ is the angle between $\mathbf{x}$ and $\mathbf{y}$.

A **matrix** is a rectangular array of values arranged into rows and columns. Here is a Matrix matrix $\mathbf{A}$ of size 3×4:

$$\begin{matrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} & \mathbf{A}_{1,3} & \mathbf{A}_{1,4} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} & \mathbf{A}_{2,3} & \mathbf{A}_{2,4} \\ \mathbf{A}_{3,1} & \mathbf{A}_{3,2} & \mathbf{A}_{3,3} & \mathbf{A}_{3,4} \end{matrix}$$

The first index of $\mathbf{A}_{i,j}$ specifies the row and the second the column. In programming languages, $\mathbf{A}_{i,j}$ is often written A[i,j] or A[i][j].

The sum of two matrices is defined by adding their corresponding elements; for example $(\mathbf{A} + \mathbf{B})_{i,j} = \mathbf{A}_{i,j} + \mathbf{B}_{i,j}$. (The sum is undefined if **A** and **B** have different sizes.) We can also define the multiplication of a matrix by a scalar: $(c\mathbf{A})_{i,j} = c\mathbf{A}_{i,j}$. Matrix multiplication (the product of two matrices) is more complicated. The product **AB** is defined only if **A** is of size $a \times b$ and **B** is of size $b \times c$ (i.e., the second matrix has the same number of rows as the first has columns); the result is a matrix of size $a \times c$. If the matrices are of appropriate size, then the result is

$$(\mathbf{AB})_{i,k} = \sum_j \mathbf{A}_{i,j}\mathbf{B}_{j,k}.$$

Matrix multiplication is not commutative, even for square matrices: $\mathbf{AB} \neq \mathbf{BA}$ in general. It is, however, associative: $(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$. Note that the dot product can be expressed in terms of a transpose and a matrix multiplication: $\mathbf{x} \bullet \mathbf{y} = \mathbf{x}^T\mathbf{y}$.

The **identity matrix I** has elements $\mathbf{I}_{i,j}$ equal to 1 when $i = j$ and equal to 0 otherwise. It has the property that $\mathbf{AI} = \mathbf{A}$ for all **A**. The **transpose** of **A**, written $\mathbf{A}^T$ is formed by turning rows into columns and vice versa, or, more formally, by $\mathbf{A}^T{}_{i,j} = \mathbf{A}_{j,i}$. The inverse of a square matrix **A** is another square matrix $\mathbf{A}^{-1}$ such that $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$. For a **singular** matrix, the inverse does not exist. For a nonsingular matrix, it can be computed in $O(n^3)$ time.

Matrices are used to solve systems of linear equations in $O(n^3)$ time; the time is dominated by inverting a matrix of coefficients. Consider the following set of equations, for which we want a solution in x, y, and z:

$$
\begin{aligned}
+2x + y - z &= 8\\
-3x - y + 2z &= -11\\
-2x + y + 2z &= -3.
\end{aligned}
$$

We can represent this system as the matrix equation **Ax = b,** where

$$
\mathbf{A} = \begin{bmatrix} 2 & 1 & -1 \\ -3 & -1 & 2 \\ -2 & 1 & 2 \end{bmatrix}, \qquad \mathbf{x} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} 8 \\ -11 \\ -3 \end{bmatrix}.
$$

To solve **Ax** = **b** we multiply both sides by $\mathbf{A}^{-1}$, yielding $\mathbf{A}^{-1}\,\mathbf{A}\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$, which simplifies to $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. After inverting **A** and multiplying by **b**, we get the answer

$$
\mathbf{x} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \\ -1 \end{bmatrix}.
$$

A few more miscellaneous points: we use $\log(x)$ for the natural logarithm, $\log_e(x)$. We use $\operatorname{argmax}_x f(x)$ for the value of x for which $f(x)$ is maximal.

A.3 Probability Distributions

A probability is a measure over a set of events that satisfies three axioms:

1. The measure of each event is between 0 and 1. We write this as $0 \leq P(X = x_i) \leq 1$, where X is a random variable representing an event and x_i are the possible values of X. In general, random variables are denoted by uppercase letters and their values by lowercase letters.
2. The measure of the whole set is 1; that is, $\sum_{i=1}^{n} P(X = x_i) = 1$.
3. The probability of a union of disjoint events is the sum of the probabilities of the individual events; that is, $P(X = x_1 \vee X = x_2) = P(X = x_1) + P(X = x_2)$, in the case where x_1 and x_2 are disjoint.

A **probabilistic model** consists of a sample space of mutually exclusive possible outcomes, together with a probability measure for each outcome. For example, in a model of the weather tomorrow, the outcomes might be *sun*, *cloud*, *rain*, and *snow*. A subset of these outcomes constitutes an event. For example, the event of precipitation is the subset consisting of {*rain*, *snow*}.

We use $\mathbf{P}(X)$ to denote the vector of values $\langle P(X = x_1), \ldots, P(X = x_n) \rangle$. We also use $P(x_i)$ as an abbreviation for $P(X = x_i)$ and $\Sigma_x P(x)$ for $\sum_{i=1}^{n} P(X = x_i)$.

The conditional probability $P(B \mid A)$ is defined as $P(B \cap A)/P(A)$. A and B are conditionally independent if $P(B \mid A) = P(B)$ (or equivalently, $P(A \mid B) = P(A)$).

For continuous variables, there are an infinite number of values, and unless there are point spikes, the probability of any one exact value is 0. So it makes more sense to talk about the value being within a range. We do that with a **probability density function**, which has a slightly different meaning from the discrete probability function. Since $P(X = x)$—the probability that X has the value x exactly—is zero, we instead measure how likely it is that X falls into an interval around x, compared to the width of the interval, and take the limit as the interval width goes to zero:

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

The density function must be nonnegative for all x and must have

$$\int_{-\infty}^{\infty} P(x)dx = 1.$$

We can also define the **cumulative distribution** $F_X(x)$, which is the probability of a random variable being less than x:

$$F_X(x) = P(X \le x) = \int_{-\infty}^{x} P(u)du.$$

Note that the probability density function has units, whereas the discrete probability function is unitless. For example, if values of X are measured in seconds, then the density is measured in Hz (i.e., 1/sec). If values of **X** are points in three-dimensional space measured in meters, then density is measured in $1/m^3$.

One of the most important probability distributions is the **Gaussian distribution**, also known as the **normal distribution**. We use the notation $N(x;\mu, \sigma^2)$ for the normal distribution that is a function of x with mean μ and standard deviation σ (and therefore variance σ^2). It is defined as

$$N(x;\mu,\sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/(2\sigma^2)},$$

where x is a continuous variable ranging from — ∞ to +∞. With mean $\mu = 0$ and variance $\sigma^2 = 1$, we get the special case of the **standard normal**

distribution. For a distribution over a vector **x** in n dimensions, there is the **multivariate Gaussian** distribution:

$$N(x;\mu,\sum) = \frac{1}{\sqrt{(2\pi)^n|\sum|}} e^{\frac{1}{-2}((x-\mu)^\top \sum^{-1}(\mathbf{x}-\mu))},$$

where μ is the mean vector and Σ is the **covariance matrix** (see below). The cumulative distribution for a univariate normal distribution is given by

$$F(x) = \int\limits_{-\infty}^{x} N(z;\mu,\sigma^2)dz = \tfrac{1}{2}(1+\mathrm{erf}\,(\tfrac{x-\mu}{\sigma\sqrt{2}})),$$

where erf(x) is the so-called **error function**, which has no closed-form representation.

The **central limit theorem** states that the distribution formed by sampling n independent random variables and taking their mean tends to a normal distribution as n tends to infinity. This holds for almost any collection of random variables, even if they are not strictly independent, unless the variance of any finite subset of variables dominates the others.

The **expectation** of a random variable, $E(X)$, is the mean or average value, weighted by the probability of each value. For a discrete variable it is:

$$E(X) = \sum_i x_i P(X = x_i).$$

For a continuous variable, replace the summation with an integral and use the probability density function, $P(x)$:

$$E(X) = \int\limits_{-\infty}^{\infty} xP(x)dx.$$

For any function f, we also have

$$E(f(X)) = \int_{-\infty}^{\infty} f(x)P(x)dx.$$

Finally, when necessary, one may specify the distribution for the random variable as a subscript to the expectation operator:

$$E_{X\sim Q(x)}(g(X)) = \int_{-\infty}^{\infty} g(x)Q(x)dx.$$

Besides the expectation, other important statistical properties of a distribution include the **variance**, which is the expected value of the square of the difference from the mean, μ, of the distribution:

$$Var(X) = E((X-\mu)^2)$$

and the **standard deviation**, which is the square root of the variance.

The **root mean square (RMS)** of a set of values (often samples of a random variable) is the square root of the mean of the squares of the values,

$$RMS(x_1, \ldots x_n) = \sqrt{\frac{x_1^2 + \ldots + x_n^2}{n}}.$$

The **covariance** of two random variables is the expectation of the product of their differences from their means:

$$\text{cov}(X, Y) = E((X-\mu_X)(Y-\mu_Y)).$$

The **covariance matrix**, often denoted Σ, is a matrix of covariances between elements of a vector of random variables. Given **X** = $\langle X_1,\ldots X_n\rangle^\top$, the entries of the covariance matrix are as follows:

$$\sum\nolimits_{i,j} = \text{cov}(X_i, X_j) = E((X_i - \mu_i)(X_j - \mu_j)).$$

We say we **sample** from a probability distribution, when we pick a value at random. We don’t know what any one pick will bring, but in the limit a large collection of samples will approach the same probability density function as the distribution it is sampled from. The **uniform distribution** is one where every element is equally (uniformly) probable. So when we say we “sample uniformly (at random) from the integers 0 to 99” it means that we are equally likely to pick any integer in that range.

Bibliographical and Historical Notes

The $O()$ notation so widely used in computer science today was first introduced in the context of number theory by the mathematician P. G. H. Bachmann (1894). The concept of NP-completeness was invented by Cook (1971), and the modern method for establishing a reduction from one problem to another is due to Karp (1972). Cook and Karp have both won the Turing award for their work.

Textbooks on the analysis and design of algorithms include Sedgewick and Wayne (2011) and Cormen, Leiserson, Rivest and Stein (2009). These books place an emphasis on designing and analyzing algorithms to solve tractable problems. For the theory of NP-completeness and other forms of intractability, see Garey and Johnson (1979) or Papadimitriou (1994). Good texts on probability include Chung (1979), Ross (2015), and Bertsekas and Tsitsiklis (2008).

APPENDIX B

NOTES ON LANGUAGES AND ALGORITHMS

B.1 Defining Languages with Backus-Naur Form (BNF)

In this book we define several languages, including the languages of propositional logic (235), first-order logic (276), and a subset of English (page 893). A formal language is defined as a set of strings where each string is a sequence of symbols. The languages we are interested in consist of an infinite set of strings, so we need a concise way to characterize the set. We do that with a **grammar**. The particular type of grammar we use is called a **context-free grammar**, because each expression has the same form in any context. We write our grammars in a formalism called **Backus-Naur form (BNF)**. There are four components to a BNF grammar:

- A set of **terminal symbols**. These are the symbols or words that make up the strings of the language. They could be letters (**A, B, C,**...) or words (**a, aardvark, abacus,** ...), or whatever symbols are appropriate for the domain.
- A set of **nonterminal symbols** that categorize subphrases of the language. For example, the nonterminal symbol *NounPhrase* in English denotes an infinite set of strings including “you” and “the big slobbery dog.”
- A **start symbol**, which is the nonterminal symbol that denotes the complete set of strings of the language. In English, this is *Sentence*; for arithmetic, it might be *Expr,* and for programming languages it is *Program*.
- A set of **rewrite rules**, of the form *LHS* → *RHS,* where *LHS* is a nonterminal symbol and *RHS* is a sequence of zero or more symbols.

These can be either terminal or nonterminal symbols, or the symbol $\in$, which is used to denote the empty string.

A rewrite rule of the form

$$Sentence \rightarrow NounPhrase\ VerbPhrase$$

means that whenever we have two strings categorized as a *NounPhrase* and a *VerbPhrase*, we can append them together and categorize the result as a *Sentence*. As an abbreviation, the two rules ($S \rightarrow A$) and ($S \rightarrow B$) can be written ($S \rightarrow A$ | B). To illustrate these concepts, here is a BNF grammar for simple arithmetic expressions:

$$\begin{array}{lcl} Expr & \rightarrow & Expr\ Operator\ Expr \mid \mathbf{(}\ Expr\ \mathbf{)} \mid Number \\ Number & \rightarrow & Digit \mid Number\ Digit \\ Digit & \rightarrow & \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9} \\ Operator & \rightarrow & + \mid - \mid \div \mid \times \end{array}$$

We cover languages and grammars in more detail in Chapter 24. Be aware that other books use slightly different notations for BNF; for example, you might see ⟨*Digit*⟩ instead of *Digit* for a nonterminal, ‘word’ instead of **word** for a terminal, or : : = instead of → in a rule.

B.2 Describing Algorithms with Pseudocode

The algorithms in this book are described in **pseudocode**. Most of the pseudocode should be familiar to programmers who use languages like Java, C++, or especially Python. In some places we use mathematical formulas or ordinary English to describe parts that would otherwise be more cumbersome. A few idiosyncrasies should be noted:

- **Persistent variables**: We use the keyword **persistent** to say that a variable is given an initial value the first time a function is called and retains that value (or the value given to it by a subsequent assignment statement) on all subsequent calls to the function. Thus, persistent variables are like global variables in that they outlive a single call to their function; but they are accessible only within the function. The agent programs in the book use persistent variables for *memory*. Programs with persistent variables can be implemented as *objects* in object-oriented languages such as C++, Java, Python, and Smalltalk. In functional languages, they can be implemented by *functional closures* over an environment containing the required variables.
- **Functions as values**: Functions have capitalized names, and variables have lowercase italic names. So most of the time, a function call looks like FN(x). However, we allow the value of a variable to be a function; for example, if the value of the variable f is the square root function, then $f(9)$ returns 3.
- **Indentation is significant**: Indentation is used to mark the scope of a loop or conditional, as in the languages Python and CoffeeScript, and unlike Java, C++, and Go (which use braces) or Lua and Ruby (which use **end**).

- **Destructuring assignment**: The notation "*x*, *y* ← *pair*" means that the right-hand side must evaluate to a two-element collection, and the first element is assigned to *x* and the second to y. The same idea is used in " **for** *x*,*y* **in** *pairs* **do**" and can be used to swap two variables: "*x*, *y* ← *y*, *x*"
- **Default values for parameters**: The notation "**function** F(*x*, y = 0) **returns** a number" means that *y* is an optional argument with default value 0; that is, the calls F(3, 0) and F(3) are equivalent.
- **yield**: a function that contains the keyword **yield** is a **generator** that generates a sequence of values, one each time the **yield** expression is encountered. After yielding, the function continues execution with the next statement. The languages Python, Ruby, C#, and Javascript (ECMAScript) have this same feature.
- **Loops**: There are four kinds of loops:
  - "**for** *x* **in** *c* **do**" executes the loop with the variable *x* bound to successive elements of the collection *c*.
  - "**for** *i* = 1 to *n* **do**" executes the loop with *i* bound to successive integers from 1 to *n* inclusive.
  - "**while** *condition* **do**" means the condition is evaluated before each iteration of the loop, and the loop exits if the condition is false.
  - "**repeat** ... **until** *condition*" means that the loop is executed unconditionally the first time, then the condition is evaluated, and the loop exits if the condition is true; otherwise the loop keeps executing (and testing at the end).
- **Lists**: [*x*,*y*, *z*] denotes a list of three elements. The "+" operator concatenates lists: [1,2] + [3,4] = [1,2,3,4]. A list can be used as a

stack: Pop removes and returns the last element of a list, TOP returns the last element.

- **Sets**: $\{x,y,z\}$ denotes a set of three elements. $\{x: p(x)\}$ denotes the set of all elements x for which $p(x)$ is true.
- **Arrays start at 1**: the first index of an array is 1 as in usual mathematical notation (and in R and Julia), not 0 (as in Python and Java and C).

B.3 Online Supplemental Material

The book has a Web site with supplemental material, instructions for sending suggestions, and opportunities for joining discussion lists:

- `aima.cs.berkeley.edu`

The algorithms in the book, and multiple additional programming exercises, have been implemented in Python and Java (and some in other languages) at the online code repository, accessible from the Web site and currently hosted at:

- `github.com/aimacode`

Bibliography

The following abbreviations are used for frequently cited conferences and journals:

Aaronson, S. (2014). My conversation with ’’Eugene Goostman,” the chatbot that’s all over the news for allegedly passing the Turing test. Shtetl-Optimized, www.scottaaronson.com/blog/?p=1858.

Aarts, E. and Lenstra, J. K. (2003). *Local Search in Combinatorial Optimization.* Princeton University Press.

Aarup, M., Arentoft, M. M., Parrod, Y., Stader, J., and Stokes, I. (1994). OPTIMUM-AIV: A knowledge-based planning and scheduling system for spacecraft AIV. In Fox, M. and Zweben, M. (Eds.), *Knowledge Based Scheduling*. Morgan Kaufmann.

Abbas, A. (2018). *Foundations of Multiattribute Utility.* Cambridge University Press.

Abbeel, P. and Ng, A. Y. (2004). Apprenticeship learning via inverse reinforcement learning. In *ICML-04.*

Abney, S., McAllester, D. A., and Pereira, F. (1999). Relating probabilistic grammars and automata. In *ACL-99.*

Abramson, B. (1987). *The expected-outcome model of two-player games*. Ph.D. thesis, Columbia University.

Abramson, B. (1990). Expected-outcome: A general model of static evaluation. *PAMI*, *12*, 182–193.

Abreu, D. and Rubinstein, A. (1988). The structure of Nash equilibrium in repeated games with finite automata. *Econometrica*, *56*, 1259–1281.

Achlioptas, D. (2009). Random satisfiability. In Biere, A., Heule, M., van Maaren, H., and Walsh, T. (Eds.), *Handbook of Satisfiability*. IOS Press.

Ackerman, E. and Guizzo, E. (2016). The next generation of Boston Dynamics' Atlas robot is quiet, robust, and tether free. *IEEE Spectrum*, *24*, 2016.

Ackerman, N., Freer, C., and Roy, D. (2013). On the computability of conditional probability. arXiv 1005.3014.

Ackley, D. H. and Littman, M. L. (1991). Interactions between learning and evolution. In Langton, C., Taylor, C., Farmer, J. D., and Rasmussen, S. (Eds.), *Artificial Life II*. Addison-Wesley.

Adida, B. and Birbeck, M. (2008). RDFa primer. Tech. rep., W3C.

Adolph, K. E., Kretch, K. S., and LoBue, V. (2014). Fear of heights in infants? *Current Directions in Psychological Science*, *23*, 60–66.

Agerbeck, C. and Hansen, M. O. (2008). A multiagent approach to solving *NP*-complete problems. Master's thesis, Technical Univ. of Denmark.

Aggarwal, G., Goel, A., and Motwani, R. (2006). Truthful auctions for pricing search keywords. In *EC-06*.

Agha, G. (1986). *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press.

Agichtein, E. and Gravano, L. (2003). Querying text databases for efficient information extraction. In *Proc. IEEE Conference on Data Engineering*.

Agmon, S. (1954). The relaxation method for linear inequalities. *Canadian Journal of Mathematics*, *6*, 382–392.

Agostinelli, F., McAleer, S., Shmakov, A., and Baldi, P. (2019). Solving the Rubik's Cube with deep reinforcement learning and search. *Nature Machine Intelligence*, *1*, 356–363.

Agrawal, P., Nair, A. V., Abbeel, P., Malik, J., and Levine, S. (2017). Learning to poke by poking: Experiential learning of intuitive physics. In *NeurIPS 29*.

Agre, P. E. and Chapman, D. (1987). Pengi: an implementation of a theory of activity. In *IJCAI-87*.

Aizerman, M., Braverman, E., and Rozonoer, L. (1964). Theoretical foundations of the potential function method in pattern recognition learning. *Automation and Remote Control*, *25*, 821–837.

Akametalu, A. K., Fisac, J. F., Gillula, J. H., Kaynama, S., Zeilinger, M. N., and Tomlin, C. J. (2014). Reachability-based safe learning with Gaussian processes. In *53rd IEEE Conference on Decision and Control*.

Akgun, B., Cakmak, M., Jiang, K., and Thomaz, A. (2012). Keyframe-based learning from demonstration. *International Journal of Social Robotics*, *4*, 343–355.

Aldous, D. and Vazirani, U. (1994). "Go with the winners" algorithms. In *FOCS-94*.

Alemi, A. A., Chollet, F., Een, N., Irving, G., Szegedy, C., and Urban, J. (2017). DeepMath - Deep sequence models for premise selection. In *NeurIPS 29*.

Allais, M. (1953). Le comportment de l'homme rationnel devant la risque: critique des postulats et axiomes de l'école Américaine. *Econometrica, 21*, 503–546.

Allan, J., Harman, D., Kanoulas, E., Li, D., Van Gysel, C., and Vorhees, E. (2017). Trec 2017 common core track overview. In *Proc. TREC*.

Allen, J. F. (1983). Maintaining knowledge about temporal intervals. *CACM*, *26*, 832–843.

Allen, J. F. (1984). Towards a general theory of action and time. *AIJ*, 23, 123–154.

Allen, J. F. (1991). Time and time again: The many ways to represent time. *Int. J. Intelligent Systems, 6*, 341–355.

Allen, J. F., Hendler, J., and Tate, A. (Eds.). (1990). *Readings in Planning*. Morgan Kaufmann.

Allen, P. and Greaves, M. (2011). The singularity isn't near. *Technology review, 12*, 7–8.

Allen-Zhu, Z., Li, Y., and Song, Z. (2018). A convergence theory for deep learning via overparameterization. arXiv:1811.03962.

Alterman, R. (1988). Adaptive planning. *Cognitive Science, 12*, 393–422.

Amarel, S. (1967). An approach to heuristic problemsolving and theorem proving in the propositional calculus. In Hart, J. and Takasu, S. (Eds.), *Systems and Computer Science*. University of Toronto Press.

Amarel, S. (1968). On representations of problems of reasoning about actions. In Michie, D. (Ed.), *Machine Intelligence 3*, Vol. 3. Elsevier.

Amir, E. and Russell, S. J. (2003). Logical filtering. In *IJCAI-03*.

Amit, Y. and Geman, D. (1997). Shape quantization and recognition with randomized trees. *Neural Computation*, *9*, 1545–1588.

Amodei, D. and Hernandez, D. (2018). AI and compute. OpenAI blog, blog.openai.com/ai-and-compute/.

Amodei, D., Olah, C., Steinhardt, J., Christiano, P, Schulman, J., and Mané, D. (2016). Concrete problems in AI safety. arXiv:1606.06565.

Andersen, S. K., Olesen, K. G., Jensen, F. V., and Jensen, F. (1989). HUGIN—A shell for building Bayesian belief universes for expert systems. In *IJCAI-89*.

Anderson, J. R. (1980). *Cognitive Psychology and Its Implications*. W. H. Freeman.

Anderson, J. R. (1983). *The Architecture of Cognition*. Harvard University Press.

Anderson, K., Sturtevant, N. R., Holte, R. C., and Schaeffer, J. (2008). Coarse-to-fine search techniques. Tech. rep., University of Alberta.

Andoni, A. and Indyk, P. (2006). Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *FOCS-06*.

Andor, D., Alberti, C., Weiss, D., Severyn, A., Presta, A., Ganchev, K., Petrov, S., and Collins, M. (2016). Globally normalized transition-based neural networks. arXiv:1603.06042.

Andre, D., Friedman, N., and Parr, R. (1998). Generalized prioritized sweeping. In *NeurIPS 10*.

Andre, D. and Russell, S. J. (2002). State abstraction for programmable reinforcement learning agents. In *AAAI-02*.

Andreae, P. (1985). *Justified Generalisation: Learning Procedures from Examples*. Ph.D. thesis, MIT.

Andrieu, C., Doucet, A., and Holenstein, R. (2010). Particle Markov chain Monte Carlo methods. *J. Royal Statistical Society*, *72*, 269–342.

Andrychowicz, M., Baker, B., Chociej, M., Jozefowicz, R., McGrew, B., Pachocki, J., Petron, A., Plappert, M., Powell, G., Ray, A., *et al.* (2018a). Learning dexterous in-hand manipulation. arXiv:1808.00177.

Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Abbeel, P., and Zaremba, W. (2018b). Hindsight experience replay. In *NeurIPS 30*.

Aneja, J., Deshpande, A., and Schwing, A. (2018). Convolutional image captioning. In *CVPR-18*.

Aoki, M. (1965). Optimal control of partially observable Markov systems. *J. Franklin Institute*, *280*, 367–386.

Appel, K. and Haken, W. (1977). Every planar map is four colorable: Part I: Discharging. *Illinois J. Math.*, *21*, 429–490.

Appelt, D. (1999). Introduction to information extraction. *AI Communications*, *12*, 161–172.

Apt, K. R. (1999). The essence of constraint propagation. *Theoretical Computer Science*, *221*, 179–210.

Apt, K. R. (2003). *Principles of Constraint Programming*. Cambridge University Press.

Apté, C., Damerau, F., and Weiss, S. (1994). Automated learning of decision rules for text categorization. *ACM Transactions on Information Systems, 12,* 233–251.

Arbuthnot, J. (1692). *Of the Laws of Chance.* Motte, London. Translation into English, with additions, of Huygens (1657).

Archibald, C., Altman, A., and Shoham, Y. (2009). Analysis of a winning computational billiards player. In *IJCAI-09.*

Arfaee, S. J., Zilles, S., and Holte, R. C. (2010). Bootstrap learning of heuristic functions. In *Third Annual Symposium on Combinatorial Search*.

Argall, B. D., Chernova, S., Veloso, M., and Browning, B. (2009). A survey of robot learning from demonstration. *Robotics and autonomous systems, 57,* 469–483.

Ariely, D. (2009). *Predictably Irrational* (Revised edition). Harper.

Arkin, R. (1998). *Behavior-Based Robotics*. MIT Press.

Arkin, R. (2015). The case for banning killer robots: Counterpoint. *CACM*, *58*.

Armando, A., Carbone, R., Compagna, L., Cuellar, J., and Tobarra, L. (2008). Formal analysis of SAML 2.0 web browser single sign-on: Breaking the SAML-based single sign-on for Google apps. In *Proc. 6th ACM Workshop on Formal Methods in Security Engineering*.

Armstrong, S. and Levinstein, B. (2017). Low impact artificial intelligences. arXiv:1705.10720.

Arnauld, A. (1662). *La logique, ou l'art de penser*. Chez Charles Savreux, Paris.

Arora, N. S., Russell, S. J., and Sudderth, E. (2013). NET-VISA: Network processing vertically integrated seismic analysis. *Bull. Seism. Soc. Amer., 103*, 709–729.

Arora, S. (1998). Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems. *JACM*, *45*, 753–782.

Arpit, D., Jastrzebski, S., Ballas, N., Krueger, D., Bengio, E., Kanwal, M. S., Maharaj, T., Fischer, A., Courville, A., Bengio, Y., and Lacoste-Julien, S. (2017). A closer look at memorization in deep networks. arXiv:1706.05394.

Arrow, K. J. (1951). *Social Choice and Individual Values.* Wiley.

Arulampalam, M. S., Maskell, S., Gordon, N., and Clapp, T. (2002). A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking. *IEEE Transactions on Signal Processing, 50*, 174–188.

Arulkumaran, K., Deisenroth, M. P., Brundage, M., and Bharath, A. A. (2017). Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine, 34*, 26–38.

Arunachalam, R. and Sadeh, N. M. (2005). The supply chain trading agent competition. *Electronic Commerce Research and Applications*, *Spring*, 66–84.

Ashby, W. R. (1940). Adaptiveness and equilibrium. *J. Mental Science, 86*, 478–83.

Ashby, W. R. (1948). Design for a brain. *Electronic Engineering, December*, 379–383.

Ashby, W. R. (1952). *Design for a Brain.* Wiley.

Asimov, I. (1942). Runaround. *Astounding Science Fiction, March.*

Asimov, I. (1950). *I, Robot.* Doubleday.

Asimov, I. (1958). The feeling of power. *If: Worlds of Science Fiction, February.*

Astrom, K. J. (1965). Optimal control of Markov decision processes with incomplete state estimation. *J. Math. Anal. Applic., 10*, 174–205.

Atkeson, C. G., Moore, A. W., and Schaal, S. (1997). Locally weighted learning for control. In *Lazy learning*. Springer.

Audi, R. (Ed.). (1999). *The Cambridge Dictionary of Philosophy.* Cambridge University Press.

Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine Learning, 47*, 235–256.

Aumann, R. and Brandenburger, A. (1995). Epistemic conditions for nash equilibrium. *Econometrica, 67*, 1161–1180.

Axelrod, R. (1985). *The Evolution of Cooperation.* Basic Books.

Ba, J. L., Kiros, J. R., and Hinton, G. E. (2016). Layer normalization. arXiv:1607.06450.

Baader, F., Calvanese, D., McGuinness, D., Nardi, D., and Patel-Schneider, P. (2007). *The Description Logic Handbook* (2nd edition). Cambridge University Press.

Baader, F. and Snyder, W. (2001). Unification theory. In Robinson, J. and Voronkov, A. (Eds.), *Handbook of Automated Reasoning*. Elsevier.

Bacchus, F. (1990). *Representing and Reasoning with Probabilistic Knowledge.* MIT Press.

Bacchus, F. and Grove, A. (1995). Graphical models for preference and utility. In *UAI-95*.

Bacchus, F. and Grove, A. (1996). Utility independence in a qualitative decision theory. In *KR-96*.

Bacchus, F., Grove, A., Halpern, J. Y., and Koller, D. (1992). From statistics to beliefs. In *AAAI-92*.

Bacchus, F. and van Beek, P. (1998). On the conversion between non-binary and binary constraint satisfaction problems. In *AAAI-98*.

Bacchus, F. and van Run, P. (1995). Dynamic variable ordering in CSPs. In *CP-95*.

Bacchus, F., Dalmao, S., and Pitassi, T. (2003). Value elimination: Bayesian inference via backtracking search. In *UAI-03*.

Bachmann, P. G. H. (1894). *Die analytische Zahlentheorie*. B. G. Teubner, Leipzig.

Backus, J. W. (1959). The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. *Proc. Int'l Conf. on Information Processing*.

Bacon, F. (1609). *Wisdom of the Ancients*. Cassell and Company.

Baeza-Yates, R. and Ribeiro-Neto, B. (2011). *Modern Information Retrieval* (2nd edition). Addison-Wesley.

Bagdasaryan, E., Veit, A., Hua, Y., Estrin, D., and Shmatikov, V. (2018). How to backdoor federated learning. arXiv:1807.00459.

Bagnell, J. A. and Schneider, J. (2001). Autonomous helicopter control using reinforcement learning policy search methods. In *ICRA-01*.

Bahdanau, D., Cho, K., and Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. In *ICLR-15*.

Bahubalendruni, M. R. and Biswal, B. B. (2016). A review on assembly sequence generation and its automation. *Proc. Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, *230*, 824–838.

Bai, A. and Russell, S. J. (2017). Efficient reinforcement learning with hierarchies of machines by leveraging internal transitions. In *IJCAI-17*.

Bai, H., Cai, S., Ye, N., Hsu, D., and Lee, W. S. (2015). Intention-aware online POMDP planning for autonomous driving in a crowd. In *ICRA-15*.

Bajcsy, A., Losey, D. P., O'Malley, M. K., and Dragan, A. D. (2017). Learning robot objectives from physical human interaction. *Proceedings of Machine Learning Research*, *78*, 217–226.

Baker, C. L., Saxe, R., and Tenenbaum, J. B. (2009). Action understanding as inverse planning. *Cognition*, *113*, 329–349.

Baker, J. (1975). The Dragon system—An overview. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, *23*, 24–29.

Baker, J. (1979). Trainable grammars for speech recognition. In *Speech Communication Papers for the 97th Meeting of the Acoustical Society of America*.

Baldi, P., Chauvin, Y., Hunkapiller, T., and McClure, M. (1994). Hidden Markov models of biological primary sequence information. *PNAS*, *91*,

1059–1063.

Baldwin, J. M. (1896). A new factor in evolution. *American Naturalist, 30,* 441–451. Continued on pages 536–553.

Ballard, B. W. (1983). The *-minimax search procedure for trees containing chance nodes. *AIJ*, *21*, 327–350.

Baluja, S. (1997). Genetic algorithms and explicit search statistics. In *NeurIPS 9*.

Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J. D. (1986). Magic sets and other strange ways to implement logic programs. In *PODS-86.*

Banko, M. and Brill, E. (2001). Scaling to very very large corpora for natural language disambiguation. In *ACL-01*.

Banko, M., Brill, E., Dumais, S. T., and Lin, J. (2002). AskMSR: Question answering using the worldwide web. In *Proc. AAAI Spring Symposium on Mining Answers from Texts and Knowledge Bases.*

Banko, M., Cafarella, M. J., Soderland, S., Broadhead, M., and Etzioni, O. (2007). Open information extraction from the web. In *IJCAI-07*.

Banko, M. and Etzioni, O. (2008). The tradeoffs between open and traditional relation extraction. In *ACL-08*.

Bansal, K., Loos, S., Rabe, M. N., Szegedy, C., and Wilcox, S. (2019). HOList: An environment for machine learning of higher-order theorem proving (extended version). arXiv:1904.03241.

Bar-Hillel, Y. (1954). Indexical expressions. *Mind, 63*, 359–379.

Bar-Shalom, Y. (Ed.). (1992). *Multitarget Multisensor Tracking: Advanced Applications*. Artech House.

Bar-Shalom, Y. and Fortmann, T. E. (1988). *Tracking and Data Association*. Academic Press.

Bar-Shalom, Y., Li, X.-R., and Kirubarajan, T. (2001). *Estimation, Tracking and Navigation: Theory, Algorithms and Software*. Wiley.

Barber, D. (2012). *Bayesian Reasoning and Machine Learning*. Cambridge University Press.

Barr, A. and Feigenbaum, E. A. (Eds.). (1981). *The Handbook of Artificial Intelligence,* Vol. 1. HeurisTech Press and William Kaufmann.

Barreiro, J., Boyce, M., Do, M., Frank, J., Iatauro, M., Kichkaylo, T., Morris, P., Ong, J., Remolina, E., Smith, T., *et al*. (2012). EUROPA: A platform for AI planning, scheduling, constraint programming, and optimization. *4th International Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS)*.

Barreno, M., Nelson, B., Joseph, A. D., and Tygar, J. D. (2010). The security of machine learning. *Machine Learning*, *81*, 121–148.

Barrett, S. and Stone, P. (2015). Cooperating with unknown teammates in complex domains: A robot soccer case study of ad hoc teamwork. In *AAAI-15*.

Barták, R., Salido, M. A., and Rossi, F. (2010). New trends in constraint satisfaction, planning, and scheduling: A survey. *The Knowledge Engineering Review*, *25*, 249–279.

Bartholdi, J. J., Tovey, C. A., and Trick, M. A. (1989). The computational difficulty of manipulating an election. *Social Choice and Welfare*, *6*, 227–241.

Barto, A. G., Bradtke, S. J., and Singh, S. (1995). Learning to act using real-time dynamic programming. *AIJ*, *73*, 81–138.

Barto, A. G., Sutton, R. S., and Brouwer, P. S. (1981). Associative search network: A reinforcement learning associative memory. *Biological Cybernetics*, *40*, 201–211.

Barwise, J. and Etchemendy, J. (2002). *Language, Proof and Logic*. CSLI Press.

Baum, E., Boneh, D., and Garrett, C. (1995). On genetic algorithms. In *COLT-95*.

Baum, E. and Smith, W. D. (1997). A Bayesian approach to relevance in game playing. *AIJ*, *97*, 195–242.

Baum, L. E. and Petrie, T. (1966). Statistical inference for probabilistic functions of finite state Markov chains. *Annals of Mathematical Statistics*, *41*, 1554–1563.

Baxter, J. and Bartlett, P. (2000). Reinforcement learning in POMDPs via direct gradient ascent. In *ICML-00*.

Bayardo, R. J. and Agrawal, R. (2005). Data privacy through optimal k-anonymization. In *Proc. 21st Int'l Conf. on Data Engineering*.

Bayardo, R. J. and Miranker, D. P. (1994). An optimal backtrack algorithm for tree-structured constraint satisfaction problems. *AIJ*, *71*, 159–181.

Bayardo, R. J. and Schrag, R. C. (1997). Using CSP look-back techniques to solve real-world SAT instances. In *AAAI-97*.

Bayes, T. (1763). An essay towards solving a problem in the doctrine of chances. *Phil. Trans. Roy. Soc.*, *53*, 370–418.

Beal, J. and Winston, P. H. (2009). The new frontier of human-level artificial intelligence. *IEEE Intelligent Systems*, *24*, 21–23.

Beardon, A. F., Candeal, J. C., Herden, G., Indurain, E., and Mehta, G. B. (2002). The non-existence of a utility function and the structure of non-representable preference relations. *Journal of Mathematical Economics*, *37*, 17–38.

Beattie, C., Leibo, J. Z., Teplyashin, D., Ward, T., Wainwright, M., Küttler, H., Lefrancq, A., Green, S., Valdés, V, Sadik, A., Schrittwieser, J., Anderson, K., York, S., Cant, M., Cain, A., Bolton, A., Gaffney, S., King, H., Hassabis, D., Legg, S., and Petersen, S. (2016). DeepMind lab. arXiv:1612.03801.

Bechhofer, R. (1954). A single-sample multiple decision procedure for ranking means of normal populations with known variances. *Annals of Mathematical Statistics*, *25*, 16–39.

Beck, J. C., Feng, T. K., and Watson, J.-P. (2011). Combining constraint programming and local search for job-shop scheduling. *INFORMS Journal on Computing*, *23*, 1–14.

Beckert, B. and Posegga, J. (1995). Leantap: Lean, tableau-based deduction. *JAR*, *15*, 339–358.

Beeri, C., Fagin, R., Maier, D., and Yannakakis, M. (1983). On the desirability of acyclic database schemes. *JACM*, *30*, 479–513.

Bekey, G. (2008). *Robotics: State Of The Art And Future Challenges*. Imperial College Press.

Belkin, M., Hsu, D., Ma, S., and Mandal, S. (2019). Reconciling modern machine-learning practice and the classical bias–variance trade-off. *PNAS, 116*, 15849–15854.

Bell, C. and Tate, A. (1985). Using temporal constraints to restrict search in a planner. In *Proc. Third Alvey IKBS SIG Workshop.*

Bell, J. L. and Machover, M. (1977). *A Course in Mathematical Logic.* Elsevier.

Bellamy, E. (2003). *Looking Backward: 2000-1887.* Broadview Press.

Bellamy, R. K. E., Dey, K., Hind, M., Hoffman, S. C., Houde, S., Kannan, K., Lohia, P., Martino, J., Mehta, S., Mojsilovic, A., Nagar, S., Ramamurthy, K. N., Richards, J. T., Saha, D., Sattigeri, P., Singh, M., Varshney, K. R., and Zhang, Y. (2018). AI fairness 360: An extensible toolkit for detecting, understanding, and mitigating unwanted algorithmic bias. arXiv:1810.01943.

Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *JAIR, 47*, 253–279.

Bellman, R. E. (1952). On the theory of dynamic programming. *PNAS, 38*, 716–719.

Bellman, R. E. (1958). On a routing problem. *Quarterly of Applied Mathematics, 16*.

Bellman, R. E. (1961). *Adaptive Control Processes: A Guided Tour*. Princeton University Press.

Bellman, R. E. (1965). On the application of dynamic programming to the determination of optimal play in chess and checkers. *PNAS*, 53, 244–246.

Bellman, R. E. (1984). *Eye of the Hurricane.* World Scientific.

Bellman, R. E. and Dreyfus, S. E. (1962). *Applied Dynamic Programming.* Princeton University Press.

Bellman, R. E. (1957). *Dynamic Programming.* Princeton University Press.

Ben-Tal, A. and Nemirovski, A. (2001). *Lectures on Modern Convex Optimization: Analysis, Algorithms, and Engineering Applications*. SIAM (Society for Industrial and Applied Mathematics).

Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, *5*, 157–166.

Bengio, Y. and Bengio, S. (2001). Modeling highdimensional discrete data with multi-layer neural networks. In *NeurIPS 13*.

Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003). A neural probabilistic language model. *JMLR, 3,* 1137–1155.

Bengio, Y. and LeCun, Y. (2007). Scaling learning algorithms towards AI. In Bottou, L., Chapelle, O., DeCoste, D., and Weston, J. (Eds.), *Large-Scale Kernel Machines*. MIT Press.

Benjamin, M. (2013). *Drone Warfare: Killing by Remote Control.* Verso Books.

Bentham, J. (1823). *Principles of Morals and Legislation*. Oxford University Press, Oxford. Original work published in 1789.

Benzmüller, C. and Paleo, B. W. (2013). Formalization, mechanization and automation of Godel's proof of God's existence. arXiv:1308.4526.

Beresniak, A., Medina-Lara, A., Auray, J. P, De Wever, A., Praet, J.-C., Tarricone, R., Torbica, A., Dupont, D., Lamure, M., and Duru, G. (2015). Validation of the underlying assumptions of the quality-adjusted life-years outcome: Results from the ECHOUTCOME European project. *PharmacoEconomics*, *33*, 61–69.

Berger, J. O. (1985). *Statistical Decision Theory and Bayesian Analysis*. Springer Verlag.

Bergstra, J. and Bengio, Y. (2012). Random search for hyper-parameter optimization. *JMLR*, *13*, 281–305.

Berk, R., Heidari, H., Jabbari, S., Kearns, M., and Roth, A. (2017). Fairness in criminal justice risk assessments: The state of the art. arXiv:1703.09207.

Berkson, J. (1944). Application of the logistic function to bio-assay. *JASA*, 39, 357–365.

Berleur, J. and Brunnstein, K. (2001). *Ethics of Computing: Codes, Spaces for Discussion and Law*. Chapman and Hall.

Berlin, K., Koren, S., Chin, C.-S., Drake, J. P., Landolin, J. M., and Phillippy, A. M. (2015). Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. *Nature Biotechnology*, *33*, 623.

Berliner, H. J. (1979). The B* tree search algorithm: A best-first proof procedure. *AIJ*, *12*, 23–40.

Berliner, H. J. (1980a). Backgammon computer program beats world champion. *AIJ*, *14*, 205–220.

Berliner, H. J. (1980b). Computer backgammon. *Scientific American*, *249*, 64–72.

Bermúdez-Chacón, R., Gonnet, G. H., and Smith, K. (2015). Automatic problem-specific hyperparameter optimization and model selection for supervised machine learning. Tech. rep., ETH Zurich.

Bernardo, J. M. and Smith, A. (1994). *Bayesian Theory.* Wiley.

Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The semantic web. *Scientific American*, *284*, 34–43.

Bernoulli, D. (1738). Specimen theoriae novae de mensura sortis. *Proc. St. Petersburg Imperial Academy of Sciences*, *5*, 175–192.

Bernstein, P. L. (1996). *Against the Gods: The Remarkable Story of Risk*. Wiley.

Berrada, L., Zisserman, A., and Kumar, M. P. (2019). Training neural networks for and by interpolation. arXiv:1906.05661.

Berrou, C., Glavieux, A., and Thitimajshima, P. (1993). Near Shannon limit error control-correcting coding and decoding: Turbo-codes. 1. In *Proc. IEEE International Conference on Communications*.

Berry, D. A. and Fristedt, B. (1985). *Bandit Problems: Sequential Allocation of Experiments.* Chapman and Hall.

Bertele, U. and Brioschi, F. (1972). *Nonserial Dynamic Programming.* Academic Press.

Bertoli, P., Cimatti, A., and Roveri, M. (2001a). Heuristic search + symbolic model checking = efficient conformant planning. In *IJCAI-01*.

Bertoli, P., Cimatti, A., Roveri, M., and Traverso, P. (2001b). Planning in nondeterministic domains under partial observability via symbolic model checking. In *IJCAI-01*.

Bertot, Y., Casteran, P., Huet, G., and Paulin-Mohring, C. (2004). *Interactive Theorem Proving and Program Development*. Springer.

Bertsekas, D. (1987). *Dynamic Programming: Deterministic and Stochastic Models.* Prentice-Hall.

Bertsekas, D. and Tsitsiklis, J. N. (1996). *NeuroDynamic Programming*. Athena Scientific.

Bertsekas, D. and Tsitsiklis, J. N. (2008). *Introduction to Probability* (2nd edition). Athena Scientific.

Bertsekas, D. and Shreve, S. E. (2007). *Stochastic Optimal Control: The Discrete-Time Case*. Athena Scientific.

Bertsimas, D., Delarue, A., and Martin, S. (2019). Optimizing schools' start time and bus routes. *PNAS*, *116 13*, 5943–5948.

Bertsimas, D. and Dunn, J. (2017). Optimal classification trees. *Machine Learning*, *106*, 1039–1082.

Bessen, J. (2015). *Learning by Doing: The Real Connection between Innovation, Wages, and Wealth*. Yale University Press.

Bessière, C. (2006). Constraint propagation. In Rossi, F., van Beek, P., and Walsh, T. (Eds.), *Handbook of Constraint Programming*. Elsevier.

Beutel, A., Chen, J., Doshi, T., Qian, H., Woodruff, A., Luu, C., Kreitmann, P., Bischof, J., and Chi, E. H. (2019). Putting fairness principles into practice: Challenges, metrics, and improvements. arXiv:1901.04562.

Bhar, R. and Hamori, S. (2004). *Hidden Markov Models: Applications to Financial Economics*. Springer.

Bibel, W. (1993). *Deduction: Automated Logic*. Academic Press.

Bien, J., Tibshirani, R., *et al*. (2011). Prototype selection for interpretable classification. *Annals of Applied Statistics*, *5*, 2403–2424.

Biere, A., Heule, M., van Maaren, H., and Walsh, T. (Eds.). (2009). *Handbook of Satisfiability*. IOS Press.

Bies, A., Mott, J., and Warner, C. (2015). English news text treebank: Penn treebank revised. Linguistic Data Consortium.

Billings, D., Burch, N., Davidson, A., Holte, R. C., Schaeffer, J., Schauenberg, T., and Szafron, D. (2003). Approximating game-theoretic optimal strategies for full-scale poker. In *IJCAI-03*.

Billingsley, P. (2012). *Probability and Measure* (4th edition). Wiley.

Binder, J., Koller, D., Russell, S. J., and Kanazawa, K. (1997a). Adaptive probabilistic networks with hidden variables. *Machine Learning, 29,* 213–244.

Binder, J., Murphy, K., and Russell, S. J. (1997b). Space-efficient inference in dynamic probabilistic networks. In *IJCAI-97*.

Bingham, E., Chen, J., Jankowiak, M., Obermeyer, F., Pradhan, N., Karaletsos, T., Singh, R., Szerlip, P, Horsfall, P., and Goodman, N. D. (2019). Pyro: Deep universal probabilistic programming. *JMLR, 20,* 1–26.

Binmore, K. (1982). *Essays on Foundations of Game Theory*. Pitman.

Biran, O. and Cotton, C. (2017). Explanation and justification in machine learning: A survey. In *Proc.IJCAI-17 Workshop on Explainable AI.*

Bishop, C. M. (1995). *Neural Networks for Pattern Recognition.* Oxford University Press.

Bishop, C. M. (2007). *Pattern Recognition and Machine Learning.* Springer-Verlag.

Bisson, T. (1990). They're made out of meat. *Omni Magazine.*

Bistarelli, S., Montanari, U., and Rossi, F. (1997). Semiring-based constraint satisfaction and optimization. *JACM, 44*, 201–236.

Bitner, J. R. and Reingold, E. M. (1975). Backtrack programming techniques. *CACM, 18*, 651–656.

Bizer, C., Auer, S., Kobilarov, G., Lehmann, J., and Cyganiak, R. (2007). DBPedia – querying Wikipedia like a database. In *16th International Conference on World Wide Web*.

Blazewicz, J., Ecker, K., Pesch, E., Schmidt, G., and Weglarz, J. (2007). *Handbook on Scheduling: Models and Methods for Advanced Planning.* Springer-Verlag.

Blei, D. M., Ng, A. Y., and Jordan, M. I. (2002). Latent Dirichlet allocation. In *NeurIPS 14*.

Bliss, C. I. (1934). The method of probits. *Science*, 79, 38–39.

Block, H. D., Knight, B., and Rosenblatt, F. (1962). Analysis of a four-layer series-coupled perceptron. *Rev. Modern Physics*, *34*, 275–282.

Block, N. (2009). Comparing the major theories of consciousness. In Gazzaniga, M. S. (Ed.), *The Cognitive Neurosciences*. MIT Press.

Blum, A. L. and Furst, M. (1997). Fast planning through planning graph analysis. *AIJ*, *90*, 281–300.

Blum, A. L. (1996). On-line algorithms in machine learning. In *Proc. Workshop on On-Line Algorithms, Dagstuhl*.

Blum, A. L., Hopcroft, J., and Kannan, R. (2020). *Foundations of Data Science*. Cambridge University Press.

Blum, A. L. and Mitchell, T. M. (1998). Combining labeled and unlabeled data with co-training. In *COLT-98*.

Blumer, A., Ehrenfeucht, A., Haussler, D., and Warmuth, M. (1989). Learnability and the Vapnik-Chervonenkis dimension. *JACM*, *36*, 929–965.

Bobrow, D. G. (1967). Natural language input for a computer problem solving system. In Minsky, M. L. (Ed.), *Semantic Information Processing*. MIT Press.

Bod, R. (2008). The data-oriented parsing approach: Theory and application. In *Computational Intelligence: A Compendium*. Springer-Verlag.

Bod, R., Scha, R., and Sima'an, K. (2003). *Data-Oriented Parsing*. CSLI Press.

Boddington, P. (2017). *Towards a Code of Ethics for Artificial Intelligence*. Springer-Verlag.

Boden, M. A. (Ed.). (1990). *The Philosophy of Artificial Intelligence*. Oxford University Press.

Bolognesi, A. and Ciancarini, P. (2003). Computer programming of kriegspiel endings: The case of KR vs. K. In *Advances in Computer Games 10*.

Bolton, R. J. and Hand, D. J. (2002). Statistical fraud detection: A review. *Statistical science*, *17*, 235–249.

Bonawitz, K., Ivanov, V., Kreuter, B., Marcedone, A., McMahan, H. B., Patel, S., Ramage, D., Segal, A., and Seth, K. (2017). Practical secure aggregation for privacy-preserving machine learning. In *Proc. ACM SIGSAC Conference on Computer and Communications Security*.

Bond, A. H. and Gasser, L. (Eds.). (1988). *Readings in Distributed Artificial Intelligence*. Morgan Kaufmann.

Bonet, B. (2002). An epsilon-optimal grid-based algorithm for partially observable Markov decision processes. In *ICML-02*.

Bonet, B. and Geffner, H. (1999). Planning as heuristic search: New results. In *ECP-99*.

Bonet, B. and Geffner, H. (2000). Planning with incomplete information as heuristic search in belief space. In *ICAPS-00*.

Bonet, B. and Geffner, H. (2005). An algorithm better than AO*? In *AAAI-05*.

Boole, G. (1847). *The Mathematical Analysis of Logic: Being an Essay towards a Calculus of Deductive Reasoning*. Macmillan, Barclay, and Macmillan.

Booth, T. L. (1969). Probabilistic representation of formal languages. In *IEEE Conference Record of the 1969 Tenth Annual Symposium on*

Switching and Automata Theory.

Borel, E. (1921). La théorie du jeu et les équations intégrales à noyau symétrique. *Comptes Rendus Hebdomadaires des Séances de l'Académie des Sciences, 173,* 1304–1308.

Borenstein, J., Everett, B., and Feng, L. (1996). *Navigating Mobile Robots: Systems and Techniques.* A. K. Peters, Ltd.

Borenstein, J. and Koren., Y. (1991). The vector field histogram–Fast obstacle avoidance for mobile robots. *IEEE Transactions on Robotics and Automation, 7,* 278–288.

Borgida, A., Brachman, R. J., McGuinness, D., and Alperin Resnick, L. (1989). CLASSIC: A structural data model for objects. *SIGMOD Record, 18,* 58–67.

Boroditsky, L. (2003). Linguistic relativity. In Nadel, L. (Ed.), *Encyclopedia of Cognitive Science.* Macmillan.

Boser, B., Guyon, I., and Vapnik, V. N. (1992). A training algorithm for optimal margin classifiers. In *COLT-92*.

Bosse, M., Newman, P., Leonard, J., Soika, M., Feiten, W., and Teller, S. (2004). Simultaneous localization and map building in large-scale cyclic environments using the Atlas framework. *Int. J. Robotics Research, 23,* 1113–1139.

Bostrom, N. (2005). A history of transhumanist thought. *Journal of Evolution and Technology, 14*, 1– 25.

Bostrom, N. (2014). *Superintelligence: Paths, Dangers, Strategies.* Oxford University Press.

Bottou, L. and Bousquet, O. (2008). The tradeoffs of large scale learning. In *NeurIPS 20*.

Bottou, L., Curtis, F. E., and Nocedal, J. (2018). Optimization methods for large-scale machine learning. *SIAM Review, 60*, 223–311.

Boué, L. (2019). Real numbers, data science and chaos: How to fit any dataset with a single parameter. arXiv:1904.12320.

Bousmalis, K., Irpan, A., Wohlhart, P, Bai, Y., Kelcey, M., Kalakrishnan, M., Downs, L., Ibarz, J., Pastor, P., Konolige, K., Levine, S., and Vanhoucke, V. (2017). Using simulation and domain adaptation to improve efficiency of deep robotic grasping. arXiv:1709.07857.

Boutilier, C. (2002). A POMDP formulation of preference elicitation problems. In *AAAI-02*.

Boutilier, C. and Brafman, R. I. (2001). Partial-order planning with concurrent interacting actions. *JAIR, 14,* 105–136.

Boutilier, C., Dearden, R., and Goldszmidt, M. (2000). Stochastic dynamic programming with factored representations. AIJ, *121*, 49–107.

Boutilier, C., Reiter, R., and Price, B. (2001). Symbolic dynamic programming for first-order MDPs. In *IJCAI-01*.

Biran, O. and Cotton, C. (2017). Explanation and justification in machine learning: A survey. In *Proc. IJCAI-17 Workshop on Explainable AI.*

Boutilier, C., Brafman, R. I., Domshlak, C., Hoos, H. H., and Poole, D. (2004). CP-nets: A tool for representing and reasoning with conditional ceteris paribus preference statements. *JAIR, 21*, 135–191.

Boutilier, C., Friedman, N., Goldszmidt, M., and Koller, D. (1996). Context-specific independence in Bayesian networks. In *UAI-96.*

Bouzy, B. and Cazenave, T. (2001). Computer Go: An AI oriented survey. *AIJ*, *132*, 39–103.

Bowling, M., Burch, N., Johanson, M., and Tammelin, O. (2015). Heads-up limit hold’em poker is solved. *Science*, *347*, 145–149.

Bowling, M., Johanson, M., Burch, N., and Szafron, D. (2008). Strategy evaluation in extensive games with importance sampling. In *ICML-08*.

Bowman, S., Angeli, G., Potts, C., and Manning, C. (2015). A large annotated corpus for learning natural language inference. In *EMNLP-15.*

Box, G. E. P. (1957). Evolutionary operation: A method of increasing industrial productivity. *Applied Statistics*, *6*, 81–101.

Box, G. E. P., Jenkins, G., Reinsel, G., and Ljung, G. M. (2016). *Time Series Analysis: Forecasting and Control* (5th edition). Wiley.

Box, G. E. P. and Tiao, G. C. (1973). *Bayesian Inference in Statistical Analysis*. Addison-Wesley.

Boyan, J. A. and Moore, A. W. (1998). Learning evaluation functions for global optimization and Boolean satisfiability. In *AAAI-98*.

Boyd, S. and Vandenberghe, L. (2004). *Convex Optimization*. Cambridge University Press.

Boyen, X., Friedman, N., and Koller, D. (1999). Discovering the hidden structure of complex dynamic systems. In *UAI-99.*

Boyer, R. S. and Moore, J. S. (1979). *A Computational Logic*. Academic Press.

Boyer, R. S. and Moore, J. S. (1984). Proof checking the RSA public key encryption algorithm. *American Mathematical Monthly*, *91*, 181–189.

Brachman, R. J. (1979). On the epistemological status of semantic networks. In Findler, N. V. (Ed.), *Associative Networks: Representation and Use of Knowledge by Computers*. Academic Press.

Brachman, R. J. and Levesque, H. J. (Eds.). (1985). *Readings in Knowledge Representation.* Morgan Kaufmann.

Bradt, R. N., Johnson, S. M., and Karlin, S. (1956). On sequential designs for maximizing the sum of n observations. *Ann. Math. Statist.*, *27*, 1060–1074.

Brafman, O. and Brafman, R. (2009). *Sway: The Irresistible Pull of Irrational Behavior*. Broadway Business.

Brafman, R. I. and Domshlak, C. (2008). From one to many: Planning for loosely coupled multi-agent systems. In *ICAPS-08*.

Brafman, R. I. and Tennenholtz, M. (2000). A near optimal polynomial time algorithm for learning in certain classes of stochastic games. *AIJ*, *121*, 31–47.

Braitenberg, V. (1984). *Vehicles: Experiments in Synthetic Psychology*. MIT Press.

Brandt, F., Conitzer, V., Endriss, U., Lang, J., and Procaccia, A. D. (Eds.). (2016). *Handbook of Computational Social Choice*. Cambridge University Press.

Brants, T. (2000). TnT: A statistical part-of-speech tagger. In *Proc. Sixth Conference on Applied Natural Language Processing*.

Brants, T., Popat, A. C., Xu, P., Och, F. J., and Dean, J. (2007). Large language models in machine translation. In *EMNLP-CoNLL-07*.

Bratko, I. (2009). *Prolog Programming for Artificial Intelligence* (4th edition). Addison-Wesley.

Bratman, M. E. (1987). *Intention, Plans, and Practical Reason.* Harvard University Press.

Breck, E., Cai, S., Nielsen, E., Salib, M., and Sculley, D. (2016). What's your ML test score? A rubric for ML production systems. In *Proc. NIPS 2016 Workshop on Reliable Machine Learning in the Wild.*

Breese, J. S. (1992). Construction of belief and decision networks. *Computational Intelligence*, *8*, 624–647.

Breese, J. S. and Heckerman, D. (1996). Decision-theoretic troubleshooting: A framework for repair and experiment. In *UAI-96.*

Breiman, L., Friedman, J., Olshen, R. A., and Stone, C. J. (1984). *Classification and Regression Trees.* Wadsworth International Group.

Breiman, L. (2001). Random forests. *Machine Learning*, 45(1), 5–32.

Brelaz, D. (1979). New methods to color the vertices of agraph. *CACM*, *22*, 251–256.

Brent, R. P. (1973). *Algorithms for Minimization without Derivatives.* Prentice-Hall.

Bresnan, J. (1982). *The Mental Representation of Grammatical Relations.* MIT Press.

Brewka, G., Dix, J., and Konolige, K. (1997). *Nononotonic Reasoning: An Overview.* Center for the Study of Language and Information (CSLI).

Brickley, D. and Guha, R. V. (2004). RDF vocabulary description language 1.0: RDF schema. Tech. rep., W3C.

Briggs, R. (1985). Knowledge representation in Sanskrit and artificial intelligence. *AIMag*, *6*, 32–39.

Brill, E. (1992). A simple rule-based part of speech tagger. In *Proc. Third Conference on Applied Natural Language Processing*.

Brin, D. (1998). *The Transparent Society.* Perseus.

Brin, S. and Page, L. (1998). The anatomy of a large-scale hypertextual web search engine. In *Proc. Seventh World Wide Web Conference.*

Bringsjord, S. (2008). If I were judge. In Epstein, R., Roberts, G., and Beber, G. (Eds.), *Parsing the Turing Test*. Springer.

Broadbent, D. E. (1958). *Perception and Communication*. Pergamon.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). OpenAI gym. arXiv:1606.01540.

Brooks, R. A. (1986). A robust layered control system for a mobile robot. *IEEE J. of Robotics and Automation*, *2*, 14–23.

Brooks, R. A. (1989). Engineering approach to building complete, intelligent beings. *Proc. SPIE–the International Society for Optical Engineering*, *1002*, 618–625.

Brooks, R. A. (1991). Intelligence without representation. *AIJ*, *47*, 139–159.

Brooks, R. A. and Lozano-Perez, T. (1985). A subdivision algorithm in configuration space for findpath with rotation. *IEEE Transactions on Systems, Man and Cybernetics*, *15*, 224–233.

Brooks, R. A. (2017). The seven deadly sins of AI predictions. *MIT Technology Review*, *Oct 6*.

Brooks, S., Gelman, A., Jones, G., and Meng, X.-L. (2011). *Handbook of Markov Chain Monte Carlo*. Chapman & Hall/CRC.

Brown, C., Finkelstein, L., and Purdom, P. (1988). Backtrack searching in the presence of symmetry. In Mora, T. (Ed.), *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*. Springer-Verlag.

Brown, K. C. (1974). A note on the apparent bias of net revenue estimates. *J. Finance*, *29*, 1215–1216.

Brown, N. and Sandholm, T. (2017). Libratus: The superhuman AI for no-limit poker. In *IJCAI-17*.

Brown, N. and Sandholm, T. (2019). Superhuman AI for multiplayer poker. *Science*, *365*, 885–890.

Brown, P. F., Cocke, J., Della Pietra, S. A., Della Pietra, V. J., Jelinek, F., Mercer, R. L., and Roossin, P. (1988). A statistical approach to language translation. In *COLING-88*.

Brown, P. F., Desouza, P. V., Mercer, R. L., Pietra, V. J. D., and Lai, J. C. (1992). Class-based n-grammodels of natural language. *Computational linguistics*, *18*(4).

Browne, C., Powley, E. J., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Liebana, D. P., Samothrakis, S., and Colton, S. (2012). A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, *4*, 1–43.

Brownston, L., Farrell, R., Kant, E., and Martin, N. (1985). *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*. Addison-Wesley.

Bruce, V., Green, P., and Georgeson, M. (2003). *Visual Perception: Physiology, Psychology and Ecology*. Routledge and Kegan Paul.

Brügmann, B. (1993). Monte Carlo Go. Tech. rep., Department of Physics, Syracuse University.

Bryce, D. and Kambhampati, S. (2007). A tutorial on planning graph-based reachability heuristics. *AIMag*, *Spring*, 47–83.

Bryce, D., Kambhampati, S., and Smith, D. E. (2006). Planning graph heuristics for belief space search. *JAIR*, *26*, 35–99.

Brynjolfsson, E. and McAfee, A. (2011). *Race Against the Machine*. Digital Frontier Press.

Brynjolfsson, E. and McAfee, A. (2014). *The Second Machine Age*. W. W. Norton.

Brynjolfsson, E., Rock, D., and Syverson, C. (2018). Artificial intelligence and the modern productivity paradox: A clash of expectations and statistics. In Agrawal, A., Gans, J., and Goldfarb, A. (Eds.), *The Economics of Artificial Intelligence: An Agenda*. University of Chicago Press.

Bryson, A. E. and Ho, Y.-C. (1969). *Applied Optimal Control*. Blaisdell.

Bryson, A. E. (1962). A gradient method for optimizing multi-stage allocation processes. In *Proc. of a Harvard Symposium on Digital Computers and Their Applications*.

Bryson, J. J. (2012). A role for consciousness in action selection. *International Journal of Machine Consciousness*, *4*, 471–482.

Bryson, J. J. and Winfield, A. (2017). Standardizing ethical design for artificial intelligence and autonomous systems. *Computer, 50*, 116–119.

Buchanan, B. G., Mitchell, T. M., Smith, R. G., and Johnson, C. R. (1978). Models of learning systems. In *Encyclopedia of Computer Science and Technology*, Vol. 11. Dekker.

Buchanan, B. G. and Shortliffe, E. H. (Eds.). (1984). *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project.* Addison-Wesley.

Buchanan, B. G., Sutherland, G. L., and Feigenbaum, E. A. (1969). Heuristic DENDRAL: A program for generating explanatory hypotheses in organic chemistry. In Meltzer, B., Michie, D., and Swann, M. (Eds.), *Machine Intelligence 4*. Edinburgh University Press.

Buck, C., Heafield, K., and Van Ooyen, B. (2014). N-gram counts and language models from the common crawl. In *Proc. International Conference on Language Resources and Evaluation*.

Buehler, M., Iagnemma, K., and Singh, S. (Eds.). (2006). *The 2005 DARPA Grand Challenge: The Great Robot Race*. Springer-Verlag.

Buffon, G. (1777). Essai d’arithmetique morale. Supplement to Histoire naturelle, vol. IV.

Bunt, H. C. (1985). The formal representation of (quasi-) continuous concepts. In Hobbs, J. R. and Moore, R. C. (Eds.), *Formal Theories of the Commonsense World*. Ablex.

Buolamwini, J. and Gebru, T. (2018). Gender shades: Intersectional accuracy disparities in commercial gender classification. In *Conference on Fairness, Accountability and Transparency.*

Burgard, W., Cremers, A. B., Fox, D., Hahnel, D., Lakemeyer, G., Schulz, D., Steiner, W., and Thrun, S. (1999). Experiences with an interactive museum tourguide robot. AIJ, *114*, 3–55.

Burkov, A. (2019). *The Hundred-Page Machine Learning Book*. Burkov.

Burns, E., Hatem, M., Leighton, M. J., and Ruml, W. (2012). Implementing fast heuristic search code. In *Symposium on Combinatorial Search*.

Buro, M. (1995). ProbCut: An effective selective extension of the alpha-beta algorithm. *J. International Computer Chess Association, 18*, 71–76.

Buro, M. (2002). Improving heuristic mini-max search by supervised learning. *AIJ*, *134*, 85–99.

Burstein, J., Leacock, C., and Swartz, R. (2001). Automated evaluation of essays and short answers. In *Fifth International Computer Assisted Assessment Conference*.

Burton, R. (2009). *On Being Certain: Believing You Are Right Even When You're Not.* St. Martin’s Griffin.

Buss, D. M. (2005). *Handbook of Evolutionary Psychology*. Wiley.

Butler, S. (1863). Darwin among the machines. *The Press (Christchurch, New Zealand), June 13.*

Bylander, T. (1994). The computational complexity of propositional STRIPS planning. AIJ, 69, 165–204.

Byrd, R. H., Lu, P., Nocedal, J., and Zhu, C. (1995). A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific and Statistical Computing*, *16*, 1190–1208.

Cabeza, R. and Nyberg, L. (2001). Imaging cognition II: An empirical review of 275 PET and fMRI studies. *J. Cognitive Neuroscience*, *12*, 1–47.

Cafarella, M. J., Halevy, A., Zhang, Y., Wang, D. Z., and Wu, E. (2008). Webtables: Exploring the power of tables on the web. In *VLDB-08*.

Calvanese, D., Lenzerini, M., and Nardi, D. (1999). Unifying class-based representation formalisms. *JAIR*, *11*, 199–240.

Camacho, R. and Michie, D. (1995). Behavioral cloning: A correction. *AIMag*, *16*, 92.

Campbell, D. E. and Kelly, J. (2002). Impossibility theorems in the Arrovian framework. In Arrow, K. J., Sen, A. K., and Suzumura, K. (Eds.), *Handbook of Social Choice and Welfare Volume 1*. Elsevier Science.

Campbell, M. S., Hoane, A. J., and Hsu, F.-H. (2002). Deep Blue. *AIJ*, *134*, 57–83.

Cannings, C., Thompson, E., and Skolnick, M. H. (1978). Probability functions on complex pedigrees. *Advances in Applied Probability*, *10*, 26–61.

Canny, J. and Reif, J. (1987). New lower bound techniques for robot motion planning problems. In *FOCS-87*.

Canny, J. (1986). A computational approach to edge detection. *PAMI*, *8*, 679–698.

Canny, J. (1988). *The Complexity of Robot Motion Planning*. MIT Press.

Capen, E., Clapp, R., and Campbell, W. (1971). Competitive bidding in high-risk situations. *J. Petroleum Technology, 23*, 641–653.

Carbonell, J. G. (1983). Derivational analogy and its role in problem solving. In *AAAI-83*.

Carbonell, J. G., Knoblock, C. A., and Minton, S. (1989). PRODIGY: An integrated architecture for planning and learning. Technical report, Computer Science Department, Carnegie-Mellon University.

Carbonnel, C. and Cooper, M. C. (2016). Tractability in constraint satisfaction problems: A survey. *Constraints*, *21*(2), 115–144.

Cardano, G. (1663). *Liber de ludo aleae*. Lyons.

Carlini, N., Athalye, A., Papernot, N., Brendel, W., Rauber, J., Tsipras, D., Goodfellow, I., Madry, A., and Kurakin, A. (2019). On evaluating adversarial robustness. arXiv:1902.06705.

Carnap, R. (1928). *Der logische Aufbau der Welt*. Weltkreis-verlag. Translated into English as The Logical Structure of the World (Carnap, 1967).

Carnap, R. (1948). On the application of inductive logic. *Philosophy and Phenomenological Research*, *8*, 133–148.

Carnap, R. (1950). *Logical Foundations of Probability*. University of Chicago Press.

Carpenter, B., Gelman, A., Hoffman, M., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M., Guo, J., Li, P., and Riddell, A. (2017). Stan: A probabilistic programming language. *Journal of Statistical Software, 76*, 1–32.

Carroll, S. (2007). *The Making of the Fittest: DNA and the Ultimate Forensic Record of Evolution.* Norton.

Casati, R. and Varzi, A. (1999). *Parts and Places: The Structures of Spatial Representation*. MIT Press.

Cassandra, A. R., Kaelbling, L. P., and Littman, M. L. (1994). Acting optimally in partially observable stochastic domains. In *AAAI-94.*

Cassandras, C. G. and Lygeros, J. (2006). *Stochastic Hybrid Systems.* CRC Press.

Castro, R., Coates, M., Liang, G., Nowak, R., and Yu, B. (2004). Network tomography: Recent developments. *Statistical Science, 19*, 499–517.

Cauchy, A. (1847). Méthode générale pour la résolution des systèmes d'équations simultanées. *Comp. Rend. Sci. Paris, 25*, 536–538.

Cesa-Bianchi, N. and Lugosi, G. (2006). *Prediction, Learning, and Games.* Cambridge University Press.

Chajewska, U., Koller, D., and Parr, R. (2000). Making rational decisions using adaptive utility elicitation. In *AAAI-00.*

Chakrabarti, P. P., Ghose, S., Acharya, A., and de Sarkar, S. C. (1989). Heuristic search in restricted memory. *AIJ, 41*, 197–222.

Chalkiadakis, G., Elkind, E., and Wooldridge, M. (2011). *Computational Aspects of Cooperative Game Theory*. Morgan Kaufmann.

Chalmers, D. J. (1992). Subsymbolic computation and the Chinese room. In Dinsmore, J. (Ed.), *The symbolic and connectionist paradigms: Closing the gap*. Lawrence Erlbaum.

Chandola, V., Banerjee, A., and Kumar, V. (2009). Anomaly detection: A survey. *ACM Computing Surveys*, *41*.

Chandra, A. K. and Harel, D. (1980). Computable queries for relational data bases. *J. Computer and System Sciences*, *21*, 156–178.

Chang, C.-L. and Lee, R. C.-T. (1973). *Symbolic Logic and Mechanical Theorem Proving*. Academic Press.

Chang, H. S., Fu, M. C., Hu, J., and Marcus, S. I. (2005). An adaptive sampling algorithm for solving Markov decision processes. *Operations Research*, 53, 126–139.

Chao, W.-L., Hu, H., and Sha, F. (2018). Being negative but constructively: Lessons learnt from creating better visual question answering datasets. In *ACL-18*.

Chapman, D. (1987). Planning for conjunctive goals. *AIJ*, *32*, 333–377.

Charniak, E. (1993). *Statistical Language Learning*. MIT Press.

Charniak, E. (1996). Tree-bank grammars. In *AAAI-96*.

Charniak, E. (1997). Statistical parsing with a context-free grammar and word statistics. In *AAAI-97*.

Charniak, E. and Goldman, R. (1992). A Bayesian model of plan recognition. AIJ, *64*, 53–79.

Charniak, E., Riesbeck, C., McDermott, D., and Meehan, J. (1987). *Artificial Intelligence Programming*(2nd edition). Lawrence Erlbaum.

Charniak, E. (1991). Bayesian networks without tears. *AIMag*, *12*, 50–63.

Charniak, E. (2018). *Introduction to Deep Learning*. MIT Press.

Chaslot, G., Bakkes, S., Szita, I., and Spronck, P. (2008). Monte-Carlo tree search: A new framework for game AI. In *Proc. Fourth Artificial Intelligence and Interactive Digital Entertainment Conference.*

Chater, N. and Oaksford, M. (Eds.). (2008). *The Probabilistic Mind: Prospects for Bayesian Cognitive Science*. Oxford University Press.

Chatfield, C. (1989). *The Analysis of Time Series: An Introduction* (4th edition). Chapman and Hall.

Chavira, M. and Darwiche, A. (2008). On probabilistic inference by weighted model counting. *AIJ*, *172*, 772–799.

Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P. (2002). SMOTE: Synthetic minority over-sampling technique. *JAIR*, *16*, 321–357.

Cheeseman, P. (1985). In defense of probability. In *IJCAI-85*.

Cheeseman, P. (1988). An inquiry into computer understanding. *Computational Intelligence*, *4*, 58–66.

Cheeseman, P., Kanefsky, B., and Taylor, W. (1991). Where the really hard problems are. In *IJCAI-91*.

Cheeseman, P., Self, M., Kelly, J., and Stutz, J. (1988). Bayesian classification. In *AAAI-88.*

Cheeseman, P. and Stutz, J. (1996). Bayesian classification (AutoClass): Theory and results. In Fayyad, U., Piatesky-Shapiro, G., Smyth, P., and Uthurusamy, R. (Eds.), *Advances in Knowledge Discovery and Data Mining*. AAAI Press/MIT Press.

Chen, D. and Manning, C. (2014). A fast and accurate dependency parser using neural networks. In *EMNLP-14*.

Chen, J., Holte, R. C., Zilles, S., and Sturtevant, N. R. (2017). Front-to-end bidirectional heuristic search with near-optimal node expansions. *IJCAI-17*.

Chen, M. X., Firat, O., Bapna, A., Johnson, M., Macherey, W., Foster, G., Jones, L., Parmar, N., Schuster, M., Chen, Z., Wu, Y., and Hughes, M. (2018). The best of both worlds: Combining recent advances in neural machine translation. In *ACL-18*.

Chen, S. F. and Goodman, J. (1996). An empirical study of smoothing techniques for language modeling. In *ACL-96*.

Chen, T. and Guestrin, C. (2016). XGBoost: A scalable tree boosting system. In *KDD-16*.

Cheng, J. and Druzdzel, M. J. (2000). AIS-BN: An adaptive importance sampling algorithm for evidential reasoning in large Bayesian networks. *JAIR*, *13*, 155–188.

Cheng, J., Greiner, R., Kelly, J., Bell, D. A., and Liu, W. (2002). Learning Bayesian networks from data: An information-theory based approach. *AIJ*, *137*, 43–90.

Chiu, C., Sainath, T., Wu, Y., Prabhavalkar, R., Nguyen, P, Chen, Z., Kannan, A., Weiss, R., Rao, K., Gonina, K., Jaitly, N., Li, B., Chorowski, J., and Bacchiani, M. (2017). State-of-the-art speech recognition with sequence-to-sequence models. arXiv:1712.01769.

Chklovski, T. and Gil, Y. (2005). Improving the design of intelligent acquisition interfaces for collecting world knowledge from web contributors. In *Proc. Third International Conference on Knowledge Capture.*

Chollet, F. (2019). On the measure of intelligence. arXiv:1911.01547.

Chollet, F. (2017). *Deep Learning with Python.* Manning.

Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on Information Theory*, *2*, 113–124.

Chomsky, N. (1957). *Syntactic Structures*. Mouton.

Choromanska, A., Henaff, M., Mathieu, M., Arous, G. B., and LeCun, Y. (2014). The loss surface of multilayer networks. arXiv:1412.0233.

Choset, H. (1996). *Sensor Based Motion Planning: The Hierarchical Generalized Voronoi Graph.* Ph.D. thesis, California Institute of Technology.

Choset, H., Hutchinson, S., Lynch, K., Kantor, G., Burgard, W., Kavraki, L., and Thrun, S. (2005). *Principles of Robot Motion: Theory, Algorithms, and Implementation*. MIT Press.

Chouldechova, A. (2017). Fair prediction with disparate impact: A study of bias in recidivism prediction instruments. *Big Data*, *5*, 153–163.

Chouldechova, A. and Roth, A. (2018). The frontiers of fairness in machine learning. arXiv:1810.08810.

Christian, B. (2011). *The Most Human Human.* Doubleday.

Christin, A., Rosenblat, A., and Boyd, D. (2015). Courts and predictive algorithms. *Data & Civil Rights.*

Chung, K. L. (1979). *Elementary Probability Theory with Stochastic Processes* (3rd edition). SpringerVerlag.

Church, A. (1936). A note on the Entscheidungsproblem. *JSL, 1,* 40–41 and 101–102.

Church, A. (1956). *Introduction to Mathematical Logic*. Princeton University Press.

Church, K. (1988). A stochastic parts program and noun phrase parser for unrestricted texts. In *Proc. Second Conference on Applied Natural Language Processing*.

Church, K. and Patil, R. (1982). Coping with syntactic ambiguity or how to put the block in the box on the table. *Computational Linguistics, 8,* 139–149.

Church, K. (2004). Speech and language processing: Can we use the past to predict the future. In *Proc. Conference on Text, Speech, and Dialogue.*

Church, K. and Gale, W. A. (1991). A comparison of the enhanced Good–Turing and deleted estimation methods for estimating probabilities of English bigrams. *Computer Speech and Language*, *5*, 19–54.

Church, K. and Hestness, J. (2019). A survey of 25 years of evaluation. *Natural Language Engineering*, *25*, 753–767.

Churchland, P. M. (2013). *Matter and Consciousness* (3rd edition). MIT Press.

Ciancarini, P. and Favini, G. P. (2010). Monte Carlo tree search in Kriegspiel. AIJ, *174*, 670–684.

Ciancarini, P. and Wooldridge, M. (2001). *Agent-Oriented Software Engineering.* Springer-Verlag.

Cimatti, A., Roveri, M., and Traverso, P. (1998). Automatic OBDD-based generation of universal plans in non-deterministic domains. In *AAAI-98.*

Claret, G., Rajamani, S. K., Nori, A. V., Gordon, A. D., and Borgström, J. (2013). Bayesian inference using data flow analysis. In *Proc. 9th Joint Meeting on Foundations of Software Engineering.*

Clark, A. (1998). *Being There: Putting Brain, Body, and World Together Again.* MIT Press.

Clark, A. (2015). *Surfing Uncertainty: Prediction, Action, and the Embodied Mind.* Oxford University Press.

Clark, K. L. (1978). Negation as failure. In Gallaire, H. and Minker, J. (Eds.), *Logic and Data Bases.* Plenum.

Clark, P., Cowhey, I., Etzioni, O., Khot, T., Sabhar-wal, A., Schoenick, C., and Tafjord, O. (2018). Think you have solved question answering? Try ARC, the AI2 reasoning challenge. arXiv:1803.05457.

Clark, P., Etzioni, O., Khot, T., Mishra, B. D., Richardson, K., *et al.* (2019). From ‘F’ to ‘A’ on the NY Regents science exams: An overview of the Aristo project. arXiv:1909.01958.

Clark, S. and Curran, J. R. (2004). Parsing the WSJ using CCG and log-linear models. In *ACL-04.*

Clarke, A. C. (1968). *2001: A Space Odyssey.* Signet.

Clarke, E. and Grumberg, O. (1987). Research on automatic verification of finite-state concurrent systems. *Annual Review of Computer Science*, *2*, 269–290.

Clearwater, S. H. (Ed.). (1996). *Market-Based Control*. World Scientific.

Clocksin, W. F. and Mellish, C. S. (2003). *Programming in Prolog* (5th edition). Springer-Verlag.

Clocksin, W. F. (2003). *Clause and Effect: Prolog Programming for the Working Programmer*. Springer.

Coase, R. H. (1960). The problem of social cost. *Journal of Law and Economics*, pp. 1–44.

Coates, A., Abbeel, P., and Ng, A. Y. (2009). Apprenticeship learning for helicopter control. *Association for Computing Machinery*, 52(7).

Cobham, A. (1964). The intrinsic computational difficulty of functions. In *Proc. International Congress for Logic, Methodology, and Philosophy of Science*.

Cohen, P. R. (1995). *Empirical Methods for Artificial Intelligence*. MIT Press.

Cohen, P. R. and Levesque, H. J. (1990). Intention is choice with commitment. AIJ, *42*, 213–261.

Cohen, P. R., Morgan, J., and Pollack, M. E. (1990). *Intentions in Communication*. MIT Press.

Cohen, P. R. and Perrault, C. R. (1979). Elements of a plan-based theory of speech acts. *Cognitive Science*, *3*, 177–212.

Cohn, A. G., Bennett, B., Gooday, J. M., and Gotts, N. (1997). RCC: A calculus for region based qualitative spatial reasoning. *GeoInformatica, 1*, 275–316.

Collin, Z., Dechter, R., and Katz, S. (1999). Self-stabilizing distributed constraint satisfaction. *Chicago J. of Theoretical Computer Science*, *1999*.

Collins, M. (1999). *Head-driven Statistical Models for Natural Language Processing.* Ph.D. thesis, University of Pennsylvania.

Collins, M. and Duffy, K. (2002). New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron. In *ACL-02*.

Colmerauer, A. and Roussel, P. (1993). The birth of Prolog. *SIGPLAN Notices*, *28*, 37–52.

Colmerauer, A., Kanoui, H., Pasero, R., and Roussel, P. (1973). Un système de communication homme-machine en Français. Rapport, Groupe d'intelligence Artificielle, Université d'Aix-Marseille II.

Condon, J. H. and Thompson, K. (1982). Belle chess hardware. In Clarke, M. R. B. (Ed.), *Advances in Computer Chess 3.* Pergamon.

Congdon, C. B., Huber, M., Kortenkamp, D., Bidlack, C., Cohen, C., Huffman, S., Koss, F., Raschke, U., and Weymouth, T. (1992). CARMEL versus Flakey: A comparison of two robots. Tech. rep., American Association for Artificial Intelligence.

Conlisk, J. (1989). Three variants on the Allais example. *American Economic Review, 79*, 392–407.

Connell, J. (1989). *A Colony Architecture for an Artificial Creature*. Ph.D. thesis, Artificial Intelligence Laboratory, MIT.

Conway, D. and White, J. (2012). *Machine Learning for Hackers*. O'Reilly.

Cook, S. A. (1971). The complexity of theorem-proving procedures. In *STOC-71*.

Cook, S. A. and Mitchell, D. (1997). Finding hard instances of the satisfiability problem: A survey. In Du, D., Gu, J., and Pardalos, P. (Eds.), *Satisfiability problems: Theory and applications*. American Mathematical Society.

Cooper, G. (1990). The computational complexity of probabilistic inference using Bayesian belief networks. *AIJ*, *42*, 393–405.

Cooper, G. and Herskovits, E. (1992). A Bayesian method for the induction of probabilistic networks from data. *Machine Learning*, *9*, 309–347.

Copeland, J. (1993). *Artificial Intelligence: A Philosophical Introduction*. Blackwell.

Corbett-Davies, S. and Goel, S. (2018). The measure and mismeasure of fairness: A critical review of fair machine learning. arXiv:1808.00023.

Corbett-Davies, S., Pierson, E., Feller, A., Goel, S., and Huq, A. (2017). Algorithmic decision making and the cost of fairness. arXiv:1701.08230.

Cormen, T. H., Leiserson, C. E., Rivest, R., and Stein, C. (2009). *Introduction to Algorithms* (3rd edition). MIT Press.

Cortes, C. and Vapnik, V. N. (1995). Support vector networks. *Machine Learning*, *20*, 273–297.

Cournot, A. (Ed.). (1838). *Recherches sur les principes mathématiques de la théorie des richesses*. L. Hachette, Paris.

Cover, T. and Thomas, J. (2006). *Elements of Information Theory* (2nd edition). Wiley.

Cowan, J. D. and Sharp, D. H. (1988a). Neural nets. *Quarterly Reviews of Biophysics*, *21*, 365–427.

Cowan, J. D. and Sharp, D. H. (1988b). Neural nets and artificial intelligence. *Daedalus*, 117, 85–121.

Cowell, R., Dawid, A. P, Lauritzen, S., and Spiegelhalter, D. J. (2002). *Probabilistic Networks and Expert Systems*. Springer.

Cox, I. (1993). A review of statistical data association techniques for motion correspondence. *IJCV*, *10*, 53–66.

Cox, I. and Hingorani, S. L. (1994). An efficient implementation and evaluation of Reid's multiple hypothesis tracking algorithm for visual tracking. In *ICPR-94*.

Cox, I. and Wilfong, G. T. (Eds.). (1990). *Autonomous Robot Vehicles*. Springer Verlag.

Cox, R. T. (1946). Probability, frequency, and reasonable expectation. *American Journal of Physics*, *14*, 1–13.

Craig, J. (1989). *Introduction to Robotics: Mechanics and Control (2nd edition)*. Addison-Wesley.

Craik, K. (1943). *The Nature of Explanation*. Cambridge University Press.

Cramton, P., Shoham, Y., and Steinberg, R. (Eds.). (2006). *Combinatorial Auctions*. MIT Press.

Craven, M., DiPasquo, D., Freitag, D., McCallum, A., Mitchell, T. M., Nigam, K., and Slattery, S. (2000). Learning to construct knowledge bases from the World Wide Web. *AIJ*, *118*, 69–113.

Crawford, J. M. and Auton, L. D. (1993). Experimental results on the crossover point in satisfiability problems. In *AAAI-93*.

Crick, F. (1999). The impact of molecular biology on neuroscience. *Phil. Trans. Roy. Soc., B*, *354*, 2021–2025.

Crick, F. and Koch, C. (2003). A framework for consciousness. *Nature Neuroscience*, *6*, 119.

Crisan, D. and Doucet, A. (2002). A survey of convergence results on particle filtering methods for practitioners. *IEEE Trans. Signal Processing*, *50*, 736–746.

Cristianini, N. and Hahn, M. (2007). *Introduction to Computational Genomics: A Case Studies Approach.* Cambridge University Press.

Cristianini, N. and Schölkopf, B. (2002). Support vector machines and kernel methods: The new generation of learning machines. *AIMag*, *23*, 31–41.

Cristianini, N. and Shawe-Taylor, J. (2000). *An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods.* Cambridge University Press.

Crockett, L. (1994). *The Turing Test and the Frame Problem: AI's Mistaken Understanding of Intelligence*. Ablex.

Croft, W. B., Metzler, D., and Strohman, T. (2010). *Search Engines: Information Retrieval in Practice*. Addison-Wesley.

Cross, S. E. and Walker, E. (1994). DART: Applying knowledge based planning and scheduling to crisis action planning. In Zweben, M. and Fox, M. S. (Eds.), *Intelligent Scheduling*. Morgan Kaufmann.

Cruse, A. (2011). *Meaning in Language: An Introduction to Semantics and Pragmatics*. Oxford University Press.

Culberson, J. and Schaeffer, J. (1996). Searching with pattern databases. In *Advances in Artificial Intelligence (Lecture Notes in Artificial Intelligence 1081).* Springer–Verlag.

Culberson, J. and Schaeffer, J. (1998). Pattern databases. *Computational Intelligence*, *14*, 318–334.

Cummins, D. and Allen, C. (1998). *The Evolution of Mind*. Oxford University Press.

Cushing, W., Kambhampati, S., Mausam, and Weld, D. S. (2007). When is temporal planning *really* temporal? In *IJCAI-07*.

Cusumano-Towner, M. F., Saad, F., Lew, A. K., and Mansinghka, V. K. (2019). Gen: A general-purpose probabilistic programming system with programmable inference. In *PLDI-19.*

Cybenko, G. (1988). Continuous valued neural networks with two hidden layers are sufficient. Technical report, Department of Computer Science, Tufts University.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Controls, Signals, and Systems*, *2*, 303–314.

Cyert, R. and de Groot, M. (1979). Adaptive utility. In Allais, M. and Hagen, O. (Eds.), *Expected Utility Hypothesis and the Allais Paradox*. D. Reidel.

Dagan, I., Glickman, O., and Magnini, B. (2005). The PASCAL recognising textual entailment challenge. In *Machine Learning Challenges Workshop*.

Daganzo, C. (1979). *Multinomial Probit: The Theory and Its Application to Demand Forecasting*. Academic Press.

Dagum, P. and Luby, M. (1993). Approximating probabilistic inference in Bayesian belief networks is NP-hard. AIJ, *60*, 141–153.

Dagum, P. and Luby, M. (1997). An optimal approximation algorithm for Bayesian inference. *AIJ*, *93*, 1–27.

Dai, A. M. and Le, Q. V. (2016). Semi-supervised sequence learning. In *NeurIPS28*.

Dalal, N. and Triggs, B. (2005). Histograms of oriented gradients for human detection. In *CVPR-05*.

Dalvi, N. N., Ré, C., and Suciu, D. (2009). Probabilistic databases. *CACM*, *52*, 86–94.

Daly, R., Shen, Q., and Aitken, S. (2011). Learning Bayesian networks: Approaches and issues. *Knowledge Engineering Review*, *26*, 99–157.

Damasio, A. R. (1999). *The Feeling of What Happens: Body and Emotion in the Making of Consciousness.* Houghton Mifflin.

Danaher, J. and McArthur, N. (2017). *Robot Sex: Social and Ethical Implications*. MIT Press.

Dantzig, G. B. (1949). Programming of interdependent activities: II. Mathematical model. *Econometrica*, *17*, 200–211.

Darwiche, A. (2001). Recursive conditioning. *AIJ*, *126*, 541.

Darwiche, A. and Ginsberg, M. L. (1992). A symbolic generalization of probability theory. In *AAAI-92*.

Darwiche, A. (2009). *Modeling and reasoning with Bayesian networks*. Cambridge University Press.

Darwin, C. (1859). *On The Origin of Species by Means of Natural Selection*. J. Murray.

Dasgupta, P., Chakrabarti, P. P., and de Sarkar, S. C. (1994). Agent searching in a tree and the optimality of iterative deepening. *AIJ*, *71*, 195–208.

Dasgupta, P. and Maskin, E. (2008). On the robustness of majority rule. *Journal of the European Economic Association*, *6*, 949–973.

Dauphin, Y., Pascanu, R., Gulcehre, C., Cho, K., Gan– guli, S., and Bengio, Y. (2015). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *NeurIPS 27*.

Davidson, D. (1980). *Essays on Actions and Events*. Oxford University Press.

Davidson, D. (1986). A nice derangement of epitaphs. *Philosophical Grounds of Rationality*, *4*, 157–174.

Davis, E. (1986). *Representing and Acquiring Geographic Knowledge*. Pitman and Morgan Kaufmann.

Davis, E. (1990). *Representations of Commonsense Knowledge*. Morgan Kaufmann.

Davis, E. (2005). Knowledge and communication: A first–order theory. AIJ, *166*, 81–140.

Davis, E. (2006). The expressivity of quantifying over regions. *J. Logic and Computation*, *16*, 891–916.

Davis, E. (2007). Physical reasoning. In van Harmelan, F., Lifschitz, V., and Porter, B. (Eds.), *The Handbook of Knowledge Representation*. Elsevier.

Davis, E. (2008). Pouring liquids: A study in commonsense physical reasoning. *AIJ*, *172*.

Davis, E. (2017). Logical formalizations of common-sense reasoning: A survey. *JAIR*, *59*, 651–723.

Davis, E. and Morgenstern, L. (2004). Introduction: Progress in formal commonsense reasoning. *AIJ*, *153*, 1–12.

Davis, E. and Morgenstern, L. (2005). A first-order theory of communication and multi-agent plans. *J. Logic and Computation*, *15*, 701–749.

Davis, M. (1957). A computer program for Presburger's algorithm. In *Proving Theorems* (*as Done by Man, Logician, or Machine*). Proc. Summer Institute for Symbolic Logic. Second edition; publication date is 1960.

Davis, M., Logemann, G., and Loveland, D. (1962). A machine program for theorem-proving. *CACM*, *5*, 394–397.

Davis, M. and Putnam, H. (1960). A computing procedure for quantification theory. *JACM*, *7*, 201–215.

Dayan, P. (1992). The convergence of TD(λ) for general λ. *Machine Learning, 8,* 341–362.

Dayan, P. and Abbott, L. F. (2001). *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems.* MIT Press.

Dayan, P and Hinton, G. E. (1993). Feudal reinforcement learning. In *NeurIPS 5.*

Dayan, P and Niv, Y. (2008). Reinforcement learning and the brain: The good, the bad and the ugly. *Current Opinion in Neurobiology, 18,* 185–196.

de Condorcet, M. (1785). *Essay on the Application of Analysis to the Probability of Majority Decisions.* Imprimerie Royale.

de Dombal, F. T., Leaper, D. J., Horrocks, J. C., and Staniland, J. R. (1974). Human and computer-aided diagnosis of abdominal pain: Further report with emphasis on performance of clinicians. *British Medical Journal*, *1,* 376–380.

de Dombal, F. T., Staniland, J. R., and Clamp, S. E. (1981). Geographical variation in disease presentation. *Medical Decision Making, 1,* 59–69.

de Farias, D. P and Roy, B. V. (2003). The linear programming approach to approximate dynamic programming. *Operations Research, 51*, 839–1016.

de Finetti, B. (1937). Le prévision: ses lois logiques, ses sources subjectives. *Ann. Inst. Poincaré, 7,* 1–68.

de Finetti, B. (1993). On the subjective meaning of probability. In Monari, P. and Cocchi, D. (Eds.), *Probabilita e Induzione.* Clueb.

de Freitas, J. F. G., Niranjan, M., and Gee, A. H. (2000). Sequential Monte Carlo methods to train neural network models. *Neural Computation, 12,*

933–953.

de Ghellinck, G. (1960). Les problèmes de décisions séquentielles. *Cahiers du Centre d'Études de Recherche Opérationnelle, 2*, 161–179.

de Kleer, J. (1975). Qualitative and quantitative knowledge in classical mechanics. Tech. rep., MIT Artificial Intelligence Laboratory.

de Kleer, J. (1989). A comparison of ATMS and CSP techniques. In *IJCAI-89.*

de Kleer, J. and Brown, J. S. (1985). A qualitative physics based on confluences. In Hobbs, J. R. and Moore, R. C. (Eds.), *Formal Theories of the Commonsense World*. Ablex.

de Marcken, C. (1996). *Unsupervised Language Acquisition.* Ph.D.thesis, MIT.

De Morgan, A. (1864). On the syllogism, No. IV, and on the logic of relations. *Transaction of the Cambridge Philosophical Society, X*, 331–358.

de Salvo Braz, R., Amir, E., and Roth, D. (2007). Lifted first-order probabilistic inference. In Getoor, L. and Taskar, B. (Eds.), *Introduction to Statistical Relational Learning*. MIT Press.

Deacon, T. W. (1997). *The Symbolic Species: The Coevolution of Language and the Brain.* W. W. Norton.

Deale, M., Yvanovich, M., Schnitzius, D., Kautz, D., Carpenter, M., Zweben, M., Davis, G., and Daun, B. (1994). The space shuttle ground processing scheduling system. In Zweben, M. and Fox, M. (Eds.), *Intelligent Scheduling*. Morgan Kaufmann.

Dean, J., Patterson, D. A., and Young, C. (2018). A new golden age in computer architecture: Empowering the machine–learning revolution. *IEEE Micro*, *38*, 21–29.

Dean, T., Basye, K., Chekaluk, R., and Hyun, S. (1990). Coping with uncertainty in a control system for navigation and exploration. In *AAAI-90*.

Dean, T. and Boddy, M. (1988). An analysis of time-dependent planning. In *AAAI-88*.

Dean, T., Firby, R. J., and Miller, D. (1990). Hierarchical planning involving deadlines, travel time, and resources. *Computational Intelligence*, *6*, 381–398.

Dean, T., Kaelbling, L. P., Kirman, J., and Nicholson, A. (1993). Planning with deadlines in stochastic domains. In *AAAI-93*.

Dean, T. and Kanazawa, K. (1989a). A model for projection and action. In *IJCAI-89*.

Dean, T. and Kanazawa, K. (1989b). A model for reasoning about persistence and causation. *Computational Intelligence*, *5*, 142–150.

Dean, T. and Wellman, M. P. (1991). *Planning and Control*. Morgan Kaufmann.

Dearden, R., Friedman, N., and Andre, D. (1999). Model-based Bayesian exploration. In *UAI-99.*

Dearden, R., Friedman, N., and Russell, S. J. (1998). Bayesian Q-learning. In *AAAI-98*.

Debevec, P., Taylor, C., and Malik, J. (1996). Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach. In *Proc. 23rd Annual Conference on Computer Graphics (SIGGRAPH).*

Debreu, G. (1960). Topological methods in cardinal utility theory. In Arrow, K. J., Karlin, S., and Suppes, P. (Eds.), *Mathematical Methods in the Social Sciences, 1959*. Stanford University Press.

Dechter, A. and Dechter, R. (1987). Removing redundancies in constraint networks. In *AAAI-87*.

Dechter, R. (1990a). Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. *AIJ*, *41*, 273–312.

Dechter, R. (1990b). On the expressiveness of networks with hidden variables. In *AAAI-90*.

Dechter, R. (1999). Bucket elimination: A unifying framework for reasoning. *AIJ*, *113*, 41–85.

Dechter, R. and Pearl, J. (1985). Generalized best-first search strategies and the optimality of A*. *JACM*, *32*, 505–536.

Dechter, R. and Pearl, J. (1987). Network-based heuristics for constraint-satisfaction problems. *AIJ*, *34*, 1–38.

Dechter, R. and Pearl, J. (1989). Tree clustering for constraint networks. *AIJ*, *38*, 353–366.

Dechter, R. and Rish, I. (2003). Mini-buckets: A general scheme for bounded inference. *JACM*, *50*, 107–153.

Dechter, R. (2003). *Constraint Processing*. Morgan Kaufmann.

Dechter, R. (2019). *Reasoning with Probabilistic and Deterministic Graphical Models: Exact Algorithms* (2nd edition). Morgan & Claypool.

Dechter, R. and Frost, D. (2002). Backjump-based backtracking for constraint satisfaction problems. *AIJ*, *136*, 147–188.

Dechter, R. and Mateescu, R. (2007). AND/OR search spaces for graphical models. *AIJ*, 171, 73–106.

DeCoste, D. and Schölkopf, B. (2002). Training invariant support vector machines. *Machine Learning*, *46*, 161–190.

Dedekind, R. (1888). *Was sind und was sollen die Zahlen.* Braunschweig, Germany.

Deerwester, S. C., Dumais, S. T., Landauer, T. K., Furnas, G. W., and Harshman, R. A. (1990). Indexing by latent semantic analysis. *J. American Society for Information Science*, *41*, 391–407.

DeGroot, M. H. (1970). *Optimal Statistical Decisions.* McGraw-Hill.

DeGroot, M. H. and Schervish, M. J. (2001). *Probability and Statistics* (3rd edition). Addison Wesley.

Dehaene, S. (2014). *Consciousness and the Brain: Deciphering How the Brain Codes Our Thoughts.* Penguin Books.

Del Moral, P., Doucet, A., and Jasra, A. (2006). Sequential Monte Carlo samplers. *J. Royal Statistical Society*, *68*, 411–436.

Del Moral, P. (2004). *Feynman–Kac Formulae, Genealogical and Interacting Particle Systems with Applications.* Springer-Verlag.

Delgrande, J. and Schaub, T. (2003). On the relation between Reiter's default logic and its (major) variants. In *Seventh European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty*.

Delling, D., Sanders, P., Schultes, D., and Wagner, D. (2009). Engineering route planning algorithms. In Lerner, J., Wagner, D., and Zweig, K. (Eds.), *Algorithmics, LNCS.* Springer-Verlag.

Dempster, A. P. (1968). A generalization of Bayesian inference. *J. Royal Statistical Society, 30 (Series B),* 205–247.

Dempster, A. P., Laird, N., and Rubin, D. (1977). Maximum likelihood from incomplete data via the EM algorithm. *J. Royal Statistical Society, 39 (Series B),*1–38.

Denardo, E. V. (1967). Contraction mappings in the theory underlying dynamic programming. *SIAM Review*, *9*, 165–177.

Deng, J., Dong, W., Socher, R., Li, L.–J., Li, K., and Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. In *CVPR-09.*

Deng, L. (2016). Deep learning: From speech recognition to language and multimodal processing. *APSIPA Transactions on Signal and Information Processing, 5.*

Deng, L., Yu, D., *et al.* (2014). Deep learning: Methods and applications. *Foundations and Trends in Signal Processing, 7*, 197–387.

Deng, X. and Papadimitriou, C. H. (1990). Exploring an unknown graph. In *FOCS–90*.

Deng, X. and Papadimitriou, C. H. (1994). On the complexity of cooperative solution concepts. *Mathematics of Operations Research*, *19,*

257–266.

Denney, E., Fischer, B., and Schumann, J. (2006). An empirical evaluation of automated theorem provers in software certification. *Int. J. AI Tools*, *15*, 81–107.

D'Épenoux, F. (1963). A probabilistic production and inventory problem. *A probabilistic production and inventory problem*, *10*, 98–108.

Dervovic, D., Herbster, M., Mountney, P., Severini, S., Usher, N., and Wossnig, L. (2018). Quantum linear systems algorithms: A primer. arXiv:1802.08227.

Descartes, R. (1637). Discourse on method. In Cottingham, J., Stoothoff, R., and Murdoch, D. (Eds.), *The Philosophical Writings of Descartes*, Vol. I. Cambridge University Press, Cambridge.

Descotte, Y. and Latombe, J.–C. (1985). Making compromises among antagonist constraints in a planner. *AIJ*, *27*, 183–217.

Deshpande, I., Hu, Y.-T., Sun, R., Pyrros, A., Siddiqui, N., Koyejo, S., Zhao, Z., Forsyth, D., and Schwing, A. (2019). Max-sliced Wasserstein distance and its use for GANs. In *CVPR-19.*

Deutscher, G. (2010). *Through the Language Glass: Why the World Looks Different in Other Languages.* Metropolitan Books.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidi rectional transformers for language understanding. arXiv:1810.04805.

Devlin, K. (2018). *Turned On: Science, Sex and Robots*. Bloomsbury.

Devroye, L. (1987). *A course in density estimation.* Birkhauser.

Dias, M. B., Zlot, R., Kalra, N., and Stentz, A. (2006). Market-based multirobot coordination: A survey and analysis. *Proc. IEEE*, *94*, 1257–1270.

Dickmanns, E. D. and Zapp, A. (1987). Autonomous high speed road vehicle guidance by computer vision. In *Automatic Control–World Congress, 1987: Selected Papers from the 10th Triennial World Congress of the International Federation of Automatic Control*.

Dietterich, T. (2000). Hierarchical reinforcement learning with the MAXQ value function decomposition. *JAIR*, *13*, 227–303.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, *1*, 269–271.

Dijkstra, E. W. (1984). The threats to computing science. In *ACM South Central Regional Conference*.

Ding, Y., Sohn, J. H., Kawczynski, M. G., Trivedi, H., Harnish, R., Jenkins, N. W., Lituiev, D., Copeland, T. P., Aboian, M. S., Mari Aparici, C., *et al.* (2018). A deep learning model to predict a diagnosis of alzheimer disease by using 18F–FDG PET of the brain. *Radiology*, p. 180958.

Dinh, H., Russell, A., and Su, Y. (2007). On the value of good advice: The complexity of A* with accurate heuristics. In *AAAI-07*.

Dissanayake, G., Newman, P., Clark, S., Durrant-Whyte, H., and Csorba, M. (2001). A solution to the simultaneous lcalisation and map building (SLAM) problem. *IEEE Transactions on Robotics and Automation*, *17*, 229–241.

Dittmer, S. and Jensen, F. (1997). Myopic value of information in influence diagrams. In *UAI-97*.

Do, M. and Kambhampati, S. (2003). Planning as constraint satisfaction: solving the planning graph by compiling it into CSP. *AIJ*, *132*, 151–182.

Do, M. B. and Kambhampati, S. (2001). Sapa: A domain-independent heuristic metric temporal planner. In *ECP-01*.

Doctorow, C. (2001). Metacrap: Putting the torch to seven straw-men of the meta-utopia. www.well.com/doctorow/metacrap.htm.

Doctorow, C. and Stross, C. (2012). *The Rapture of the Nerds: A Tale of the Singularity, Posthumanity, and Awkward Social Situations*. Tor Books.

Dodd, L. (1988). The inside/outside algorithm: Grammatical inference applied to stochastic context-free grammars. Tech. rep., Royal Signals and Radar Establishment, Malvern.

Domingos, P. and Pazzani, M. (1997). On the optimality of the simple Bayesian classifier under zero–one loss. *Machine Learning*, *29*, 103–30.

Domingos, P. (2012). A few useful things to know about machine learning. *Commun. ACM*, 55(10), 78– 87.

Domingos, P. (2015). *The Master Algorithm: How the Quest for the Ultimate Learning Machine Will Remake Our World*. Basic Books.

Dong, X., Gabrilovich, E., Heitz, G., Horn, W., Lao, N., Murphy, K., Strohmann, T., Sun, S., and Zhang, W. (2014). Knowledge vault: A web-scale approach to probabilistic knowledge fusion. In *KDD-14*.

Doorenbos, R. (1994). Combining left and right unlinking for matching a large number of learned rules. In *AAAI-94*.

Doran, J. and Michie, D. (1966). Experiments with the graph traverser program. *Proc. Roy. Soc.*, *294*, *Series A*, 235–259.

Dorf, R. C. and Bishop, R. H. (2004). *Modern Control Systems* (10th edition). Prentice–Hall.

Dorigo, M., Birattari, M., Blum, C., Clerc, M., Stutzle, T., and Winfield, A. (2008). *Ant Colony Optimization and Swarm Intelligence: 6th International Conference, ANTS 2008, Brussels, Belgium, September 2224, 2008, Proceedings*, Vol. 5217. Springer-Verlag.

Doshi-Velez, F. and Kim, B. (2017). Towards a rigorous science of interpretable machine learning. arXiv:1702.08608.

Doucet, A. (1997). *Monte Carlo methods for Bayesian estimation of hidden Markov models: Application to radiation signals.* Ph.D. thesis, Université de ParisSud.

Doucet, A., de Freitas, J. F. G., and Gordon, N. (2001). *Sequential Monte Carlo Methods in Practice.* Springer-Verlag.

Doucet, A., de Freitas, J. F. G., Murphy, K., and Russell, S. J. (2000). Rao-Blackwellised particle filtering for dynamic Bayesian networks. In *UAI-00.*

Doucet, A. and Johansen, A. M. (2011). A tutorial on particle filtering and smoothing: Fifteen years later. In Crisan, D. and Rozovskii, B. (Eds.), *Oxford Handbook of Nonlinear Filtering*. Oxford.

Dowty, D., Wall, R., and Peters, S. (1991). *Introduction to Montague Semantics*. D. Reidel.

Doyle, J. (1979). A truth maintenance system. *AIJ*, *12*, 231–272.

Doyle, J. (1983). What is rational psychology? Toward a modern mental philosophy. *AIMag*, *4*, 50–53.

Drabble, B. (1990). Mission scheduling for spacecraft: Diaries of T-SCHED. In *Expert Planning Systems*. Institute of Electrical Engineers.

Dragan, A. D., Lee, K. C., and Srinivasa, S. (2013). Legibility and predictability of robot motion. In *HRI-13*.

Dredze, M., Crammer, K., and Pereira, F. (2008). Confidence-weighted linear classification. In *ICML-08*.

Dressel, J. and Farid, H. (2018). The accuracy, fairness, and limits of predicting recidivism. *Science Advances*, *4*, eaao5580.

Dreyfus, H. L. (1972). *What Computers Can't Do: A Critique of Artificial Reason*. Harper and Row.

Dreyfus, H. L. (1992). *What Computers Still Can't Do: A Critique of Artificial Reason*. MIT Press.

Dreyfus, H. L. and Dreyfus, S. E. (1986). *Mind over Machine: The Power of Human Intuition and Expertise in the Era of the Computer*. Blackwell.

Dreyfus, S. E. (1962). The numerical solution of variational problems. *J. Math. Anal. and Appl.*, *5*, 30–45.

Dreyfus, S. E. (1969). An appraisal of some shortest– paths algorithms. *Operations Research*, *17*, 395–412.

Dreyfus, S. E. (1990). Artificial neural networks, back propagation, and the Kelley–Bryson gradient procedure. *J. Guidance, Control, and Dynamics*, *13*, 926– 928.

Du, S. S., Lee, J. D., Li, H., Wang, L., and Zhai, X. (2018). Gradient descent finds global minima of deep neural networks. arXiv:1811.03804.

Dubois, D. and Prade, H. (1994). A survey of belief revision and updating rules in various uncertainty models. *Int. J. Intelligent Systems*, *9*, 61–100.

Duda, R. O. and Hart, P. E. (1973). *Pattern classification and scene analysis*. Wiley.

Duda, R. O., Hart, P. E., and Stork, D. G. (2001). *Pattern Classification* (2nd edition). Wiley.

Dudek, G. and Jenkin, M. (2000). *Computational Principles of Mobile Robotics.* Cambridge University Press.

Duffy, D. (1991). *Principles of Automated Theorem Proving.* John Wiley & Sons.

Dunn, H. L. (1946). Record linkage”. *Am. J. Public Health*, *36*, 1412–1416.

Durfee, E. H. and Lesser, V. R. (1989). Negotiating task decomposition and allocation using partial global planning. In Huhns, M. and Gasser, L. (Eds.), *Distributed AI*, Vol. 2. Morgan Kaufmann.

Durme, B. V. and Pasca, M. (2008). Finding cars, goddesses and enzymes: Parametrizable acquisition of labeled instances for open-domain information extraction. In *AAAI-08.*

Dwork, C. (2008). Differential privacy: A survey of results. In *International Conference on Theory and Applications of Models of Computation.*

Dwork, C., Hardt, M., Pitassi, T., Reingold, O., and Zemel, R. (2012). Fairness through awareness. In *Proc. 3rd innovations in theoretical computer science conference*.

Dwork, C., Roth, A., *et al.* (2014). The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer*

Science, *9*, 211–407.

Dyson, F. (2004). A meeting with Enrico Fermi. *Nature*, *427*, 297.

Dyson, G. (1998). *Darwin among the machines : the evolution of global intelligence*. Perseus Books.

Earley, J. (1970). An efficient context-free parsing algorithm. *CACM*, *13*, 94–102.

Ebendt, R. and Drechsler, R. (2009). Weighted A* search–unifying view and application. *AIJ*, *173*, 1310– 1342.

Eckerle, J., Chen, J., Sturtevant, N. R., Zilles, S., and Holte, R. C. (2017). Sufficient conditions for node expansion in bidirectional heuristic search. In *ICAPS-17*.

Eckhouse, L., Lum, K., Conti-Cook, C., and Ciccolini, J. (2019). Layers of bias: A unified approach for understanding problems with risk assessment. *Criminal Justice and Behavior*, *46*, 185–209.

Edelkamp, S. (2009). Scaling search with symbolic pattern databases. In *Model Checking and Artificial Intelligence (MOCHART)*.

Edelkamp, S. and Schrödl, S. (2012). *Heuristic Search*. Morgan Kaufmann.

Edmonds, J. (1965). Paths, trees, and flowers. *Canadian J. of Mathematics*, *17*, 449–467.

Edwards, P. (Ed.). (1967). *The Encyclopedia of Philosophy*. Macmillan.

Eiter, T., Leone, N., Mateis, C., Pfeifer, G., and Scarcello, F. (1998). The KR system dlv: Progress report, comparisons and benchmarks. In *KR-98*.

Elio, R. (Ed.). (2002). *Common Sense, Reasoning, and Rationality*. Oxford University Press.

Elkan, C. (1997). Boosting and naive Bayesian learning. Tech. rep., Department of Computer Science and Engineering, University of California, San Diego.

Ellsberg, D. (1962). *Risk, Ambiguity, and Decision*. Ph.D. thesis, Harvard University.

Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, *14*, 179–211.

Elman, J. L., Bates, E., Johnson, M., Karmiloff–Smith, A., Parisi, D., and Plunkett, K. (1997). *Rethinking Innateness*. MIT Press.

Elo, A. E. (1978). *The rating of chess players: Past and present*. Arco Publishing.

Elsken, T., Metzen, J. H., and Hutter, F. (2018). Neural architecture search: Asurvey. arXiv:1808.05377.

Empson, W. (1953). *Seven Types of Ambiguity*. New Directions.

Enderton, H. B. (1972). *A Mathematical Introduction to Logic*. Academic Press.

Engel, J., Resnick, C., Roberts, A., Dieleman, S., Norouzi, M., Eck, D., and Simonyan, K. (2017). Neural audio synthesis of musical notes with wavenet autoencoders. In *Proc. 34th International Conference on Machine Learning-Volume 70.*

Epstein, R., Roberts, G., and Beber, G. (Eds.). (2008). *Parsing the Turing test*. Springer.

Erdmann, M. A. and Mason, M. (1988). An exploration of sensorless manipulation. *IEEE Journal of Robotics and Automation, 4,* 369–379.

Ernst, H. A. (1961). *MH-1, a Computer-Operated Mechanical Hand*. Ph.D. thesis, MIT.

Ernst, M., Millstein, T., and Weld, D. S. (1997). Automatic SAT-compilation of planning problems. In *IJCAI-97*.

Erol, K., Hendler, J., and Nau, D. S. (1994). HTN planning: Complexity and expressivity. In *AAAI-94*.

Erol, K., Hendler, J., and Nau, D. S. (1996). Complexity results for HTN planning. *AlJ*, *18,* 69–93.

Erol, Y., Li, L., Ramsundar, B., and Russell, S. J. (2013). The extended parameter filter. In *ICML-13*.

Erol, Y., Wu, Y., Li, L., and Russell, S. J. (2017). A nearly-black–box online algorithm for joint parameter and state estimation in temporal models. In *AAAI-17.*

Esteva, A., Kuprel, B., Novoa, R. A., Ko, J., Swetter, S. M., Blau, H. M., and Thrun, S. (2017). Dermatologist-level classification of skin cancer with deep neural networks. *Nature, 542,* 115.

Etzioni, A. (2004). *From Empire to Community: A New Approach to lnternational Relation*. Palgrave Macmillan.

Etzioni, A. and Etzioni, O. (2017a). Incorporating ethics into artificial intelligence. *The Journal of Ethics*, *21*, 403–418.

Etzioni, A. and Etzioni, O. (2017b). Should artificial intelligence be regulated? *Issues in Science and Technology*, *Summer*.

Etzioni, O. (1989). Tractable decision-analytic control. In *Proc. First International Conference on Knowledge Representation and Reasoning*.

Etzioni, O., Banko, M., Soderland, S., and Weld, D. S. (2008). Open information extraction from the web. *CACM*, *51*.

Etzioni, O., Hanks, S., Weld, D. S., Draper, D., Lesh, N., and Williamson, M. (1992). An approach to planning with incomplete information. In *KR-92*.

Etzioni, O., Banko, M., and Cafarella, M. J. (2006). Machine reading. In *AAAI-06*.

Etzioni, O., Cafarella, M. J., Downey, D., Popescu, A.-M., Shaked, T., Soderland, S., Weld, D. S., and Yates, A. (2005). Unsupervised named-entity extraction from the web: An experimental study. *AIJ*, *165*(1), 91–134.

Evans, T. G. (1968). A program for the solution of a class of geometric-analogy intelligence-test questions. In Minsky, M. L. (Ed.), *Semantic Information Processing*. MIT Press.

Fagin, R., Halpern, J. Y., Moses, Y., and Vardi, M. Y. (1995). *Reasoning about Knowledge*. MIT Press.

Fahlman, S. E. (1974). A planning system for robot construction tasks. *AIJ*, 5, 1–49.

Faugeras, O. (1992). Whatcan be seen in three dimensions with an uncalibrated stereo rig? In *ECCV, Vol. 588 of Lecture Notes in Computer Science.*

Faugeras, O., Luong, Q.-T., and Papadopoulo, T. (2001). *The Geometry of Multiple Images.* MIT Press.

Fawcett, T. and Provost, F. (1997). Adaptive fraud detection. *Data mining and knowledge discovery, 1*, 291–316.

Fearing, R. S. and Hollerbach, J. M. (1985). Basic solid mechanics for tactile sensing. *Int. J. Robotics Research, 4*, 40–54.

Featherstone, R. (1987). *Robot Dynamics Algorithms.* Kluwer Academic Publishers.

Feigenbaum, E. A. (1961). The simulation of verbal learning behavior. *Proc. Western Joint Computer Conference, 19*, 121–131.

Feigenbaum, E. A., Buchanan, B. G., and Lederberg, J. (1971). On generality and problem solving: A case study using the DENDRAL program. In Meltzer, B. and Michie, D. (Eds.), *Machine Intelligence 6.* Edinburgh University Press.

Feldman, J. and Sproull, R. F. (1977). Decision theory and artificial intelligence II: The hungry monkey. Technical report, Computer Science Department, University of Rochester.

Feldman, J. and Yakimovsky, Y. (1974). Decision theory and artificial intelligence I: Semantics-based region analyzer. *AIJ, 5*, 349–371.

Feldman, M. (2017). Oak Ridge readies Summit supercomputer for 2018 debut. *Top500.org, bit.ly/2ERRFr9.*

Fellbaum, C. (2001). *Wordnet: An Electronic Lexical Database*. MIT Press.

Fellegi, I. and Sunter, A. (1969). A theory for record linkage”. JASA, *64*, 1183–1210.

Felner, A., Korf, R. E., and Hanan, S. (2004). Additive pattern database heuristics. *JAIR*, 22, 279–318.

Felner, A. (2018). Position paper: Using early goal test in A*. In *Eleventh Annual Symposium on Combinatorial Search.*

Felner, A., Korf, R. E., Meshulam, R., and Holte, R. C. (2007). Compressed pattern databases. *JAIR, 30.*

Felner, A., Zahavi, U., Holte, R. C., Schaeffer, J., Sturtevant, N. R., and Zhang, Z. (2011). Inconsistent heuristics in theory and practice. *AIJ, 175,* 1570–1603.

Felzenszwalb, P. and McAllester, D. A. (2007). The generalized A* architecture. *JAIR.*

Fenton, N. and Neil, M. (2018). *Risk Assessment and Decision Analysis with Bayesian Networks* (2nd edition). Chapman and Hall.

Ferguson, T. (1992). Mate with knight and bishop in kriegspiel. *Theoretical Computer Science*, *96*, 389–403.

Ferguson, T. (1995). Mate with the two bishops in kriegspiel. www.math.ucla.edu/˜tom/papers.

Ferguson, T. (2001). *Optimal Stopping and Applications*. www.math.ucla.edu/ tom/Stopping/Contents.html.

Ferguson, T. (1973). Bayesian analysis of some nonparametric problems. *Annals of Statistics*, *1*, 209–230.

Fern, A., Natarajan, S., Judah, K., and Tadepalli, P. (2014). A decision-theoretic model of assistance. *JAIR, 50,* 71–104.

Fernandez, J. M. F. and Mahlmann, T. (2018). The Dota 2 bot competition. *IEEE Transactions on Games*.

Ferraris, P. and Giunchiglia, E. (2000). Planning as satisability in nondeterministic domains. In *AAAI-00*.

Ferriss, T. (2007). *The 4-Hour Workweek.* Crown.

Ferrucci, D., Brown, E., Chu-Carroll, J., Fan, J., Gondek, D., Kalyanpur, A. A., Lally, A., Murdock, J. W., Nyberg, E., Prager, J., Schlaefer, N., and Welty, C. (2010). Building Watson: An overview of the DeepQA project. *AI Magazine, Fall.*

Faugeras, O. (1992). What can be seen in three dimensions with an uncalibrated stereo rig? In *ECCV*, Vol. 588 of *Lecture Notes in Computer Science*.

Fikes, R. E., Hart, P. E., and Nilsson, N. J. (1972). Learning and executing generalized robot plans. *AIJ*, *3*, 251–288.

Fikes, R. E. and Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *AIJ*, *2*, 189–208.

Fikes, R. E. and Nilsson, N. J. (1993). STRIPS, a retrospective. *AIJ*, *59*, 227–232.

Fine, S., Singer, Y., and Tishby, N. (1998). The hierarchical hidden Markov model: Analysis and applications. *Machine Learning*, *32*.

Finn, C., Abbeel, P., and Levine, S. (2017). Model-agnostic meta-learning for fast adaptation of deep networks. In *Proc. 34th International Conference on Machine Learning-Volume 70.*

Finney, D. J. (1947). *Probit analysis: A statistical treatment of the sigmoid response curve.* Cambridge University Press.

Firoiu, V., Whitney, W. F., and Tenenbaum, J. B. (2017). Beating the world's best at Super Smash Bros. with deep reinforcement learning. arXiv:1702.06230.

Firth, J. (1957). *Papers in Linguistics.* Oxford University Press.

Fisher, R. A. (1922). On the mathematical foundations of theoretical statistics. *Phil. Trans. Roy. Soc., A, 222*, 309–368.

Fix, E. and Hodges, J. L. (1951). Discriminatory analysis—Nonparametric discrimination: Consistency properties. Tech. rep., USAF School of Aviation Medicine.

Floreano, D., Zufferey, J. C., Srinivasan, M. V., and Ellington, C. (2009). *Flying Insects and Robots.* Springer.

Floyd, R. W. (1962). Algorithm 97: Shortest path. *CACM*, *5*, 345.

Fogel, D. B. (2000). *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence.* IEEE Press.

Fogel, L. J., Owens, A. J., and Walsh, M. J. (1966). *Artificial Intelligence through Simulated Evolution.* Wiley.

Forbes, J., Huang, T., Kanazawa, K., and Russell, S. J. (1995). The BATmobile: Towards a Bayesian automated taxi. In *IJCAI-95*.

Forbus, K. D. (1985). Qualitative process theory. In Bobrow, D. (Ed.), *Qualitative Reasoning About Physical Systems*. MIT Press.

Forbus, K. D. and de Kleer, J. (1993). *Building Problem Solvers*. MIT Press.

Forbus, K. D., Hinrichs, T. R., De Kleer, J., and Usher, J. M. (2010). FIRE: Infrastructure for experience-based systems with common sense. In *AAAI Fall Symposium: Commonsense Knowledge.*

Ford, K. M. and Hayes, P. J. (1995). Turing Test considered harmful. In *IJCAI-95*.

Ford, L. R. (1956). Network flow theory. Tech. rep., RAND Corporation.

Ford, M. (2015). *Rise of the Robots: Technology and the Threat of a Jobless Future.* Basic Books.

Ford, M. (2018). *Architects of Intelligence*. Packt.

Forestier, J.-P. and Varaiya, P. (1978). Multilayer control of large Markov chains. *IEEE Transactions on Automatic Control*, *23*, 298–304.

Forgy, C. (1981). OPS5 user's manual. Technical report, Computer Science Department, Carnegie-Mellon University.

Forgy, C. (1982). A fast algorithm for the many patterns/many objects match problem. *AIJ*, *19*, 17–37.

Forster, E. M. (1909). *The Machine Stops*. Sheba Blake.

Forsyth, D. and Ponce, J. (2002). *Computer Vision: A Modern Approach.* Prentice Hall.

Fouhey, D., Kuo, W.-C., Efros, A., and Malik, J. (2018). From lifestyle vlogs to everyday interactions. In *CVPR-18.*

Fourier, J. (1827). Analysedestravaux de l'Académie Royale des Sciences, pendant l'année 1824; partie mathématique. *Histoire de l'Académie Royale des Sciences de France*, *7*, xlvii–lv.

Fox, C. and Tversky, A. (1995). Ambiguity aversion and comparative ignorance. *Quarterly Journal of Economics*, *110*, 585–603.

Fox, D., Burgard, W., Dellaert, F., and Thrun, S. (1999). Monte Carlo localization: Efficient position estimation for mobile robots. In *AAAI-99.*

Fox, M. S. (1990). constraint-guided scheduling: A short history of research at CMU. *Computers in Industry*, *14*, 79–88.

Fox, M. S., Allen, B., and Strohm, G. (1982). Job shop scheduling: An investigation in constraint-directed reasoning. In *AAAI-82*.

Franco, J. and Paull, M. (1983). Probabilistic analysis of the Davis Putnam procedure for solving the satisfiability problem. *Discrete Applied Mathematics*, *5*, 77–87.

Francois-Lavet, V., Henderson, P, Islam, R., Bellemare, M. G., and Pineau, J. (2018). An introduction to deep reinforcement learning. *Foundations and Trends in Machine Learning*, *11*, 219–354.

Frank, I., Basin, D. A., and Matsubara, H. (1998). Finding optimal strategies for imperfect information games. In *AAAI-98*.

Frank, R. H. and Cook, P J. (1996). *The Winner-Take-All Society*. Penguin.

Frans, K., Ho, J., Chen, X., Abbeel, P, and Schulman, J. (2018). Meta learning shared hierarchies. In *ICLR-18.*

Franz, A. and Brants, T. (2006). All our n-gram are belong to you. Google blog, ai.googleblog.com/2006/08/all-our-n-gram-are-belong-to-you.html.

Frege, G. (1879). *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. Halle, Berlin. English translation appears in van Heijenoort (1967).

Freitag, D. and McCallum, A. (2000). Information extraction with hmm structures learned by stochastic optimization. In *AAAI-00*.

Freuder, E. C. (1978). Synthesizing constraint expressions. *CACM*, *21*, 958–966.

Freuder, E. C. (1982). A sufficient condition for backtrack-free search. *JACM*, *29*, 24–32.

Freuder, E. C. (1985). A sufficient condition for backtrack-bounded search. *JACM*, *32*, 755–761.

Freund, Y. and Schapire, R. E. (1996). Experiments with a new boosting algorithm. In *ICML-96*.

Freund, Y. and Schapire, R. E. (1999). Large margin classification using the perceptron algorithm. *Machine Learning*, *37*, 277–296.

Frey, B. J. (1998). *Graphical models for machine learning and digital communication*. MIT Press.

Frey, C. B. and Osborne, M. A. (2017). The future of employment: How susceptible are jobs to computerisation? *Technological forecasting and social change*, *114*, 254–280.

Friedberg, R. M. (1958). A learning machine: Part I. *IBM Journal of Research and Development*, *2*, 2–13.

Friedberg, R. M., Dunham, B., and North, T. (1959). A learning machine: Part II. *IBM Journal of Research and Development*, *3*, 282–287.

Friedman, G. J. (1959). Digital simulation of an evolutionary process. *General Systems Yearbook*, *4*, 171–184.

Friedman, J., Hastie, T., and Tibshirani, R. (2000). Additive logistic regression: A statistical view of boosting. *Annals of Statistics, 28*, 337–374.

Friedman, J. (2001). Greedy function approximation: A gradient boosting machine. *Annals of statistics*, *29,* 1189–1232.

Friedman, N. (1998). The Bayesian structural EM algorithm. In *UAI-98.*

Friedman, N. and Goldszmidt, M. (1996). Learning Bayesian networks with local structure. In *UAI-96.*

Friedman, N. and Koller, D. (2003). Being Bayesian about Bayesian network structure: A Bayesian approach to structure discovery in Bayesian networks. *Machine Learning*, *50*, 95–125.

Friedman, N., Murphy, K., and Russell, S. J. (1998). Learning the structure of dynamic probabilistic networks. In *UAI-98*.

Friedman, N. (2004). Inferring cellular networks using probabilistic graphical models. *Science*, *303*.

Fruhwirth, T. and Abdennadher, S. (2003). *Essentials of constraint programming*. Cambridge University Press.

Fuchs, J. J., Gasquet, A., Olalainty, B., and Currie, K. W. (1990). PlanERS-1: An expert planning system for generating spacecraft mission plans. In *First International Conference on Expert Planning Systems.* Institute of Electrical Engineers.

Fudenberg, D. and Tirole, J. (1991). *Game theory*. MIT Press.

Fukunaga, A. S., Rabideau, G., Chien, S., and Yan, D. (1997). ASPEN: A framework for automated planning and scheduling of spacecraft control and operations. In *Proc. International Symposium on AI, Robotics and Automation in Space*.

Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, *36*, 193–202.

Fukushima, K. and Miyake, S. (1982). Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and cooperation in neural nets*. Springer.

Fuller, S. B., Straw, A. D., Peek, M. Y., Murray, R. M., and Dickinson, M. H. (2014). Flying Drosophila stabilize their vision-based velocity controller by sensing wind with their antennae. *Proc. National Academy of Sciences of the United States of America*, *111 13*, E1182–91.

Fung, C., Yoon, C. J. M., and Beschastnikh, I. (2018). Mitigating sybils in federated learning poisoning. arXiv:1808.04866.

Fung, R. and Chang, K. C. (1989). Weighting and integrating evidence for stochastic simulation in Bayesian networks. In *UAI 5*.

Gaddum, J. H. (1933). Reports on biological standard III: Methods of biological assay depending on a quantal response. Special report series of the medical research council, Medical Research Council.

Gaifman, H. (1964a). Concerning measures in first order calculi. *lsrael J. Mathematics*, *2*, 1–18.

Gaifman, H. (1964b). Concerning measures on Boolean algebras. *Pacific J. Mathematics*, *14*, 61–73.

Gallaire, H. and Minker, J. (Eds.). (1978). *Logic and Databases*. Plenum.

Gallier, J. H. (1986). *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Harper and Row.

Galton, F. (1886). Regression towards mediocrity in hereditary stature. *l. Anthropological lnstitute of Great Britain and lreland*, *15*, 246–263.

Gamba, A., Gamberini, L., Palmieri, G., and Sanna, R. (1961). Further experiments with PAPA. *Nuovo Cimento Supplemento, 20*, 221–231.

Gandomi, A. and Haider, M. (2015). Beyond the hype: Big data concepts, methods, and analytics. *International journal of information management*, *35*, 137– 144.

Gao, J. (2014). Machine learning applications for data center optimization. Google Research.

García, J. and Fernández, F. (2015). A comprehensive survey on safe reinforcement learning. *JMLR, 16,* 1437–1480.

Gardner, M. (1968). *Logic Machines, Diagrams and Boolean Algebra.* Dover.

Garey, M. R. and Johnson, D. S. (1979). *Computers and lntractability.* W. H. Freeman.

Gaschnig, J. (1977). A general backtrack algorithm that eliminates most redundant tests. In *IJCAI-77*.

Gaschnig, J. (1979). Performance measurement and analysis of certain search algorithms. Technical report, Computer Science Department, Carnegie-Mellon University.

Gasser, R. (1995). *Efficiently harnessing computational resources for exhaustive search.* Ph.D. thesis, ETH Zürich.

Gat, E. (1998). Three-layered architectures. In Kortenkamp, D., Bonasso, R. P., and Murphy, R. (Eds.), *AI-based Mobile Robots: Case Studies of Successful Robot Systems*. MIT Press.

Gatys, L. A., Ecker, A. S., and Bethge, M. (2016). Image style transfer using convolutional neural networks. In *CVPR-16.*

Gauci, J., Conti, E., Liang, Y., Virochsiri, K., He, Y., Kaden, Z., Narayanan, V., and Ye, X. (2018). Horizon: Facebook's open source applied reinforcement learning platform. arXiv:1811.00260.

Gauss, C. F. (1809). *Theoria Motus Corporum Coelestium in Sectionibus Conicis Solem Ambientium.* Sumtibus F. Perthes et I. H. Besser, Hamburg.

Gauss, C. F. (1829). Beiträge zur theorie der algebraischen gleichungen. *Werke*, *3*, 71–102.

Gazzaniga, M. (2018). *The Consciousness Instinct.* Farrar, Straus and Girou.

Gebru, T., Morgenstern, J., Vecchione, B., Vaughan, J. W., Wallach, H. M., III, H. D., and Crawford, K. (2018). Datasheets for datasets. arXiv:1803.09010.

Geiger, D., Verma, T., and Pearl, J. (1990a). d-separation: From theorems to algorithms. In Henrion, M., Shachter, R. D., Kanal, L. N., and Lemmer, J. F. (Eds.), *UAI-90.* Elsevier.

Geiger, D., Verma, T., and Pearl, J. (1990b). Identifying independence in Bayesian networks. *Networks*, *20*, 507–534.

Gelb, A. (1974). *Applied Optimal Estimation.* MIT Press.

Gelernter, H. (1959). Realization of a geometry-theorem proving machine. In *Proc. an International Conference on Information Processing*. UNESCO House.

Gelfond, M. and Lifschitz, V. (1988). Compiling circumscriptive theories into logic programs. In *NonMonotonic Reasoning: 2nd International Workshop Proceedings*.

Gelfond, M. (2008). Answer sets. In van Harmelan, F., Lifschitz, V., and Porter, B. (Eds.), *Handbook of Knowledge Representation.* Elsevier.

Gelman, A. (2004). Exploratory data analysis for complex models. *Journal of Computational and Graphical Statistics, 13*, 755–779.

Gelman, A., Carlin, J. B., Stern, H. S., and Rubin, D. (1995). *Bayesian Data Analysis.* Chapman & Hall.

Geman, S. and Geman, D. (1984). Stochastic relaxation, Gibbs distributions, and Bayesian restoration of images. *PAMI, 6,* 721–741.

Gene Ontology Consortium, The. (2008). The gene ontology project in 2008. *Nucleic Acids Research, 36* (D440–D444).

Genesereth, M. R. (1984). The use of design descriptions in automated diagnosis. *AIJ, 24*, 411–436.

Genesereth, M. R. and Nilsson, N. J. (1987). *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann.

Genesereth, M. R. and Nourbakhsh, I. (1993). Timesaving tips for problem solving with incomplete information. In *AAAI-93.*

Genesereth, M. R. and Smith, D. E. (1981). Metalevel architecture. Memo, Computer Science Department, Stanford University.

Gent, I., Petrie, K., and Puget, J.-F. (2006). Symmetry in constraint programming. In Rossi, F., van Beek, P., and Walsh, T. (Eds.), *Handbook of Constraint Programming*. Elsevier.

Géron, A. (2019). *Hands-On Machine Learning with Scikit-Learn, Kerasm and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly.

Gers, F. A., Schmidhuber, J., and Cummins, F. (2000). Learning to forget: Continual prediction with LSTM. *Neural Computation*, *12*, 2451–2471.

Getoor, L. and Taskar, B. (Eds.). (2007). *Introduction to Statistical Relational Learning*. MIT Press.

Ghaheri, A., Shoar, S., Naderan, M., and Hoseini, S. S. (2015). The applications of genetic algorithms in medicine. *Oman medical journal*, *30*, 406–416.

Ghahramani, Z. (1998). Learning dynamic Bayesian networks. In *Adaptive Processing of Sequences and Data Structures*.

Ghahramani, Z. (2005). Tutorial on nonparametric Bayesian methods. Given at the UAI-05 Conference.

Ghallab, M., Howe, A., Knoblock, C. A., and McDermott, D. (1998). PDDL—The planning domain definition language. Tech. rep., Yale Center for Computational Vision and Control.

Ghallab, M. and Laruelle, H. (1994). Representation and control in IxTeT, a temporal planner. In *AIPS-94*.

Ghallab, M., Nau, D. S., and Traverso, P. (2004). *Automated Planning: Theory and practice*. Morgan Kaufmann.

Ghallab, M., Nau, D. S., and Traverso, P. (2016). *Automated Planning and aAting*. Cambridge University Press.

Gibbs, R. W. (2006). Metaphor interpretation as embodied simulation. *Mind, 21*, 434–458.

Gibson, J. J. (1950). *The Perception of the Visual World.* Houghton Mifflin.

Gibson, J. J. (1979). *The Ecological Approach to Visual Perception*. Houghton Mifflin.

Gibson, J. J., Olum, P., and Rosenblatt, F. (1955). Parallax and perspective during aircraft landings. *American Journal of Psychology, 68*, 372–385.

Gilks, W. R., Richardson, S., and Spiegelhalter, D. J. (Eds.). (1996). *Markov chain Monte Carlo in practice*. Chapman and Hall.

Gilks, W. R., Thomas, A., and Spiegelhalter, D. J. (1994). A language and program for complex Bayesian modelling. *The Statistician, 43,* 169–178.

Gilks, W. R. and Berzuini, C. (2001). Following a moving target—Monte Carlo inference for dynamic Bayesian models. *J. Royal Statistical Society, 63, 127–146.*

Gilks, W. R. and Wild, P. P. (1992). Adaptive rejection sampling for Gibbs sampling. *Applied Statistics, 41,* 337–348.

Gillies, D. B. (1959). Solutions to general non-zero-sum games. In Tucker, A. W. and Luce, L. D. (Eds.), *Contributions to the Theory of Games, volume IV.* Princeton University Press.

Gilmore, P C. (1960). A proof method for quantification theory: Its justification and realization. *IBM Journal of Research and Development, 4,* 28–35.

Gilpin, A., Sandholm, T., and Sorensen, T. (2008). A heads-up no-limit Texas Hold'em poker player: Discretized betting models and automatically generated equilibrium-finding programs. In *AAMAS-08.*

Ginsberg, M. L. (1993). *Essentials of Artificial Intelligence.* Morgan Kaufmann.

Ginsberg, M. L. (2001). GIB: Imperfect infoormation in a computationally challenging game. *JAIR, 14,* 303–358.

Gionis, A., Indyk, P., and Motwani, R. (1999). Similarity search in high dimensions vis hashing. In *Proc. 25th Very Large Database (VLDB) Conference.*

Girshick, R., Donahue, J., Darrell, T., and Malik, J. (2016). Region-based convolutional networks for accurate object detection and segmentation. *PAMI*, 38, 142–58.

Gittins, J. C. (1989). *Multi-Armed Bandit Allocation Indices*. Wiley.

Gittins, J. C. and Jones, D. M. (1974). A dynamic allocation index for the sequential design of experiments. In Gani, J. (Ed.), *Progress in Statistics.* North-Holland.

Glanc, A. (1978). On the etymology of the word "robot". *SIGART Newsletter, 67*, 12.

Glickman, M. E. (1999). Parameter estimation in large dynamic paired comparison experiments. *Applied Statistics*, *48*, 377–394.

Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In *AISTATS'2011*.

Glover, F. and Laguna, M. (Eds.). (1997). *Tabu search.* Kluwer.

Gluss, B. (1959). An optimum policy for detecting a fault in a complex system. *Operations Research*, *7*, 468–477.

Godefroid, P. (1990). Using partial orders to improve automatic verification methods. In *Proc. 2nd Int'l Workshop on Computer Aided Verification.*

Gödel, K. (1930). *Über die Vollständigkeit des Logikkalküls.* Ph.D. thesis, University of Vienna.

Gödel, K. (1931). Über formal unentscheidbare Sätze der Principia mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, *38*, 173–198.

Goebel, J., Volk, K., Walker, H., and Gerbault, F. (1989). Automatic classification of spectra from the infrared astronomical satellite (IRAS). *Astronomy and Astrophysics*, *222*, L5–L8.

Goertzel, B. and Pennachin, C. (2007). *Artificial General Intelligence*. Springer.

Gogate, V. and Domingos, P. (2011). Approximation by quantization. In *UAI-11*.

Gold, E. M. (1967). Language identification in the limit. *Information and Control*, *10*, 447–474.

Goldberg, A. V., Kaplan, H., and Werneck, R. F. (2006). Reach for A*: Efficient point-to-point shortest path algorithms. In *Workshop on algorithm engineering and experiments.*

Goldberg, Y. (2017). Neural network methods for natural language processing. *Synthesis Lectures on Human Language Technologies*, *10*.

Goldberg, Y., Zhao, K., and Huang, L. (2013). Efficient implementation of beam-search incremental parsers. In *ACL-13*.

Goldman, R. and Boddy, M. (1996). Expressive planning and explicit knowledge. In *AIPS-96*.

Goldszmidt, M. and Pearl, J. (1996). Qualitative probabilities for default reasoning, belief revision, and causal modeling. *AIJ, 84*, 57–112.

Golomb, S. and Baumert, L. (1965). Backtrack proramming. *JACM, 14*, 516–524.

Golub, G., Heath, M., and Wahba, G. (1979). Generalized cross-validation as a method for choosing a good ridge parameter. *Technometrics, 21*.

Gomes, C., Selman, B., Crato, N., and Kautz, H. (2000). Heavy–tailed phenomena in satisfiability and constrain processing. *JAR, 24*, 67–100.

Gomes, C., Kautz, H., Sabharwal, A., and Selman, B. (2008). Satisfiability solvers. In van Harmelen, F., Lifschitz, V., and Porter, B. (Eds.), *Handbook of Knowledge Representation*. Elsevier.

Gomes, C. and Selman, B. (2001). Algorithm portfolios. AIJ, *126*, 43–62.

Gomes, C., Selman, B., and Kautz, H. (1998). Boosting combinatorial search through randomization. In *AAAI-98*.

Gonthier, G. (2008). Formal proof–The four-color theorem. *Notices of the AMS, 55*, 1382–1393.

Good, I. J. (1961). Acausal calculus. *British Journal of the Philosophy of Science, 11*, 305–318.

Good, I. J. (1965a). The mystery of Go. *New Scientist*, 427, 172–174.

Good, I. J. (1965b). Speculations concerning the first ultraintelligent machine. In Alt, F. L. and Rubinoff, M. (Eds.), *Advances in Computers*, Vol. 6. Academic Press.

Good, I. J. (1983). *Good Thinking: The Foundations of Probability and Its Applications.* University of Minnesota Press.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning.* MIT Press.

Goodfellow, I., Bulatov, Y., Ibarz, J., Arnoud, S., and Shet, V. (2014). Multi-digit number recognition from Street View imagery using deep convolutional neural networks. In *International Conference on Learning Representations*.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2015a). Generative adversarial nets. In *NeurIPS 27*.

Goodfellow, I., Vinyals, O., and Saxe, A. M. (2015b). Qualitatively characterizing neural network optimization problems. In *International Conference on Learning Representations*.

Goodman, J. (2001). A bit of progress in language modeling. Tech. rep., Microsoft Research.

Goodman, N. D., Mansinghka, V. K., Roy, D., Bonawitz, K., and Tenenbaum, J. B. (2008). Church: A language for generative models. In *UAI-08.*

Goodman, N. (1977). *The Structure of Appearance* (3rd edition). D. Reidel.

Gopnik, A. and Glymour, C. (2002). Causal maps and Bayes nets: A cognitive and computational account of theory-formation. In Caruthers, P., Stich, S., and Siegal, M. (Eds.), *The Cognitive Basis of Science*. Cambridge University Press.

Gordon, A. D., Graepel, T., Rolland, N., Russo, C., Borgström, J., and Guiver, J. (2014). Tabular: A schema–driven probabilistic programming language. In *POPL-14*.

Gordon, A. S. and Hobbs, J. R. (2017). *A Formal Theory of Commonsense Psychology: How People Think People Think*. Cambridge University Press.

Gordon, M. J., Milner, A. J., and Wadsworth, C. P. (1979). *Edinburgh LCF*. Springer-Verlag.

Gordon, N. (1994). *Bayesian methods for tracking*. Ph.D. thesis, Imperial College.

Gordon, N., Salmond, D. J., and Smith, A. F. M. (1993). Novel approach to nonlinear/non-Gaussian Bayesian state estimation. *IEE Proceedings F (Radar and Signal Processing), 140*, 107–113.

Gordon, S. A. (1994). A faster Scrabble move generation algorithm. *Software Practice and Experience, 24*, 219–232.

Gorry, G. A. (1968). Strategies for computer-aided diagnosis. *Math. Biosciences, 2*, 293–318.

Gorry, G. A., Kassirer, J. P., Essig, A., and Schwartz, W. B. (1973). Decision analysis as the basis for computer-aided management of acute renal failure. *American Journal of Medicine, 55*, 473–484.

Gottlob, G., Leone, N., and Scarcello, F. (1999a). A comparison of structural CSP decomposition methods. In *IJCAI-99*.

Gottlob, G., Leone, N., and Scarcello, F. (1999b). Hypertree decompositions and tractable queries. In *PODS-99*.

Goyal, Y., Khot, T., Summers–Stay, D., Batra, D., and Parikh, D. (2017). Making the V in VQA matter: Elevating the role of image understanding in visual question answering. In *CVPR-17*.

Grace, K., Salvatier, J., Dafoe, A., Zhang, B., and Evans, O. (2017). When will AI exceed human performance? Evidence from AI experts. arXiv:1705.08807.

Graham, S. L., Harrison, M. A., and Ruzzo, W. L. (1980). An improved context-free recognizer. *ACM Transactions on Programming Languages and Systems*, *2*, 415–462.

Grassmann, H. (1861). *Lehrbuch der Arithmetik*. Th. Chr. Fr. Enslin, Berlin.

Grayson, C. J. (1960). Decisions under uncertainty: Drilling decisions by oil and gas operators. Tech. rep., Harvard Business School.

Green, B., Wolf, A., Chomsky, C., and Laugherty, K. (1961). BASEBALL: An automatic question answerer. In *Proc. Western Joint Computer Conference*.

Green, C. (1969a). Application of theorem proving to problem solving. In *IJCAI-69*.

Green, C. (1969b). Theorem-proving by resolution as a basis for question-answering systems. In Meltzer, B., Michie, D., and Swann, M. (Eds.), *Machine Intelligence 4*. Edinburgh University Press.

Green, C. and Raphael, B. (1968). The use of theorem-proving techniques in question-answering systems. In *Proc. 23rd ACM National Conference.*

Gribkoff, E., Van den Broeck, G., and Suciu, D. (2014). Understanding the complexity of lifted inference and asymmetric weighted model counting. In *UAI-14*.

Griffiths, T. L., Kemp, C., and Tenenbaum, J. B. (2008). Bayesian models of cognition. In Sun, R. (Ed.), *The Cambridge handbook of computational cognitive modeling*. Cambridge University Press.

Grinstead, C. and Snell, J. (1997). *Introduction to Probability*. American Mathematical Society.

Grosz, B. J. and Stone, P. (2018). A century long commitment to assessing artificial intelligence and its impact on society. *Communications of the ACM, 61*.

Grove, W. and Meehl, P. (1996). Comparative efficiency of informal (subjective, impressionistic) and formal (mechanical, algorithmic) prediction procedures: The clinical statistical controversy. *Psychology, Public Policy, and Law, 2*, 293–323.

Gruber, T. (2004). Interview of Tom Gruber. *AISSIGSEMIS Bulletin*, *1*.

Gu, J. (1989). *Parallel Algorithms and Architectures for Very Fast AI Search.* Ph.D. thesis, Univ. of Utah.

Guard, J., Oglesby, F., Bennett, J., and Settle, L. (1969). Semi-automated mathematics. *JACM, 16,* 4962.

Guestrin, C., Koller, D., Gearhart, C., and Kanodia, N. (2003a). Generalizing plans to new environments inrelational MDPs. In *IJCAI-03*.

Guestrin, C., Koller, D., Parr, R., and Venkataraman, S. (2003b). Efficient solution algorithms for factored MDPs. *JAIR*, *19*, 399–468.

Guestrin, C., Lagoudakis, M. G., and Parr, R. (2002). Coordinated reinforcement learning. In *ICML-02*.

Guibas, L. J., Knuth, D. E., and Sharir, M. (1992). Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, *7*, 381–413.

Gulshan, V., Peng, L., Coram, M., Stumpe, M. C., Wu, D., Narayanaswamy, A., Venugopalan, S., Widner, K., Madams, T., Cuadros, J., *et al.* (2016). Development and validation of a deep learning algorithm for detection of diabetic retinopathy in retinal fundus photographs. *Jama*, *316*, 2402–2410.

Gunkel, D. J. (2018). *Robot Rights*. MIT Press.

Gunning, D. (2016). Explainable artificial intelligence (xai). Tech. rep., DARPA.

Guo, C., Goldstein, T., Hannun, A., and van der Maaten, L. (2019). Certified data removal from machine learning models. arXiv:1911.03030.

Gururangan, S., Swayamdipta, S., Levy, O., Schwartz, R., Bowman, S., and Smith, N. A. (2018). Annotation artifacts in natural language inference data. arXiv:1803.02324.

Guyon, I., Bennett, K., Cawley, G. C., Escalante, H. J., Escalera, S., Ho, T. K., Macià, N., Ray, B., Saeed, M., Statnikov, A. R., and Viegas, E. (2015).

Design of the 2015 ChaLearn AutoML challenge. In *IJCNN-15*.

Guyon, I. and Elisseeff, A. (2003). An introduction to variable and feature selection. *JMLR*, *3*, 1157–1182.

Hacking, I. (1975). *The Emergence of Probability*. Cambridge University Press.

Hadfield-Menell, D., Dragan, A. D., Abbeel, P., and Russell, S. J. (2017a). Cooperative inverse reinforcement learning. In *NeurIPS 29*.

Hadfield-Menell, D., Dragan, A. D., Abbeel, P., and Russell, S. J. (2017b). The off-switch game. In *IJCAI-17*.

Hadfield-Menell, D. and Russell, S. J. (2015). Multitasking: Efficient optimal planning for bandit superprocesses. In *UAI-15*.

Hailperin, T. (1984). Probability logic. *Notre Dame J. Formal Logic*, *25*, 198–212.

Hald, A. (1990). *A History of Probability and Statistics and Their Applications before 1750*. Wiley.

Hales, T. (2005). A proof of the Kepler conjecture. *Annals of mathematics*, *162*, 1065–1185.

Hales, T., Adams, M., Bauer, G., Dang, T. D., Harrison, J., Le Truong, H., Kaliszyk, C., Magron, V., McLaughlin, S., Nguyen, T. T., *etal.* (2017). A formal proof of the Kepler conjecture. In *Forum of Mathematics, Pi*.

Halevy, A. (2007). Dataspaces: A new paradigm for data integration. In *Brazilian Symposium on Databases*.

Halevy, A., Norvig, P., and Pereira, F. (2009). The unreasonable effectiveness of data. *IEEE Intelligent Systems, March/April,* 8–12.

Halpern, J. Y. (1990). An analysis of first-order logics of probability. *AIJ, 46*, 311–350.

Halpern, J. Y. (1999). Technical addendum, Cox's theorem revisited. *JAIR, 11*, 429–435.

Halpern, J. Y. and Weissman, V. (2008). Using firstorder logic to reason about policies. *ACM Transactions on Information and System Security, 11,* 1–41.

Hammersley, J. M. and Handscomb, D. C. (1964). *Monte Carlo Methods.* Methuen.

Han, J., Pei, J., and Kamber, M. (2011). *DataMining: Concepts and Techniques.* Elsevier.

Han, X. and Boyden, E. (2007). Multiple-color optical activation, silencing, and desynchronization of neural activity, with single-spike temporal resolution. *PLoS One, e299.*

Handschin, J. E. and Mayne, D. Q. (1969). Monte Carlo techniques to estimate the conditional expectation in multi-stage nonlinear filtering. *Int. J. Control, 9,* 547–559.

Hans, A., Schneegaβ, D., Schäfer, A. M., and Udluft, S. (2008). Safe exploration for reinforcement learning. In *ESANN.*

Hansen, E. (1998). Solving POMDPs by searching in policy space. In *UAI-98.*

Hansen, E. and Zilberstein, S. (2001). LAO*: a heuristic search algorithm that finds solutions with loops. *AIJ, 129,* 35–62.

Hansen, P. and Jaumard, B. (1990). Algorithms for the maximum satisfiability problem. *Computing, 44,* 279–303.

Hanski, I. and Cambefort, Y. (Eds.). (1991). *Dung Beetle Ecology.* Princeton University Press.

Hansson, O. and Mayer, A. (1989). Heuristic search as evidential reasoning. In *UAI 5*.

Haralick, R. M. and Elliott, G. L. (1980). Increasing tree search efficiency for constraint satisfaction problems. *AIJ, 14*, 263–313.

Hardin, G. (1968). The tragedy of the commons. *Science, 162*, 1243–1248.

Hardt, M., Price, E., Srebro, N., *et al.* (2017). Equality of opportunity in supervised learning. In *NeurIPS 29*.

Harris, T. (2016). How technology is hijacking your mind–From a magician and Google design ethicist. medium.com/thrive-global/how-technology-hijacks-peoples-minds-from-a-magician-and-googles-design-ethicist-56d62ef5edf3.

Harris, Z. (1954). Distributional structure. *Word, 10.*

Harrison, J. and March, J. G. (1984). Decision making and postdecision surprises. *Administrative Science Quarterly, 29*, 26–42.

Harrow, A. W., Hassidim, A., and Lloyd, S. (2009). Quantum algorithm for linear systems of equations. *Physical Review Letters, 103 15*, 150502.

Harsanyi, J. (1967). Games with incomplete information played by Bayesian players. *Management Science*, *14*, 159–182.

Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, *SSC-4(2)*, 100–107.

Hart, T. P. and Edwards, D. J. (1961). The tree prune (TP) algorithm. Artificial intelligence project memo, MIT.

Hartley, H. (1958). Maximum likelihood estimation from incomplete data. *Biometrics*, *14*, 174–194.

Hartley, R. and Zisserman, A. (2000). *Multiple view geometry in computer vision*. Cambridge University Press.

Hashimoto, K., Xiong, C., Tsuruoka, Y., and Socher, R. (2016). A joint many-task model: Growing a neural network for multiple NLP tasks. arXiv:1611.01587.

Haslum, P., Botea, A., Helmert, M., Bonet, B., and Koenig, S. (2007). Domain-independent construction of pattern database heuristics for cost-optimal planning. In *AAAI-07*.

Haslum, P. and Geffner, H. (2001). Heuristic planning with time and resources. In *Proc. IJCAI-01 Workshop on Planning with Resources*.

Haslum, P. (2006). Improving heuristics through relaxed search – An analysis of TP4 and HSP*a in the 2004 planning competition. *JAIR*, *25*, 233–267.

Hastie, T. and Tibshirani, R. (1996). Discriminant adaptive nearest neighbor classification and regression. In *NeurIPS 8*.

Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference and Prediction* (2nd edition). Springer-Verlag.

Hastings, W. K. (1970). Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, *57*, 97–109.

Hatem, M. and Ruml, W. (2014). Simpler bounded suboptimal search. In *AAAI-14*.

Haugeland, J. (1985). *Artificial Intelligence: The Very Idea*. MIT Press.

Havelund, K., Lowry, M., Park, S., Pecheur, C., Penix, J., Visser, W., and White, J. L. (2000). Formal analysis of the remote agent before and after flight. In *Proc. 5th NASA Langley Formal Methods Workshop*.

Havenstein, H. (2005). Spring comes to AI winter. *Computer World*, Fe. 14.

Hawkins, J. (1961). Self–organizing systems: A review and commentary. *Proc. IRE*, *49*, 31–48.

Hay, N., Russell, S. J., Shimony, S. E., and Tolpin, D. (2012). Selecting computations: Theory and applications. In *UAI-12*.

Hayes, P. J. (1978). The naive physics manifesto. In Michie, D. (Ed.), *Expert Systems in the Microelectronic Age*. Edinburgh University Press.

Hayes, P. J. (1979). The logic of frames. In Metzing, D. (Ed.), *Frame Conceptions and Text Understanding*. de Gruyter.

Hayes, P. J. (1985a). Naive physics I: Ontology for liquids. In Hobbs, J. R. and Moore, R. C. (Eds.), *Formal Theories of the Commonsense World*, chap. 3. Ablex.

Hayes, P. J. (1985b). The second naive physics manifesto. In Hobbs, J. R. and Moore, R. C. (Eds.), *Formal Theories of the Commonsense World*, chap. 1. Ablex.

Hays, J. and Efros, A. (2007). Scene completion Using millions of photographs. *ACM Transactions on Graphics (SIGGRAPH), 26.*

He, H., Bai, Y., Garcia, E. A., and Li, S. (2008). ADASYN: Adaptive synthetic sampling approach for imbalanced learning. In *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *CVPR-16*.

Heawood, P. J. (1890). Map colouring theorem. *Quarterly Journal of Mathematics*, *24*, 332–338.

Hebb, D. O. (1949). *The Organization of Behavior.* Wiley.

Heckerman, D. (1986). Probabilistic interpretation for MYCIN’s certainty factors. In Kanal, L. N. and Lemmer, J. F. (Eds.), *UAI2*. Elsevier.

Heckerman, D. (1991). *Probabilistic Similarity Networks*. MIT Press.

Heckerman, D. (1998). A tutorial on learning with Bayesian networks. In Jordan, M. I. (Ed.), *Learning in graphical models*. Kluwer.

Heckerman, D., Geiger, D., and Chickering, D. M. (1994). Learning Bayesian networks: The combination of knowledge and statistical data. Technical report, Microsoft Research.

Heidegger, M. (1927). *Being and Time.* SCM Press.

Heinlein, R. A. (1973). *Time Enough for Love*. Putnam.

Held, M. and Karp, R. M. (1970). The traveling salesman problem and minimum spanning trees. *Operations Research*, *18*, 1138–1162.

Helmert, M. (2001). On the complexity of planning in transportation domains. In *ECP-01*.

Helmert, M. (2006). The fast downward planning system. *JAIR*, *26*, 191–246.

Helmert, M. and Röger, G. (2008). How good is almost perfect? In *AAAI-08*.

Helmert, M., Röger, G., and Karpas, E. (2011). Fast downward stone soup: A baseline for building planner portfolios. In *ICAPS*.

Hendeby, G., Karlsson, R., and Gustafsson, F. (2010). Particle filtering: The need for speed. *EURASIP J. Adv. Sig. Proc., June*.

Henrion, M. (1988). Propagation of uncertainty in Bayesian networks by probabilistic logic sampling. In Lemmer, J. F. and Kanal, L. N. (Eds.), *UAI2*. Elsevier.

Henzinger, T. A. and Sastry, S. (Eds.). (1998). *Hybrid Systems: Computation and Control*. Springer-Verlag.

Herbrand, J. (1930). *Recherches sur la Théorie de la Démonstration*. Ph.D. thesis, University of Paris.

Herbrich, R., Minka, T., and Graepel, T. (2007). TrueSkill: A Bayesian skill rating system. In *NeurIPS 19*.

Hernández–Orallo, J. (2016). Evaluation in artificial intelligence: From task-oriented to ability–oriented measurement. *Artificial Intelligence Review, 48*, 397–447.

Hess, C. and Ostrom, E. (2007). *Understanding Knowledge as a Commons.* MIT Press.

Hewitt, C. (1977). Viewing control structures as patterns of passing messages. *AIJ*, *8*, 323–364.

Hewitt, C. (1969). PLANNER: a language for proving theorems in robots. In *IJCAI-69*.

Hezaveh, Y. D., Levasseur, L. P., and Marshall, P J.(2017). Fast automated analysis of strong gravitational lenses with convolutional neural networks. *Nature*, *548*, 555–557.

Hierholzer, C. (1873). Über die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. *Mathematische Annalen*, *6*, 30–32.

Hilbert, M. and Lopez, P. (2011). The world's technological capacity to store, communicate, and compute information. *Science*, *332*, 60–65.

Hilgard, E. R. and Bower, G. H. (1975). *Theories of Learning* (4th edition). Prentice-Hall.

Hind, M., Mehta, S., Mojsilovic, A., Nair, R., Ramamurthy, K. N., Olteanu, A., and Varshney, K. R.(2018). Increasing trust in AI services through supplier's declarations of conformity. arXiv:1808.07261.

Hintikka, J. (1962). *Knowledge and Belief.* Cornell University Press.

Hinton, G. E. and Anderson, J. A. (1981). *Parallel Models of Associative Memory*. Lawrence Erlbaum.

Hinton, G. E. and Nowlan, S. J. (1987). How learning can guide evolution. *Complex Systems*, *1*, 495–502.

Hinton, G. E. and Sejnowski, T. (1983). Optimal perceptual inference. In *CVPR-83*.

Hinton, G. E. and Sejnowski, T. (1986). Learning and relearning in Boltzmann machines. In Rumelhart, D. E. and McClelland, J. L. (Eds.), *Parallel Distributed Processing*. MIT Press.

Hinton, G. E. (1987). Learning translation invariant recognition in a massively parallel network. In Goos, G. and Hartmanis, J. (Eds.), *PARLE: Parallel Architectures and Languages Europe*. Springer-Verlag.

Heess, N., Wayne, G., Silver, D., Lillicrap, T., Erez, T., and Tassa, Y. (2016). Learning continuous control policies by stochastic value gradients. In *NeurIPS 28*.

Hinton, G. E., Deng, L., Yu, D., Dahl, G., Mohamed, A. R., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T., and Kingsbury, B. (2012). Deep neural networks for acoustic modeling in speech recognition. *Signal Processing Magazine*, *29*, 82 – 97.

Hinton, G. E., Osindero, S., and Teh, Y. W. (2006). A fast learning algorithm for deep belief nets. *Neural Computation*, *18*, 1527–1554.

Hirth, M., Hofifeld, T., and Tran–Gia, P. (2013). Analyzing costs and accuracy of validation mechanisms for crowdsourcing platforms. *Mathematical and Computer Modelling*, *57*, 2918–2932.

Ho, M. K., Littman, M. L., MacGlashan, J., Cushman, F., and Austerweil, J. L. (2017). Showing versus doing: Teaching by demonstration. In *NeurIPS 29.*

Ho, T. K. (1995). Random decision forests. In *Proc. 3rd Int’l Conf. on Document Analysis and Recognition.*

Hobbs, J. R. (1990). *Literature and Cognition.* CSLI Press.

Hobbs, J. R. and Moore, R. C. (Eds.). (1985). *Formal Theories of the Commonsense World.* Ablex.

Hochreiter, S. (1991). Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Technische Universität München.

Hochreiter, S. and Schmidhuber, J. (1997). Long short–term memory. *Neural Computation, 9,* 1735–1780.

Hoffman, M., Bach, F. R., and Blei, D. M. (2011). Online learning for latent Dirichlet allocation. In *NeurIPS 23*.

Hoffmann, J. (2001). FF: The fast-forward planning system. *AIMag, 22,* 57–62.

Hoffmann, J. and Brafman, R. I. (2006). Conformant planning via heuristic forward search: A new approach. *AIJ, 170*, 507–541.

Hoffmann, J. and Brafman, R. I. (2005). Contingent planning via heuristic forward search with implicit belief states. In *ICAPS-05.*

Hoffmann, J. (2005). Where “ignoring delete lists” works: Local search topology in planning benchmarks. *JAIR, 24*, 685–758.

Hoffmann, J. and Nebel, B. (2001). The FF planning system: Fast plan generation through heuristic search. *JAIR*, *14*, 253–302.

Hoffmann, J., Sabharwal, A., and Domshlak, C. (2006). Friends or foes? An AI planning perspective on abstraction and search. In *ICAPS-06.*

Hofleitner, A., Herring, R., Abbeel, P., and Bayen, A. M. (2012). Learning the dynamics of arterial traffic from probe data using a dynamic Bayesian network. *IEEE Transactions on Intelligent Transportation Systems*, *13*, 1679–1693.

Hogan, N. (1985). Impedance control: An approach to manipulation. Parts I, II, and III. *J. Dynamic Systems, Measurement, and Control, 107,* 1–24.

Hoiem, D., Efros, A., and Hebert, M. (2007). Recovering surface layout from an image. *IJCV, 75,* 151–172.

Holland, J. H. (1975). *Adaption in Natural and Artificial Systems*. University of Michigan Press.

Holland, J. H. (1995). *Hidden Order: How Adaptation Builds Complexity.* Addison-Wesley.

Holte, R. C., Felner, A., Sharon, G., and Sturtevant, N. R. (2016). Bidirectional search that is guaranteed to meet in the middle. In *AAAI-16*.

Holzmann, G. J. (1997). The Spin model checker. *IEEE Transactions on Software Engineering*, *23*, 279–295.

Hood, A. (1824). Case 4th–28 July 1824 (Mr. Hood’s cases of injuries of the brain). *Phrenological Journal and Miscellany*, *2*, 82–94.

Hooker, J. (1995). Testing heuristics: We have it all wrong. *J. Heuristics*, *1*, 33–42.

Hoos, H. H. and Stützle, T. (2004). *Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann.

Hoos, H. H. and Tsang, E. (2006). Local search methods. In Rossi, F., van Beek, P., and Walsh, T. (Eds.), *Handbook of Constraint Processing*. Elsevier.

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *PNAS*, *79*, 2554–2558.

Horn, A. (1951). On sentences which are true of direct unions of algebras. *JSL*, *16*, 14–21.

Horn, B. K. P. (1970). Shape from shading: A method for obtaining the shape of a smooth opaque object from one view. Technical report, MIT Artificial Intelligence Laboratory.

Horn, B. K. P. and Brooks, M. J. (1989). *Shape from Shading*. MIT Press.

Horn, K. V. (2003). Constructing a logic of plausible inference: A guide to Cox's theorem. *IJAR*, *34*, 3–24.

Horning, J. J. (1969). *A Study of Grammatical Inference*. Ph.D. thesis, Stanford University.

Horswill, I. (2000). Functional programming of behavior-based systems. *Autonomous Robots*, *9*, 83–93.

Horvitz, E. J. (1987). Problem-solving design: Reasoning about computational value, trade-offs, and resources. In *Proc. Second Annual NASA Research Forum*.

Horvitz, E. J. and Barry, M. (1995). Display of information for time-critical decision making. In *UAI-95*.

Horvitz, E. J., Breese, J. S., Heckerman, D., and Hovel, D. (1998). The Lumiere project: Bayesian user modeling for inferring the goals and needs of software users. In *UAI-98*.

Horvitz, E. J., Breese, J. S., and Henrion, M. (1988). Decision theory in expert systems and artificial intelligence. *IJAR*, *2*, 247–302.

Horvitz, E. J. and Breese, J. S. (1996). Ideal partition of resources for metareasoning. In *AAAI-96*.

Hotelling, H. (1933). Analysis of a complex of statistical variables into principal components. *J. Ed. Psych.*, *24*, 417–441.

Howard, J. and Gugger, S. (2020). *Deep Learning for Coders with fastai and PyTorch*. O'Reilly.

Howard, J. and Ruder, S. (2018). Fine–tuned language models for text classification. arXiv:1801.06146.

Howard, R. A. (1960). *Dynamic Programming and Markov Processes*. MIT Press.

Howard, R. A. (1966). Information value theory. *IEEE Transactions on Systems Science and Cybernetics*, *SSC-2*, 22–26.

Howard, R. A. (1989). Microrisks for medical decision analysis. *Int. J. Technology Assessment in Health Care*, *5*, 357–370.

Howard, R. A. and Matheson, J. E. (1984). Influence diagrams. In Howard, R. A. and Matheson, J. E. (Eds.), *Readings on the Principles and Applications of Decision Analysis*. Strategic Decisions Group.

Howe, D. (1987). The computational behaviour of Girard's paradox. In *LICS-87*.

Howson, C. (2003). Probability and logic. *J. Applied Logic*, *1*, 151–165.

Hsiao, K., Kaelbling, L. P., and Lozano-Perez, T. (2007). Grasping POMDPs. In *ICRA-07*.

Hsu, F.-H. (2004). *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion*. Princeton University Press.

Hsu, F.-H., Anantharaman, T. S., Campbell, M. S., and Nowatzyk, A. (1990). A grandmaster chess machine. *Scientific American, 263,* 44–50.

Hu, J. and Wellman, M. P. (1998). Multiagent reinforcement learning: Theoretical framework and an algorithm. In *ICML-98*.

Hu, J. and Wellman, M. P. (2003). Nash Q-learning for general–sum stochastic games. *JMLR, 4,* 1039–1069.

Huang, T., Koller, D., Malik, J., Ogasawara, G., Rao, B., Russell, S. J., and Weber, J. (1994). Automatic symbolic traffic scene analysis using belief networks. In *AAAI-94*.

Huang, T. and Russell, S. J. (1998). Object identification: A Bayesian analysis with application to traffic surveillance. *AIJ, 103,* 1–17.

Hubel, D. H. and Wiesel, T. N. (1962). Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *J. Physiology, 160,* 106–154.

Hubel, D. H. and Wiesel, T. N. (1968). Receptive fields and functional architecture of monkey striate cortex. *J. Physiology, 195*, 215–243.

Hubel, D. H. (1988). *Eye, Brain, and Vision.* W. H. Freeman.

Hubel, D. H. and Wiesel, T. N. (1959). Receptive fields of single neurons in the cat's striate cortex. *Journal of Physiology*, *148*, 574–591.

Huddleston, R. D. and Pullum, G. K. (2002). *The Cambridge Grammar of the English Language*. Cambridge University Press.

Huffman, D. A. (1971). Impossible objects as nonsense sentences. In Meltzer, B. and Michie, D. (Eds.), *Machine Intelligence 6*. Edinburgh University Press.

Hughes, B. D. (1995). *Random Walks and Random Environments, Vol. 1: Random Walks*. Oxford University Press.

Hughes, G. E. and Cresswell, M. J. (1996). *A New Introduction to Modal Logic*. Routledge.

Huhns, M. N. and Singh, M. (Eds.). (1998). *Readings in Agents*. Morgan Kaufmann.

Hume, D. (1739). *A Treatise of Human Nature* (2nd edition). Republished by Oxford University Press, 1978, Oxford.

Humphrys, M. (2008). How my program passed the Turing test. In Epstein, R., Roberts, G., and Beber, G. (Eds.), *Parsing the Turing Test*. Springer.

Hunsberger, L. and Grosz, B. J. (2000). A combinatorial auction for collaborative planning. In *Int. Conference on Multi-Agent Systems*.

Hunt, W. and Brock, B. (1992). A formal HDL and its use in the FM9001 verification. *Phil. Trans. Roy. Soc.*, *339*.

Hunter, L. and States, D. J. (1992). Bayesian classification of protein structure. *IEEE Expert*, *7*, 67–75.

Hur, C.-K., Nori, A. V., Rajamani, S. K., and Samuel, S. (2014). Slicing probabilistic programs. In *PLDI-14.*

Hurst, M. (2000). *The Interpretation of Text in Tables.* Ph.D. thesis, Edinburgh.

Hurwicz, L. (1973). The design of mechanisms for resource allocation. *American Economic Review Papers and Proceedings*, *63*, 1–30.

Huth, M. and Ryan, M. (2004). *Logic in Computer Science: Modelling and Reasoning About Systems* (2nd edition). Cambridge University Press.

Huttenlocher, D. and Ullman, S. (1990). Recognizing solid objects by alignment with an image. *IJCV*, *5*, 195–212.

Hutter, F., Kotthoff, L., and Vanschoren, J. (2019). *Automated Machine Learning*. Springer.

Huygens, C. (1657). De ratiociniis in ludo aleae. In van Schooten, F. (Ed.), *Exercitionum Mathematicorum.* Elsevirii, Amsterdam. Translated into English by John Arbuthnot (1692).

Huyn, N., Dechter, R., and Pearl, J. (1980). Probabilistic analysis of the complexity of A*. *AIJ*, *15*, 241–254.

Huynh, V. A. and Roy, N. (2009). icLQG: Combining local and global optimization for control in information space. In *ICRA-09.*

Hwa, R. (1998). An empirical evaluation of probabilistic lexicalized tree insertion grammars. In *ACL-98.*

Hwang, C. H. and Schubert, L. K. (1993). EL: A formal, yet natural, comprehensive knowledge representation. In *AAAI-93.*

Hyafil, L. and Rivest, R. (1976). Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*, *5*, 15–17.

Ieong, S. and Shoham, Y. (2005). Marginal contribution nets: A compact representation scheme for coalitional games. In *Proc. Sixth ACM Conference on Electronic Commerce (EC'05).*

Ingerman, P. Z. (1967). Panini–Backus form suggested. *CACM*, *10,* 137.

Intille, S. and Bobick, A. (1999). A framework for recognizing multi-agent action from visual evidence. In *AAAI-99*.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv:1502.03167.

Irpan, A. (2018). Deep reinforcement learning doesn't work yet. www.alexirpan.com/2018/02/14/rl–hard.html.

Isard, M. and Blake, A. (1996). Contour tracking by stochastic propagation of conditional density. In *ECCV-96.*

Isola, P, Zhu, J.-Y., Zhou, T., and Efros, A. (2017). Image-to-image translation with conditional adversarial networks. In *CVPR-17.*

Jaakkola, T. and Jordan, M. I. (1996). Computing upper and lower bounds on likelihoods in intractable networks. In *UAI-96.*

Jacobson, D. H. and Mayne, D. Q. (1970). *Differential Dynamic Programming.* North-Holland.

Jaderberg, M., Czarnecki, W. M., Dunning, I., Marris, L., Lever, G., Castaneda, A. G., Beattie, C., Rabinowitz, N. C., Morcos, A. S., Ruderman, A., *et al.*(2019). Human-level performance in 3D multiplayer games with population-based reinforcement learning. *Science*, *364*, 859–865.

Jaderberg, M., Dalibard, V., Osindero, S., Czarnecki, W. M., Donahue, J., Razavi, A., Vinyals, O., Green, T., Dunning, I., Simonyan, K., Fernando, C., and Kavukcuoglu, K. (2017). Population based training of neuralnetworks. arXiv:1711.09846.

Jaffar, J. and Lassez, J.-L. (1987). Constraint logic programming. In *Proc. Fourteenth ACM POPL Conference*. Association for Computing Machinery.

Jaffar, J., Michaylov, S., Stuckey, P. J., and Yap, R. H. C. (1992). The CLP(R) language and system. *ACM Transactions on Programming Languages and Systems*, *14*, 339–395.

Jain, D., Barthels, A., and Beetz, M. (2010). Adaptive Markov logic networks: Learning statistical relational models with dynamic parameters. In *ECAI-10*.

Jain, D., Kirchlechner, B., and Beetz, M. (2007). Extending Markov logic to model probability distributions in relational domains. In *30th Annual German Conference on AI (K1).*

James, G., Witten, D., Hastie, T., and Tibshirani, R. (2013). *An Introduction to Statistical Learning with Applications in R.* Springer-Verlag.

Jarrett, K., Kavukcuoglu, K., Ranzato, M., and LeCun, Y. (2009). What is the best multi-stage architecture for object recognition? In *ICCV-09.*

Jaynes, E. T. (2003). *Probability Theory: The Logic of Science.* Cambridge Univ. Press.

Jeffrey, R. C. (1983). *The Logic of Decision* (2nd edition). University of Chicago Press.

Jeffreys, H. (1948). *Theory of Probability.* Oxford.

Jelinek, F. (1976). Continuous speech recognition by statistical methods. *Proc. IEEE*, *64*, 532–556.

Jelinek, F. and Mercer, R. L. (1980). Interpolated estimation of Markov source parameters from sparse data. In *Proc. Workshop on Pattern Recognition in Practice.*

Jennings, H. S. (1906). *Behavior of the Lower Organisms.* Columbi aUniversity Press.

Jenniskens, P., Betlem, H., Betlem, J., and Barifaijo, E. (1994). The Mbale meteorite shower. *Meteoritics*, *29*, 246–254.

Jensen, F. V. (2007). *Bayesian Networks and Decision Graphs.* Springer-Verlag.

Ji, Z., Lipton, Z. C., and Elkan, C. (2014). Differential privacy and machine learning: A survey and review. arXiv:1412.7584.

Jiang, H. and Nachum, O. (2019). Identifying and correcting label bias in machine learning. arXiv:1901.04966.

Jimenez, P. and Torras, C. (2000). An efficient algorithm for searching implicit AND/OR graphs with cycles. *AIJ*, *124*, 1–30.

Joachims, T. (2001). A statistical learning model of text classification with support vector machines. In *SIGIR-01*.

Johnson, M. (1998). PCFG models of linguistic tree representations. *Comput. Linguist.*, *24*, 613–632.

Johnson, W. W. and Story, W. E. (1879). Notes on the “15” puzzle. *American Journal of Mathematics*, *2*, 397–404.

Johnston, M. D. and Adorf, H.-M. (1992). Scheduling with neural networks: The case of the Hubble space telescope. *Computers and Operations Research, 19*, 209–240.

Jonathan, P. J. Y., Fung, C. C., and Wong, K. W. (2009). Devious chatbots-interactive malware with a plot. In *FIRA RoboWorld Congress.*

Jones, M. and Love, B. C. (2011). Bayesian fundamentalism or enlightenment? On the explanatory status and theoretical contributions of Bayesian models of cognition. *BBS, 34*, 169–231.

Jones, R. M., Laird, J., and Nielsen, P. E. (1998). Automated intelligent pilots for combat flight simulation. In *AAAI-98.*

Jones, R., McCallum, A., Nigam, K., and Riloff, E. (1999). Bootstrapping for text learning tasks. In *Proc. IJCAI-99 Workshop on Text Mining: Foundations, Techniques, and Applications.*

Jones, T. (2007). *Artificial Intelligence: A Systems Approach.* Infinity Science Press.

Jonsson, A., Morris, P., Muscettola, N., Rajan, K., and Smith, B. (2000). Planning in interplanetary space: Theory and practice. In *AIPS-00*.

Jordan, M. I. (2005). Dirichlet processes, Chinese restaurant processes and all that. Tutorial presentation at the NeurIPS Conference.

Jordan, M. I. (1986). Serial order: A parallel distributed processing approach. Tech. rep., UCSD Institute for Cognitive Science.

Jordan, M. I., Ghahramani, Z., Jaakkola, T., and Saul, L. K. (1999). An introduction to variational methods for graphical models. *Machine Learning, 37*, 183–233.

Jouannaud, J.-P. and Kirchner, C. (1991). Solving equations in abstract algebras: A rule-based survey of unification. In Lassez, J.-L. and Plotkin, G. (Eds.), *Computational Logic*. MIT Press.

Joulin, A., Grave, E., Bojanowski, P., and Mikolov, T. (2016). Bag of tricks for efficient text classification. arXiv:1607.01759.

Jouppi, N. P., Young, C., Patil, N., Patterson, D. A., *et al.* (2017). In-datacenter performance analysis of a tensor processing unit. In *ACM/IEEE 44th International Symposium on Computer Architecture*.

Joy, B. (2000). Why the future doesn't need us. *Wired*, *8*.

Jozefowicz, R., Vinyals, O., Schuster, M., Shazeer, N., and Wu, Y. (2016). Exploring the limits of language modeling. arXiv:1602.02410.

Jozefowicz, R., Zaremba, W., and Sutskever, I. (2015). An empirical exploration of recurrent network architectures. In *ICML-15*.

Juels, A. and Wattenberg, M. (1996). Stochastic hillclimbing as a baseline method for evaluating genetic algorithms. In *NeurIPS 8*.

Julesz, B. (1971). *Foundations of Cyclopean Perception*. University of Chicago Press.

Julian, K. D., Kochenderfer, M. J., and Owen, M. P. (2018). Deep neural network compression for aircraft collision avoidance systems. arXiv:1810.04240.

Juliani, A., Berges, V., Vckay, E., Gao, Y., Henry, H., Mattar, M., and Lange, D. (2018). Unity: A general platform for intelligent agents. arXiv:1809.02627.

Junker, U. (2003). The logic of ilog (j)configurator: Combining constraint programming with a description logic. In *Proc. IJCAI-03 Configuration Workshop*.

Jurafsky, D. and Martin, J. H. (2020). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition* (3rd edition). Prentice-Hall.

Kadane, J. B. and Simon, H. A. (1977). Optimal strategies for a class of constrained sequential problems. *Annals of Statistics*, *5*, 237–255.

Kadane, J. B. and Larkey, P. D. (1982). Subjective probability and the theory of games. *Management Science*, *28*, 113–120.

Kaelbling, L. P, Littman, M. L., and Cassandra, A. R. (1998). Planning and acting in partially observable stochastic domains. *AIJ*, *101*, 99–134.

Kaelbling, L. P. and Rosenschein, S. J. (1990). Action and planning in embedded agents. *Robotics and Autonomous Systems*, *6*, 35–48.

Kager, R. (1999). *Optimality Theory.* Cambridge University Press.

Kahn, H. and Marshall, A. W. (1953). Methods of reducing sample size in Monte Carlo computations. *Operations Research*, *1*, 263–278.

Kahn, H. (1950a). Random sampling (Monte Carlo) techniques in neutron attenuation problems–I. *Nucleonics*, *6*, 27–passim.

Kahn, H. (1950b). Random sampling (Monte Carlo) techniques in neutron attenuation problems–II. *Nucleonics*, *6*, 60–65.

Kahneman, D. (2011). *Thinking, Fast and Slow.* Farrar, Straus and Giroux.

Kahneman, D., Slovic, P., and Tversky, A. (Eds.). (1982). *Judgment under Uncertainty: Heuristics and Biases.* Cambridge University Press.

Kahneman, D. and Tversky, A. (1979). Prospect theory: An analysis of decision under risk. *Econometrica*, 47, 263–291.

Kaindl, H. and Khorsand, A. (1994). Memory-bounded bidirectional search. In *AAAI-94*.

Kalman, R. (1960). A new approach to linear filtering and prediction problems. *J. Basic Engineering*, *82*, 35–46.

Kambhampati, S. (1994). Exploiting causal structure to control retrieval and refitting during plan reuse. *Computational Intelligence*, *10*, 213–244.

Kanade, T., Thorpe, C., and Whittaker, W. (1986). Autonomous land vehicle project at CMU. In *ACM Fourteenth Annual Conference on Computer Science*.

Kanal, E. (2017). Machine learning in cybersecurity. CMU SEI Blog, insights.sei. cmu.edu/sei-blog/2017/06/machine-learning-in-cybersecurity.html.

Kanazawa, A., Black, M., Jacobs, D., and Malik, J. (2018a). End-to-end recovery of human shape and pose. In *CVPR-18*.

Kanazawa, A., Tulsiani, M., Efros, A., and Malik, J. (2018b). Learning category–specific mesh reconstruction from image collections. In *ECCV-18*.

Kanazawa, K., Koller, D., and Russell, S. J. (1995). Stochastic simulation algorithms for dynamic probabilistic networks. In *UAI-95*.

Kang, S. M. and Wildes, R. P. (2016). Review of action recognition and detection methods. arXiv:1610.06906.

Kanter, J. M. and Veeramachaneni, K. (2015). Deep feature synthesis: Towards automating data science endeavors. In *Proc. IEEE Int'l Conf. on Data Science and Advanced Analytics.*

Kantorovich, L. V. (1939). Mathematical methods of organizing and planning production. Published in translation in *Management Science*, 6(4), 366–422, 1960.

Kaplan, D. and Montague, R. (1960). A paradox regained. *Notre Dame Formal Logic, 1*, 79–90.

Karaboga, D. and Basturk, B. (2007). Apowerful and efficient algorithm for numerical function optimization: Artificial bee colony (ABC) algorithm. *Journal of global optimization, 39*, 459–471.

Karamchandani, A., Bjerager, P., and Cornell, C. A. (1989). Adaptive importance sampling. In *Proc. Fifth International Conference on Structural Safety and Reliability*.

Karmarkar, N. (1984). A new polynomial–time algorithm for linear programming. *Combinatorica, 4*, 373–395.

Karp, R. M. (1972). Reducibility among combinatorial problems. In Miller, R. E. and Thatcher, J. W. (Eds.), *Complexity of Computer Computations.* Plenum.

Karpathy, A. (2015). The unreasonable effectiveness of recurrent neural networks. Andrej Karpathy blog, karpathy.github.io/2015/05/21/rnn-effectiveness/.

Karpathy, A. and Fei-Fei, L. (2015). Deep visual-semantic alignments for generating image descriptions. In *CVPR-15*.

Karras, T., Aila, T., Laine, S., and Lehtinen, J. (2017). Progressive growing of GANs for improved quality, stability, and variation. arXiv:1710.10196.

Karsch, K., Hedau, V., Forsyth, D., and Hoiem, D. (2011). Rendering synthetic objects into legacy photographs. In *SIGGRAPH Asia*.

Kartam, N. A. and Levitt, R. E. (1990). A constraint-based approach to construction planning of multi-story buildings. In *Expert Planning Systems*. Institute of Electrical Engineers.

Kasami, T. (1965). An efficient recognition and syntax analysis algorithm for context–free languages. Tech. rep., Air Force Cambridge Research Laboratory.

Katehakis, M. N. and Veinott, A. F. (1987). The multiarmed bandit problem: Decomposition and computation. *Mathematics of Operations Research*, *12*, 185–376.

Katz, B. (1997). Annotating the world wide web using natural language. In *RIAO '97*.

Kaufmann, M., Manolios, P., and Moore, J. S. (2000). *Computer-Aided Reasoning: An Approach*. Kluwer.

Kautz, H. (2006). Deconstructing planning as satisfiability. In *AAAI-06*.

Kautz, H., McAllester, D. A., and Selman, B. (1996). Encoding plans in propositional logic. In *KR-96*.

Kautz, H. and Selman, B. (1992). Planning as satisfiability. In *ECAI-92*.

Kautz, H. and Selman, B. (1998). BLACKBOX: A new approach to the application of theorem proving to problem solving. Working Notes of the AIPS–98 Workshop on Planning as Combinatorial Search.

Kavraki, L., Svestka, P., Latombe, J.-C., and Overmars, M. (1996). Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, *12*, 566–580.

Kazemi, S. M., Kimmig, A., Van den Broeck, G., and Poole, D. (2017). New liftable classes for first-order probabilistic inference. In *NeurIPS 29*.

Kearns, M. (1990). *The Computational Complexity of Machine Learning*. MIT Press.

Kearns, M., Mansour, Y., and Ng, A. Y. (2000). Approximate planning in large POMDPs via reusable trajectories. In *NeurIPS 12*.

Kearns, M. and Singh, S. (1998). Near-optimal reinforcement learning in polynomial time. In *ICML-98*.

Kearns, M. and Vazirani, U. (1994). *An Introduction to Computational Learning Theory*. MIT Press.

Kearns, M. (1988). Thoughts on hypothesis boosting.

Kearns, M., Mansour, Y., and Ng, A. Y. (2002). A sparse sampling algorithm for near-optimal planning in large Markov decision processes. *Machine Learning*, *49*, 193–208.

Kebeasy, R. M., Hussein, A. I., and Dahy, S. A. (1998). Discrimination between natural earthquakes and nuclear explosions using the Aswan Seismic Network. *Annali di Geofisica*, *41*, 127–140.

Keeney, R. L. (1974). Multiplicative utility functions. *Operations Research*, *22*, 22–34.

Keeney, R. L. and Raiffa, H. (1976). *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. Wiley.

Kelley, H. J. (1960). Gradient theory of optimal flight paths. *ARS Journal*, *30*, 947–954.

Kemp, M. (Ed.). (1989). *Leonardo on Painting: An Anthology of Writings.* Yale University Press.

Kempe, A. B. (1879). On the geographical problem of the four-colors. *American Journal of Mathematics*, *2*, 193–200.

Kephart, J. O. and Chess, D. M. (2003). The vision of autonomic computing. *IEEE Computer*, *36*, 41–50.

Kersting, K., Raedt, L. D., and Kramer, S. (2000). Interpreting Bayesian logic programs. In *Proc. AAAI-00 Workshop on Learning Statistical Models from Relational Data*.

Keskar, N. S., McCann, B., Varshney, L., Xiong, C., and Socher, R. (2019). CTRL: A conditional transformer language model for controllable generation. arXiv:1909.

Keynes, J. M. (1921). *A Treatise on Probability*. Macmillan.

Khare, R. (2006). Microformats: The next (small) thing on the semantic web. *IEEE Internet Computing*, *10*, 68–75.

Khatib, O. (1986). Real–time obstacle avoidance for robot manipulator and mobile robots. *Int. J. Robotics Research*, *5*, 90–98.

Kim, B., Khanna, R., and Koyejo, O. O. (2017). Examples are not enough, learn to criticize! Criticism for interpretability. In *NeurIPS 29*.

Kim, J. H. (1983). *CONVINCE: A Conversational Inference Consolidation Engine*. Ph.D. thesis, Department of Computer Science, UCLA.

Kim, J. H. and Pearl, J. (1983). A computational model for combined causal and diagnostic reasoning in inference systems. In *IJCAI-83*.

Kim, J.-H., Lee, C.-H., Lee, K.-H., and Kuppuswamy, N. (2007). Evolving personality of a genetic robot in ubiquitous environment. In *Proc. 16th IEEE International Symposium on Robot and Human Interactive Communication*.

Kim, T. W. (2018). Explainable artificial intelligence (XAI), the goodness criteria and the grasp-ability test. arXiv:1810.09598.

Kingma, D. P. and Welling, M. (2013). Auto-encoding variational Bayes. arXiv:1312.6114.

Kirk, D. E. (2004). *Optimal Control Theory: An Introduction*. Dover.

Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, *220*, 671–680.

Kisynski, J. and Poole, D. (2009). Lifted aggregation in directed first-order probabilistic models. In *IJCAI-09*.

Kitaev, N., Kaiser, L., and Levskaya, A. (2020). Reformer: The efficient transformer. arXiv:2001.04451.

Kitaev, N. and Klein, D. (2018). Constituency parsing with a self-attentive encoder. arXiv:1805.01052.

Kitani, K. M., abd James Andrew Bagnell, B. D. Z., and Hebert, M. (2012). Activity forecasting. In *ECCV-12*.

Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., and Osawa, E. (1997). RoboCup: The robot world cup initiative. In *Proc. First International Conference on Autonomous Agents*.

Kjaerulff, U. (1992). A computational scheme for reasoning in dynamic probabilistic networks. In *UAI-92*.

Klarman, H. E., Francis, J., and Rosenthal, G. D. (1968). Cost effectiveness analysis applied to the treatment of chronic renal disease. *Medical Care*, *6*, 48–54.

Klein, D. and Manning, C. (2001). Parsing with tree-bank grammars: Empirical bounds, theoretical models, and the structure of the Penn treebank. In *ACL-01*.

Klein, D. and Manning, C. (2003). A* parsing: Fast exact Viterbi parse selection. In *HLT-NAACL-03*.

Kleinberg, J. M., Mullainathan, S., and Raghavan, M. (2016). Inherent trade–offs in the fair determination of risk scores. arXiv:1609.05807.

Klemperer, P. (2002). What really matters in auction design. *J. Economic Perspectives*, *16*.

Kneser, R. and Ney, H. (1995). Improved backing–off for M–gram language modeling. In *ICASSP-95*.

Knoblock, C. A. (1991). Search reduction in hierarchical problem solving. In *AAAI-91*.

Knuth, D. E. (1964). Representing numbers using only one 4. *Mathematics Magazine*, *37*, 308–310.

Knuth, D. E. (1975). An analysis of alpha–beta pruning. *AIJ*, *6*, 293–326.

Knuth, D. E. (2015). *The Art of Computer Programming*, Vol. 4, Fascicle 6: Satisfiability. Addison-Wesley.

Knuth, D. E. and Bendix, P. B. (1970). Simple word problems in universal algebras. In Leech, J. (Ed.), *Computational Problems in Abstract Algebra*. Pergamon.

Kober, J., Bagnell, J. A., and Peters, J. (2013). Reinforcement learning in robotics: A survey. *International Journal of Robotics Research*, *32*, 1238–1274.

Koch, C. (2019). *The Feeling of Life Itself.* MIT Press.

Kochenderfer, M. J. (2015). *Decision Making Under Uncertainty: Theory and Application.* MIT Press.

Kocsis, L. and Szepesvari, C. (2006). Bandit-based Monte-Carlo planning. In *ECML-06.*

Koditschek, D. (1987). Exact robot navigation by means of potential functions: Some topological considerations. In *ICRA-87.*

Koehn, P. (2009). *Statistical Machine Translation.* Cambridge University Press.

Koelsch, S. and Siebel, W. A. (2005). Towards a neural basis of music perception. *Trends in Cognitive Sciences, 9*, 578–584.

Koenderink, J. J. (1990). *Solid Shape*. MIT Press.

Koenderink, J. J. and van Doorn, A. J. (1991). Affine structure from motion. *J. Optical Society of America* A, 8, 377–385.

Koenig, S. (1991). Optimal probabilistic and decision-theoretic planning using Markovian decision theory. Master’s report, Computer Science Division, University of California, Berkeley.

Koenig, S. (2000). Exploring unknown environments with real–time search or reinforcement learning. In *NeurIPS 12*.

Koenig, S. (2001). Agent-centered search. *AIMag*, *22*, 109–131.

Koenig, S. and Likhachev, M. (2002). D* Lite. *AAAI-15*, *15*.

Koenig, S., Likhachev, M., and Furcy, D. (2004). Lifelong planning A*. *AIJ*, *155*, 93–146.

Kolesky, D. B., Truby, R. L., Gladman, A. S., Busbee, T. A., Homan, K. A., and Lewis, J. A. (2014). 3D bioprinting of vascularized, heterogeneous cell–laden tissue constructs. *Advanced Materials*, *26*, 3124–3130.

Koller, D., Meggido, N., and von Stengel, B. (1996). Efficient computation of equilibria for extensive two– person games. *Games and Economic Behaviour*, *14*, 247–259.

Koller, D. and Pfeffer, A. (1997). Representations and solutions for game-theoretic problems. *AIJ*, *94*, 167–215.

Koller, D. and Pfeffer, A. (1998). Probabilistic frame-based systems. In *AAAI-98*.

Koller, D. and Friedman, N. (2009). *Probabilistic Graphical Models: Principles and Techniques.* MIT Press.

Koller, D., McAllester, D. A., and Pfeffer, A. (1997). Effective Bayesian inference for stochastic programs. In *AAAI-97*.

Koller, D. and Parr, R. (2000). Policy iteration for factored MDPs. In *UAI-00*.

Koller, D. and Sahami, M. (1997). Hierarchicallyclas-sifying documents using very few words. In *ICML-97*.

Kolmogorov, A. N. (1941). Interpolation und extrapolation von stationaren zufalligen folgen. *Bulletin of the Academy of Sciences of the USSR, Ser. Math. 5*, 3–14.

Kolmogorov, A. N. (1950). *Foundations of the Theory of Probability*. Chelsea.

Kolmogorov, A. N. (1963). On tables of random numbers. *Sankhya, the Indian Journal of Statistics: Series A*, *25*(4), 369–376.

Kolmogorov, A. N. (1965). Three approaches to the quantitative definition of information. *Problems in Information Transmission*, *1*, 1–7.

Kolter, J. Z., Abbeel, P., and Ng, A. Y. (2008). Hierarchical apprenticeship learning, with application to quadruped locomotion. In *NeurIPS 20*.

Kondrak, G. and van Beek, P. (1997). A theoretical evaluation of selected backtracking algorithms. *AIJ*, *89*, 365–387.

Konečnỳ, J., McMahan, H. B., Yu, F. X., Richtárik, P., Suresh, A. T., and Bacon, D. (2016). Federated learning: Strategies for improving communication efficiency. arXiv:1610.05492.

Konolige, K. (1997). COLBERT: A language for reactive control in Saphira. In *Künstliche Intelligenz: Advances in Artificial Intelligence*, LNAI.

Konolige, K. (2004). Large-scale map-making. In *AAAI-04*.

Konolige, K. (1982). A first order formalization of knowledge and action for a multi-agent planning system. In Hayes, J. E., Michie, D., and Pao, Y.-

H. (Eds.), *Machine Intelligence 10.* Ellis Horwood.

Konolige, K. (1994). Easy to be hard: Difficult problems for greedy algorithms. In *KR-94.*

Koopmans, T. C. (1972). Representation of preference orderings over time. In McGuire, C. B. and Radner, R. (Eds.), *Decision and Organization.* Elsevier.

Korb, K. B. and Nicholson, A. (2010). *Bayesian Artificial Intelligence*. CRC Press.

Korf, R. E. (1985a). Depth-first iterative-deepening: an optimal admissible tree search. *AIJ, 27*, 97–109.

Korf, R. E. (1985b). Iterative–deepening A*: An optimal admissible tree search. In *IJCAI-85.*

Korf, R. E. (1987). Planning as search: A quantitative approach. *AIJ, 33*, 65–88.

Korf, R. E. (1990). Real–time heuristic search. *AIJ*, *42*, 189–212.

Korf, R. E. (1993). Linear-space best-first search. *AIJ, 62*, 41–78.

Korf, R. E. and Chickering, D. M. (1996). best-first minimax search. *AIJ*, *84*, 299–337.

Korf, R. E. and Felner, A. (2002). Disjoint pattern database heuristics. *AIJ*, *134*, 9–22.

Korf, R. E. and Zhang, W. (2000). Divide-and- conquer frontier search applied to optimal sequence alignment. In *AAAI-00*.

Korf, R. E. (1997). Finding optimal solutions to Rubik's Cube using pattern databases. In *AAAI-97*.

Korf, R. E. and Reid, M. (1998). Complexity analysis of admissible heuristic search. In *AAAI-98*.

Koutsoupias, E. and Papadimitriou, C. H. (1992). On the greedy algorithm for satisfiability. *Information Processing Letters*, *43*, 53–55.

Kovacs, D. L. (2011). BNF definition of PDDL3.1. Unpublished manuscript from the IPC-2011 website.

Kowalski, R. (1974). Predicate logic as a programming language. In *Proc. IFIP Congress*.

Kowalski, R. (1979). *Logic for Problem Solving*. Elsevier.

Kowalski, R. (1988). The early years of logic programming. *CACM*, *31*, 38–43.

Kowalski, R. and Sergot, M. (1986). A logic-based calculus of events. *New Generation Computing*, *4*, 67–95.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press.

Koza, J. R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press.

Koza, J. R., Bennett, F. H., Andre, D., and Keane, M. A. (1999). *Genetic Programming III: Darwinian Invention and Problem Solving*. Morgan Kaufmann.

Krakovna, V. (2018). Specification gaming examples in AI.

Kraska, T., Beutel, A., Chi, E. H., Dean, J., and Polyzotis, N. (2017). The case for learned index structures. arXiv:1712.01208.

Kraus, S. (2001). *Strategic Negotiation in Multiagent Environments*. MIT Press.

Kraus, S., Ephrati, E., and Lehmann, D. (1991). Negotiation in a non-cooperative environment. AIJ, *3*, 255–281.

Krause, A. and Guestrin, C. (2005). Optimal nonmyopic value of information in graphical models: Efficient algorithms and theoretical limits. In *IJCAI-05*.

Krause, A. and Guestrin, C. (2009). Optimal value of information in graphical models. *JAIR*, *35*, 557–591.

Krause, A., McMahan, B., Guestrin, C., and Gupta, A. (2008). Robust submodular observation selection. *JMLR*, *9*, 2761–2801.

Kripke, S. A. (1963). Semantical considerations on modal logic. *Acta Philosophica Fennica*, *16*, 83–94.

Krishna, V. (2002). *Auction Theory*. AcademicPress.

Krishnamurthy, V. (2016). *Partially Observed*

Markov Decision Processes: From Filtering to Controlled Sensing. Cambridge University Press.

Krishnanand, K. and Ghose, D. (2009). Glowworm swarm optimisation: A new method for optimising multi-modal functions. *International Journal of Computational Intelligence Studies*, *1*, 93–119.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2013). ImageNet classification with deep convolutional neural networks. In *NeurIPS 25*.

Krogh, A., Brown, M., Mian, I. S., Sjolander, K., and Haussler, D. (1994). Hidden Markov models in computational biology: Applications to protein modeling. *J. Molecular Biology*, *235*, 1501–1531.

Krogh, A. and Hertz, J. A. (1992). A simple weight decay can improve generalization. In *NeurIPS 4*.

Kruppa, E. (1913). Zur Ermittlung eines Objecktes aus zwei Perspektiven mit innerer Orientierung. *SitzBer. Akad. Wiss., Wien, Math. Naturw., Kl. Abt. IIa*, *122*, 1939–1948.

Kübler, S., McDonald, R., and Nivre, J. (2009). *Dependency Parsing*. Morgan & Claypool.

Kuffner, J. J. and LaValle, S. (2000). RRT-connect: An efficient approach to single-query path planning. In *ICRA–00*.

Kuhn, H. W. (1953). Extensive games and the problem of information. In Kuhn, H. W. and Tucker, A. W. (Eds.), *Contributions to the Theory of Games II*. Princeton University Press.

Kuhn, H. W. (1955). The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, *2*, 83–97.

Kuipers, B. J. (1985). Qualitative simulation. In Bo- brow, D. (Ed.), *Qualitative Reasoning About Physical Systems*. MIT Press.

Kuipers, B. J. and Levitt, T. S. (1988). Navigation and mapping in large-scale space. *AIMag*, *9*, 25–43.

Kuipers, B. J. (2001). Qualitative simulation. In Meyers, R. A. (Ed.), *Encyclopedia of Physical Science and Technology*. Academic Press.

Kulkarni, T., Kohli, P., Tenenbaum, J. B., and Mansinghka, V. K. (2015). Picture: Aprobabilistic programming language for scene perception. In *CVPR-15*.

Kumar, P. R. and Varaiya, P. (1986). *Stochastic Systems: Estimation, Identification, and Adaptive Control*. Prentice-Hall.

Kumar, S. (2017). A survey of deep learning methods for relation extraction. arXiv:1705.03645.

Kumar, V. and Kanal, L. N. (1988). The CDP: A unifying formulation for heuristic search, dynamic programming, and branch-and-bound. In Kanal, L. N. and Kumar, V. (Eds.), *Search in Artificial Intelligence*. Springer-Verlag.

Kurien, J., Nayak, P., and Smith, D. E. (2002). Fragment-based conformant planning. In *AIPS-02*.

Kurth, T., Treichler, S., Romero, J., Mudigonda, M., Luehr, N., Phillips, E. H., Mahesh, A., Matheson, M., Deslippe, J., Fatica, M., Prabhat, and Houston, M. (2018). Exascale deep learning for climate analytics. arXiv:1810.01993.

Kurzweil, R. (2005). *The Singularity is Near.* Viking.

Kwok, C., Etzioni, O., and Weld, D. S. (2001). Scaling question answering to the web. In *Proc. 10th International Conference on the World Wide Web.*

LaMettrie, J. O. (1748). *L'homme machine.* E. Luzac, Leyde, France.

La Mura, P. and Shoham, Y. (1999). Expected utility networks. In *UAI-99.*

Laborie, P (2003). Algorithms for propagating resource constraints in AI planning and scheduling. *AIJ*, *143*, 151–188.

Ladkin, P. (1986a). Primitives and units for time specification. In *AAAI-86*.

Ladkin, P. (1986b). Time representation: a taxonomy of interval relations. In *AAAI-86*.

Lafferty, J., McCallum, A., and Pereira, F. (2001). Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *ICML-01*.

Lai, T. L. and Robbins, H. (1985). Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, *6*, 4–22.

Laird, J., Newell, A., and Rosenbloom, P. S. (1987). SOAR: An architecture for general intelligence. *AIJ*, *33*, 1–64.

Laird, J., Rosenbloom, P S., and Newell, A. (1986). Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning*, *1*, 11–46.

Laird, J. (2008). Extending the Soar cognitive architecture. In *Artificial General Intelligence Conference.*

Lake, B., Salakhutdinov, R., and Tenenbaum, J. B. (2015). Human-level concept learning through probabilistic program induction. *Science, 350*, 1332–1338.

Lakoff, G. (1987). *Women, Fire, and Dangerous Things: What Categories Reveal About the Mind.* University of Chicago Press.

Lakoff, G. and Johnson, M. (1980). *Metaphors We Live By.* University of Chicago Press.

Lakoff, G. and Johnson, M. (1999). *Philosophy in the Flesh : The Embodied Mind and Its Challenge to Western Thought*. Basic Books.

Lam, J. and Greenspan, M. (2008). Eye-in-hand visual servoing for accurate shooting in pool robotics. In *5th Canadian Conference on Computer and Robot Vision.*

Lamarck, J. B. (1809). *Philosophie zoologique.* Chez Dentu et L'Auteur, Paris.

Lample, G. and Conneau, A. (2019). Cross-lingual language model pretraining. arXiv:1901.07291.

Landhuis, E. (2004). Lifelong debunker takes on arbiter of neutral choices: Magician-turned-mathematician uncovers bias in a flip of a coin. *Stanford Report*, June 7.

Langdon, W. and Poli, R. (2002). *Foundations of Genetic Programming*. Springer.

Langton, C. (Ed.). (1995). *Artificial Life.* MIT Press.

LaPaugh, A. S. (2010). Algorithms and theory of computation handbook. In Atallah, M. J. and Blanton, M. (Eds.), *VLSI Layout Algorithms.* Chapman & Hall/CRC.

Laplace, P. (1816). *Essai philosophique sur les probabilités* (3rd edition). Courcier Imprimeur, Paris.

Larochelle, H. and Murray, I. (2011). The neural autoregressive distribution estimator. In *AISTATS-11*.

Larson, S. C. (1931). The shrinkage of the coefficient of multiple correlation. *J. Educational Psychology, 22*, 45–55.

Laskey, K. B. (1995). Sensitivity analysis for probability assessments in Bayesian networks. *IEEE Transactions on Systems, Man and Cybernetics*, *25*, 901–909.

Laskey, K. B. (2008). MEBN: A language for first order Bayesian knowledge bases. *AIJ*, *172*, 140–178.

Latombe, J.-C. (1991). *Robot Motion Planning*. Kluwer.

Lauritzen, S. (1995). The EM algorithm for graphical association models with missing data. *Computational Statistics and Data Analysis*, *19*, 191–201.

Lauritzen, S., Dawid, A. P., Larsen, B., and Leimer, H. (1990). Independence properties of directed Markov fields. *Networks*, *20*, 491–505.

Lauritzen, S. and Spiegelhalter, D. J. (1988). Local computations with probabilities on graphical structures and their application to expert systems. *J. Royal Statistical Society*, *B 50*, 157–224.

Lauritzen, S. and Wermuth, N. (1989). Graphical models for associations between variables, some of which are qualitative and some quantitative. *Annals of Statistics*, *17*, 31–57.

LaValle, S. (2006). *PlanningAlgorithms*. Cambridge University Press.

Lawler, E. L., Lenstra, J. K., Kan, A., and Shmoys, D. B. (1992). *The Travelling Salesman Problem*. Wiley Interscience.

Lawler, E. L., Lenstra, J. K., Kan, A., and Shmoys, D. B. (1993). Sequencing and scheduling: Algorithms and complexity. In Graves, S. C., Zipkin, P. H., and Kan, A. H. G. R. (Eds.), *Logistics of Production and*

Inventory: Handbooks in Operations Research and Management Science, Volume 4. North-Holland.

Lawler, E. L. and Wood, D. E. (1966). Branch-and-bound methods: A survey. *Operations Research*, *14*, 699–719.

Lazanas, A. and Latombe, J.-C. (1992). Landmark-based robot navigation. In *AAAI-92*.

Le, T. A., Baydin, A. G., and Wood, F. (2017). Inference compilation and universal probabilistic programming. In *AISTATS-17*.

Lebedev, M. A. and Nicolelis, M. A. (2006). Brain-machine interfaces: Past, present and future. *Trends in Neurosciences*, *29*, 536–546.

Lecoutre, C. (2009). *Constraint Networks: Techniques and Algorithms*. Wiley-IEEE Press.

LeCun, Y., Denker, J., and Solla, S. (1990). Optimal brain damage. In *NeurIPS 2*.

LeCun, Y., Jackel, L., Boser, B., and Denker, J. (1989). Handwritten digit recognition: Applications of neural network chips and automatic learning. *IEEE Communications Magazine*, *27*, 41–46.

LeCun, Y., Jackel, L., Bottou, L., Brunot, A., Cortes, C., Denker, J., Drucker, H., Guyon, I., Muller, U., Sackinger, E., Simard, P., and Vapnik, V. N. (1995). Comparison of learning algorithms for handwritten digit recognition. In *Int. Conference on Artificial Neural Networks*.

LeCun, Y., Bengio, Y., and Hinton, G. E. (2015). Deep learning. *Nature*, *521*, 436–444.

Lee, D., Seo, H., and Jung, M. W. (2012). Neural basis of reinforcement learning and decision making. *Annual Review of Neuroscience*, *35*, 287–308.

Lee, K.-F. (2018). *AI Superpowers: China, Silicon Valley, and the New World Order*. Houghton Mifflin.

Leech, G., Rayson, P., and Wilson, A. (2001). *Word Frequencies in Written and Spoken English: Based on the British National Corpus*. Longman.

Legendre, A. M. (1805). *Nouvelles méthodes pour la détermination des orbites des comètes.* Chez Firmin Didot, Paris.

Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kon-tokostas, D., Mendes, P. N., Hellmann, S., Morsey, M., van Kleef, P., Auer, S., and Bizer, C. (2015). DBpedia – A large-scale, multilingual knowledge base extracted fromWikipedia. *Semantic Web, 6*, 167–195.

Lehrer, J. (2009). *How We Decide.* Houghton Mifflin.

Leike, J., Martic, M., Krakovna, V., Ortega, P. A., Everitt, T., Lefrancq, A., Orseau, L., and Legg, S. (2017). Alsafetygridworlds. arXiv:1711.09883.

Lelis, L., Arfaee, S. J., Zilles, S., and Holte, R. C. (2012). Learning heuristic functions faster by using predicted solution costs. In *Proc. FifthAnnual Symposium on Combinatorial Search.*

Lenat, D. B. (1975). BEINGS: Knowledge as interacting experts. In *IJCAI-75.*

Lenat, D. B. and Guha, R. V. (1990). *Building Large Knowledge-Based Systems: Representation and Inference in the CYC Project.* Addison-Wesley.

Leonard, H. S. and Goodman, N. (1940). The calculus of individuals and its uses. *JSL*, *5*, 45–55.

Leonard, J. and Durrant-Whyte, H. (1992). *Directed Sonar Sensing for Mobile Robot Navigation.* Kluwer.

Lepage, G. P. (1978). A new algorithm for adaptive multidimensional integration. *Journal of Computational Physics*, *27*, 192–203.

Lerner, U. (2002). *Hybrid Bayesian Networks for Reasoning About Complex Systems.* Ph.D. thesis, Stanford University.

Leśniewski, S. (1916). Podstawy ogólnej teorii mnogości. Poplawski.

Lesser, V. R. and Corkill, D. D. (1988). The distributed vehicle monitoring testbed: A tool for investigating distributed problem solving networks. In Engelmore, R. and Morgan, T. (Eds.), *Blackboard Systems*. Addison-Wesley.

Letz, R., Schumann, J., Bayerl, S., and Bibel, W. (1992). SETHEO: A high-performance theorem prover. *JAR*, *8*, 183–212.

Levesque, H. J. and Brachman, R. J. (1987). Expressiveness and tractability in knowledge representation and reasoning. *Computational Intelligence*, *3*, 78–93.

Levin, D. A., Peres, Y., and Wilmer, E. L. (2008). *Markov Chains and Mixing Times.* American Mathematical Society.

Levine, S., Finn, C., Darrell, T., and Abbeel, P. (2016). End-to-end training of deep visuomotor policies. *JMLR*, *17*, 1334–1373.

Levine, S., Pastor, P., Krizhevsky, A., Ibarz, J., and Quillen, D. (2018). Learning hand–eye coordination for robotic grasping with deep learning and large-scale data collection. *International Journal of Robotics Research, 37*, 421–436.

Levy, D. (1989). The million pound bridge program. In Levy, D. and Beal, D. (Eds.), *Heuristic Programming in Artificial Intelligence.* Ellis Horwood.

Levy, D. (2008). *Love and Sex with Robots: The Evolution of Human-Robot Relationships.* Harper.

Levy, O. and Goldberg, Y. (2014). Linguistic regularities in sparse and explicit word representations. In *Proc. Eighteenth Conference on Computational Natural Language Learning*.

Leyton–Brown, K. and Shoham, Y. (2008). *Essentials of Game Theory: A Concise, Multidisciplinary Introduction.* Morgan & Claypool.

Li, C. M. and Anbulagan (1997). Heuristics based on unit propagation for satisfiability problems. In *IJCAI-97.*

Li, K. and Malik, J. (2018a). Implicit maximum likelihood estimation. arXiv:1809.09087.

Li, K. and Malik, J. (2018b). On the implicit assumptions of GANs. arXiv:1811.12402.

Li, M., Vitányi, P., *et al.* (2008). *An Introduction to Kolmogorov Complexity and Its Applications* (3rd edition). Springer-Verlag.

Li, T.-M., Gharbi, M., Adams, A., Durand, F., and Ragan-Kelley, J. (2018). Differentiable programming for image processing and deep learning in Halide. *ACM Transactions on Graphics, 37*, 139.

Li, W. and Todorov, E. (2004). Iterative linear quadratic regulator design for nonlinear biological movement systems. In *Proc. 1st International*

Conference on Informatics in Control, Automation and Robotics.

Li, X. and Yao, X. (2012). Cooperatively coevolving particle swarms for large scale optimization. *IEEE Trans. Evolutionary Computation, 16*, 210–224.

Li, Z., Li, P., Krishnan, A., and Liu, J. (2011). Large-scale dynamic gene regulatory network inference combining differential equation models with local dynamic Bayesian network analysis. *Bioinformatics, 27 19*, 2686–91.

Liang, P., Jordan, M. I., and Klein, D. (2011). Learning dependency–based compositional semantics. arXiv:1109.6841.

Liang, P. and Potts, C. (2015). Bringing machine learning and compositional semantics together. *Annual Review of Linguistics*, *1*, 355–376.

Liberatore, P. (1997). The complexity of the language **A**. *Electronic Transactions on Artificial Intelligence, 1*, 13–38.

Lifschitz, V. (2001). Answer set programming and plan generation. *AIJ*, *138*, 39–54.

Lighthill, J. (1973). Artificial intelligence: A general survey. In Lighthill, J., Sutherland, N. S., Needham, R. M., Longuet–Higgins, H. C., and Michie, D. (Eds.), *Artificial Intelligence: A Paper Symposium.* Science Research Council of Great Britain.

Lillicrap, T., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2015). Continuous control with deep reinforcement learning. arXiv:1509.02971.

Lin, S. (1965). Computer solutions of the travelling salesman problem. *Bell Systems Technical Journal*, *44(10)*, 2245–2269.

Lin, S. and Kernighan, B. W. (1973). An effective heuristic algorithm for the travelling-salesman problem. *Operations Research*, *21*, 498–516.

Lindley, D. V. (1956). On a measure of the information provided by an experiment. *Annals of Mathematical Statistics*, *27*, 986–1005.

Lindsay, R. K., Buchanan, B. G., Feigenbaum, E. A., and Lederberg, J. (1980). *Applications of Artificial Intelligence for Organic Chemistry: The DENDRAL Project.* McGraw-Hill.

Lindsten, F., Jordan, M. I., and Schön, T. B. (2014). Particle Gibbs with ancestor sampling. *JMLR*, *15*, 2145–2184.

Littman, M. L. (1994). Markov games as a framework for multi-agent reinforcement learning. In *ICML-94*.

Littman, M. L., Cassandra, A. R., and Kaelbling, L. P. (1995). Learning policies for partially observable environments: Scalingup. In *ICML-95*.

Littman, M. L. (2015). Reinforcement learning improves behaviour from evaluative feedback. *Nature*, *521*, *445–451*.

Littman, M. L., Topcu, U., Fu, J., Isbell, C., Wen, M., and MacGlashan, J. (2017). Environment-independent taskspecifications viaGLTL. arXiv:1704.04341.

Liu, B., Gemp, I., Ghavamzadeh, M., Liu, J., Mahadevan, S., and Petrik, M. (2018). Proximal gradient temporal difference learning: Stable reinforcement learning with polynomial sample complexity. *JAIR*, *63*, 461–494.

Liu, H., Simonyan, K., Vinyals, O., Fernando, C., and Kavukcuoglu, K. (2017). Hierarchical representations for efficient architecture search. arXiv:1711.00436.

Liu, H., Simonyan, K., and Yang, Y. (2019). DARTS: Differentiable architecture search. In *ICLR-19*.

Liu, J. and Chen, R. (1998). Sequential Monte Carlo methods for dynamic systems. JASA, 93, 1022–1031.

Liu, J. and West, M. (2001). Combined parameter and state estimation in simulation-based filtering. In Doucet, A., de Freitas, J. F. G., and Gordon, N. (Eds.), *Sequential Monte Carlo Methods in Practice.* Springer.

Liu, L. T., Dean, S., Rolf, E., Simchowitz, M., and Hardt, M. (2018a). Delayed impact of fair machine learning. arXiv:1803.04383.

Liu, M.-Y., Breuel, T., and Kautz, J. (2018b). Unsupervised image-to-image translation networks. In *NeurIPS 30*.

Liu, X., Faes, L., Kale, A. U., Wagner, S. K., Fu, D. J., Bruynseels, A., Mahendiran, T., Moraes, G., Shamdas, M., Kern, C., Ledsam, J. R., Schmid, M., Balaskas, K., Topol, E., Bachmann, L. M., Keane, P A., and Denniston, A. K. (2019a). A comparison of deep learning performance against health-care professionals in detecting diseases from medical imaging: A systematic review and meta-analysis. *The Lancet Digital Health.*

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. (2019b). RoBERTa: A robustiy optimized BERT pretraining approach. arXiv:1907.11692.

Liu, Y., Jain, A., Eng, C., Way, D. H., Lee, K., Bui, P., Kanada, K., de Oliveira Marinho, G., Gallegos, J., Gabriele, S., Gupta, V., Singh, N., Natarajan, V., Hofmann-Wellenhof, R., Corrado, G., Peng, L., Webster, D. R., Ai, D., Huang, S., Liu, Y., Dunn, R. C., and Coz, D. (2019c). A deep learning system for differential diagnosis of skin diseases. arXiv:1909.

Liu, Y., Gadepalli, K. K., Norouzi, M., Dahl, G., Kohlberger, T., Venugopalan, S., Boyko, A. S., Timofeev, A., Nelson, P Q., Corrado, G., Hipp, J. D., Peng, L., and Stumpe, M. C. (2017). Detecting cancer metastases on gigapixel pathology images. arXiv:1703.02442.

Liu, Y., Kohlberger, T., Norouzi, M., Dahl, G., Smith, J. L., Mohtashamian, A., Olson, N., Peng, L., Hipp, J. D., and Stumpe, M. C. (2018). Artificial intelligence-based breast cancer nodal metastasis detection: Insights into the black box for pathologists. *Archives of Pathology & Laboratory Medicine, 143*, 859–868.

Livescu, K., Glass, J., and Bilmes, J. (2003). Hidden feature modeling for speech recognition using dynamic Bayesian networks. In *EUROSPEECH-2003*.

Lloyd, S. (2000). Ultimate physical limits to computation. *Nature*, *406*, 1047–1054.

Lloyd, W. F. (1833). *Two Lectures on the Checks to Population*. Oxford University.

Llull, R. (1305). *Ars Magna*. Published as Salzinger, I. *et al.* (Eds.), *Raymundi Lulli Opera omnia*, Mainz, 1721–1742.

Loftus, E. and Palmer, J. (1974). Reconstruction of automobile destruction: An example of the interaction between language and memory. *J. VerbalLearning and Verbal Behavior, 13,* 585–589.

Lohn, J. D., Kraus, W. F., and Colombano, S. P. (2001). Evolutionary optimization of yagi-uda antennas. In *Proc. Fourth International Conference on Evolvable Systems*.

Longuet-Higgins, H. C. (1981). A computer algorithm for reconstructing a scene from two projections. *Nature*, *293*, 133–135.

Loos, S., Irving, G., Szegedy, C., and Kaliszyk, C. (2017). Deep network guided proof search. In *Proc. 21st Int'l Conf. on Logic for Programming, Artificial Intelligence and Reasoning*.

Lopez de Segura, R. (1561). *Libro de la invencion liberal y arte del juego del axedrez.* Andres de Angulo.

Lorentz, R. (2015). Early playout termination in MCTS. In Plaat, A., van den Herik, J., and Kosters, W. (Eds.), *Advances in Computer Games*. SpringerVerlag.

Love, N., Hinrichs, T., and Genesereth, M. R. (2006). General game playing: Game description language specification. Tech. rep., Stanford University Computer Science Dept.

Lovejoy, W. S. (1991). A survey of algorithmic methods for partially observed Markov decision processes. *Annals of Operations Research*, *28*, 47–66.

Lovelace, A. (1843). Sketch of the analytical engine invented by Charles Babbage. Notes appended to Lovelace’s translation of an article of the above title written by L. F. Menabrea based on lectures by Charles babbage in 1840. The translation appeared in R. Taylor (Ed.), *Scientific Memoirs, vol. III*. R. and J. E. Taylor, London.

Loveland, D. (1970). A linear format for resolution. In *Proc. IRIA Symposium on Automatic Demonstration*.

Lowe, D. (1987). Three-dimensional object recognition from single two-dimensional images. *AIJ*, *31*, 355–395.

Lowe, D. (2004). Distinctive image features from scale–invariant keypoints. *IJCV*, *60*, 91–110.

Löwenheim, L. (1915). Über möglichkeiten im Relativkalkül. *Mathematische Annalen*, *76*, 441–470.

Lowerre, B. T. (1976). *The* HARPY *Speech Recognition System*. Ph.D. thesis, Computer Science Department, Carnegie-Mellon University.

Lowry, M. (2008). Intelligent software engineering tools for NASA's crew exploration vehicle. In *ISMIS-08*.

Loyd, S. (1959). *Mathematical Puzzles of Sam Loyd: Selected and Edited by Martin Gardner*. Dover.

Lozano–Perez, T. (1983). Spatial planning: A configuration space approach. *IEEE Transactions on Computers*, *C-32*, 108–120.

Lozano–Perez, T., Mason, M., and Taylor, R. (1984). Automatic synthesis of fine-motion strategies for robots. *Int. J. Robotics Research*, *3*, 3–24.

Lu, F. and Milios, E. (1997). Globally consistent range scan alignment for environment mapping. *Autonomous Robots*, *4*, 333–349.

Lubberts, A. and Miikkulainen, R. (2001). Coevolving a Go-playing neural network. In *GECCO-01*.

Luby, M., Sinclair, A., and Zuckerman, D. (1993). Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, *47*, 173–180.

Lucas, J. R. (1961). Minds, machines, and Godel. *Philosophy*, *36*.

Lucas, J. R. (1976). This Gödel is killing me: A rejoinder. *Philosophia*, *6*, 145–148.

Lucas, P., van der Gaag, L., and Abu-Hanna, A. (2004). Bayesian networks in biomedicine and healthcare. *Artificial Intelligence in Medicine*.

Luce, D. R. and Raiffa, H. (1957). *Games and Decisions*. Wiley.

Lukasiewicz, T. (1998). Probabilistic logic programming. In *ECAI-98*.

Lundberg, S. M. and Lee, S.-I. (2018). A unified approach to interpreting model predictions. In *NeurIPS 30*.

Lunn, D., Jackson, C., Best, N., Thomas, A., and Spiegelhalter, D. J. (2013). *The BUGS Book: A Practical Introduction to Bayesian Analysis.* Chapman and Hall.

Lunn, D., Thomas, A., Best, N., and Spiegelhalter, D. J. (2000). WinBUGS–a Bayesian modelling framework: Concepts, structure, and extensibility. *Statistics and Computing*, *10*, 325–337.

Luo, S., Bimbo, J., Dahiya, R., and Liu, H. (2017). Robotic tactile perception of object properties: A review. *Mechatronics*, *48*, 54–67.

Lyman, P. and Varian, H. R. (2003). How much information? www.sims.berkeley.edu/how-much-info-2003.

Lynch, K. and Park, F. C. (2017). *Modern Robotics.* Cambridge University Press.

Machina, M. (2005). Choice under uncertainty. In *Encyclopedia of Cognitive Science*. Wiley.

MacKay, D. J. C. (2002). *Information Theory, Inference and Learning Algorithms*. Cambridge University Press.

MacKenzie, D. (2004). *Mechanizing Proof*. MIT Press.

Mackworth, A. K. (1977). Consistency in networks of relations. *AIJ*, *8*, 99–118.

Mackworth, A. K. and Freuder, E. C. (1985). The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *AIJ*, 25, 65–74.

Madhavan, R. and Schlenoff, C. I. (2003). Moving object prediction for off-road autonomous navigation. In *Unmanned Ground Vehicle Technology V*.

Mailath, G. and Samuelson, L. (2006). *Repeated Games and Reputations: Long-Run Relationships*. Oxford University Press.

Majercik, S. M. and Littman, M. L. (2003). Contingent planning under uncertainty via stochastic satisfiability. *AIJ*, *147*, 119–162.

Malhotra, P., Vig, L., Shroff, G., and Agarwal, P. (2015). Long short term memory networks for anomaly detection in time series. In *ISANN-15*.

Malik, D., Palaniappan, M., Fisac, J. F., Hadfield-Menell, D., Russell, S. J., and Dragan, A. D. (2018). An efficient, generalized bellman update for cooperative inverse reinforcement learning. In *ICML-18*.

Malone, T. W. (2004). *The Future of Work*. Harvard Business Review Press.

Maneva, E., Mossel, E., and Wainwright, M. (2007). A new look at survey propagation and its generalizations. arXiv:cs/0409012.

Manna, Z. and Waldinger, R. (1971). Toward automatic program synthesis. *CACM*, *14*, 151–165.

Manna, Z. and Waldinger, R. (1985). *The Logical Basis for Computer Programming: Volume 1: Deductive Reasoning*. Addison-Wesley.

Manne, A. S. (1960). Linear programming and sequential decisions. *Management Science*, *6*, 259–267.

Manning, C. and Schutze, H. (1999). *Foundations of Statistical Natural Language Processing*. MIT Press.

Manning, C., Raghavan, P., and Schutze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press.

Mannion, M. (2002). Using first-order logic for product line model validation. In *Software Product Lines: Second International Conference.*

Mansinghka, V. K., Selsam, D., and Perov, Y. (2013). Venture: A higher-order probabilistic programming platform with programmable inference. arXiv:1404.0099.

Marbach, P. and Tsitsiklis, J. N. (1998). Simulation-based optimization of Markov reward processes. Technical report, Laboratory for Information and Decision Systems, MIT.

Marcus, G. (2009). *Kluge: The Haphazard Evolution of the Human Mind.* Mariner Books.

Marcus, M. P., Santorini, B., and Marcinkiewicz, M. A. (1993). Building a large annotated corpus of English: The Penn treebank. *Computational*

Linguistics, *19*, 313–330.

Marinescu, R. and Dechter, R. (2009). AND/OR branch-and-bound search for combinatorial optimization in graphical models. *AIJ*, *173*, 1457–1491.

Markov, A. (1913). An example of statistical investigation in the text of “Eugene Onegin” illustrating coupling of “tests” in chains. *Proc. Academy of Sciences of St. Petersburg*, *7*, 153–162.

Marler, R. T. and Arora, J. S. (2004). Survey of multiobjective optimization methods for engineering. *Structural and Multidisciplinary Optimization*, *26*, 369–395.

Maron, M. E. (1961). Automatic indexing: An experimental inquiry. *JACM*, *8*, 404–417.

Màrquez, L. and Rodríguez, H. (1998). Part-of-speech tagging using decision trees. In *ECML-98*.

Marr, D. and Poggio, T. (1976). Cooperative computation of stereo disparity. *Science*, *194*, 283–287.

Marr, D. (1982). *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information*. W. H. Freeman.

Marriott, K. and Stuckey, P. J. (1998). *Programming with Constraints: An Introduction*. MIT Press.

Marsland, S. (2014). *Machine Learning: An Algorithmic Perspective* (2nd edition). CRC Press.

Martelli, A. and Montanari, U. (1973). Additive AND/OR graphs. In *IJCAI-73*.

Martelli, A. (1977). On the complexity of admissible search algorithms. *AIJ*, *8*, 1–13.

Marthi, B., Pasula, H., Russell, S. J., and Peres, Y. (2002). Decayed MCMC filtering. In *UAI-02*.

Marthi, B., Russell, S. J., Latham, D., and Guestrin, C. (2005). Concurrent hierarchical reinforcement learning. In *IJCAI-05*.

Marthi, B., Russell, S. J., and Wolfe, J. (2007). Angelic semantics for high-level actions. In *ICAPS–07*.

Marthi, B., Russell, S. J., and Wolfe, J. (2008). Angelic hierarchical planning: Optimal and online algorithms. In *ICAPS–08*.

Martin, D., Fowlkes, C., and Malik, J. (2004). Learning to detect natural image boundaries using local brightness, color, and texture cues. *PAMI*, *26*, 530–549.

Martin, F. G. (2012). Will massive open online courses change how we teach? *CACM*, *55*, 26–28.

Martin, J. H. (1990). *A Computational Model of Metaphor Interpretation*. Academic Press.

Mason, M. (1993). Kicking the sensing habit. *AIMag*, *14*, 58–59.

Mason, M. (2001). *Mechanics of Robotic Manipulation*. MIT Press.

Mason, M. and Salisbury, J. (1985). *Robot Hands and the Mechanics of Manipulation*. MIT Press.

Mataric, M. J. (1997). Reinforcement learning in the multi-robot domain. *Autonomous Robots*, *4*, 73–83.

Mates, B. (1953). *StoicLogic*. University of California Press.

Matuszek, C., Cabral, J., Witbrock, M., and DeOliveira, J. (2006). An introduction to the syntax and semantics of Cyc. In *Proc. AAAI Spring Symposium on Formalizing and Compiling Background Knowledge and Its Applications to Knowledge Representation and Question Answering*.

Mausam and Kolobov, A. (2012). *Planning with Markov Decision Processes: An AI Perspective*. Morgan & Claypool.

Maxwell, J. (1868). On governors. *Proc. Roy. Soc.*, *16*, 270–283.

Mayer, J., Khairy, K., and Howard, J. (2010). Drawing an elephant with four complex parameters. *American Journal of Physics*, *78*, 648–649.

Mayor, A. (2018). *Gods and Robots: Myths, Machines, and Ancient Dreams of Technology*. Princeton University Press.

McAllester, D. A. (1980). An outlook on truth maintenance. AI memo, MIT AI Laboratory.

McAllester, D. A. (1988). Conspiracy numbers for minmax search. *AIJ*, 35, 287–310.

McAllester, D. A. (1998). What is the most pressing issue facing AI and the AAAI today? Candidate statement, election for Councilor of the American Association for Artificial Intelligence.

McAllester, D. A. and Rosenblitt, D. (1991). Systematic nonlinear planning. In *AAAI-91*.

McAllester, D. A. (1990). Truth maintenance. In *AAAI-90*.

McAllester, D. A., Milch, B., and Goodman, N. D. (2008). Random-world semantics and syntactic independence for expressive languages. Technical report, MIT.

McCallum, A. (2003). Efficiently inducing features of conditional random fields. In *UAI-03*.

McCallum, A., Schultz, K., and Singh, S. (2009). FACTORIE: Probabilistic programming via imperatively defined factor graphs. In *NeurIPS 22*.

McCarthy, J. (1958). Programs with common sense. In *Proc. Symposium onMechanisation of Thought Processes*.

McCarthy, J. (1963). Situations, actions, and causal laws. Memo, Stanford University Artificial Intelligence Project.

McCarthy, J. (1968). Programs with common sense. In Minsky, M. L. (Ed.), *Semantic Information Processing*. MIT Press.

McCarthy, J. (1980). Circumscription: A form of non-monotonic reasoning. AIJ, *13*, 27–39.

McCarthy, J. (2007). From here to human-level AI. AIJ, 171.

McCarthy, J. and Hayes, P. J. (1969). Some philosophical problems from the standpoint of artificial intelligence. In Meltzer, B., Michie, D., and Swann, M. (Eds.), *Machine Intelligence 4*. Edinburgh University Press.

McCawley, J. D. (1988). *The Syntactic Phenomena of English*. University of Chicago Press.

McCorduck, P. (2004). *Machines Who Think: A Personal Inquiry Into the History and Prospects of Artificial Intelligence* (Revised edition). A K

Peters.

McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, *5*, 115–137.

McCune, W. (1997). Solution of the Robbins problem. *JAR*, *19*, 263–276.

McCune, W. (1990). Otter 2.0. In *International Conference on Automated Deduction*.

McDermott, D. (1976). Artificial intelligence meets natural stupidity. *SIGART Newsletter*, *57*, 4–9.

McDermott, D. (1978a). Planning and acting. *Cognitive Science*, *2*, 71–109.

McDermott, D. (1978b). Tarskian semantics, or no notation without denotation! *Cognitive Science*, *2*, 277–282.

McDermott, D. (1985). Reasoning about plans. In Hobbs, J. and Moore, R. (Eds.), *Formal theories of the commonsense world.* Ablex.

McDermott, D. (1987). A critique of pure reason. *Computational Intelligence*, *3*, 151–237.

McDermott, D. (1996). A heuristic estimator for means-ends analysis in planning. In *ICAPS–96.*

McDermott, D. and Doyle, J. (1980). non-monotonic logic: i. *AIJ*, *13*, 41–72.

McDermott, J. (1982). R1: A rule–based configurer of computer systems. *AIJ*, *19*, 39–88.

McEliece, R. J., MacKay, D. J. C., and Cheng, J.-F. (1998). Turbo decoding as an instance of Pearl's "belief propagation" algorithm. *lEEE Journal on Selected Areas in Communications, 16*, 140–152.

McGregor, J. J. (1979). Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences*, *19*, 229–250.

McIlraith, S. and Zeng, H. (2001). Semantic web services. *IEEE Intelligent Systems, 16*, 46–53.

McKinney, W. (2012). *Python for Data Analysis: Data Wrangling with Pandas*. O'Reilly.

McLachlan, G. J. and Krishnan, T. (1997). *The EM Algorithm and Extensions*. Wiley.

McMahan, H. B. and Andrew, G. (2018). A general approach to adding differential privacy to iterative training procedures. arXiv:1812.06210.

McMillan, K. L. (1993). *Symbolic Model Checking.* Kluwer.

McWhorter, J. H. (2014). *The Language Hoax: Why the World Looks the Same in Any Language.* Oxford University Press.

Meehl, P. (1955). *Clinical vs. Statistical Prediction.* University of Minnesota Press.

Mehrabi, N., Morstatter, F., Saxena, N., Lerman, K., and Galstyan, A. (2019). A survey on bias and fairness in machine learning. arXiv:1908.09635.

Mendel, G. (1866). Versuche uber pflanzen-hybriden. *Verhandlungen des Naturforschenden Vereins, Abhandlungen, Brunn, 4*, 3–47. Translated into

English by C. T. Druery, published by Bateson (1902).

Mercer, J. (1909). Functions of positive and negative type and their connection with the theory of integral equations. *Phil. Trans. Roy. Soc., A, 209*, 415–446.

Merleau–Ponty, M. (1945). *Phenomenology of Perception*. Routledge.

Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, A., and Teller, E. (1953). Equations of state calculations by fast computing machines. *J. Chemical Physics*, *21*, 1087–1091.

Metropolis, N. and Ulam, S. (1949). The beginning of the Monte Carlo method. *Journal of the American Statistical Association*, *44*, 335–341.

Mézard, M., Parisi, G., and Virasoro, M. (1987). *Spin Glass Theory and Beyond: An Introduction to the Replica Method and Its Applications*. World Scientific.

Michie, D. (1966). Game–playing and game-learning automata. In Fox, L. (Ed.), *Advances in Programming and non-Numerical Computation*. Pergamon.

Michie, D. (1972). Machine intelligence at Edinburgh. *Management Informatics*, *2*, 7–12.

Michie, D. and Chambers, R. A. (1968). BOXES: An experiment in adaptive control. In Dale, E. and Michie, D. (Eds.), *Machine Intelligence 2*. Elsevier.

Michie, D. (1963). Experiments on the mechanization of game-learning Part I. Characterization of the model and its parameters. *The Computer Journal*, *6*, 232–236.

Miikkulainen, R., Liang, J., Meyerson, E., Rawal, A., Fink, D., Francon, O., Raju, B., Shahrzad, H., Navruzyan, A., Duffy, N., *et al.* (2019). Evolving deep neural networks. In *Artificial Intelligence in the Age of Neural Networks and Brain Computing*. Elsevier.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space. arXiv:1301.3781.

Mikolov, T., Karafiat, M., Burget, L., Cernocky, J., and Khudanpur, S. (2010). Recurrent neural network based language model. In *Eleventh Annual Conference of the International Speech Communication Association*.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., and Dean, J. (2014). Distributed representations of words and phrases and their compositionality. In *NeurIPS 26*.

Milch, B. (2006). *Probabilistic Models with Unknown Objects.* Ph.D. thesis, UC Berkeley.

Milch, B., Marthi, B., Sontag, D., Russell, S. J., Ong, A., and Kolobov, A. (2005). BLOG: Probabilistic models with unknown objects. In *IJCAI-05.*

Milch, B., Zettlemoyer, L., Kersting, K., Haimes, M., and Kaelbling, L. P. (2008). Lifted probabilistic inference with counting formulas. In *AAAI-08.*

Milgrom, P. (1997). Putting auction theory to work: The simultaneous ascending auction. Tech. rep., Stanford University Department of Economics.

Mill, J. S. (1863). *Utilitarianism.* Parker, Son and Bourn, London.

Miller, A. C., Merkhofer, M. M., Howard, R. A., Matheson, J. E., and Rice, T. R. (1976). Development of automated aids for decision analysis. Technical report, SRI International.

Miller, T., Howe, P, and Sonenberg, L. (2017). Explainable AI: Beware of inmates running the asylum. In *Proc. IJCAI-17 Workshop on Explainable AI*.

Minka, T. (2010). Bayesian linear regression. Unpublished manuscript.

Minka, T., Cleven, R., and Zaykov, Y. (2018). TrueSkill 2: An improved Bayesian skill rating system. Tech. rep., Microsoft Research.

Minker, J. (2001). *Logic–Based Artificial Intelligence*. Kluwer.

Minsky, M. L. (1975). A framework for representing knowledge. In Winston, P. H. (Ed.), *The Psychology of Computer Vision*. McGraw-Hill.

Minsky, M. L. (1986). *The Society of Mind*. Simon and Schuster.

Minsky, M. L. (2007). *The Emotion Machine: Commonsense Thinking, Artificial Intelligence, and the Future of the Human Mind*. Simon and Schuster.

Minsky, M. L. and Papert, S. (1969). *Perceptrons: An Introduction to Computational Geometry*. MIT Press.

Minsky, M. L. and Papert, S. (1988). *Perceptrons: An Introduction to Computational Geometry* (Expanded edition). MIT Press.

Minsky, M. L., Singh, P., and Sloman, A. (2004). The St. Thomas common sense symposium: Designing architectures for human-level intelligence. *AIMag*, *25*, 113–124.

Minton, S., Johnston, M. D., Philips, A. B., and Laird, P. (1992). Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *AIJ*, *58*, 161–205.

Mirjalili, S. M. and Lewis, A. (2014). Grey wolf optimizer. *Advances in Engineering Software*, *69*, 46–61.

Misak, C. (2004). *The Cambridge Companion to Peirce*. Cambridge University Press.

Mitchell, M., Wu, S., Zaldivar, A., Barnes, P., Vasserman, L., Hutchinson, B., Spitzer, E., Raji, I. D., and Gebru, T. (2019). Model cards for model reporting. *Proc. of the Conference on Fairness, Accountability, and Transparency*.

Mitchell, M. (1996). *An Introduction to Genetic Algorithms*. MIT Press.

Mitchell, M. (2019). *Artificial Intelligence: A Guide for Thinking Humans*. Farrar, Straus and Giroux.

Mitchell, M., Holland, J. H., and Forrest, S. (1996). When will a genetic algorithm outperform hill climbing? In *NeurIPS 6*.

Mitchell, T. M. (1997). *Machine Learning*. McGrawHill.

Mitchell, T. M. (2005). Reading the web: A breakthrough goal for AI. *AIMag*, *26*.

Mitchell, T. M. (2007). Learning, information extraction and the web. In *ECML-07*.

Mitchell, T. M., Cohen, W., Hruschka, E., Talukdar, P, Yang, B., Betteridge, J., Carlson, A., Dalvi, B., Gardner, M., Kisiel, B., *et al.* (2018). Never-ending learning. *CACM*, *61*, 103–115.

Mitchell, T. M., Shinkareva, S. V., Carlson, A., Chang, K.-M., Malave, V. L., Mason, R. A., and Just, M. A. (2008). Predicting human brain activity associated with the meanings of nouns. *Science*, *320*, 1191–1195.

Mittelstadt, B. (2019). Principles alone cannot guarantee ethical AI. *Nature Machine Intelligence*, *1*, 501–507.

Mitten, L. G. (1960). An analytic solutlon to the least cost testing sequence problem. *Journal of Industrial Engineering*, *11*, 17.

Miyato, T., Kataoka, T., Koyama, M., and Yoshida, Y. (2018). Spectral normalization for generative adversarial networks. arXiv:1802.05957.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. A. (2013). Playing Atari with deep reinforcement learning. arXiv:1312.5602.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M. A., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*, 529–533.

Mohr, R. and Henderson, T. C. (1986). Arc and path consistency revisited. *AIJ*, *28*, 225–233.

Montague, R. (1970). English as a formal language. In Visentini, B. (Ed.), *Linguaggi nella Società e nella Tecnica*. Edizioni di Comunità.

Montague, R. (1973). The proper treatment of quantification in ordinary English. In Hintikka, K. J. J., Moravcsik, J. M. E., and Suppes, P. (Eds.), *Approaches to Natural Language*. D. Reidel.

Montanari, U. (1974). Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, *7*, 95–132.

Montemerlo, M. and Thrun, S. (2004). Large-scale robotic 3-D mapping of urban structures. In *Proc. International Symposium on Experimental Robotics*.

Montemerlo, M., Thrun, S., Koller, D., and Wegbreit, B. (2002). FastSLAM: A factored solution to the simultaneous localization and mapping problem. In *AAAI-02*.

Mooney, R. (1999). Learning for semantic interpretation: Scaling up without dumbing down. In *Proc. 1st Workshop on Learning Language in Logic*.

Mirjalili, S. M. and Lewis, A. (2014). Grey wolf optimizer. *Advances in Engineering Software*, *69*, 46–61.

Moore, A. M. and Wong, W.-K. (2003). Optimal reinsertion: A new search operator for accelerated and more accurate Bayesian network structure learning. In *ICML-03*.

Moore, A. W. and Atkeson, C. G. (1993). Prioritized sweeping—Reinforcement learning with less data and less time. *Machine Learning*, *13*, 103–130.

Moore, A. W. and Lee, M. S. (1997). Cached sufficient statistics for efficient machine learning with large datasets. *JAIR*, *8*, 67–91.

Moore, E. F. (1959). The shortestpaththroughamaze. In *Proc. International Symposium on the Theory of Switching, Part II*. Harvard University Press.

Moore, R. C. (1980). Reasoning about knowledge and action. Artificial intelligence center technical note, SRI International.

Moore, R. C. (1985). A formal theory of knowledge and action. In Hobbs, J. R. and Moore, R. C. (Eds.), *Formal Theories of the Commonsense World.* Ablex.

Moore, R. C. and DeNero, J. (2011). L1 and L2 regularization for multiclass hinge loss models. In *Symposium on Machine Learning in Speech and Natural Language Processing*.

Moravčík, M., Schmid, M., Burch, N., Lisy, V., Morrill, D., Bard, N., Davis, T., Waugh, K., Johanson, M., and Bowling, M. (2017). Deepstack: Expert-level artificial intelligence in no-limit poker. arXiv:1701.01724.

Moravec, H. P. (1983). The Stanford cart and the CMU rover. *Proc. IEEE, 71*, 872–884.

Moravec, H. P. and Elfes, A. (1985). High resolution maps from wide angle sonar. In *ICRA–85.*

Moravec, H. P. (2000). *Robot: MereMachine to Transcendent Mind*. Oxford University Press.

Morgan, C. L. (1896). *Habit and Instinct.* Edward Arnold.

Morgan, T. J. H. and Griffiths, T. L. (2015). What the Baldwin Effect affects. In *COGSCI-15.*

Morjaria, M. A., Rink, F. J., Smith, W. D., Klempner, G., Burns, C., and Stein, J. (1995). Elicitation of probabilities for belief networks: Combining qualitative and quantitative information. In *UAI-95.*

Morrison, P. and Morrison, E. (Eds.). (1961). *Charles Babbage and His Calculating Engines: Selected Writings by Charles Babbage and Others.* Dover.

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. (2001). Chaff: Engineering an efficient SAT solver. In *Proc. 38th Design Automation Conference.*

Mott, A., Job, J., Vlimant, J.-R., Lidar, D., and Spiropulu, M. (2017). Solving a Higgs optimization problem with quantum annealing for machine learning. *Nature*, *550*, 375.

Motzkin, T. S. and Schoenberg, I. J. (1954). The relaxation method for linear inequalities. *Canadian Journal of Mathematics*, *6*, 393–404.

Moutarlier, P. and Chatila, R. (1989). Stochastic multisensory data fusion for mobile robot location and environment modeling. In *ISRR-89*.

Mueller, E. T. (2006). *Commonsense Reasoning*. Morgan Kaufmann.

Muggleton, S. H. and De Raedt, L. (1994). Inductive logic programming: Theory and methods. *J. Logic Programming*, *19/20*, 629–679.

Müller, M. (2002). Computer Go. *AIJ*, *134*, 145–179.

Mumford, D. and Shah, J. (1989). Optimal approximations by piece–wise smooth functions and associated variational problems. *Commun. Pure Appl Math.*, *42*, 577–685.

Mundy, J. and Zisserman, A. (Eds.). (1992). *Geometric Invariance in Computer Vision.* MIT Press.

Munos, R., Stepleton, T., Harutyunyan, A., and Bellemare, M. G. (2017). Safe and efficient off–policy reinforcement learning. In *NeurIPS 29*.

Murphy, K. (2002). *Dynamic Bayesian Networks: Representation, Inference and Learning*. Ph.D. thesis, UC Berkeley.

Murphy, K. (2012). *Machine Learning: A Probabilistic Perspective*. MIT Press.

Murphy, K. and Mian, I. S. (1999). Modelling gene expression data using Bayesian networks. Tech. rep., Computer Science Division, UC Berkeley.

Murphy, K. and Russell, S. J. (2001). Rao- Blackwellised particle filtering for dynamic Bayesian networks. In Doucet, A., de Freitas, J. F. G., and Gordon, N. J. (Eds.), *Sequential Monte Carlo Methods in Practice*. Springer-Verlag.

Murphy, K. and Weiss, Y. (2001). The factored frontier algorithm for approximate inference in DBNs. In *UAI-01*.

Murphy, R. (2000). *Introduction to AI Robotics*. MIT Press.

Murray, L. M. (2013). Bayesian state-space modelling on high-performance hardware using LibBi. arXiv:1306.3277.

Murray, R. M. (2017). *A Mathematical Introduction to Robotic Manipulation*. CRC Press.

Murray-Rust, P, Rzepa, H. S., Williamson, J., and Willighagen, E. L. (2003). Chemical markup, XML and the world-wide web. 4. CML schema. *J. Chem. Inf. Comput. Sci., 43*, 752–772.

Murthy, C. and Russell, J. R. (1990). A constructive proof of Higman’s lemma. In *LICS-90.*

Muscettola, N. (2002). Computing the envelope for stepwise-constantresource allocations. In *CP-02.*

Muscettola, N., Nayak, P., Pell, B., and Williams, B. (1998). Remote agent: To boldly go where no AI system has gone before. *AIJ*, *103*, 5–48.

Muslea, I. (1999). Extraction patterns for information extraction tasks: A survey. In *Proc. AAAI-99 Workshop on Machine Learning for Information Extraction*.

Muth, J. T., Vogt, D. M., Truby, R. L., Mengtic, Y., Kolesky, D. B., Wood, R. J., and Lewis, J. A. (2014). Embedded 3D printing of strain sensors within highly stretchable elastomers. *Advanced Materials*, *26*, 6307–6312.

Myerson, R. (1981). Optimal auction design. *Mathematics of Operations Research*, *6*, 58–73.

Myerson, R. (1986). Multistage games with communication. *Econometrica*, *54*, 323–358.

Myerson, R. (1991). *Game Theory: Analysis of Conflict*. Harvard University Press.

Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted Boltzmann machines. In *ICML-10*.

Nalwa, V. S. (1993). *A Guided Tour of Computer Vision*. Addison-Wesley.

Narayanan, A., Shi, E., and Rubinstein, B. I. (2011). Link prediction by de-anonymization: How we won the Kaggle social network challenge. In *IJCNN-11*.

Narayanan, A. and Shmatikov, V. (2006). How to break anonymity of the Netflix prize dataset. arXiv:cs/0610105.

Nash, J. (1950). Equilibrium points in N–person games. *PNAS*, *36*, 48–49.

Nash, P. (1973). *Optimal Allocation of Resources Between Research Projects*. Ph.D. thesis, University of Cambridge.

Nayak, P. and Williams, B. (1997). Fast context switching in real–time propositional reasoning. In *AAAI-97*.

Neches, R., Swartout, W. R., and Moore, J. D. (1985). Enhanced maintenance and explanation of expert systems through explicit models of their development. *IEEE Transactions on Software Engineering*, *SE-11*, 1337–1351.

Nemhauser, G. L., Wolsey, L. A., and Fisher, M. L. (1978). An analysis of approximations for maximizing submodular set functions I. *Mathematical Programming*, *14*, 265–294.

Nesterov, Y. and Nemirovski, A. (1994). *Interior-Point Polynomial Methods in Convex Programming*. SIAM (Society for Industrial and Applied Mathematics).

Newell, A. (1982). The knowledge level. AIJ, *18*, 82–127.

Newell, A. (1990). *Unified Theories of Cognition*. Harvard University Press.

Newell, A. and Ernst, G. (1965). The search for generality. In *Proc. IFIP Congress*.

Newell, A., Shaw, J. C., and Simon, H. A. (1957). Empirical explorations with the logic theory machine. *Proc. Western Joint Computer Conference*, *15*, 218–239. Reprinted in Feigenbaum and Feldman (1963).

Newell, A. and Simon, H. A. (1961). GPS, a program that simulates human thought. In Billing, H. (Ed.), *Lernende Automaten*. R. Oldenbourg.

Newell, A. and Simon, H. A. (1972). *Human Problem Solving*. Prentice-Hall.

Newell, A. and Simon, H. A. (1976). Computer science as empirical inquiry: Symbols and search. *CACM*, *19*, 113–126.

Newton, I. (1664–1671). Methodus fluxionum et serierum infinitarum. Unpublished notes.

Ng, A. Y. (2004). Feature selection, L_1 vs. L_2 regularization, and rotational invariance. In *ICML-04*.

Ng, A. Y. (2019). *Machine Learning Yearning.* www.mlyearning.org.

Ng, A. Y., Harada, D., and Russell, S. J. (1999). Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML-99*.

Ng, A. Y. and Jordan, M. I. (2000). PEGASUS:Apol– icy search method for large MDPs and POMDPs. In *UAI-00*.

Ng, A. Y. and Jordan, M. I. (2002). On discriminative vs. generative classifiers: A comparison of logistic regression and naive Bayes. In *NeurIPS 14*.

Ng, A. Y., Kim, H. J., Jordan, M. I., and Sastry, S. (2001). Autonomous helicopter flight via reinforcement learning. In *NeurIPS 16*.

Ng, A. Y. and Russell, S. J. (2000). Algorithms for inverse reinforcement learning. In *ICML-00*.

Nicholson, A. and Brady, J. M. (1992). The data association problem when monitoring robot vehicles using dynamic belief networks. In *ECAI-92*.

Nielsen, M. A. (2015). *Neural Networks and Deep Learning*. Determination Press.

Nielsen, T. and Jensen, F. (2003). Sensitivity analysis in influence diagrams. *IEEE Transactions on Systems, Man and Cybernetics*, *33*, 223–234.

Niemelä, I., Simons, P., and Syrjanen, T. (2000). Smodels: A system for answer set programming. In *Proc. 8th International Workshop on non-Monotonic Reasoning*.

Nikolaidis, S. and Shah, J. (2013). Human-robot cross-training: computational formulation, modeling and evaluation of a human team training strategy. In *HRI-13*.

Niles, I. and Pease, A. (2001). Towards a standard upper ontology. In *Proc. International Conference on Formal Ontology in Information Systems*.

Nilsson, D. and Lauritzen, S. (2000). Evaluating influence diagrams using LIMIDs. In *UAI-00.*

Nilsson, N. J. (1965). *Learning Machines: Foundations of Trainable Pattern-Classifying Systems.* McGraw-Hill.

Nilsson, N. J. (1971). *Problem-Solving Methods in Artificial Intelligence.* McGraw-Hill.

Nilsson, N. J. (1984). Shakey the robot. Technical note, SRI International.

Nilsson, N. J. (1986). Probabilistic logic. *AIJ*, *28*, 71–87.

Nilsson, N. J. (1995). Eye on the prize. *AIMag*, *16*, 9–17.

Nilsson, N. J. (2009). *The Quest for Artificial Intelligence: A History of Ideas and Achievements.* Cambridge University Press.

Nisan, N. (2007). Introduction to mechanism design (for computer scientists). In Nisan, N., Roughgarden, T., Tardos, E., and Vazirani, V. V. (Eds.), *Algorithmic Game Theory.* Cambridge University Press.

Nisan, N., Roughgarden, T., Tardos, E., and Vazirani, V. (Eds.). (2007). *Algorithmic Game Theory.* Cambridge University Press.

Niv, Y. (2009). Reinforcement learning in the brain. *Journal of Mathematical Psychology, 53*, 139–154.

Nivre, J., De Marneffe, M.-C., Ginter, F., Goldberg, Y., Hajic, J., Manning, C., McDonald, R., Petrov, S., *et al.* (2016). Universal dependencies v1: A multilingual treebank collection. In *Proc. International Conference on Language Resources and Evaluation.*

Nodelman, U., Shelton, C., and Koller, D. (2002). Continuous time Bayesian networks. In *UAI-02.*

Noe, A. (2009). *Out of Our Heads: Why You Are Not Your Brain, and Other Lessons from the Biology of Consciousness.* Hill and Wang.

Nordfors, D., Cerf, V., and Senges, M. (2018). *Disrupting Unemployment.* Amazon Digital Services.

Norvig, P. (1988). Multiple simultaneous interpretations of ambiguous sentences. In *COGSCI-88*.

Norvig, P. (1992). *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp.* Morgan Kaufmann.

Norvig, P. (2009). Natural language corpus data. In Segaran, T. and Hammerbacher, J. (Eds.), *Beautiful Data.* O'Reilly.

Nowick, S. M., Dean, M. E., Dill, D. L., and Horowitz, M. (1993). The design of a high-performance cache controller: A case study in asynchronous synthesis. *Integration: The VLSI Journal*, *15*, 241–262.

Och, F. J. and Ney, H. (2003). A systematic comparison of various statistical alignment models. *Computational Linguistics*, *29*, 19–51.

Och, F. J. and Ney, H. (2004). The alignment template approach to statistical machine translation. *Computational Linguistics*, *30*, 417–449.

Och, F. J. and Ney, H. (2002). Discriminative training and maximum entropy models for statistical machine translation. In *COLING–02*.

Ogawa, S., Lee, T.-M., Kay, A. R., and Tank, D. W. (1990). Brain magnetic resonance imaging with contrast dependent on blood oxygenation. *PNAS*, *87*, 9868–9872.

Oh, M.-S. and Berger, J. O. (1992). Adaptive importance sampling in Monte Carlo integration. *Journal of Statistical Computation and Simulation*, *41*, 143–168.

Oh, S., Russell, S. J., and Sastry, S. (2009). Markov chain Monte Carlo data association for multi-target tracking. *IEEE Transactions on Automatic Control*, *54*, 481–497.

Oizumi, M., Albantakis, L., and Tononi, G. (2014). From the phenomenology to the mechanisms of consciousness: Integrated information theory 3.0. *PLoS Computational Biology*, *10*, e1003588.

Olesen, K. G. (1993). Causal probabilistic networks with both discrete and continuous variables. *PAMI*, *15*, 275–279.

Oliver, N., Garg, A., and Horvitz, E. J. (2004). Layered representations for learning and inferring office activity from multiple sensory channels. *Computer Vision and Image Understanding*, *96*, 163–180.

Oliver, R. M. and Smith, J. Q. (Eds.). (1990). *Influence Diagrams, Belief Nets and Decision Analysis*. Wiley.

Omohundro, S. (2008). The basicAIdrives. In *AGI-08 Workshop on the Sociocultural, Ethical and Futurological Implications of Artificial Intelligence*.

O'Neil, C. (2017). *Weapons of Math Destruction: How Big Data Increases Inequality and Threatens Democracy*. Broadway Books.

O'Neil, C. and Schutt, R. (2013). *Doing Data Science: Straight Talk from the Frontline*. O'Reilly.

O'Reilly, U.-M. and Oppacher, F. (1994). Program search with a hierarchical variable length representation: Genetic programming, simulated annealing and hill climbing. In *Proc. Third Conference on Parallel Problem Solving from Nature*.

Osborne, M. J. (2004). *An Introduction to Game Theory.* Oxford University Pres.

Osborne, M. J. and Rubinstein, A. (1994). *A Course in Game Theory.* MIT Press.

Osherson, D. N., Stob, M., and Weinstein, S. (1986). *Systems That Learn: An Introduction to Learning Theory for Cognitive and Computer Scientists.*

MIT Press.

Ostrom, E. (1990). *Governing the Commons.* Cambridge University Press.

Padgham, L. and Winikoff, M. (2004). *Developing Intelligent Agent Systems: A Practical Guide*. Wiley.

Paige, B. and Wood, F. (2014). A compilation target for probabilistic programming languages. In *ICML-14*.

Paige, B., Wood, F., Doucet, A., and Teh, Y. W. (2015). Asynchronous anytime sequential Monte Carlo. In *NeurIPS 27*.

Palacios, H. and Geffner, H. (2007). From conformant into classical planning: Efficient translations that may be complete too. In *ICAPS-07*.

Palmer, S. (1999). *Vision Science: Photons to Phenomenology*. MIT Press.

Papadimitriou, C. H. (1994). *Computational Complexity.* Addison-Wesley.

Papadimitriou, C. H. and Tsitsiklis, J. N. (1987). The complexity of Markov decision processes. *Mathematics of Operations Research, 12*, 441–450.

Papadimitriou, C. H. and Yannakakis, M. (1991). Shortest paths without a map. *Theoretical Computer Science, 84*, 127–150.

Papavassiliou, V. and Russell, S. J. (1999). Convergence of reinforcement learning with general function approximators. In *IJCAI-99*.

Parisi, G. (1988). *Statistical Field Theory.* Addison-Wesley.

Parisi, M. M. G. and Zecchina, R. (2002). Analytic and algorithmic solution of random satisfiability problems. *Science, 297*, 812–815.

Park, J. D. and Darwiche, A. (2004). Complexity results and approximation strategies for MAP explanations. *JAIR*, 21, 101–133.

Parker, D. B. (1985). Learning logic. Technical report, Center for Computational Research in Economics and Management Science, MIT.

Parker, L. E. (1996). On the design of behavior-based multi-robot teams. *J. Advanced Robotics*, *10*, 547–578.

Parr, R. and Russell, S. J. (1998). Reinforcement learning with hierarchies of machines. In *NeurIPS 10*.

Parzen, E. (1962). On estimation of a probability density function and mode. *Annals of Mathematical Statistics*, *33*, 1065–1076.

Pasca, M. and Harabagiu, S. M. (2001). High performance question/answering. In *SIGIR-01*.

Pasca, M., Lin, D., Bigham, J., Lifchits, A., and Jain, A. (2006). Organizing and searching the world wide web of facts—Step one: The one–million fact extraction challenge. In *AAAI-06*.

Paskin, M. (2002). Maximum entropy probabilistic logic. Tech. report, UC Berkeley.

Pasula, H., Marthi, B., Milch, B., Russell, S. J., and Shpitser, I. (2003). Identity uncertainty and citation matching. In *NeurIPS 15*.

Pasula, H., Russell, S. J., Ostland, M., and Ritov, Y. (1999). Tracking many objects with many sensors. In *IJCAI-99*.

Patel-Schneider, P. (2014). Analyzing schema.org. In *Proc. International Semantic Web Conference.*

Patrick, B. G., Almulla, M., and Newborn, M. (1992). An upper bound on the time complexity of iterative-deepening-A*. *AIJ*, *5*, 265–278.

Paul, R. P. (1981). *Robot Manipulators: Mathematics, Programming, and Control*. MIT Press.

Pauls, A. and Klein, D. (2009). K-best A* parsing. In *ACL-09*.

Peano, G. (1889). *Arithmetices principia, nova methodo exposita*. Fratres Bocca, Turin.

Pearce, J., Tambe, M., and Maheswaran, R. (2008). Solving multiagent networks using distributed constraint optimization. *AIMag*, *29*, 47–62.

Pearl, J. (1982a). Reverend Bayes on inference engines: A distributed hierarchical approach. In *AAAI-82*.

Pearl, J. (1982b). The solution for the branching factor of the alpha–beta pruning algorithm and its optimality. *CACM*, *25*, 559–564.

Pearl, J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Pearl, J. (1985). Bayesian networks: A model of self-activated memory for evidential reasoning. In *COGSCI-85*.

Pearl, J. (1986). Fusion, propagation, and structuring in belief networks. *AIJ*, *29*, 241–288.

Pearl, J. (1987). Evidential reasoning using stochastic simulation of causal models. *AIJ*, *32*, 247–257.

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann.

Pearl, J. (2000). *Causality: Models, Reasoning, and Inference*. Cambridge University Press.

Pearl, J. and McKenzie, D. (2018). *The Book of Why.* Basic Books.

Pearl, J. and Verma, T. (1991). A theory of inferred causation. In *KR-91*.

Pearson, K. (1895). Contributions to the mathematical theory of evolution, II: Skew variation in homogeneous material. *Phil. Trans. Roy. Soc., 186*, 343–414.

Pearson, K. (1901). On lines and planes of closest fit to systems of points in space. *Philosophical Magazine, 2*, 559–572.

Parker, A., Nau, D. S., and Subrahmanian, V. S. (2005). Game–tree search with combinatorially large belief states. In *IJCAI-05*.

Pease, A. and Niles, I. (2002). IEEE standard upper ontology: A progress report. *Knowledge Engineering Review, 17*, 65–70.

Pednault, E. P. D. (1986). Formulating multiagent, dynamic-world problems in the classical planning framework. In *Reasoning About Actions and Plans: Proc. 1986 Workshop.*

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., *et al.* (2011). Scikit-learn: Machine learning in Python. *JMLR, 12*, 2825–2830.

Peirce, C. S. (1870). Description of a notation for the logic of relatives, resulting from an amplification of the conceptions of Boole’s calculus of logic. *Memoirs of the American Academy of Arts and Sciences, 9*, 317–378.

Peirce, C. S. (1883). A theory of probable inference. Note B. The logic of relatives. In Peirce, C. S. (Ed.), *Studies in Logic*, Little, Brown.

Peirce, C. S. (1909). Existential graphs. Unpublished manuscript; reprinted in (Buchler 1955).

Peleg, B. and Sudholter, P. (2002). *Introduction to the Theory of Cooperative Games* (2nd edition). SpringerVerlag.

Pelikan, M., Goldberg, D. E., and Cantu-Paz, E. (1999). BOA: The Bayesian optimization algorithm. In *GECCO-99.*

Pemberton, J. C. and Korf, R. E. (1992). Incremental planning on graphs with cycles. In *AIPS-92*.

Penberthy, J. S. and Weld, D. S. (1992). UCPOP: A sound, complete, partial order planner for ADL. In *KR-92*.

Peng, J. and Williams, R. J. (1993). Efficient learning and planning within the Dyna framework. *Adaptive Behavior*, 2, 437–454.

Pennington, J., Socher, R., and Manning, C. (2014). Glove: Global vectors for word representation. In *EMNLP-14.*

Penrose, R. (1989). *The Emperor's New Mind.* Oxford University Press.

Penrose, R. (1994). *Shadows of the Mind*. Oxford University Press.

Peot, M. and Smith, D. E. (1992). Conditional nonlinear planning. In *ICAPS-92*.

Pereira, F. and Schabes, Y. (1992). Inside–outside reestimation from partially bracketed corpora. In *ACL-92*.

Pereira, F. and Warren, D. H. D. (1980). Definite clause grammars for language analysis: A survey of the formalism and a comparison with augmented transition networks. *AIJ*, *13*, 231–278.

Peters, J. and Schaal, S. (2008). Reinforcement learning of motor skills with policy gradients. *Neural Networks*, *21*, 682–697.

Peters, J., Janzing, D., and Scholkopf, B. (2017). *Elements of Causal Inference: Foundations and Learning Algorithms*. MIT press.

Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. (2018). Deep contextualized word representations. arXiv:1802.05365.

Peterson, C. and Anderson, J. R. (1987). A mean field theory learning algorithm for neural networks. *Complex Systems*, *1*, 995–1019.

Petosa, N. and Balch, T. (2019). Multiplayer AlphaZero. arXiv:1910.13012.

Pfeffer, A. (2001). IBAL: A probabilistic rational programming language. In *IJCAI-01*.

Pfeffer, A., Koller, D., Milch, B., and Takusagawa, K. T. (1999). SPOOK: A system for probabilistic object-orientedknowledgerepresentation. In *UAI-99*.

Pfeffer, A. (2016). *Practical Probabilistic Programming*. Manning.

Pfeffer, A. (2000). *Probabilistic Reasoning for Complex Systems*. Ph.D. thesis, Stanford University.

Pfeffer, A. (2007). The design and implementation of IBAL: A general-purpose probabilistic language. In Getoor, L. and Taskar, B. (Eds.), *Introduction to Statistical Relational Learning*. MIT Press.

Pfeifer, R., Bongard, J., Brooks, R. A., and Iwasawa, S. (2006). *How the Body Shapes the Way We Think: A New View of Intelligence.* Bradford.

Pham, H., Guan, M. Y., Zoph, B., Le, Q. V., and Dean, J. (2018). Efficient neural architecture search via parameter sharing. arXiv:1802.03268.

Pineau, J., Gordon, G., and Thrun, S. (2003). Point-based value iteration: An anytime algorithm for POMDPs. In *IJCAI-03*.

Pinedo, M. (2008). *Scheduling: Theory, Algorithms, and Systems.* SpringerVerlag.

Pinkas, G. and Dechter, R. (1995). Improving connectionist energy minimization. *JAIR*, *3*, 223–248.

Pinker, S. (1995). Language acquisition. In Gleitman, L. R., Liberman, M., and Osherson, D. N. (Eds.), *An Invitation to Cognitive Science* (2nd edition). MIT Press.

Pinker, S. (2003). *The Blank Slate: The Modern Denial of Human Nature.* Penguin.

Pinto, D., McCallum, A., Wei, X., and Croft, W. B. (2003). Table extraction using conditional random fields. In *SIGIR-03*.

Pinto, L. and Gupta, A. (2016). Supersizing selfsupervision: Learning to grasp from 50k tries and 700 robot hours. In *ICRA-16*.

Platt, J. (1999). Fast training of support vector machines using sequential minimal optimization. In *Advances in Kernel Methods: Support Vector Learning*. MIT Press.

Plotkin, G. (1972). Building-in equational theories. In Meltzer, B. and Michie, D. (Eds.), *Machine Intelligence 7*. Edinburgh University Press.

Plummer, M. (2003). JAGS: A program for analysis of Bayesian graphical models using Gibbs sampling. In *Proc. Third Int'l Workshop on Distributed*

Statistical Computing.

Pnueli, A. (1977). The temporal logic of programs. In *FOCS–77*.

Pohl, I. (1971). Bi-directional search. In Meltzer, B. and Michie, D. (Eds.), *Machine Intelligence 6*. Edinburgh University Press.

Pohl, I. (1973). The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving. In *IJCAI-73*.

Pohl, I. (1977). Practical and theoretical considerations in heuristic search algorithms. In Elcock, E. W. and Michie, D. (Eds.), *Machine Intelligence 8*. Ellis Horwood.

Pohl, I. (1970). Heuristic search viewed as path finding in a graph. *AIJ*, *1*, 193–204.

Poli, R., Langdon, W., and McPhee, N. (2008). *A Field Guide to Genetic Programming*. Lulu.com.

Pomerleau, D. A. (1993). *Neural Network Perception for Mobile Robot Guidance*. Kluwer.

Poole, B., Lahiri, S., Raghu, M., Sohl–Dickstein, J., and Ganguli, S. (2017). Exponential expressivity in deep neural networks through transient chaos. In *NeurIPS 29*.

Poole, D. (1993). Probabilistic Horn abduction and Bayesian networks. *AIJ*, *64*, 81–129.

Poole, D. (2003). First-order probabilistic inference. In *IJCAI-03*.

Poole, D. and Mackworth, A. K. (2017). *Artificial Intelligence: Foundations of Computational Agents* (2 edition). Cambridge University Press.

Poppe, R. (2010). A survey on vision-based human action recognition. *Image Vision Comput.*, *28*, 976–990.

Popper, K. R. (1959). *The Logic of Scientific Discovery*. Basic Books.

Popper, K. R. (1962). *Conjectures and Refutations: The Growth of Scientific Knowledge*. Basic Books. Portner, P. and Partee, B. H. (2002). *Formal Semantics: The Essential Readings*. Wiley-Blackwell.

Post, E. L. (1921). Introduction to a general theory of elementary propositions. *American Journal of Mathematics*, *43*, 163–185.

Poulton, C. and Watts, M. (2016). MIT and DARPA pack Lidar sensor onto single chip. *IEEE Spectrum*, August 4.

Poundstone, W. (1993). *Prisoner's Dilemma*. Anchor.

Pourret, O., Naim, P., and Marcot, B. (2008). *Bayesian Networks: A Practical Guide to Applications*. Wiley.

Pradhan, M., Provan, G. M., Middleton, B., and Henrion, M. (1994). Knowledge engineering for large belief networks. In *UAI-94*.

Prawitz, D. (1960). An improved proof procedure. *Theoria*, *26*, 102–139.

Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (2007). *Numerical Recipes: The Art of Scientific Computing* (3rd edition). Cambridge University Press.

Preston, J. and Bishop, M. (2002). *Views into the Chinese Room: New Essays on Searle and Artificial Intelligence*. Oxford University Press.

Prieditis, A. E. (1993). Machine discovery of effective admissible heuristics. *Machine Learning*, 12, 117–141.

Prosser, P. (1993). Hybrid algorithms for constraint satisfaction problems. *Computational Intelligence*, *9*, 268–299.

Pullum, G. K. (1991). *The Great Eskimo Vocabulary Hoax (and Other Irreverent Essays on the Study of Language).* University of Chicago Press.

Pullum, G. K. (1996). Learnability, hyperlearning, and the poverty of the stimulus. In *22nd Annual Meeting of the Berkeley Linguistics Society*.

Puterman, M. L. (1994). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley.

Puterman, M. L. and Shin, M. C. (1978). Modified policy iteration algorithms for discounted Markov decision problems. *Management Science*, *24*, 1127– 1137.

Putnam, H. (1963). ‘Degree of confirmation’ and inductive logic. In Schilpp, P. A. (Ed.), *The Philosophy of Rudolf Carnap*. Open Court.

Quillian, M. R. (1961). A design for an understanding machine. Paper presented at a colloquium: Semantic Problems in Natural Language, King’s College, Cambridge, England.

Quine, W. V. (1953). Two dogmas of empiricism. In *From a Logical Point of View.* Harper and Row.

Quine, W. V. (1960). *Word and Object*. MIT Press.

Quine, W. V. (1982). *Methods of Logic* (4th edition). Harvard University Press.

Quinlan, J. R. (1979). Discovering rules from large collections of examples: A case study. In Michie, D. (Ed.), *Expert Systems in the Microelectronic Age*. Edinburgh University Press.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, *1*, 81–106.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann.

Quinlan, S. and Khatib, O. (1993). Elastic bands: Connecting path planning and control. In *ICRA-93*.

Quirk, R., Greenbaum, S., Leech, G., and Svartvik, J. (1985). *A Comprehensive Grammar of the English Language*. Longman.

Rabani, Y., Rabinovich, Y., and Sinclair, A. (1998). A computational view of population genetics. *Random Structures and Algorithms*, *12*, 313–334.

Rabiner, L. R. and Juang, B.-H. (1993). *Fundamentals of Speech Recognition.* Prentice-Hall.

Radford, A., Metz, L., and Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. arXiv:1511.06434.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. (2019). Language models are unsupervised multitask learners. *OpenAI Blog*, *1*.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. (2019). Exploring the limits of transfer learning with a unified text-to-text transformer. arXiv:1910.10683.

Rafferty, A. N., Brunskill, E., Griffiths, T. L., and Shafto, P. (2016). Faster teaching via POMDP planning. *Cognitive Science*, *40*, 1290–1332.

Rahwan, T., Michalak, T. P., Wooldridge, M., and Jennings, N. R. (2015). Coalition structure generation: A survey. *AIJ*, *229*, 139–174.

Raibert, M., Blankespoor, K., Nelson, G., and Playter, R. (2008). Bigdog, the rough-terrain quadruped robot. *IFAC Proceedings Volumes*, *41*, 10822–10825.

Rajpurkar, P., Zhang, J., Lopyrev, K., and Liang, P. (2016). Squad: 100,000+questionsfor machine comprehension of text. In *EMNLP-16*.

Ramsey, F. P. (1931). Truth and probability. In Braithwaite, R. B. (Ed.), *The Foundations of Mathematics and Other Logical Essays*. Harcourt Brace Jovanovich.

Ramsundar, B. and Zadeh, R. B. (2018). *TensorFlow for Deep Learning: From Linear Regression to Reinforcement Learning*. O'Reilly.

Rao, D. A. S. and Verweij, G. (2017). Sizing the prize: What's the real value of AI for your business and how can you capitalise? PwC.

Raphael, B. (1976). *The Thinking Computer: Mind Inside Matter*. W. H. Freeman.

Raphson, J. (1690). *Analysis aequationum universalis*. Apud Abelem Swalle, London.

Raschka, S. (2015). *Python Machine Learning*. Packt.

Rashevsky, N. (1936). Physico-mathematical aspects of excitation and conduction in nerves. In *Cold Springs Harbor Symposia on Quantitative Biology. IV: Excitation Phenomena*.

Rashevsky, N. (1938). *Mathematical Biophysics: Physico-Mathematical Foundations of Biology*. University of Chicago Press.

Rasmussen, C. E. and Williams, C. K. I. (2006). *Gaussian Processes for Machine Learning*. MIT Press.

Rassenti, S., Smith, V., and Bulfin, R. (1982). A combinatorial auction mechanism for airport time slot allocation. *Bell Journal of Economics*, *13*, 402–417.

Ratliff, N., Bagnell, J. A., and Zinkevich, M. (2006). Maximum margin planning. In *ICML-06*.

Ratliff, N., Zucker, M., Bagnell, J. A., and Srinivasa, S. (2009). CHOMP: Gradient optimization techniques for efficient motion planning. In *ICRA–09*.

Ratnaparkhi, A. (1996). A maximum entropy model for part-of-speech tagging. In *EMNLP-96*.

Ratner, D. and Warmuth, M. (1986). Finding a short-estsolutionforthe $n \times n$ extensionofthe15-puzzleis intractable. In *AAAI-86*.

Rauch, H. E., Tung, F., and Striebel, C. T. (1965). Maximum likelihood estimates of linear dynamic systems. *AIAA Journal*, *3*, 1445–1450.

Rayward-Smith, V., Osman, I., Reeves, C., and Smith, G. (Eds.). (1996). *Modern Heuristic Search Methods*. Wiley.

Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. (2018). Regularized evolution for image classifier architecture search. arXiv:1802.01548.

Rechenberg, I. (1965). Cybernetic solution path of an experimental problem. Library translation, Royal Aircraft Establishment.

Regin, J. (1994). A filtering algorithm for constraints of difference in CSPs. In *AAAI-94*.

Reid, D. B. (1979). An algorithmfor tracking multiple targets. *IEEE Trans. Automatic Control*, *24*, 843–854.

Reif, J. (1979). Complexity of the mover's problem and generalizations. In *FOCS–79*.

Reiter, R. (1980). A logic for default reasoning. *AIJ*, *13*, 81–132.

Reiter, R. (1991). The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In Lifschitz, V. (Ed.), *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy.* Academic Press.

Reiter, R. (2001). *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press.

Renner, G. and Ekart, A. (2003). Genetic algorithms in computer aided design. *ComputerAided Design*, *35*, 709–726.

Rényi, A. (1970). *Probability Theory.* Elsevier.

Resnick, P. and Varian, H. R. (1997). Recommender systems. *CACM*, *40*, 56–58.

Rezende, D. J., Mohamed, S., and Wierstra, D. (2014). Stochastic backpropagation and approximate inference in deep generative models. In *ICML-14*.

Riazanov, A. and Voronkov, A. (2002). The design and implementation of VAMPIRE. *AI Communications, 15*, 91–110.

Ribeiro, M. T., Singh, S., and Guestrin, C. (2016). Why should I trust you?: Explaining the predictions of any classifier. In *KDD-16*.

Richardson, M. and Domingos, P. (2006). Markov logic networks. *Machine Learning, 62*, 107–136.

Richter, S. and Helmert, M. (2009). Preferred operators and deferred evaluation in satisficing planning. In *ICAPS-09*.

Ridley, M. (2004). *Evolution*. Oxford Reader.

Riley, J. and Samuelson, W. (1981). Optimal auctions. *American Economic Review, 71*, 381–392.

Riley, P. (2019). Three pitfalls to avoid in machine learning. *Nature, 572*, 27–29.

Riloff, E. (1993). Automatically constructing a dictionary for information extraction tasks. In *AAAI-93*.

Ringgaard, M., Gupta, R., and Pereira, F. (2017). SLING: A framework for frame semantic parsing. arXiv:1710.07032.

Rintanen, J. (1999). Improvements to the evaluation of quantified Boolean formulae. In *IJCAI-99*.

Rintanen, J. (2007). Asymptotically optimal encodings of conformant planning in QBF. In *AAAI-07*.

Rintanen, J. (2012). Planning as satisfiability: Heuristics. *AIJ*, *193*, 45–86.

Rintanen, J. (2016). Computational complexity in automated planning and scheduling. In *ICAPS–16*.

Ripley, B. D. (1996). *Pattern Recognition and Neural Networks*. Cambridge University Press.

Rissanen, J. (1984). Universal coding, information, prediction, and estimation. *IEEE Transactions on Information Theory*, *IT-30*, 629–636.

Rissanen, J. (2007). *Information and Complexity in Statistical Modeling*. Springer.

Rivest, R. (1987). Learning decision lists. *Machine Learning*, *2*, 229–246.

Robbins, H. (1952). Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, *58*, 527–535.

Robbins, H. and Monro, S. (1951). A stochastic approximation method. *Annals of Mathematical Statistics*, *22*, 400–407.

Roberts, L. G. (1963). Machine perception of threedimensional solids. Technical report, MIT Lincoln Laboratory.

Robertson, N. and Seymour, P. D. (1986). Graph minors. II. Algorithmic aspects of tree-width. *J. Algorithms*, *7*, 309–322.

Robertson, S. E. and Sparck Jones, K. (1976). Relevance weighting of search terms. *J. American Society for Information Science*, *27*, 129–146.

Robins, J. (1986). Anew approach to causal inference in mortality studies with a sustained exposure period: Application to control of the healthy worker survivor effect. *Mathematical Modelling*, *7*, 1393–1512.

Robinson, A. and Voronkov, A. (Eds.). (2001). *Handbook of Automated Reasoning*. Elsevier.

Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *JACM*, *12*, 23–41.

Robinson, S. (2002). Computer scientists find unexpected depths in airfare search problem. *SIAM News*, *35*(6).

Roche, E. and Schabes, Y. (Eds.). (1997). *Finite–State Language Processing*. Bradford Books.

Rock, I. (1984). *Perception*. W. H. Freeman.

Rokicki, T., Kociemba, H., Davidson, M., and Dethridge, J. (2014). The diameter of the Rubik's Cube group is twenty. *SIAM Review*, *56*, 645–670.

Rolf, D. (2006). Improved bound for the PPSZ/Schoning-algorithm for 3–SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, *1*, 111–122.

Rolnick, D., Donti, P. L., Kaack, L. H., *et al.*(2019). Tackling climate change with machine learning. arXiv:1906.05433.

Rolnick, D. and Tegmark, M. (2018). The power of deeper networks for expressing natural functions. In *ICLR-18*.

Romanovskii, I. (1962). Reduction of a game with complete memory to a matrix game. *Soviet Mathematics*, *3*, 678–681.

Ros, G., Sellart, L., Materzynska, J., Vazquez, D., and Lopez, A. M. (2016). The SYNTHIA dataset: A large collection of synthetic images for semantic segmentation of urban scenes. In *CVPR-16.*

Rosenblatt, F. (1957). The perceptron: A perceiving and recognizing automaton. Report, Project PARA, Cornell Aeronautical Laboratory.

Rosenblatt, F. (1960). On the convergence of reinforcement procedures in simple perceptrons. Report, Cornell Aeronautical Laboratory.

Rosenblatt, F. (1962). *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan.

Rosenblatt, M. (1956). Remarks on some nonparametric estimates of a density function. *Annals of Mathematical Statistics*, *27*, 832–837.

Rosenblueth, A., Wiener, N., and Bigelow, J. (1943). Behavior, purpose, and teleology. *Philosophy of Science*, *10*, 18–24.

Ripley, B. D. (1996). *Pattern Recognition and Neural Networks*. Cambridge University Press.

Rosenschein, J. S. and Zlotkin, G. (1994). *Rules of Encounter.* MIT Press.

Rosenschein, S. J. (1985). Formal theories of knowledge in AI and robotics. *New Generation Computing*, *3*, 345–357.

Ross, G. (2012). Fisher and the millionaire: The statistician and the calculator. *Significance*, *9*, 46–48.

Ross, S. (2015). *A First Course in Probability* (9th edition). Pearson.

Ross, S., Gordon, G., and Bagnell, D. (2011). A reduction of imitation learning and structured prediction to no-regret online learning. In *AISTATS–*

11.

Rossi, F., van Beek, P., and Walsh, T. (2006). *Handbook of Constraint Processing.* Elsevier.

Roth, D. (1996). On the hardness of approximate reasoning. *AIJ*, *82*, 273–302.

Roussel, P. (1975). Prolog: Manual de reference et d'utilization. Tech. rep., Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille.

Rowat, P. F. (1979). *Representing the Spatial Experience and Solving Spatial Problems in a Simulated Robot Environment.* Ph.D. thesis, University of British Columbia.

Roweis, S. T. and Ghahramani, Z. (1999). A unifying review of linear Gaussian models. *Neural Computation*, *11*, 305–345.

Rowley, H., Baluja, S., and Kanade, T. (1998). Neural Network-based face detection. *PAMI*, *20*, 23–38.

Roy, N., Gordon, G., and Thrun, S. (2005). Finding approximate POMDP solutions through belief compression. *JAIR*, *23*, 1–40.

Rubin, D. (1974). Estimating causal effects of treatments in randomized and nonrandomized studies. *Journal of Educational Psychology*, *66*, 688–701.

Rubin, D. (1988). Using the SIR algorithm to simulate posterior distributions. In Bernardo, J. M., de Groot, M. H., Lindley, D. V., and Smith, A. F. M. (Eds.), *Bayesian Statistics 3.* Oxford University Press.

Rubinstein, A. (1982). Perfect equilibrium in a bargaining model. *Econometrica*, *50*, 97–109.

Rubinstein, A. (2003). Economics and psychology? The case of hyperbolic discounting. *International Economic Review*, *44*, 1207–1216.

Ruder, S. (2018). NLP's ImageNet moment has arrived. *The Gradient*, July 8.

Ruder, S., Peters, M. E., Swayamdipta, S., and Wolf, T. (2019). Transfer learning in natural language processing. In *COLING–19.*

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back–propagating errors. *Nature*, *323*, 533–536.

Rumelhart, D. E. and McClelland, J. L. (Eds.). (1986). *Parallel Distributed Processing.* MIT Press.

Rummery, G. A. and Niranjan, M. (1994). On-line Q-learning using connectionist systems. Tech. rep., Cambridge University Engineering Department.

Ruspini, E. H., Lowrance, J. D., and Strat, T. M. (1992). Understanding evidential reasoning. *IJAR*, *6*, 401–424.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. (2015). ImageNet large scale visual recognition challenge. *IJCV*, 115, 211–252.

Russell, J. G. B. (1990). Is screening for abdominal aortic aneurysm worthwhile? *Clinical Radiology*, *41*, 182–184.

Russell, S. J. (1985). The compleat guide to MRS. Report, Computer Science Department, Stanford University.

Russell, S. J. (1992). Efficient memory-bounded search methods. In *ECAI-92*.

Russell, S. J. (1998). Learning agents for uncertain environments. In *COLT-98*.

Russell, S. J. (1999). Expressive probability models in science. In *Proc. Second International Conference on Discovery Science*.

Russell, S. J. (2019). *Human Compatible*. Penguin.

Russell, S. J., Binder, J., Koller, D., and Kanazawa, K. (1995). Local learning in probabilistic networks with hidden variables. In *IJCAI-95*.

Russell, S. J. and Norvig, P. (2003). *Artificial Intelligence: A Modern Approach* (2nd edition). Prentice-Hall.

Russell, S. J. and Subramanian, D. (1995). Provably bounded-optimal agents. *JAIR*, *3*, 575–609.

Russell, S. J. and Wefald, E. H. (1989). On optimal game-tree search using rational meta-reasoning. In *IJCAI-89*.

Russell, S. J. and Wefald, E. H. (1991). *Do the Right Thing: Studies in Limited Rationality*. MIT Press.

Russell, S. J. and Wolfe, J. (2005). Efficient belief-state AND-OR search, with applications to Kriegspiel. In *IJCAI-05*.

Russell, S. J. and Zimdars, A. (2003). Q-decomposition of reinforcement learning agents. In *ICML-03*.

Rustagi, J. S. (1976). *Variational Methods in Statistics*. Academic Press.

Saad, F. and Mansinghka, V. K. (2017). A probabilistic programming approach to probabilistic data analysis. In *NeurIPS 29*.

Sabin, D. and Freuder, E. C. (1994). Contradicting conventional wisdom in constraint satisfaction. In *ECAI-94*.

Sabri, K. E. (2015). Automated verification of role-based access control policies constraints using Prover9. arXiv:1503.07645.

Sacerdoti, E. D. (1974). Planning in a hierarchy of abstraction spaces. *AIJ*, *5*, 115–135.

Sacerdoti, E. D. (1975). The nonlinear nature of plans. In *IJCAI-75*.

Sacerdoti, E. D. (1977). *A Structure for Plans and Behavior*. Elsevier.

Sadeghi, F. and Levine, S. (2016). CAD2RL: Real single-image flight without a single real image. arXiv:1611.04201.

Sadigh, D., Sastry, S., Seshia, S. A., and Dragan, A. D. (2016). Planning for autonomous cars that leverage effects on human actions. In *Proc. Robotics: Science and Systems*.

Sadler, M. and Regan, N. (2019). *Game Changer.* New in Chess.

Sadri, F. and Kowalski, R. (1995). Variants of the event calculus. In *ICLP-95*.

Sagae, K. and Lavie, A. (2006). A best-first probabilistic shift-reduce parser. In *COLING–06*.

Sahami, M., Hearst, M. A., and Saund, E. (1996). Applying the multiple cause mixture model to text categorization. In *ICML-96*.

Sahin, N. T., Pinker, S., Cash, S. S., Schomer, D., and Halgren, E. (2009). Sequential processing of lexical, grammatical, and phonological information within Broca's area. *Science*, *326*, 445–449.

Sakuta, M. and Iida, H. (2002). AND/OR-tree search for solving problems with uncertainty: A case study using screen-shogi problems. *Trans. Inf. Proc. Society of Japan*, *43*, 1–10.

Salomaa, A. (1969). Probabilistic and weighted grammars. *Information and Control*, *15*, 529–544.

Samadi, M., Feiner, A., and Schaeffer, J. (2008). Learning from multiple heuristics. In *AAAI-08.*

Samet, H. (2006). *Foundations of Multidimensional and Metric Data Structures.* Morgan Kaufmann.

Sammut, C., Hurst, S., Kedzier, D., and Michie, D. (1992). Learning to fly. In *ICML-92.*

Samuel, A. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, *3*, 210–229.

Samuel, A. (1967). Some studies in machine learning using the game of checkers II-Recent progress. *IBM Journal of Research and Development*, *11*, 601–617.

Sanchez–Lengeling, B., Wei, J. N., Lee, B. K., Gerkin, R. C., Aspuru–Guzik, A., and Wiltschko, A. B. (2019). Machine learning for scent: Learning generalizable perceptual representations of small molecules. arXiv:1910.10685.

Sandholm, T. (1999). Distributed rational decision making. In Weiß, G. (Ed.), *Multiagent Systems.* MIT Press.

Sandholm, T., Larson, K., Andersson, M., Shehory, O., and Tohme, F. (1999). Coalition structure generation with worst case guarantees. *AIJ, 111,* 209–238.

Sandholm, T. (1993). An implementation of the contract net protocol based on marginal cost calculations. In *AAAI-93*.

Sang, T., Beame, P., and Kautz, H. (2005). Performing Bayesian inference by weighted model counting. In *AAAI-05.*

Sapir, E. (1921). *Language: An Introduction to the Study of Speech*. Harcourt Brace Jovanovich.

Sarawagi, S. (2007). Information extraction. *Foundations and Trends in Databases, 1,* 261–377.

Sargent, T. J. (1978). Estimation of dynamic labor demand schedules under rational expectations. *J. Political Economy, 86,* 1009–1044.

Sartre, J.–P. (1960). *Critique de la Raison dialectique.* Editions Gallimard.

Satia, J. K. and Lave, R. E. (1973). Markovian decision processes with probabilistic observation of states. *Management Science, 20,* 1–13.

Sato, T. and Kameya, Y. (1997). PRISM: A symbolic– statistical modeling language. In *IJCAI-97.*

Saul, L. K., Jaakkola, T., and Jordan, M. I. (1996). Mean field theory for sigmoid belief networks. *JAIR*, 4, 61–76.

Saunders, W., Sastry, G., Stuhlmüller, A., and Evans, O. (2018). Trial without error: Towards safe reinforcement learning via human intervention. In *AAMAS–18*.

Savage, L. J. (1954). *The Foundations of Statistics*. Wiley.

Savva, M., Kadian, A., Maksymets, O., Zhao, Y., Wijmans, E., Jain, B., Straub, J., Liu, J., Koltun, V., Malik, J., Parikh, D., and Batra, D. (2019). Habitat: A platform for embodied AI research. arXiv:1904.01201.

Sayre, K. (1993). Three more flaws in the computational model. Paper presented at the APA (Central Division) Annual Conference, Chicago, Illinois.

Schaeffer, J. (2008). *OneJumpAhead: Computer Perfection at Checkers.* Springer-Verlag.

Schaeffer, J., Burch, N., Bjornsson, Y., Kishimoto, A., Muller, M., Lake, R., Lu, P., and Sutphen, S. (2007). Checkers is solved. *Science*, *317*, 1518–1522.

Schank, R. C. and Abelson, R. P. (1977). *Scripts, Plans, Goals, and Understanding*. Lawrence Erlbaum.

Schank, R. C. and Riesbeck, C. (1981). *Inside Computer Understanding: Five Programs Plus Miniatures*. Lawrence Erlbaum.

Schapire, R. E. and Singer, Y. (2000). Boostexter: A boosting–based system for text categorization. *Machine Learning*, *39*, 135–168.

Schapire, R. E. (1990). The strength of weak learn-ability. *Machine Learning*, *5*, 197–227.

Schapire, R. E. (2003). The boosting approach to machine learning: An overview. In Denison, D. D., Hansen, M. H., Holmes, C., Mallick, B., and Yu, B. (Eds.), *Nonlinear Estimation and Classification.* Springer.

Scharre, P. (2018). *Army of None.* W. W. Norton.

Schmid, C. and Mohr, R. (1996). Combining grey-value invariants with local constraints for object recognition. In *CVPR-96.*

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks, 61*, 85–117.

Schofield, M. and Thielscher, M. (2015). Lifting model sampling for general game playing to incomplete-information models. In *AAAI-15*.

Schölkopf, B. and Smola, A. J. (2002). *Learning with Kernels.* MIT Press.

Schöning, T. (1999). A probabilistic algorithm for k– SAT and constraint satisfaction problems. In *FOCS– 99*.

Schoppers, M. J. (1989). In defense of reaction plans as caches. *AIMag, 10*, 51–60.

Schraudolph, N. N., Dayan, P, and Sejnowski, T. (1994). Temporal difference learning of position evaluation in the game of Go. In *NeurIPS 6.*

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., Lillicrap, T., and Silver, D. (2019). Mastering Atari, Go, chess and shogi by planning with a learned model. arXiv:1911.08265.

Schröder, E. (1877). *Der Operationskreis des Logikkalkuls.* B. G. Teubner, Leipzig.

Schulman, J., Ho, J., Lee, A. X., Awwal, I., Bradlow, H., and Abbeel, P. (2013). Finding locally optimal, collision-free trajectories with sequential convex optimization. In *Proc. Robotics: Science and Systems.*

Schulman, J., Levine, S., Abbeel, P., Jordan, M. I., and Moritz, P. (2015a). Trust region policy optimization. In *ICML-15*.

Schulman, J., Levine, S., Moritz, P., Jordan, M., and Abbeel, P. (2015b). Trust region policy optimization. In *ICML-15*.

Schultz, W., Dayan, P, and Montague, P. R. (1997). A neural substrate of prediction and reward. *Science*, 275, 1593.

Schulz, D., Burgard, W., Fox, D., and Cremers, A. B. (2003). People tracking with mobile robots using sample-based joint probabilistic data association filters. *Int. J. Robotics Research*, *22*, 99–116.

Schulz, S. (2004). System Description: E 0.81. In *Proc. International Joint Conference on Automated Reasoning*, Vol. 3097 of *LNAI.*

Schulz, S. (2013). System description: E 1.8. In *Proc. Int. Conf. on Logic for Programming Artificial Intelligence and Reasoning*.

Schutze, H. (1995). *Ambiguity in Language Learning: Computational and Cognitive Models.* Ph.D. thesis, Stanford University. Also published by CSLI Press, 1997.

Schwartz, J. T., Scharir, M., and Hopcroft, J. (1987). *Planning, Geometry and Complexity of Robot Motion*. Ablex.

Schwartz, S. P (Ed.). (1977). *Naming, Necessity, and Natural Kinds*. Cornell University Press.

Scott, D. and Krauss, P. (1966). Assigning probabilities to logical formulas. In Hintikka, J. and Suppes, P. (Eds.), *Aspects of Inductive Logic.* North-Holland.

Searle, J. R. (1980). Minds, brains, and programs. *BBS, 3*, 417–457.

Searle, J. R. (1990). Is the brain's mind a computer program? *Scientific American*, *262*, 26–31.

Searle, J. R. (1992). *The Rediscovery of the Mind.* MIT Press.

Sedgewick, R. and Wayne, K. (2011). *Algorithms.* Addison-Wesley.

Sefidgar, Y. S., Agarwal, P., and Cakmak, M. (2017). Situated tangible robot programming. In *HRI-17*.

Segaran, T. (2007). *Programming Collective Intelligence: Building Smart Web 2.0 Applications.* O'Reilly.

Seipp, J. and Roger, G. (2018). Fast downward stone soup 2018. IPC 2018 Classical Track.

Seipp, J., Sievers, S., Helmert, M., and Hutter, F. (2015). Automatic configuration of sequential planning portfolios. In *AAAI-15*.

Selman, B., Kautz, H., and Cohen, B. (1996). Local search strategies for satisfiability testing. In Johnson, D. S. and Trick, M. A. (Eds.), *Cliques, Coloring, and Satisfiability.* American Mathematical Society.

Selman, B. and Levesque, H. J. (1993). The complexity of path–based defeasible inheritance. *AIJ*, *62*, 303–339.

Selman, B., Levesque, H. J., and Mitchell, D. (1992). A new method for solving hard satisfiability problems. In *AAAI-92*.

Seni, G. and Elder, J. F. (2010). Ensemble methods in data mining: Improving accuracy through combining predictions. *Synthesis Lectures on Data Mining and Knowledge Discovery*, *2*, 1–126.

Seo, M., Kembhavi, A., Farhadi, A., and Hajishirzi, H. (2017). Bidirectional attention flow for machine comprehension. In *ICLR-17*.

Shachter, R. D. (1986). Evaluating influence diagrams. *Operations Research*, *34*, 871–882.

Shachter, R. D. (1998). Bayes-ball: The rational pastime (for determining irrelevance and requisite information in belief networks and influence diagrams). In *UAI-98*.

Shachter, R. D., D'Ambrosio, B., and Del Favero, B. A. (1990). Symbolic probabilistic inference in belief networks. In *AAAI-90*.

Shachter, R. D. and Kenley, C. R. (1989). Gaussian influence diagrams. *Management Science*, *35*, 527–550.

Shachter, R. D. and Peot, M. (1989). Simulation approaches to general probabilistic inference on belief networks. In *UAI-98*.

Shafer, G. (1976). *A Mathematical Theory of Evidence*. Princeton University Press.

Shanahan, M. (1997). *Solving the Frame Problem*. MIT Press.

Shanahan, M. (1999). The event calculus explained. In Wooldridge, M. J. and Veloso, M. (Eds.), *Artificial Intelligence Today*. Springer-Verlag.

Shanahan, M. (2015). *The Technological Singularity*. MIT Press.

Shani, G., Pineau, J., and Kaplow, R. (2013). Asurvey of point-based POMDP solvers. *Autonomous Agents and Multi-Agent Systems*, *27*, 1–51.

Shankar, N. (1986). *Proof-Checking Metamathematics*. Ph.D. thesis, Computer Science Department, University of Texas at Austin.

Shannon, C. E. and Weaver, W. (1949). *The Mathematical Theory of Communication*. University of Illinois Press.

Shannon, C. E. (1950). Programming a computer for playing chess. *Philosophical Magazine*, *41*, 256–275.

Shapley, S. (1953b). Stochastic games. *PNAS*, *39*, 1095–1100.

Sharan, R. V. and Moir, T. J. (2016). An overview of applications and advancements in automatic sound recognition. *Neurocomputing*, *200*, 22–34.

Shatkay, H. and Kaelbling, L. P. (1997). Learning topological maps with weak local odometric information. In *IJCAI-97*.

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q. V., Hinton, G. E., and Dean, J. (2017). Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. arXiv:1701.06538.

Shelley, M. (1818). *Frankenstein: Or, the Modern Prometheus*. Pickering and Chatto.

Sheppard, B. (2002). World-championship-caliber scrabble. *AIJ*, *134*, 241–275.

Shi, J. and Malik, J. (2000). Normalized cuts and image segmentation. *PAMI*, *22*, 888–905.

Shieber, S. (1994). Lessons from a restricted Turing test. *CACM*, *37*, 70–78.

Shieber, S. (Ed.). (2004). *The Turing Test*. MIT Press.

Shimony, S. E. (1994). Finding MAPs for belief networks is NP-hard. *AIJ*, *68*, 399–410.

Shoham, Y. (1993). Agent-oriented programming. *AIJ*, *60*, 51–92.

Shoham, Y. (1994). *Artificial Intelligence Techniques in Prolog*. Morgan Kaufmann.

Shoham, Y. and Leyton-Brown, K. (2009). *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge Univ. Press.

Shoham, Y., Powers, R., and Grenager, T. (2004). If multi-agent learning is the answer, what is the question? In *Proc. AAAI Fall Symposium on Artificial Multi-Agent Learning*.

Shortliffe, E. H. (1976). *Computer-Based Medical Consultations: MYCIN*. Elsevier.

Siciliano, B. and Khatib, O. (Eds.). (2016). *Springer Handbook of Robotics* (2nd edition). Springer-Verlag.

Sigaud, O. and Buffet, O. (2010). *Markov Decision Processes in Artificial Intelligence*. Wiley.

Sigmund, K. (2017). *Exact Thinking in Demented Times*. Basic Books.

Silberstein, M., Weissbrod, O., Otten, L., Tzemach, A., Anisenia, A., Shtark, O., Tuberg, D., Galfrin, E., Gannon, I., Shalata, A., Borochowitz, Z.

U., Dechter, R., Thompson, E., and Geiger, D. (2013). A system for exact and approximate genetic linkage analysis of SNP data in large pedigrees. *Bioinformatics*, *29*, 197–205.

Silva, R., Melo, F. S., and Veloso, M. (2015). Towards table tennis with a quadrotor autonomous learning robot and onboard vision. In *IROS-15*.

Silver, D. and Veness, J. (2011). Monte-Carlo planning in large POMDPs. In *NeurIPS 23*.

Silver, D., Huang, A., Maddison, C. J., Guez, A., and Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, *529*, 484–489.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., *etal.* (2018). Ageneralreinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, *362*, 1140–1144.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., and Hassabis, D. (2017). Mastering the game of Go without human knowledge. *Nature*, *550*, 354–359.

Silverman, B. W. (1986). *Density Estimation for Statistics and Data Analysis*. Chapman and Hall.

Shapley, L. S. (1953a). A value for n–person games. In Kuhn, H. W. and Tucker, A. W. (Eds.), *Contributions to the Theory of Games.* Princeton University Press.

Silverstein, C., Henzinger, M., Marais, H., and Moricz, M. (1998). Analysis of a very large AltaVista query log. Tech. rep., Digital Systems Research Center.

Simmons, R. and Koenig, S. (1995). Probabilistic robot navigation in partially observable environments. In *IJCAI-95*.

Simon, D. (2006). *Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches*. Wiley.

Simon, H. A. (1947). *Administrative Behavior*. Macmillan.

Simon, H. A. (1963). Experiments with a heuristic compiler. *JACM*, *10*, 493–506.

Simon, H. A. and Newell, A. (1958). Heuristic problem solving: The next advance in operations research. *Operations Research*, *6*, 1–10.

Simon, J. C. and Dubois, O. (1989). Number of solutions to satisfiability instances—Applications to knowledge bases. *AIJ*, *3*, 53–65.

Simonis, H. (2005). Sudoku as a constraint problem. In *CP-05 Workshop on Modeling and Reformulating Constraint Satisfaction Problems*.

Singer, P. W. (2009). *Wired for War*. Penguin Press.

Singh, P., Lin, T., Mueller, E. T., Lim, G., Perkins, T., and Zhu, W. L. (2002). Open mind common sense: Knowledge acquisition from the general public. In *Proc. First International Conference on Ontologies, Databases, and Applications of Semantics for Large Scale Information Systems*.

Sisbot, E. A., Marin-Urias, L. F., Alami, R., and Simeon, T. (2007). A human aware mobile robot motion planner. *IEEE Transactions on Robotics*, *23*, 874–883.

Siskind, J. M. and Pearlmutter, B. A. (2016). Efficient implementation of a higher-order language with built-in AD. arXiv:1611.03416.

Sistla, A. P. and Godefroid, P. (2004). Symmetry and reduced symmetry in model checking. *ACM Trans. Program. Lang. Syst.*, *26*, 702–734.

Sittler, R. W. (1964). An optimal data association problem in surveillance theory. *IEEE Transactions on Military Electronics*, *8*, 125–139.

Skolem, T. (1920). Logisch-kombinatorische Untersuchungen uber die Erfullbarkeit oder Beweisbarkeit mathematischer Satze nebst einem Theoreme uber die dichte Mengen. *Videnskapsselskapets skrifter, I. Matematisk-naturvidenskabelig klasse*, *4*, 1–36.

Skolem, T. (1928). Uber die mathematische Logik. *Norsk matematisk tidsskrift*, *10*, 125–142.

Slagle, J. R. (1963). A heuristic program that solves symbolic integration problems in freshman calculus. *JACM*, *10*.

Slate, D. J. and Atkin, L. R. (1911). CHESS 4.5— Northwestern University chess program. In Frey, P. W. (Ed.), *Chess Skill in Man and Machine*. SpringerVerlag.

Slater, E. (1950). Statistics for the chess computer and the factor of mobility. In *Symposium on Information Theory*. Ministry of Supply.

Slocum, J. and Sonneveld, D. (2006). *The 15 Puzzle.* Slocum Puzzle Foundation.

Smallwood, R. D. and Sondik, E. J. (1913). The optimal control of partially observable Markov processes over a finite horizon. *Operations Research*, *21*, 1071–1088.

Smith, B. (2004). Ontology. In Floridi, L. (Ed.), *The Blackwell Guide to the Philosophy of Computing and Information*. Wiley-Blackwell.

Smith, B., Ashburner, M., Rosse, C., *et al.* (2001). The OBO Foundry: Coordinated evolution of ontologies to support biomedical data integration. *Nature Biotechnology*, *25*, 1251–1255.

Smith, D. E., Genesereth, M. R., and Ginsberg, M. L. (1986). Controlling recursive inference. *AIJ*, *30*, 343–389.

Smith, D. A. and Eisner, J. (2008). Dependency parsing by belief propagation. In *EMNLP-08.*

Smith, D. E. and Weld, D. S. (1998). Conformant Graphplan. In *AAAI-98.*

Smith, J. Q. (1988). *Decision Analysis.* Chapman and Hall.

Smith, J. E. and Winkler, R. L. (2006). The optimizer's curse: Skepticism and postdecision surprise in decision analysis. *Management Science*, *52*, 311–322.

Smith, J. M. (1982). *Evolution and the Theory of Games.* Cambridge University Press.

Smith, J. M. and Szathmary, E. (1999). *The Origins of Life: From the Birth of Life to the Origin of Language.* Oxford University Press.

Smith, M. K., Welty, C., and McGuinness, D. (2004). OWL web ontology language guide. Tech. rep., W3C.

Smith, R. G. (1980). *A Framework for Distributed Problem Solving.* UMI Research Press.

Smith, R. C. and Cheeseman, P. (1986). On the representation and estimation of spatial uncertainty. *Int. J. Robotics Research, 5*, 56–68.

Smith, S. J. J., Nau, D. S., and Throop, T. A. (1998). Success in spades: Using AI planning techniques to win the world championship of computer bridge. In *AAAI-98*.

Smith, W. E. (1956). Various optimizers for singlestage production. *Naval Research Logistics Quarterly, 3*, 59–66.

Smolensky, P (1988). On the proper treatment of con– nectionism. *BBS*, 2, 1–74.

Smolensky, P. and Prince, A. (1993). Optimality theory: Constraint interaction in generative grammar. Tech. rep., Department of Computer Science, University of Colorado at Boulder.

Smullyan, R. M. (1995). *First-Order Logic*. Dover.

Smyth, P., Heckerman, D., and Jordan, M. I. (1997). Probabilistic independence networks for hidden Markov probability models. *Neural Computation, 9*, 227–269.

Snoek, J., Larochelle, H., and Adams, R. P. (2013). Practical Bayesian optimization of machine learning algorithms. In *NeurIPS 25*.

Solomonoff, R. J. (1964). A formal theory of inductive inference. *Information and Control, 7*, 1–22, 224–254.

Solomonoff, R. J. (2009). Algorithmic probability–theory and applications. In Emmert-Streib, F. and Dehmer, M. (Eds.), *Information Theory and Statitical Learning*. Springer.

Sondik, E. J. (1971). *The Optimal Control of Partially Observable Markov Decision Processes*. Ph.D. thesis, Stanford University.

Sosic, R. and Gu, J. (1994). Efficient local search with conflict minimization: A case study of the n-queens problem. *IEEE Transactions on Knowledge and Data Engineering*, *6*, 661–668.

Sowa, J. (1999). *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Blackwell.

Spaan, M. T. J. and Vlassis, N. (2005). Perseus: Randomized point-based value iteration for POMDPs. *JAIR*, *24*, 195–220.

Sparrow, R. (2004). The Turing triage test. *Ethics and Information Technology*, *6*, 203–213.

Spiegelhalter, D. J., Dawid, A. P., Lauritzen, S., and Cowell, R. (1993). Bayesian analysis in expert systems. *Statistical Science*, *8*, 219–282.

Spirtes, P., Glymour, C., and Scheines, R. (1993). *Causation, Prediction, and Search*. Springer-Verlag.

Spitkovsky, V. I., Alshawi, H., and Jurafsky, D. (2010a). From baby steps to leapfrog: How less is more in unsupervised dependency parsing. In *NAACL HLT*.

Spitkovsky, V. I., Jurafsky, D., and Alshawi, H. (2010b). Profiting from mark-up: Hyper-text annotations for guided parsing. In *ACL–10*.

Srivas, M. and Bickford, M. (1990). Formal verification of a pipelined microprocessor. *IEEE Software*, *7*, 52–64.

Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014a). Dropout: A simple way to prevent neural networks from overfitting. *JMLR*, 15, 1929–1958.

Srivastava, S., Russell, S. J., and Ruan, P. (2014b). first-order open-universe POMDPs. In *UAI-14*.

Staab, S. (2004). *Handbook on Ontologies*. Springer.

Stallman, R. M. and Sussman, G. J. (1977). Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *AIJ*, *9*, 135–196.

Stanfill, C. and Waltz, D. (1986). Toward memory-based reasoning. *CACM*, *29*, 1213–1228.

Stanislawska, K., Krawiec, K., and Vihma, T. (2015). Genetic programming for estimation of heat flux between the atmosphere and sea ice in polar regions. In *GECCO-15*.

Stefik, M. (1995). *Introduction to Knowledge Systems*. Morgan Kaufmann.

Steiner, D. F., MacDonald, R., Liu, Y., Truszkowski, P., Hipp, J. D., Gammage, C., Thng, F., Peng, L., and Stumpe, M. C. (2018). Impact of deep learning assistance on the histopathologic review of lymph nodes for metastatic breast cancer. *Am. J. Surgical Pathology*, *42*, 1636–1646.

Steinruecken, C., Smith, E., Janz, D., Lloyd, J., and Ghahramani, Z. (2019). The Automatic Statistician. In Hutter, F., Kotthoff, L., and Vanschoren, J. (Eds.), *Automated Machine Learning*. Springer.

Stergiou, K. and Walsh, T. (1999). The difference alldifference makes. In *IJCAI-99*.

Stickel, M. E. (1992). A Prolog technology theorem prover: a new exposition and implementation in Prolog. *Theoretical Computer Science*, *104*, 109–128.

Stiller, L. (1992). KQNKRR. *J. International Computer Chess Association*, *15*, 16–18.

Stiller, L. (1996). Multilinear algebra and chess endgames. In Nowakowski, R. J. (Ed.), *Games of No Chance, MSRI, 29, 1996.* Mathematical Sciences Research Institute.

Stockman, G. (1979). A minimax algorithm better than alpha-beta? AIJ, *12*, 179–196.

Stoffel, K., Taylor, M., and Hendler, J. (1997). Efficient management of very large ontologies. In *AAAI*- 97.

Stone, M. (1974). Cross-validatory choice and assessment of statistical predictions. *J. Royal Statistical Society*, *36*, 111–133.

Stone, P. (2000). *Layered Learning in Multi-Agent Systems: A Winning Approach to Robotic Soccer.* MIT Press.

Stone, P. (2003). Multiagent competitions and research: Lessons from RoboCup and TAC. In Lima, P. U. and Rojas, P. (Eds.), *RoboCuP-2002: Robot Soccer World Cup VI.* Springer Verlag.

Stone, P. (2016). What's hot at RoboCup. In *AAAI-16.*

Stone, P., Brooks, R. A., Brynjolfsson, E., Calo, R., Etzioni, O., Hager, G., Hirschberg, J., Kalyanakrish- nan, S., Kamar, E., Kraus, S., *et al.* (2016). Artificial intelligence and life in 2030. Tech. rep., Stanford University One Hundred Year Study on Artificial Intelligence: Report of the 2015–2016 Study Panel.

Stone, P, Kaminka, G., and Rosenschein, J. S. (2009). Leading a best-response teammate in an ad hoc team. In *AAMAS Workshop in Agent Mediated Electronic Commerce.*

Stone, P., Sutton, R. S., and Kuhlmann, G. (2005). Reinforcement learning for robocup soccer keepaway. *Adaptive Behavior, 13*, 165–188.

Storvik, G. (2002). Particle filters for state-space models with the presence of unknown static parameters. *IEEE Transactions on Signal Processing, 50*, 281–289.

Strachey, C. (1952). Logical or non-mathematical programmes. In *Proc. 1952 ACM National Meeting.*

Stratonovich, R. L. (1959). Optimum nonlinear systems which bring about a separation of a signal with constant parameters from noise. *Radiofizika, 2*, 892–901.

Stratonovich, R. L. (1965). On value of information. *Izvestiya of USSR Academy of Sciences, Technical Cybernetics*, *5*, 3–12.

Sturtevant, N. R. and Bulitko, V. (2016). Scrubbing during learning in real-time heuristic search. *JAIR, 57*, 307–343.

Subramanian, D. and Wang, E. (1994). constraint-based kinematic synthesis. In *Proc. International Conference on Qualitative Reasoning*.

Suk, H.-I., Sin, B.-K., and Lee, S.-W. (2010). Hand gesture recognition based on dynamic Bayesian network framework. *Pattern Recognition, 43*, 3059–3072.

Sun, Y., Wang, S., Li, Y., Feng, S., Tian, H., Wu, H., and Wang, H. (2019). ERNIE 2.0: A continual Pre-training framework for language understanding. arXiv:1907.12412.

Sussman, G. J. (1975). *A Computer Model of Skill Acquisition*. Elsevier.

Sutcliffe, G. (2016). The CADE ATP system competition - CASC. *AIMag*, *37*, 99–101.

Sutcliffe, G. and Suttner, C. (1998). The TPTP ProblemLibrary: CNF Releasev1.2.1. *JAR*, *21*, 177–203.

Sutcliffe, G., Schulz, S., Claessen, K., and Gelder, A. V. (2006). Using the TPTP language for writing derivations and finite interpretations. In *Proc. International Joint Conference on Automated Reasoning*.

Sutherland, I. (1963). Sketchpad: A man-machine graphical communication system. In *Proc. Spring Joint Computer Conference*.

Sutskever, I., Vinyals, O., and Le, Q. V. (2015). Sequence to sequence learning with neural networks. In *NeurIPS 27*.

Sutton, C. and McCallum, A. (2007). An introduction to conditional random fields for relational learning. In Getoor, L. and Taskar, B. (Eds.), *Introduction to Statistical Relational Learning*. MIT Press.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, *3*, 944.

Sutton, R. S., McAllester, D. A., Singh, S., and Man-sour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. In *NeurIPS 12*.

Sutton, R. S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *ICML-90*.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction* (2nd edition). MIT Press.

Swade, D. (2000). *Difference Engine: Charles Bab-bage And The Quest To Build The First Computer*. Diane Publishing Co.

Sweeney, L. (2000). Simple demographics often iden-tify people uniquely. *Health (San Francisco), 671*, 1–34.

Sweeney, L. (2002a). Achieving k-anonymity privacy protection using generalization and suppression. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems, 10*, 571–588.

Sweeney, L. (2002b). k-anonymity: Amodel for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems, 10*, 557–570.

Swerling, P. (1959). First order error propagation in a stagewise smoothing procedure for satellite observations. *J. Astronautical Sciences*, *6*, 46–52.

Swift, T. and Warren, D. S. (1994). Analysis of SLG- WAM evaluation of definite programs. In *Logic Programming: Proc. 1994 International Symposium*.

Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., ERhan, D., Goodfellow, I., and Fergus, R. (2013). Intriguing properties of neural networks. arXiv:1312.6199.

Szeliski, R. (2011). *Computer Vision: Algorithms and Applications*. Springer-Verlag.

Szepesvari, C. (2010). Algorithms for reinforcement learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, *4*, 1–103.

Tadepalli, P., Givan, R., and Driessens, K. (2004). Relational reinforcement learning: An overview. In *ICML-04*.

Tait, P G. (1880). Note on the theory of the “15 puzzle”. *Proc. Royal Society of Edinburgh, 10,* 664–665.

Tamaki, H. and Sato, T. (1986). OLD resolution with tabulation. In *ICLP-86.*

Tan, P, Steinbach, M., Karpatne, A., and Kumar, V. (2019). *Introduction to Data Mining* (2nd edition). Pearson.

Tang, E. (2018). A quantum–inspired classical algorithm for recommendation systems. arXiv:1807.04271.

Tarski, A. (1935). Die Wahrheitsbegriffinden formalisierten Sprachen. *Studia Philosophica*, *1*, 261–405.

Tarski, A. (1941). *Introduction to Logic and to the Methodology of Deductive Sciences*. Dover.

Tarski, A. (1956). *Logic, Semantics, Metamathematics: Papers from 1923 to 1938*. Oxford University Press.

Tash, J. K. and Russell, S. J. (1994). Control strategies for a stochastic planner. In *AAAI-94*.

Tassa, Y., Doron, Y., Muldal, A., Erez, T., Li, Y., Casas, D. d. L., Budden, D., Abdolmaleki, A., Merel, J., Lefrancq, A., *etal.* (2018). Deepmind control suite. arXiv:1801.00690.

Tate, A. (1975a). Interacting goals and their use. In *IJCAI-75.*

Tate, A. (1975b). *Using Goal Structure to Direct Search in a Problem Solver*. Ph.D. thesis, University of Edinburgh.

Tate, A. (1977). Generating project networks. In *IJCAI-77*.

Tate, A. and Whiter, A. M. (1984). Planning with multiple resource constraints and an application to a naval planning problem. In *Proc. First Conference on AIAP-plications*.

Tatman, J. A. and Shachter, R. D. (1990). Dynamic programming and influence diagrams. *IEEE Transactions on Systems, Man and Cybernetics*, *20*, 365–379.

Tattersall, C. (1911). *A Thousand End–Games: A Collection of Chess Positions That Can be Won or Drawn by the Best Play*. British Chess Magazine.

Taylor, A. D. and Zwicker, W. S. (1999). *Simple Games: Desirability Relations, Trading, Pseudoweightings.* Princeton University Press.

Taylor, G., Stensrud, B., Eitelman, S., and Dunham, B. (2007). Towards automating airspace management. In *Proc. Computational Intelligence for Security and Defense Applications (CISDA) Conference*.

Taylor, P. (2009). *Text-to-Speech Synthesis.* Cambridge University Press.

Tedrake, R., Zhang, T. W., and Seung, H. S. (2004). Stochastic policy gradient reinforcement learning on a simple 3D biped. In *IROS–04.*

Tellex, S., Kollar, T., Dickerson, S., Walter, M. R., Banerjee, A., Teller, S., and Roy, N. (2011). Understanding natural language commands for robotic navigation and mobile manipulation. In *AAAI-11*.

Tenenbaum, J. B., Griffiths, T. L., and Niyogi, S. (2007). Intuitive theories as grammars for causal inference. In Gopnik, A. and Schulz, L. (Eds.), *Causal Learning: Psychology, Philosophy, and Computation*. Oxford University Press.

Tesauro, G. (1990). Neurogammon: A neural-network backgammonprogram. In *IJCNN–90.*

Tesauro, G. (1992). Practical issues in temporal difference learning. *Machine Learning, 8*, 257–277.

Tesauro, G. (1995). Temporal difference learning and TD-Gammon. CACM, 38, 58–68.

Tesauro, G. and Galperin, G. R. (1997). On-line policy improvement using Monte-Carlo search. In *NeurIPS 9*.

Tetlock, P. E. (2017). *Expert Political Judgment: How Good Is It? How Can We Know?* Princeton University Press.

Teyssier, M. and Koller, D. (2005). Ordering-based search: A simple and effective algorithm for learning Bayesian networks. In *UAI-05.*

Thaler, R. (1992). *The Winner's Curse: Paradoxes and Anomalies of Economic Life.* Princeton University Press.

Thaler, R. and Sunstein, C. (2009). *Nudge: Improving Decisions About Health, Wealth, and Happiness*. Penguin.

Thayer, J. T., Dionne, A., and Ruml, W. (2011). Learning inadmissible heuristics during search. In *ICAPS-11*.

Theocharous, G., Murphy, K., and Kaelbling, L. P. (2004). Representing hierarchical POMDPs as DBNs for multi-scale robot localization. In *ICRA-*

04.

Thiele, T. (1880). Om anvendelse af mindste kvadraters methode i nogle tilfælde, hvor en komplikation af visse slags uensartede tilfældige fejlkilder giver fejlene en ‘systematisk’ karakter. *Vidensk. Selsk. Skr. 5. Rk., naturvid. og mat. Afd., 12*, 381–408.

Thielscher, M. (1999). From situation calculus to fluent calculus: State update axioms as a solution to the inferential frame problem. *AIJ*, *111*, 277–299.

Thomas, P. S., daSilva, B. C., Barto, A. G., and Brunskill, E. (2017). On ensuring that intelligent machines are well-behaved. arXiv:1708.05448.

Thomaz, A., Hoffman, G., Cakmak, M., *etal.* (2016). Computational human-robot interaction. *Foundations and Trends in Robotics*, *4*, 105–223.

Thompson, K. (1986). Retrograde analysis of certain endgames. *J. International Computer Chess Association*, *9*, 131–139.

Thompson, K. (1996). 6-piece endgames. *J. International Computer Chess Association*, *19*, 215–226.

Thompson, W. R. (1933). On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, *25*, 285–294.

Thorndike, E. (1911). *Animal Intelligence*. Macmillan.

Thornton, C., Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2013). Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *KDD-13*.

Thrun, S., Burgard, W., and Fox, D. (2005). *Probabilistic Robotics*. MIT Press.

Thrun, S., Fox, D., and Burgard, W. (1998). A probabilistic approach to concurrent mapping and localization for mobile robots. *Machine Learning, 31*, 29–53.

Thrun, S. (2006). Stanley, the robot that won the DARPA Grand Challenge. *J. Field Robotics, 23*, 661–692.

Thrun, S. and Pratt, L. (2012). *Learning to Learn*. Springer.

Thurstone, L. L. (1927). A law of comparative judgment. *PsychologicalReview, 34*, 273–286.

Tian, J., Paz, A., and Pearl, J. (1998). Finding a minimal *d*-separator. Tech. rep., UCLA Department of Computer Science.

Tikhonov, A. N. (1963). Solution of incorrectly formulated problems and the regularization method. *Soviet Math. Dokl.*, 5, 1035–1038.

Tipping, M. E. and Bishop, C. M. (1999). Probabilistic principal component analysis. *J. Royal Statistical Society, 61*, 611–622.

Titterington, D. M., Smith, A. F. M., and Makov, U. E. (1985). *Statistical Analysis of Finite Mixture Distributions*. Wiley.

Toma, P. (1977). SYSTRAN as a multilingual machine translation system. In *Proc. Third European Congress on Information Systems and Networks: Overcoming the Language Barrier*.

Tomasi, C. and Kanade, T. (1992). Shape and motion from image streams under orthography: A factorization method. *IJCV, 9*, 137–154.

Topol, E. (2019). *Deep Medicine: How Artificial Intelligence Can Make Healthcare Human Again.* Basic Books.

Torralba, A., Fergus, R., and Weiss, Y. (2008). Small codes and large image databases for recognition. In *CVPR*.

Torralba, A., Linares López, C., and Borrajo, D. (2016). Abstraction heuristics for symbolic bidirectional search. In *IJCAI-16.*

Tramér, F., Zhang, F., Juels, A., Reiter, M. K., and Ristenpart, T. (2016). Stealing machine learning models via prediction APIs. In *USENIX Security Symposium.*

Tran, D., Hoffman, M., Saurous, R. A., Brevdo, E., Murphy, K., and Blei, D. M. (2017). Deep probabilistic programming. In *ICLR-17.*

Trappenberg, T. (2010). *Fundamentals of Computational Neuroscience* (2nd edition). Oxford University Press.

Tsang, E. (1993). *Foundations of Constraint Satisfaction*. Academic Press.

Tshitoyan, V., Dagdelen, J., Weston, L., Dunn, A., Rong, Z., Kononova, O., Persson, K. A., Ceder, G., and Jain, A. (2019). Unsupervised word embeddings capture latent knowledge from materials science literature. *Nature*, *571*, 95.

Tsitsiklis, J. N. and Van Roy, B. (1997). An analysis of temporal–difference learning with function approximation. *IEEE Transactions on Automatic Control*, *42*, 674–690.

Tukey, J. W. (1977). *Exploratory Data Analysis*. Addison-Wesley.

Tumer, K. and Wolpert, D. (2000). Collective intelligence and Braess' paradox. In *AAAI-00.*

Turian, J., Ratinov, L., and Bengio, Y. (2010). Word representations: a simple and general method for semi-supervised learning. In *ACL-10*.

Turing, A. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Mathematical Society, 2nd series*, *42*, 230–265.

Turing, A. (1948). Intelligent machinery. Tech. rep., National Physical Laboratory. reprinted in (Ince, 1992).

Turing, A. (1950). Computing machinery and intelligence. *Mind*, *59*, 433–460.

Turing, A., Strachey, C., Bates, M. A., and Bowden, B. V. (1953). Digital computers applied to games. In Bowden, B. V. (Ed.), *Faster than Thought*. Pitman.

Turing, A. (1947). Lecture to the London Mathematical Society on 20 February 1947.

Turing, A. (1996). Intelligent machinery, a heretical theory. *Philosophia Mathematica*, *4*, 256–260. Originally written c. 1951.

Tversky, A. and Kahneman, D. (1982). Causal schemata in judgements under uncertainty. In Kahneman, D., Slovic, P., and Tversky, A. (Eds.), *Judgement Under Uncertainty: Heuristics and Biases*. Cambridge University Press.

Tygar, J. D. (2011). Adversarial machine learning. *IEEE Internet Computing*, *15*, 4–6.

Ullman, J. D. (1985). Implementation of logical query languages for databases. *ACM Transactions on Database Systems*, *10*, 289–321.

Ullman, S. (1979). *The Interpretation of Visual Motion*. MIT Press.

Urmson, C. and Whittaker, W. (2008). Self-driving cars and the Urban Challenge. *IEEE Intelligent Systems*, *23*, 66–68.

Valiant, L. (1984). A theory of the learnable. *CACM*, *27*, 1134–1142.

Vallati, M., Chrpa, L., and Kitchin, D. E. (2015). Portfolio-based planning: State of the art, common practice and open challenges. *AI Commun.*, 28(4), 717–733.

van Beek, P. (2006). Backtracking search algorithms. In Rossi, F., van Beek, P., and Walsh, T. (Eds.), *Handbook of Constraint Programming*. Elsevier.

van Beek, P. and Chen, X. (1999). CPlan: A constraint programming approach to planning. In *AAAI-99*.

van Beek, P. and Manchak, D. (1996). The design and experimental analysis of algorithms for temporal reasoning. *JAIR*, *4*, 1–18.

van Bentham, J. and ter Meulen, A. (1997). *Handbook of Logic and Language*. MIT Press.

van den Oord, A., Dieleman, S., and Schrauwen, B. (2014). Deep content–based music recommendation. In *NeurIPS 26*.

van den Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A., and Kavukcuoglu, K. (2016a). WaveNet: A generative model for raw audio. arXiv:1609.03499.

van den Oord, A., Kalchbrenner, N., and Kavukcuoglu, K. (2016b). Pixel recurrent neural networks. arXiv:1601.06759.

van Harmelen, F., Lifschitz, V., and Porter, B. (2007). *The Handbook of Knowledge Representation*. Elsevier.

van Heijenoort, J. (Ed.). (1967). *FromFregeto Godel: A Source Book in Mathematical Logic, 1879–1931*. Harvard University Press.

Van Hentenryck, P., Saraswat, V., and Deville, Y. (1998). Design, implementation, and evaluation of the constraint language cc(FD). *J. Logic Programming*, *37*, 139–164.

van Hoeve, W.-J. (2001). The alldifferent constraint: a survey. In *6th Annual Workshop of the ERCIM Working Group on Constraints*.

van Hoeve, W.-J. and Katriel, I. (2006). Global constraints. In Rossi, F., vanBeek, P., and Walsh, T. (Eds.), *Handbook of Constraint Processing*. Elsevier.

van Lambalgen, M. and Hamm, F. (2005). *The Proper Treatment of Events*. Wiley-Blackwell.

van Nunen, J. A. E. E. (1976). A set of successive approximation methods for discounted Markovian decision problems. *Zeitschrift fur Operations Research, Serie A*, *20*, 203–208.

Van Roy, P. L. (1990). Can logic programming execute as fast as imperative programming? Report, Computer Science Division, UC Berkeley.

Vapnik, V. N. (1998). *Statistical Learning Theory.* Wiley.

Vapnik, V. N. and Chervonenkis, A. Y. (1971). On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and Its Applications*, *16*, 264–280.

Vardi, M. Y. (1996). An automata-theoretic approach to linear temporal logic. In Moller, F. and Birtwistle, G. (Eds.), *Logics for Concurrency*. Springer.

Varian, H. R. (1995). Economic mechanism design for computerized agents. In *USENIX Workshop on Electronic Commerce*.

Vasilache, N., Johnson, J., Mathieu, M., Chintala, S., Piantino, S., and LeCun, Y. (2014). Fast convolutional nets with fbfft: A GPU performance evaluation. arXiv:1412.7580.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I.(2018). Attention is all you need. In *NeurIPS 30.*

Veach, E. and Guibas, L. J. (1995). Optimally combining sampling techniques for Monte Carlo rendering. In *Proc. 22rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH).*

Venkatesh, S. (2012). *The Theory of Probability: Explorations and Applications*. Cambridge University Press.

Vere, S. A. (1983). Planning in time: Windows and durations for activities and goals. *PAMI*, *5*, 246–267.

Verma, S. and Rubin, J. (2018). Fairness definitions explained. In *2018 IEEE/ACM International Workshop on Software Fairness*.

Verma, V., Gordon, G., Simmons, R., and Thrun, S. (2004). Particle filters for rover fault diagnosis. *IEEE Robotics and Automation Magazine, June*.

Vinge, V. (1993). The coming technological singularity: How to survive in the post-human era. In *Proc. Vision-21: Interdisciplinary Science and*

Engineering in the Era of Cyberspace. NASA.

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., Hassabis, D., Apps, C., and Silver, D. (2019). Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 575, 350–354.

Vinyals, O., Ewalds, T., Bartunov, S., and Georgiev, P. (2017a). StarCraft II: A new challenge for reinforcement learning. arXiv:1708.04782.

Vinyals, O., Toshev, A., Bengio, S., and Erhan, D. (2017b). Show and tell: Lessons learned from the 2015 MSCOCO image captioning challenge. *PAMI*, *39*, 652–663.

Viola, P. and Jones, M. (2004). Robust real-time face detection. *IJCV*, *57*, 137–154.

Visser, U., Ribeiro, F., Ohashi, T., and Dellaert, F. (Eds.). (2008). *RoboCup 2007: Robot Soccer World Cup XI*. Springer.

Viterbi, A. J. (1967). Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, *13*, 260–269.

Vlassis, N. (2008). *A Concise Introduction to Multiagent Systems and Distributed Artificial Intelligence.* Morgan & Claypool.

von Mises, R. (1928). *Wahrscheinlichkeit, Statistik und Wahrheit.* J. Springer.

von Neumann, J. (1928). Zur Theorie der Gesellschaftsspiele. *Mathematische Annalen*, *100*, 295–320.

von Neumann, J. and Morgenstern, O. (1944). *Theory of Games and Economic Behavior* (first edition). Princeton University Press.

von Winterfeldt, D. and Edwards, W. (1986). *Decision Analysis and Behavioral Research.* Cambridge University Press.

Vossen, T., Ball, M., Lotem, A., and Nau, D. S. (2001). Applying integer programming to AI planning. *Knowledge Engineering Review, 16*, 85–100.

Wainwright, M. and Jordan, M. I. (2008). Graphical models, exponential families, and variational inference. *Foundations and Trends in Machine Learning, 1*, 1–305.

Walker, G. (1931). On periodicity in series of related terms. *Proc. Roy. Soc., A, 131*, 518–532.

Walker, R. J. (1960). An enumerative technique for a class of combinatorial problems. In *Proc. Sympos. Appl. Math.*, Vol. 10.

Wallace, A. R. (1858). On the tendency of varieties to depart indefinitely from the original type. *Proc. Linnean Society of London, 3*, 53–62.

Walpole, R. E., Myers, R. H., Myers, S. L., and Ye, K. E. (2016). *Probability and Statistics for Engineers and Scientists* (9th edition). Pearson.

Walsh, T. (2015). Turing'sredflag. arXiv:1510.09033.

Waltz, D. (1975). Understanding line drawings of scenes with shadows. In Winston, P. H. (Ed.), *The Psychology of Computer Vision.* McGraw-Hill.

Wang, A., Pruksachatkun, Y., Nangia, N., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. R. (2019). SuperGLUE: A stickier benchmark for general-purpose language understanding systems. arXiv:1905.00537.

Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. (2018a). GLUE: A multi-task benchmark and analysis platform for natural language understanding. arXiv:1804.07461.

Wang, J., Zhu, T., Li, H., Hsueh, C.-H., and Wu, I.-C. (2018b). Belief-state Monte Carlo tree search for phantom Go. *IEEE Transactions on Games*, *10*, 139154.

Wanner, E. (1974). *On Remembering, Forgetting and Understanding Sentences*. Mouton.

Warren, D. H. D. (1974). WARPLAN: A System for Generating Plans. Department of Computational Logic Memo, University of Edinburgh.

Warren, D. H. D. (1983). An abstract Prolog instruction set. Technical note, SRI International.

Wasserman, L. (2004). *All of Statistics*. Springer.

Watkins, C. J. (1989). *Models of Delayed Reinforcement Learning*. Ph.D. thesis, Psychology Department, Cambridge University.

Watson, J. D. and Crick, F. (1953). A structure for deoxyribose nucleic acid. *Nature*, *171*, 737.

Wattenberg, M., Viegas, F., and Johnson, I. (2016). How to use t-SNE effectively. *Distill*, *1*.

Waugh, K., Schnizlein, D., Bowling, M., and Szafron, D. (2009). Abstraction pathologies in extensive games. In *AAMAS–09*.

Weibull, J. (1995). *Evolutionary Game Theory*. MIT Press.

Weidenbach, C. (2001). SPASS: Combining superposition, sorts and splitting. In Robinson, A. and Voronkov, A. (Eds.), *Handbook of Automated Reasoning*. MIT Press.

Weiss, G. (2000a). *Multiagent Systems*. MIT Press.

Weiss, Y. (2000b). Correctness of local probability propagation in graphical models with loops. *Neural Computation*, *12*, 1–41.

Weiss, Y. and Freeman, W. (2001). Correctness of belief propagation in Gaussian graphical models of arbitrary topology. *Neural Computation*, *13*, 2173–2200.

Weizenbaum, J. (1976). *Computer Power and Human Reason*. W. H. Freeman.

Weld, D. S. (1994). An introduction to least commitment planning. *AIMag*, *15*, 27–61.

Weld, D. S. (1999). Recent advances in AI planning. *AIMag*, *20*, 93–122.

Weld, D. S., Anderson, C. R., and Smith, D. E. (1998). Extending Graphplan to handle uncertainty and sensing actions. In *AAAI-98*.

Weld, D. S. and de Kleer, J. (1990). *Readings in Qualitative Reasoning about Physical Systems*. Morgan Kaufmann.

Weld, D. S. and Etzioni, O. (1994). The first law of robotics: Acallto arms. In *AAAI-94*.

Wellman, M. P. (1985). Reasoning about preference models. Technical report, Laboratory for Computer Science, MIT.

Wellman, M. P. (1988). *Formulation of Tradeoffs in Planning under Uncertainty*. Ph.D. thesis, MIT.

Wellman, M. P. (1990a). Fundamental concepts of qualitative probabilistic networks. *AIJ*, *44*, 257–303.

Wellman, M. P. (1990b). The STRIPS assumption for planning under uncertainty. In *AAAI-90*.

Wellman, M. P., Breese, J. S., and Goldman, R. (1992). From knowledge bases to decision models. *Knowledge Engineering Review*, *7*, 35–53.

Wellman, M. P. and Doyle, J. (1992). Modular utility representation for decision–theoretic planning. In *ICAPS-92*.

Wellman, M. P, Wurman, P, O'Malley, K., Bangera, R., Lin, S., Reeves, D., and Walsh, W. (2001). Designing the market game for a trading agent competition. *IEEE Internet Computing*, *5*, 43–51.

Werbos, P. (1974). *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. Ph.D. thesis, Harvard University.

Werbos, P. (1990). Backpropagation through time: What it does and how to do it. *Proc. IEEE*, *78*, 1550–1560.

Werbos, P. (1992). Approximate dynamic programming for real-time control and neural modeling. In White, D. A. and Sofge, D. A. (Eds.), *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches*. Van Nostrand Reinhold.

Werbos, P. (1977). Advanced forecasting methods for global crisis warning and models of intelligence. *General Systems Yearbook*, *22*, 25–38.

Wesley, M. A. and Lozano-Perez, T. (1979). An algorithm for planning collision–free paths among polyhedral objects. *CACM*, 22, 560–570.

West, D. M. (2018). *The Future of Work: Robots, AI, and Automation*. Brookings Institution Press.

West, S. M., Whittaker, M., and Crawford, K. (2019). Discriminating systems: Gender, race and power in AI. Tech. rep., AI Now Institute.

Wheatstone, C. (1838). On some remarkable, and hitherto unresolved, phenomena of binocular vision. *Phil. Trans. Roy. Soc.*, *2*, 371–394.

White, C., Neiswanger, W., and Savani, Y.(2019). BANANAS: Bayesian optimization with neural architectures for neural architecture search. arXiv:1910.11858.

Whitehead, A. N. and Russell, B. (1910). *Principia Mathematica*. Cambridge University Press.

Whittle, P (1979). Discussion of Dr Gittins' paper. *J. Royal Statistical Society*, *41*, 165.

Whorf, B. (1956). *Language, Thought, and Reality*. MIT Press.

Widrow, B. (1962). Generalization and information storage in networks of ADALINE "neurons". In Yovits, M. C., Jacobi, G. T., and Goldstein, G. D. (Eds.), *Self-Organizing Systems*. Spartan.

Widrow, B. and Hoff, M. E. (1960). Adaptive switching circuits. In *IRE WESCON Convention Record*.

Wiedijk, F. (2003). Comparing mathematical provers. In *Proc. 2nd Int. Conf. on Mathematical Knowledge Management*.

Wiegley, J., Goldberg, K., Peshkin, M., and Brokowski, M. (1996). A complete algorithm for designing passive fences to orient parts. In *ICRA–96*.

Wiener, N. (1942). The extrapolation, interpolation, and smoothing of stationary time series. Tech. rep., Research Project DIC–6037, MIT.

Wiener, N. (1948). *Cybernetics*. Wiley.

Wiener, N. (1950). *The Human Use of Human Beings*. Houghton Mifflin.

Wiener, N. (1960). Some moral and technical consequences of automation. *Science*, *131*, 1355–1358.

Wiener, N. (1964). *God & Golem, Inc: A Comment on Certain Points Where Cybernetics Impinges on Religion*. MIT Press.

Wilensky, R. (1978). *Understanding Goal-Based Stories*. Ph.D. thesis, Yale University.

Wilkins, D. E. (1988). *PracticalPlanning: Extending the AI Planning Paradigm*. Morgan Kaufmann.

Wilkins, D. E. (1990). Can AI planners solve practical problems? *Computational Intelligence*, *6*, 232–246.

Wilks, Y. (2010). *Close Engagements With Artificial Companions: Key Social, Psychological, Ethical and Design Issues*. John Benjamins.

Wilks, Y. (2019). *Artificial Intelligence: Modern Magic or Dangerous Future*. Icon.

Williams, A., Nangia, N., and Bowman, S. (2018). A broad-coverage challenge corpus for sentence understanding through inference. In *NAACL*

HLT.

Williams, B., Ingham, M., Chung, S., and Elliott, P. (2003). Model-based programming of intelligent embedded systems and robotic space explorers. *Proc. IEEE*, 91(212–237).

Williams, R. J. (1992). Simple statistical gradient following algorithms for connectionist reinforcement learning. *Machine Learning, 8*, 229–256.

Williams, R. J. and Zipser, D. (1989). A learning algorithm for continually running fully recurrent neural networks. *Neural Computation, 1*, 270–280.

Williams, R. J. and Baird, L. C. I. (1993). Tight performance bounds on greedy policies based on imperfect value functions. Tech. rep., College of Computer Science, Northeastern University.

Wilson, D. H. (2011). *Robopocalypse*. Doubleday.

Wilson, R. A. and Keil, F. C. (Eds.). (1999). *The MIT Encyclopedia of the Cognitive Sciences*. MIT Press.

Wilson, R. (2004). *Four Colors Suffice*. Princeton University Press.

Wexler, Y. and Meek, C. (2009). MAS: A multiplicative approximation scheme for probabilistic inference. In *NeurIPS 21*.

Wilt, C. M. and Ruml, W. (2014). Speedy versus greedysearch. In *Seventh Annual Symposium on Combinatorial Search.*

Wilt, C. M. and Ruml, W. (2016). Effective heuristics for suboptimal best-first search.*JAIR, 57*, 273–306.

Wingate, D. and Seppi, K. D. (2005). Prioritization methods for accelerating MDP solvers. *JMLR, 6*, 851–881.

Wingate, D., Stuhlmuller, A., and Goodman, N. D. (2011). Lightweight implementations of probabilistic programming languages via transformational compilation. In *AISTATS-11*.

Winograd, S. and Cowan, J. D. (1963). *Reliable Computation in the Presence of Noise.* MIT Press.

Winograd, T. (1972). Understanding natural language. *Cognitive Psychology, 3*, 1–191.

Winston, P. H. (1970). Learning structural descriptions from examples. Technical report, Department of Electrical Engineering and Computer Science, MIT.

Wintermute, S., Xu, J., and Laird, J. (2007). SORTS: A human-level approach to real-time strategy AI. In *Proc. Third Artificial Intelligence and Interactive Digital Entertainment Conference*.

Winternitz, L. (2017). Autonomous navigation above the GNSS constellations and beyond: GPS navigation for the magnetospheric multiscale mission and SEXTANT pulsar navigation demonstration. Tech. rep., NASA Goddard Space Flight Center.

Witten, I. H. (1977). An adaptive optimal controller for discrete-time Markov environments. *Information and Control, 34,* 286–295.

Witten, I. H. and Bell, T. C. (1991). The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. *IEEE Transactions on Information Theory, 37*, 1085–1094.

Witten, I. H. and Frank, E. (2016). *Data Mining: Practical Machine Learning Tools and Techniques* (4th edition). Morgan Kaufmann.

Witten, I. H., Moffat, A., and Bell, T. C. (1999). *Managing Gigabytes: Compressing and Indexing Documents and Images* (2nd edition). Morgan Kaufmann.

Wittgenstein, L. (1922). *Tractatus Logico-Philosophicus* (2nd edition). Routledge and Kegan Paul. Reprinted 1971, edited by D. F. Pears and B. F. McGuinness.

Wittgenstein, L. (1953). *Philosophical Investigations*. Macmillan.

Wojciechowski, W. S. and Wojcik, A. S. (1983). Automated design of multiple-valued logic circuits by automated theorem proving techniques. *IEEE Transactions on Computers, C–32*, 785–798.

Wolfe, J. and Russell, S. J. (2007). Exploiting belief state structure in graph search. In *ICAPS Workshop on Planning in Games*.

Wolpert, D. (2013). Ubiquity symposium: Evolutionary computation and the processes of life: what the no free lunch theorems really mean: how to improve search algorithms. *Ubiquity, December*, 1–15.

Wolpert, D. and Macready, W. G. (1997). No free lunch theorems for optimization. *IEEE Trans. Evolutionary Computation, 1*(1), 67–82.

Wong, C., Houlsby, N., Lu, Y., and Gesmundo, A. (2018). Transfer learning with neural AutoML. In *NeurIPS 31*.

Woods, W. A. (1973). Progress in natural language understanding: An application to lunar geology. In *AFIPS Conference Proceedings*.

Woods, W. A. (1975). What's in a link? Foundations for semantic networks. In Bobrow, D. G. and Collins, A. M. (Eds.), *Representation and Understanding: Studies in Cognitive Science*. Academic Press.

Wooldridge, M. (2009). *An Introduction to MultiAgent Systems* (2nd edition). Wiley.

Wooldridge, M. and Rao, A. (Eds.). (1999). *Foundations of Rational Agency*. Kluwer.

Wos, L., Carson, D., and Robinson, G. (1964). The unit preference strategy in theorem proving. In *Proc. Fall Joint Computer Conference.*

Wos, L., Carson, D., and Robinson, G. (1965). Efficiency and completeness of the set-of-support strategy in theorem proving. *JACM*, *12*, 536–541.

Wos, L., Overbeek, R., Lusk, E., and Boyle, J. (1992). *Automated Reasoning: Introduction and Applications*(2nd edition). McGraw-Hill.

Wos, L., and Robinson, G. (1968). Paramodulation and set of support. In *Proc*. *IRIA Symposium on Automatic Demonstration*.

Wos, L., Robinson, G., Carson, D., and Shalla, L. (1967). The concept of demodulation in theorem proving. *JACM*, *14*, 698–704.

Wos, L. and Winker, S. (1983). Open questions solved with the assistance of AURA. In Bledsoe, W. W. and Loveland, D. (Eds.), *Automated Theorem Proving: After 25 Years*. American Mathematical Society.

Wos, L. and Pieper, G. (2003). *Automated Reasoning and the Discovery of Missing and Elegant Proofs*. Rinton Press.

Wray, R. E. and Jones, R. M. (2005). An introduction to Soar as an agent architecture. In Sun, R. (Ed.), *Cognition and Multi-Agent Interaction: From Cognitive Modeling to Social Simulation*. Cambridge University Press.

Wright, S. (1921). Correlation and causation. *J. Agricultural Research*, *20,* 557–585.

Wright, S. (1931). Evolution in Mendelian populations. *Genetics*, *16*, 97–159.

Wright, S. (1934). The method of path coefficients. *Annals of Mathematical Statistics*, *5*, 161–215.

Wu, F. and Weld, D. S. (2008). Automatically refining the Wikipedia infobox ontology. In *17th World Wide Web Conference (WWW2008)*.

Wu, Y., Li, L., and Russell, S. J. (2016a). SWIFT: Compiled inference for probabilistic programming languages. In *IJCAI-16*.

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., *et al.* (2016b). Google's neural machine translation system: Bridging the gap between human and machine translation. arXiv:1609.08144.

Wu, Y. and He, K. (2018). Group normalization. arXiv:1803.08494.

Xiong, W., Wu, L., Alleva, F., Droppo, J., Huang, X., and Stolcke, A. (2017). The Microsoft 2017 conversational speech recognition system. arXiv:1708.06073.

Yampolskiy, R. V. (2018). *Artificial Intelligence Safety and Security*. Chapman and Hall/CRC.

Yang, G., Lin, Y., and Bhattacharya, P. (2010). A driver fatigue recognition model based on information fusion and dynamic Bayesian network. *Inf. Sci.*, *180*, 1942–1954.

Yang, X.-S. (2009). Firefly algorithms for multimodal optimization. In *International Symposium on Stochastic Algorithms*.

Yang, X.-S. and Deb, S. (2014). Cuckoo search: Recent advances and applications. *Neural Computing and Applications*, *24*, 169–174.

Yang, Z., Dai, Z., Yang, Y., Carbonell, J. G., Salakhutdinov, R., and Le, Q. V. (2019). XLNet: Generalized autoregressive pretraining for language understanding. arXiv:1906.08237.

Yarowsky, D. (1995). Unsupervised word sense disambiguation rivaling supervised methods. In *ACL–95*.

Ye, Y. (2011). The simplex and policy-iteration methods are strongly polynomial for the Markov decision problem with a fixed discount rate. *Mathematics of Operations Research*, *36*, 593–784.

Yedidia, J., Freeman, W., and Weiss, Y. (2005). Constructing free-energy approximations and generalized belief propagation algorithms. *IEEE Transactions on Information Theory*, *51*, 2282–2312.

Yeo, H.-S., Minami, R., Rodriguez, K., Shaker, G., and Quigley, A. (2018). Exploring tangible interactions with radar sensing. *Proc. ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, *2*, 1–25.

Ying, C., Kumar, S., Chen, D., Wang, T., and Cheng, Y. (2018). Image classification at supercomputer scale. arXiv:1811.06992.

Yip, K. M.-K. (1991). *KAM:* A *System for Intelligently Guiding Numerical Experimentation by Computer*. MIT Press.

Yngve, V. (1955). A model and an hypothesis for language structure. In Locke, W. N. and Booth, A. D. (Eds.), *Machine Translation of Languages*. MIT Press. Yob, G. (1975). Hunt the wumpus! *Creative Computing*, *Sep/Oct*. Yoshikawa, T. (1990). *FoundationsofRobotics: Analysis and Control.* MIT Press.

You, Y., Pan, X., Wang, Z., and Lu, C. (2017). Virtual to real reinforcement learning for autonomous driving. arXiv:1704.03952.

Young, H. P. (2004). *Strategic Learning and Its Limits*. Oxford University Press.

Young, S., Gašić, M., Thompson, B., and Williams, J. (2013). POMDP-based statistical spoken dialog systems: Areview. *Proc. IEEE*, *101*, 1160–1179. Younger, D. H. (1967). Recognition and parsing of context–free languages in time n^3. *Information and Control*, *10*, 189–208.

Younger, D. H. (1967). Recognition and parsing of context-free languages in time n^3. Information and Control, 10, 189–208.

Yu, D. and Deng, L. (2016). *Automatic Speech Recognition.* Springer-Verlag.

Yu, H.-F., Lo, H.-Y., Hsieh, H.-P., and Lou, J.-K. (2011). Feature engineering and classifier ensemble for KDD Cup 2010. In *Proc. KDD Cup 2010 Workshop*.

Yu, K., Sciuto, C., Jaggi, M., Musat, C., and Salzmann, M. (2019). Evaluating the search phase of neural architecture search. arXiv:1902.08142.

Yudkowsky, E. (2008). Artificial intelligence as a positive and negative factor in global risk. In Bostrom, N. and Cirkovic, M. (Eds.), *Global Catastrophic Risk*. Oxford University Press.

Yule, G. U. (1927). On a method of investigating periodicities in disturbed series, with special reference to Wolfer's sunspot numbers. *Phil. Trans. Roy. Soc., A*, *226*, 267–298.

Zadeh, L. A. (1965). Fuzzy sets. *Information and Control*, *8*, 338–353.

Zadeh, L. A. (1978). Fuzzy sets as a basis for a theory of possibility. *Fuzzy Sets and Systems*, *1*, 3–28.

Zaritskii, V. S., Svetnik, V. B., and Shimelevich, L. I. (1975). Monte-Carlo technique in problems of optimal information processing. *Automation and Remote Control*, *36*, 2015–22.

Zeckhauser, R. and Shepard, D. (1976). Where now for saving lives? *Law and Contemporary Problems*, *40*, 5–45.

Zeeberg, A. (2017). D.I.Y. artificial intelligence comes to a Japanese family farm. *New Yorker*, August 10.

Zelle, J. and Mooney, R. (1996). Learning to parse database queries using inductive logic programming. In *AAAI-96*.

Zemel, R., Wu, Y., Swersky, K., Pitassi, T., and Dwork, C. (2013). Learning fair representations. In *ICML-13*.

Zemelman, B. V., Lee, G. A., Ng, M., and Miesenböck, G. (2002). Selective photostimulation of genetically chARGed neurons. *Neuron*, 33, 15–22.

Zermelo, E. (1913). Uber Eine Anwendung der Mengenlehre auf die Theorie des Schachspiels. In *Proc. Fifth International Congress of Mathematicians.*

Zermelo, E. (1976). An application of set theory to the theory of chess-playing. *Firbush News*, *6*, 37–42. English translation of (Zermelo 1913).

Zettlemoyer, L. and Collins, M. (2005). Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *UAI-05.*

Zhang, C., Bengio, S., Hardt, M., Recht, B., and Vinyals, O. (2016). Understanding deep learning requiresrethinking generalization. arXiv:1611.03530. Zhang, H. and Stickel, M. E. (1996). An efficient algorithm for unit-propagation. In *Proc. Fourth International Symposium on Artificial Intelligence and Mathematics*.

Zhang, L., Pavlovic, V., Cantor, C. R., and Kasif, S. (2003). Human-mouse gene identification by comparative evidence integration and evolutionary analysis. *Genome Research, 13,* 1190–1202.

Zhang, N. L. and Poole, D. (1994). A simple approach to Bayesian network computations. In *Proc. 10th Canadian Conference on Artificial 1ntelligence*.

Zhang, S., Yao, L., and Sun, A. (2017). Deep learning based recommender system: A survey and new perspectives. arXiv:1707.07435.

Zhang, X., Zhao, J., and LeCun, Y. (2016). Character-level convolutional networks for text classification. In *NeurIPS 28.*

Zhang, Y., Pezeshki, M., Brakel, P., Zhang, S., Laurent, C., Bengio, Y., and Courville, A. (2017). Towards end-to-end speech recognition with deep convolutional neural networks. arXiv:1701.02720.

Zhao, K. and Huang, L. (2015). Type-driven incremental semantic parsing with polymorphism. In *NAACL HLT.*

Zhou, K., Doyle, J., and Glover, K. (1995). *Robust and Optimal Control.* Pearson.

Zhou, R. and Hansen, E. (2002). Memory-bounded A* graph search. In *Proc. 15th International FLAIRS Conference*.

Zhou, R. and Hansen, E. (2006). Breadth-first heuristic search. *AIJ, 170*, 385–408.

Zhu, B., Jiao, J., and Tse, D. (2019). Deconstructing generative adversarial networks. arXiv:1901.09465.

Zhu, D. J. and Latombe, J.-C. (1991). New heuristic algorithms for efficient hierarchical path planning. *IEEE Transactions on Robotics and Automation, 7*, 920.

Zhu, J.-Y., Park, T., Isola, P, and Efros, A. (2017). Unpaired image-to-image translation using cycle-consistent adversarial networks. In *ICCV-17*.

Zhu, M., Zhang, Y., Chen, W., Zhang, M., and Zhu, J. (2013). Fast and accurate shift-reduce constituent parsing. In *ACL-13*.

Ziebart, B. D., Maas, A. L., Dey, A. K., and Bagnell, J. A. (2008). Navigate like a cabbie: Probabilistic reasoning from observed context-aware behavior. In *Proc. 10th Int. Conf. on Ubiquitous Computing*.

Ziebart, B. D., Ratliff, N., Gallagher, G., Mertz, C., Peterson, K., Bagnell, J. A., Hebert, M., Dey, A. K., and Srinivasa, S. (2009). Planning-based prediction for pedestrians. In *IROS-09*.

Zimmermann, H.-J. (Ed.). (1999). *Practical Applications of Fuzzy Technologies*. Kluwer.

Zimmermann, H.-J. (2001). *Fuzzy Set Theory–And 1ts Applications* (4th edition). Kluwer.

Zinkevich, M., Johanson, M., Bowling, M., and Piccione, C. (2008). Regret minimization in games with incomplete information. In *NeurIPS 20*.

Zipf, G. (1935). *The Psychobiology of Language*. Houghton Mifflin.

Zipf, G. (1949). *Human Behavior and the Principle of Least Effort*. Addison-Wesley.

Zobrist, A. L. (1970). *Feature Extraction and Representation for Pattern Recognition and the Game of Go.* Ph.D. thesis, University of Wisconsin.

Zollmann, A., Venugopal, A., Och, F. J., and Ponte, J. (2008). A systematic comparison of phrase-based, hierarchical and syntax-augmented statistical MT. In *COLING-08*.

Zoph, B. and Le, Q. V. (2016). Neural architecture search with reinforcement learning. arXiv:1611.01578.

Zuse, K. (1945). The Plankalkül. Report, Gesellschaft für Mathematik und Datenverarbeitung.

Zweig, G. and Russell, S. J. (1998). Speech recognition with dynamic Bayesian networks. In *AAAI-98*.

Index

Page numbers in **bold** refer to definitions of terms and algorithms.

Page numbers in *italics* refer to items in the bibliography.

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

X

Y

Z