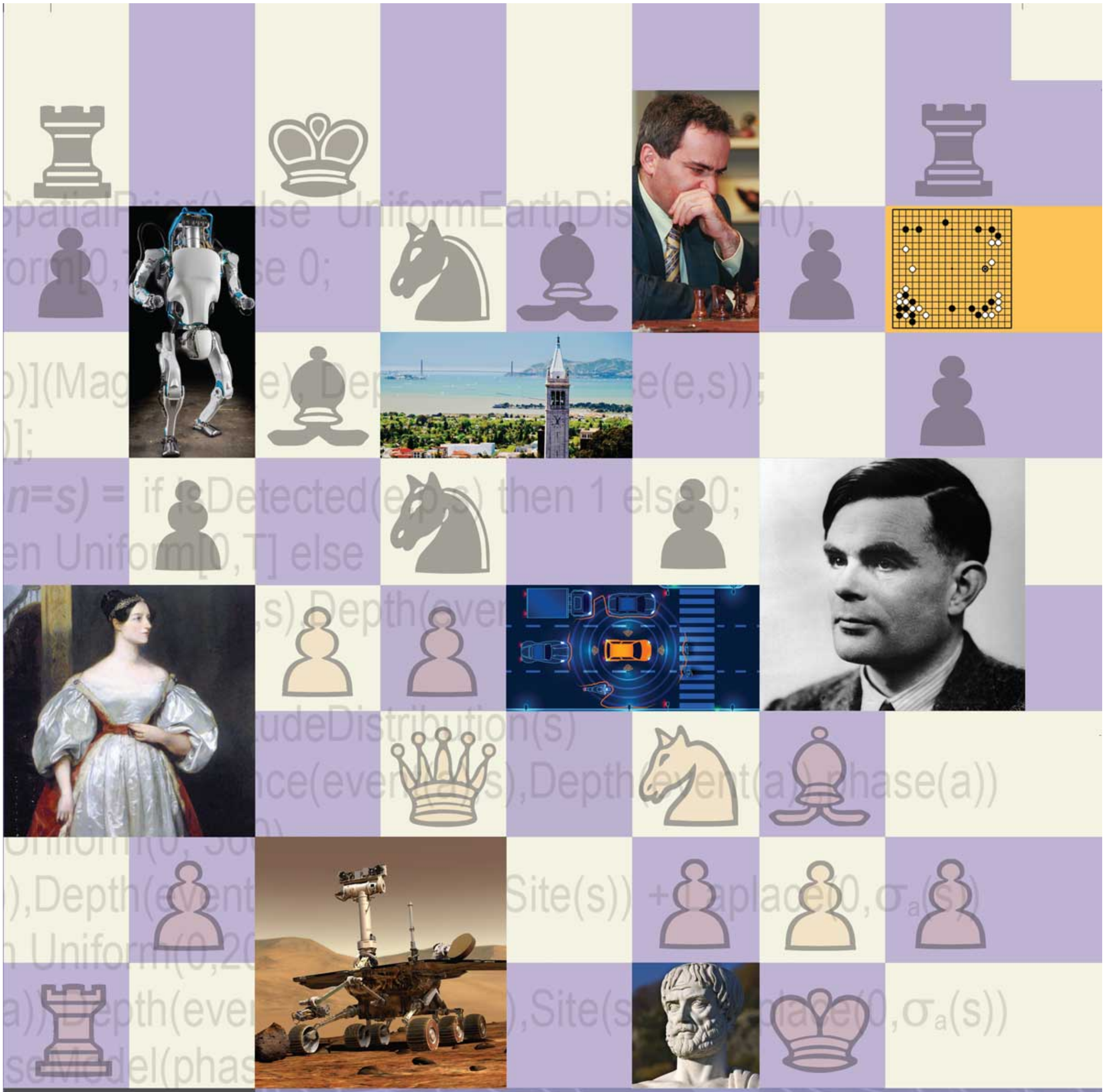

SpatialPrior() else UniformEarthDistribution();
Uniform[0,T] else 0;
))](MagnitudeDistribution(e), Depth(e), Distance(e,s));
n=s) = if IsDetected(e,p,s) then 1 else 0;
en Uniform[0,T] else
,s),Depth(event
udeDistribution(s)
nce(event(a),s),Depth(event(a)),phase(a))
Uniform(0, 50
),Depth(event Site(s)) + Laplace(0,σa(s))
Uniform(0,20
a)),Depth(event),Site(s Laplace(0,σa(s))
seModel(phas

Stuart
Russell
Peter
Norvig

Artificial Intelligence
A Modern Approach
Fourth Edition

Artificial Intelligence

A Modern Approach

Fourth 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

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

Acknowledgments of third-party content appear on the appropriate page within the text.

Cover Images:

Alan Turing – Science History Images/Alamy Stock Photo

Statue of Aristotle – Panos Karas/Shutterstock

Ada Lovelace – Pictorial Press Ltd/Alamy Stock Photo

Autonomous cars – Andrey Suslov/Shutterstock

Atlas Robot – Boston Dynamics, Inc.

Berkeley Campanile and Golden Gate Bridge – Ben Chu/Shutterstock

Background ghosted nodes – Eugene Sergeev/Alamy Stock Photo

Chess board with chess figure – Titania/Shutterstock

Mars Rover – Stocktrek Images, Inc./Alamy Stock Photo

Kasparov – KATHY WILLENS/AP Images

Library of Congress Cataloging-in-Publication Data

Names: Russell, Stuart J. (Stuart Jonathan), author. | Norvig, Peter, author.

Title: Artificial intelligence : a modern approach / Stuart J. Russell and Peter Norvig.

Description: Fourth edition. | Hoboken : Pearson, [2021] | Series: Pearson series in artificial intelligence | Includes bibliographical references and index. | Summary: “Updated edition of popular textbook on Artificial Intelligence.”— Provided by publisher.

Identifiers: LCCN 2019047498 | ISBN 9780134610993 (hardcover)

Subjects: LCSH: Artificial intelligence.

Classification: LCC Q335 .R86 2021 | DDC 006.3–dc23

LC record available at https://lccn.loc.gov/2019047498

ScoutAutomatedPrintCode

ISBN-10: 0-13-461099-7

ISBN-13: 978-0-13-461099-3

Pearson

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

For Kris, Isabella, and Juliet — P.N.

Preface

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

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

New to this edition

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

- We focus more on machine learning rather than hand-crafted knowledge engineering, due to the increased availability of data, computing resources, and new algorithms.
- Deep learning, probabilistic programming, and multiagent systems receive expanded coverage, each with their own chapter.
- The coverage of natural language understanding, robotics, and computer vision has been revised to reflect the impact of deep learning.
- The robotics chapter now includes robots that interact with humans and the application of reinforcement learning to robotics.
- Previously we defined the goal of AI as creating systems that try to maximize expected utility, where the specific utility information—the objective—is supplied by the human designers of the system. Now we no longer assume that the objective is fixed and known by the AI system; instead, the system may be uncertain about the true objectives of the humans on whose behalf it operates. It must learn what to maximize and must function appropriately even while uncertain about the objective.

- We increase coverage of the impact of AI on society, including the vital issues of ethics, fairness, trust, and safety.
- We have moved the exercises from the end of each chapter to an online site. This allows us to continuously add to, update, and improve the exercises, to meet the needs of instructors and to reflect advances in the field and in AI-related software tools.
- Overall, about 25% of the material in the book is brand new. The remaining 75% has been largely rewritten to present a more unified picture of the field. 22% of the citations in this edition are to works published after 2010.

Overview of the book

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

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

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

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

Term

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

Online resources

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

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

Book cover

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

Acknowledgments

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

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

Then some key roles:

- Cynthia Yeung and Malika Cantor (project management);
- Julie Sussman and Tom Galloway (copyediting and writing suggestions);

- Omari Stephens (illustrations);
- Tracy Johnson (editor);
- Erin Ault and Rose Kernan (cover and color conversion);
- Nalin Chhibber, Sam Goto, Raymond de Lacaze, Ravi Mohan, Ciaran O'Reilly, Amit Patel, Dragomir Radiv, and Samagra Sharma (online code development and mentoring);
- Google Summer of Code students (online code development).

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

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

About the Authors

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

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

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

Contents

I Artificial Intelligence

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.

Intelligence

Artificial Intelligence

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

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

1.1 What Is AI?

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

[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.

Rationality

From these two dimensions—human vs. rational[2] and thought vs. behavior—there are four possible combinations, and there have been adherents and research programs for all 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.

[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.

1.1.1 Acting humanly: The Turing test approach

Turing test

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

- **natural language processing** 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.

Natural language processing

Knowledge representation

Automated reasoning

Machine learning

Total Turing test

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

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

Computer vision

Robotics

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.

Introspection

Psychological experiments

Brain imaging

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

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

Cognitive science

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

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

1.1.3 Thinking rationally: The "laws of thought" approach

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

Syllogisms

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

Logicist

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

Probability

1.1.4 Acting rationally: The rational agent approach

Agent

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.

Rational agent

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.

Do the right thing

Standard model

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

Limited rationality

1.1.5 Beneficial machines

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

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

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

Value alignment problem

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

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

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

1.2 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.

Dualism

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.”

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

Empiricism

Induction

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

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

Logical positivism

Observation sentence

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

Confirmation theory

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

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

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

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

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

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

Utility

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 16.

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.

Utilitarianism

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

Deontological ethics

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.

Formal logic

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.

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

Probability

Statistics

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

Algorithm

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

Incompleteness theorem

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

Computability

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

Tractability

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

NP-completeness

1.2.3 Economics

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

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

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

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

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

Decision theory

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

Operations research

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.

Satisficing

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.

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

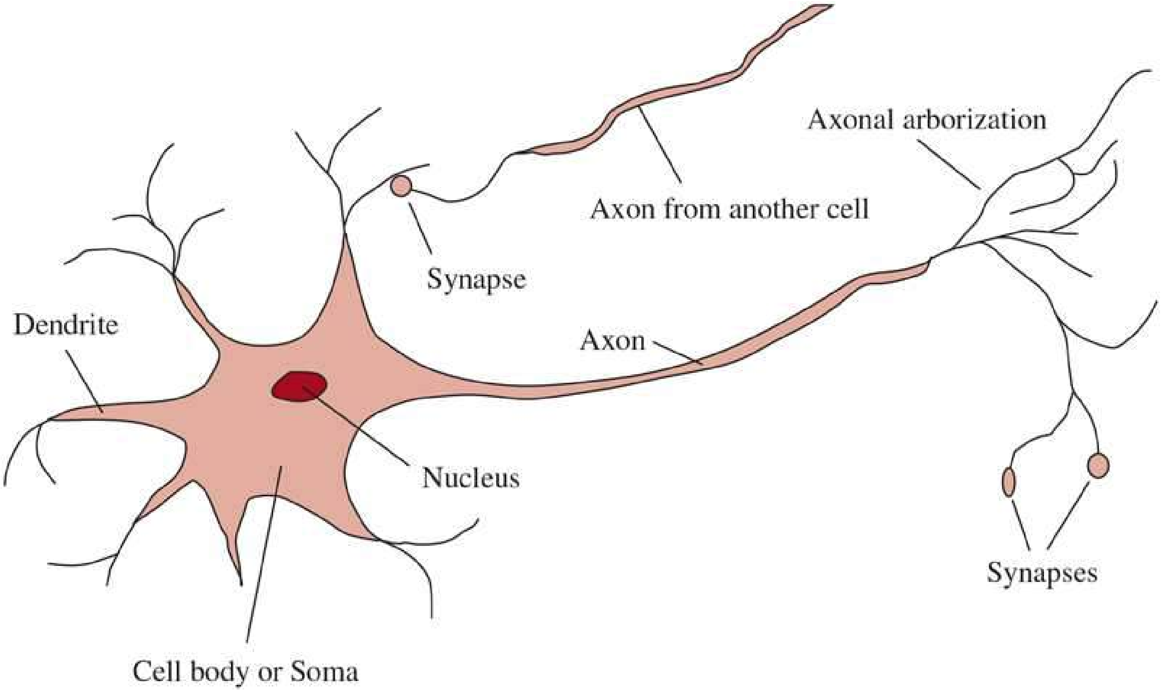
Neuroscience

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*.

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

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

Figure 1.1



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

Neuron

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; Optogenetics Zemelman *et al.*, 2002; Han and Boyden, 2007), which allow both measurement and control of individual neurons modified to be light-sensitive.

Optogenetics

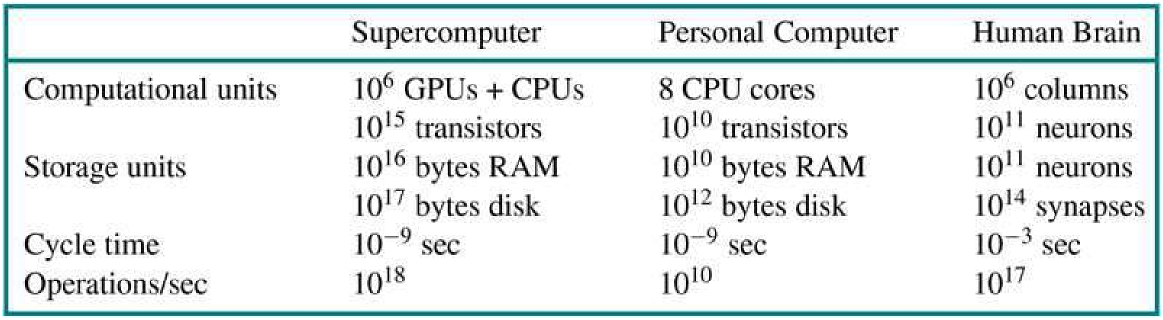
The development of **brain–machine interfaces** (Lebedev and Nicolelis, 2006) for both sensing and motor control not only promises to restore function to disabled individuals, but also sheds light on many aspects of neural systems. A remarkable finding from this work is that the brain is able to adjust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

Brain–machine interface

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

Singularity

Figure 1.2



	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}

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

Behaviorism

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

Cognitive psychology

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

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

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

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

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

Intelligence augmentation

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

1.2.6 Computer engineering

- How can we build an efficient computer?

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

[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.

Moore's law

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.

Quantum computing

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

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

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

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

1.2.7 Control theory and cybernetics

- How can artifacts operate under their own control?

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

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.

Control theory

Cybernetics

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

Homeostatic

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

Cost function

1.2.8 Linguistics

- How does language relate to thought?

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

Computational linguistics

Modern linguistics and AI, then, were “born” at about the same time, and grew up together, intersecting in a hybrid field called **computational linguistics** or **natural language processing.** The problem of understanding language turned out to be considerably more complex than it seemed in 1957. Understanding language requires an understanding of the subject matter and context, not just an understanding of the structure of sentences. This might seem obvious, but it was not widely appreciated until the 1960s. Much of the early work in **knowledge representation** (the study of how to put knowledge into a form that a computer can reason with) was tied to language and informed by research in linguistics, which was connected in turn to decades of work on the philosophical analysis of language.

1.3 The History of Artificial Intelligence

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

1.3.1 The inception of artificial intelligence (1943–1956)

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

Hebbian learning

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

There were a number of other examples of early work that can be characterized as AI, including two checkers-playing programs developed independently in 1952 by Christopher Strachey at the University of Manchester and by Arthur Samuel at IBM. However, Alan Turing's vision was the most influential. He gave lectures on the topic as early as 1947 at the London Mathematical Society and articulated a persuasive agenda in his 1950 article "Computing Machinery and Intelligence." Therein, he introduced the Turing test, machine learning, genetic algorithms, and reinforcement learning. He dealt with many of the objections raised to the possibility of AI, as described in Chapter 27. He also suggested that it would be easier to create human-level AI by developing learning algorithms and then teaching the machine rather than by programming its intelligence by hand. In subsequent lectures he warned that achieving this goal might not be the best thing for the human race.

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

[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.

> 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.

[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.

1.3.2 Early enthusiasm, great expectations (1952–1969)

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

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

Physical symbol system

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

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

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

Lisp

1958 also marked the year that Marvin Minsky moved to MIT. His initial collaboration with McCarthy did not last, however. McCarthy stressed representation and reasoning in formal logic, whereas Minsky was more interested in getting programs to work and eventually developed an anti-logic outlook. In 1963, McCarthy started the AI lab at Stanford. His plan to use logic to build the ultimate Advice Taker was advanced by J. A. Robinson's discovery in 1965 of the resolution method (a complete theorem-proving algorithm for first-order logic; see Chapter 9). Work at Stanford emphasized general-purpose methods for logical reasoning. Applications of logic included Cordell Green's question-answering and planning systems (Green, 1969b) and the Shakey robotics project at the Stanford Research Institute (SRI). The latter project, discussed further in Chapter 26, was the first to demonstrate the complete integration of logical reasoning and physical activity.

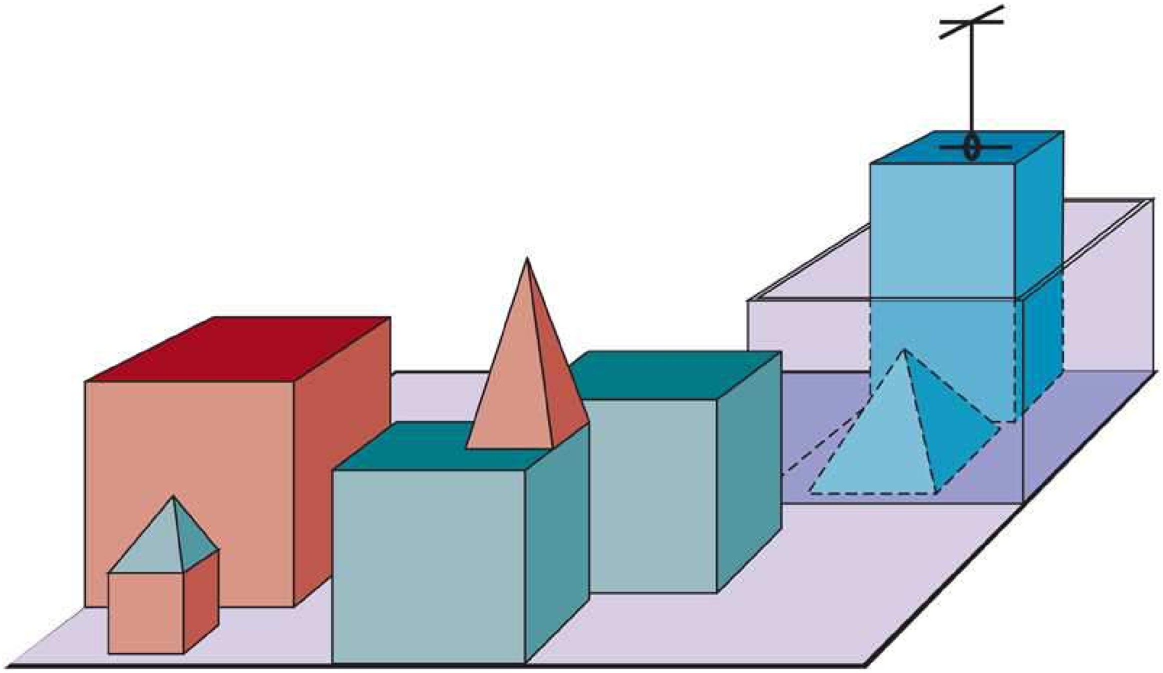
Microworld

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

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

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.”

Blocks world

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**) (Fried-Machine evolutionberg, 1958; Friedberg *et al.*, 1959) were based on the undoubtedly correct belief that by making an appropriate series of small mutations to a machine-code program, one can generate a program with good performance for any particular task. The idea, then, was to try random mutations with a selection process to preserve mutations that seemed useful. Despite thousands of hours of CPU time, almost no progress was demonstrated.

Machine evolution

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

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

1.3.4 Expert systems (1969–1986)

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

Weak method

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

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

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

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

Expert systems

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

Certainty factor

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

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

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

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

Frames

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

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

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

1.3.5 The return of neural networks (1986–present)

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

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.

connectionist

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

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

> being abandoned. There is a recognition that machine learning should not be isolated from information theory, that uncertain reasoning should not be isolated from stochastic modeling, that search should not be isolated from classical optimization and control, and that automated reasoning should not be isolated from formal methods and static analysis.

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

Hidden Markov models

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

Bayesian network

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

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

1.3.7 Big data (2001–present)

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.

Big data

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 handwritten digit recognition in the 1990s (LeCun *et al.*, 1995). It was not until 2011, however, that deep learning methods really took off. This occurred first in speech recognition and then in visual object recognition.

Deep learning

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

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

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

1.4 The State of the Art

AI Index

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 27 and 28.

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

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

- **LETHAL AUTONOMOUS WEAPONS:** These are defined by the United Nations as weapons that can locate, select, and eliminate human targets without human intervention. A primary concern with such weapons is their *scalability*: the absence of a requirement for human supervision means that a small group can deploy an arbitrarily large number of weapons against human targets defined by any feasible recognition criterion. The technologies needed for autonomous weapons are similar to those needed for self-driving cars. Informal expert discussions on the potential risks of lethal autonomous weapons began at the UN in 2014, moving to the formal pre-treaty stage of a Group of Governmental Experts in 2017.
- **SURVEILLANCE AND PERSUASION:** While it is expensive, tedious, and sometimes legally questionable for security personnel to monitor phone lines, video camera feeds, emails, and other messaging channels, AI (speech recognition, computer vision, and

natural language understanding) can be used in a scalable fashion to perform mass surveillance of individuals and detect activities of interest. By tailoring information flows to individuals through social media, based on machine learning techniques, political behavior can be modified and controlled to some extent—a concern that became apparent in elections beginning in 2016.

- **BIASED DECISION MAKING:** Careless or deliberate misuse of machine learning algorithms for tasks such as evaluating parole and loan applications can result in decisions that are biased by race, gender, or other protected categories. Often, the data themselves reflect pervasive bias in society.
- **IMPACT ON EMPLOYMENT:** Concerns about machines eliminating jobs are centuries old. The story is never simple: machines do some of the tasks that humans might otherwise do, but they also make humans more productive and therefore more employable, and make companies more profitable and therefore able to pay higher wages. They may render some activities economically viable that would otherwise be impractical. Their use generally results in increasing wealth but tends to have the effect of shifting wealth from labor to capital, further exacerbating increases in inequality. Previous advances in technology—such as the invention of mechanical looms—have resulted in serious disruptions to employment, but eventually people find new kinds of work to do. On the other hand, it is possible that AI will be doing those new kinds of work too. This topic is rapidly becoming a major focus for economists and governments around the world.
- **SAFETY-CRITICAL APPLICATIONS:** As AI techniques advance, they are increasingly used in high-stakes, safety-critical applications such as driving cars and managing the water supplies of cities. Fatal accidents have already occurred and highlight the difficulty of formal verification and statistical risk analysis for systems developed using machine learning techniques. The field of AI will need to develop technical and ethical standards at least comparable to those prevalent in other engineering and healthcare disciplines where people’s lives are at stake.
- **CYBERSECURITY:** AI techniques are useful in defending against cyberattack, for example by detecting unusual patterns of behavior, but they will also contribute to the potency, survivability, and proliferation capability of malware. For example, reinforcement learning methods have been used to create highly effective tools for automated, personalized blackmail and phishing attacks.

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

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

Human-level AI

Artificial general intelligence (AGI)

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.

Artificial superintelligence (ASI)

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

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

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

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.

[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!"

Gorilla problem

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]

King Midas problem

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.

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

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

Assistance game

Inverse reinforcement learning

Summary

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

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

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

Bibliographical and Historical Notes

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

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

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

Chapter 2
Intelligent Agents

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

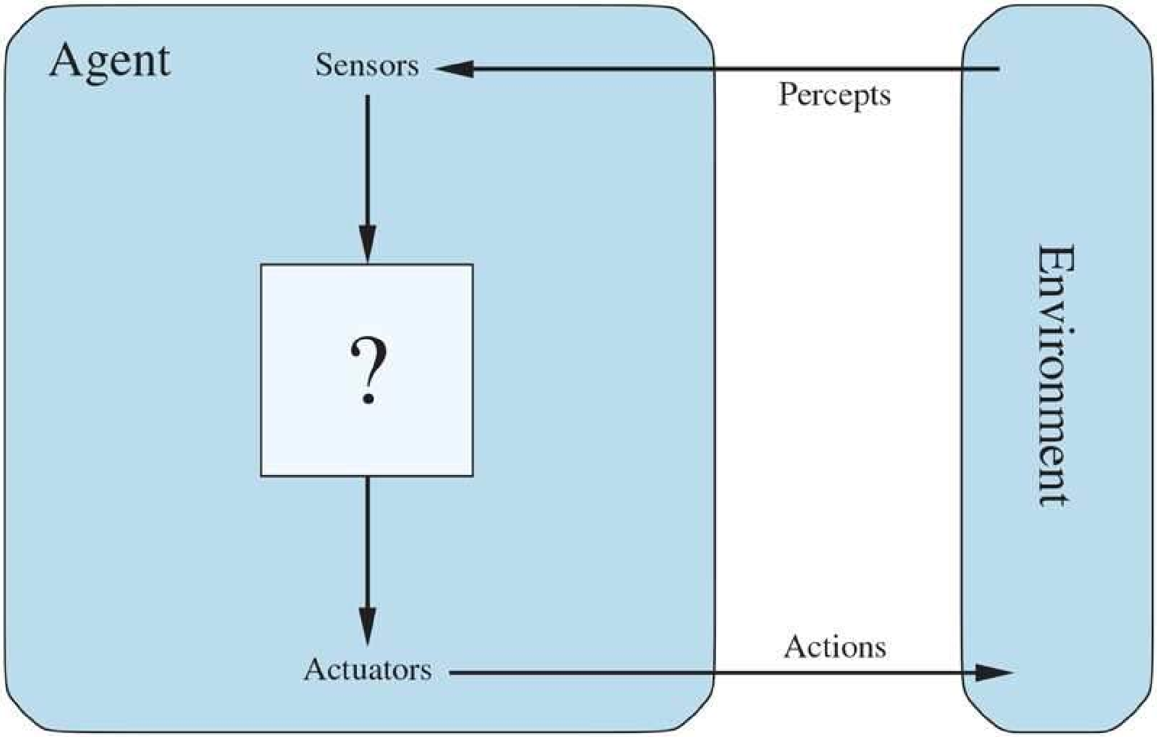
Chapter 1 identified the concept of **rational agents** as central to our approach to artificial intelligence. In this chapter, we make this notion more concrete. We will see that the concept of rationality can be applied to a wide variety of agents operating in any imaginable environment. Our plan in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

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

2.1 Agents and Environments

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.

Environment

Sensor

Actuator

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.

Percept

Percept sequence

Agent function

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.

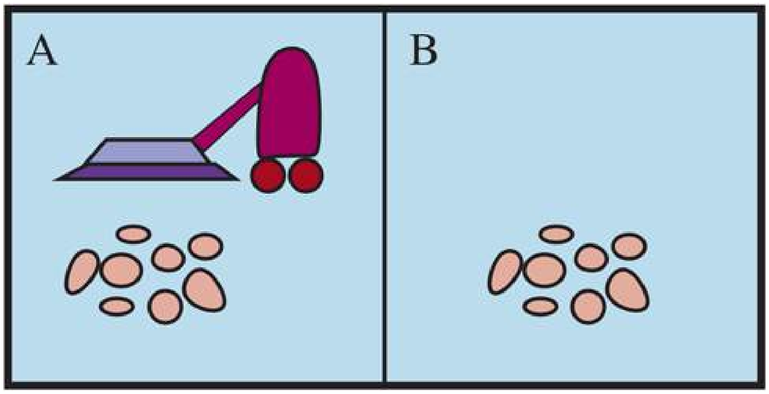
[1] If the agent uses some randomization to choose its actions, then we would have to try each sequence many times to identify the probability of each action. One might imagine that acting randomly is rather silly, but we show later in this chapter that it can be very intelligent.

Agent program

To illustrate these ideas, we use a simple example—the vacuum-cleaner world, which consists of a robotic vacuum-cleaning agent in a world consisting of squares that can be either dirty or clean. Figure 2.2 shows a configuration with just two squares, A and B. The vacuum agent perceives which square it is in and whether there is dirt in the square. The agent starts in square A. The available actions are to move to the right, move to the left, suck up the dirt, or do nothing.[2] One very simple agent function is the following: if the current square is dirty, then suck; otherwise, move to the other square. A partial tabulation of this agent function is shown in Figure 2.3 and an agent program that implements it appears in Figure 2.8 on page 49.

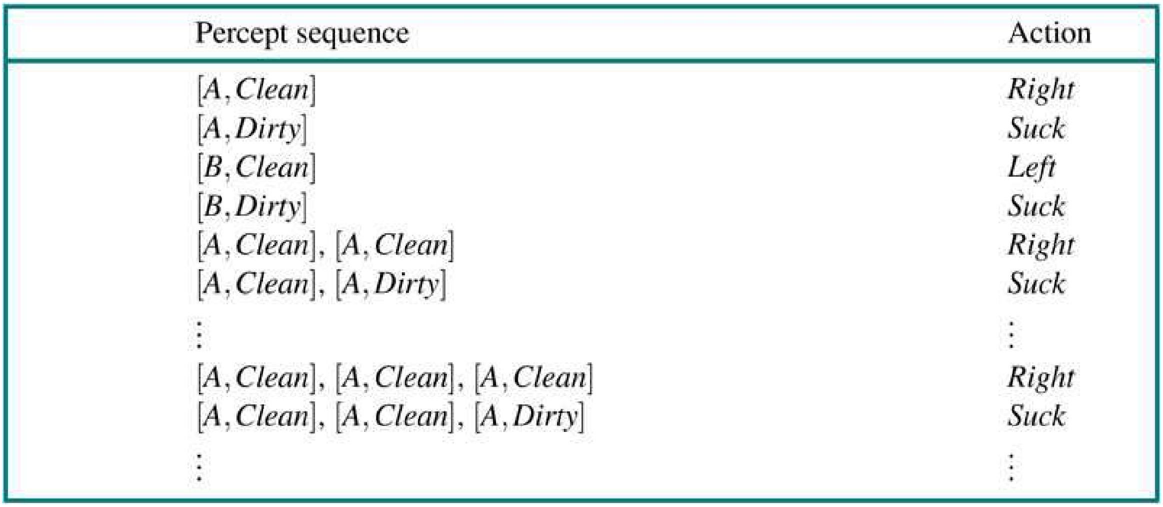
[2] In a real robot, it would be unlikely to have an actions like “move right” and “move left.” Instead the actions would be “spin wheels forward” and “spin wheels backward.” We have chosen the actions to be easier to follow on the page, not for ease of implementation in an actual robot.

Figure 2.2



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

Figure 2.3



Percept sequence	Action
$[A, \mathit{Clean}]$	*Right*
$[A, \mathit{Dirty}]$	*Suck*
$[B, \mathit{Clean}]$	*Left*
$[B, \mathit{Dirty}]$	*Suck*
$[A, \mathit{Clean}]$, $[A, \mathit{Clean}]$	*Right*
$[A, \mathit{Clean}]$, $[A, \mathit{Dirty}]$	*Suck*
⋮	⋮
$[A, \mathit{Clean}]$, $[A, \mathit{Clean}]$, $[A, \mathit{Clean}]$	*Right*
$[A, \mathit{Clean}]$, $[A, \mathit{Clean}]$, $[A, \mathit{Dirty}]$	*Suck*
⋮	⋮

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.

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.

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

2.2 Good Behavior: The Concept of Rationality

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

Rational agent

2.2.1 Performance measures

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

Consequentialism

Performance measure

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

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

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

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

2.2.2 Rationality

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

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

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.

Definition of a rational agent

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

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

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

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

2.2.3 Omniscience, learning, and autonomy

Omniscience

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

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

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

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

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

Information gathering

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

Learning

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

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

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.

Autonomy

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.

Task environment

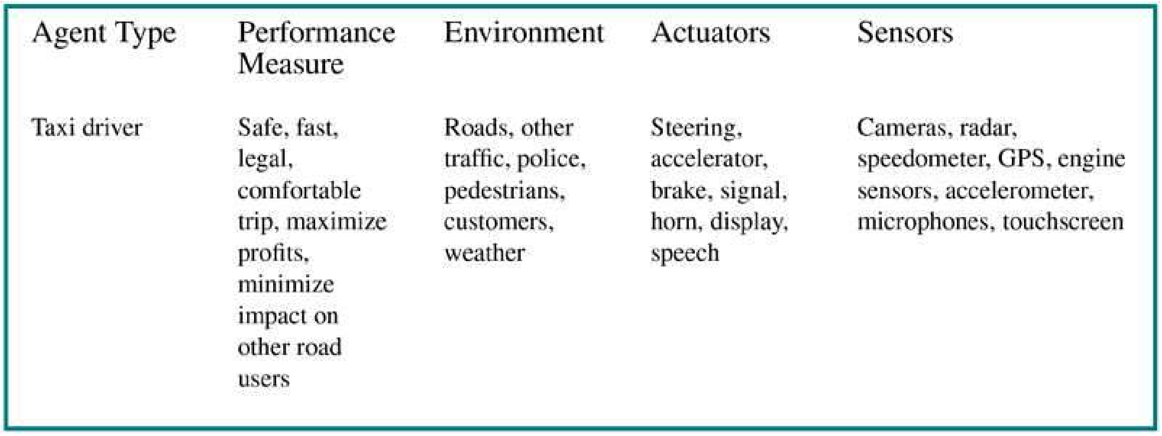
2.3.1 Specifying the task environment

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.

PEAS

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.

Figure 2.4



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

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.

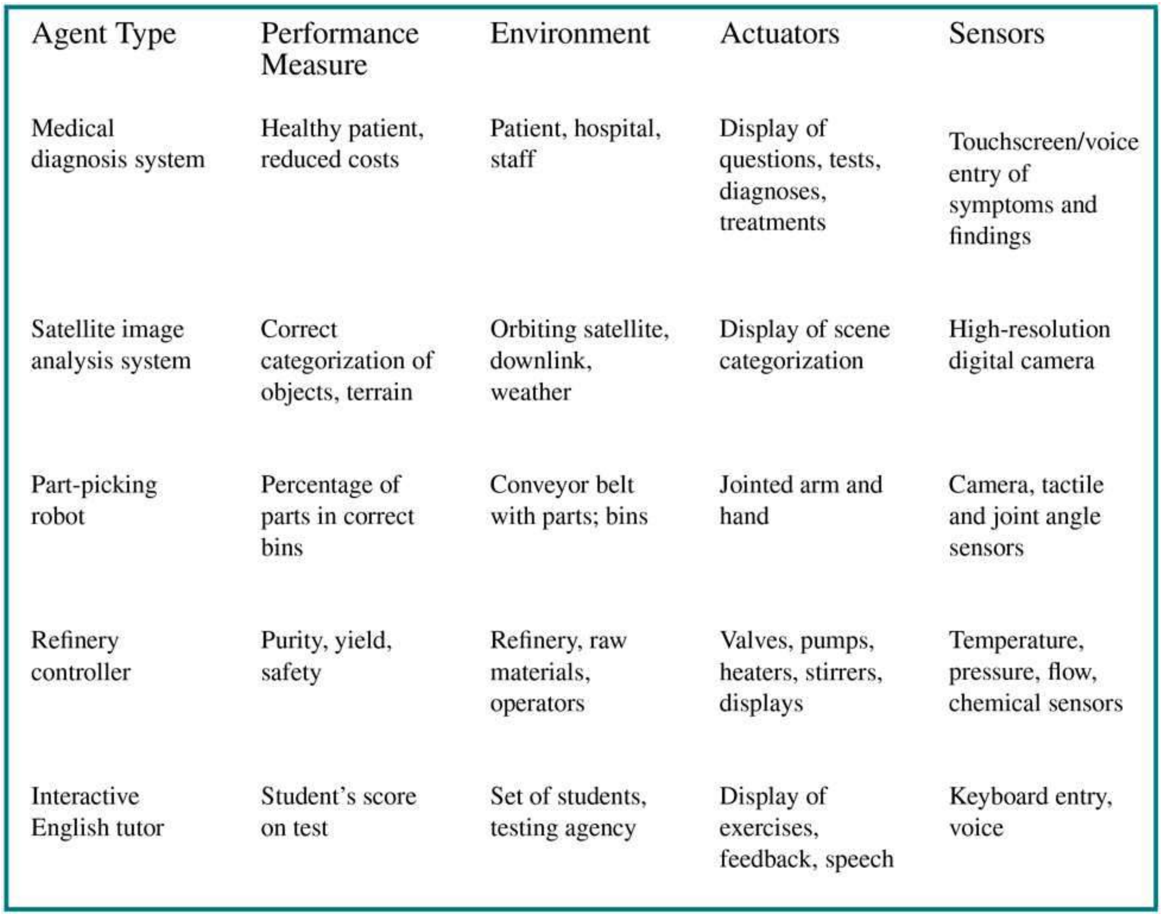
Next, what is the driving **environment** that the taxi will face? Any taxi driver must deal with a variety of roads, ranging from rural lanes and urban alleys to 12-lane freeways. The roads contain other traffic, pedestrians, stray animals, road works, police cars, puddles, and potholes. The taxi must 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.

Figure 2.5



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

Examples of agent types and their PEAS descriptions.

Software agent

Softbot

2.3.2 Properties of task environments

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

Fully observable

Partially observable

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.

Unobservable

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.

Single-agent

Multiagent

For example, in chess, the opponent entity B is trying to maximize its performance measure, which, by the rules of chess, minimizes agent A's performance measure. Thus, chess is a **competitive** multiagent environment. On the other hand, in the taxi-driving environment, avoiding collisions maximizes the performance measure of all agents, so it is a partially

cooperative multiagent environment. It is also partially competitive because, for example, only one car can occupy a parking space.

Competitive

Cooperative

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

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

Deterministic

Nondeterministic

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

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

Stochastic

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

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

Episodic

Sequential

Static

Dynamic

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.

Semidynamic

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.

Discrete

Continuous

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.

Known

Unknown

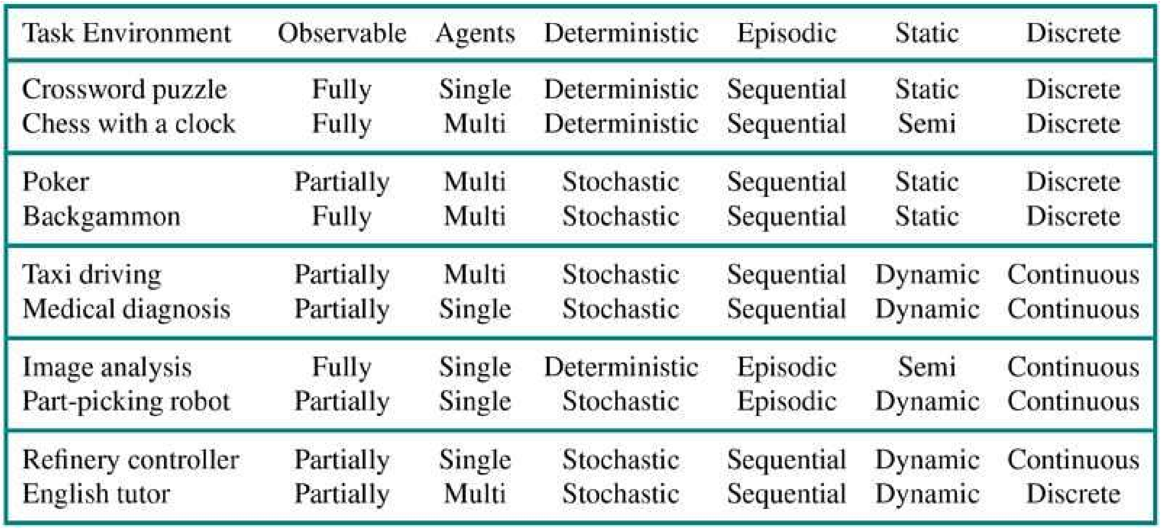
The distinction between known and unknown environments is not the same as the one between fully and partially observable environments. It is quite possible for a *known* environment to be *partially* observable—for example, in solitaire card games, I know the rules but am still unable to see the cards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

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

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

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

Figure 2.6



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

Examples of task environments and their characteristics.

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

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

Environment class

2.4 The Structure of Agents

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

$$agent = architecture + program\ .$$

Agent program

Agent architecture

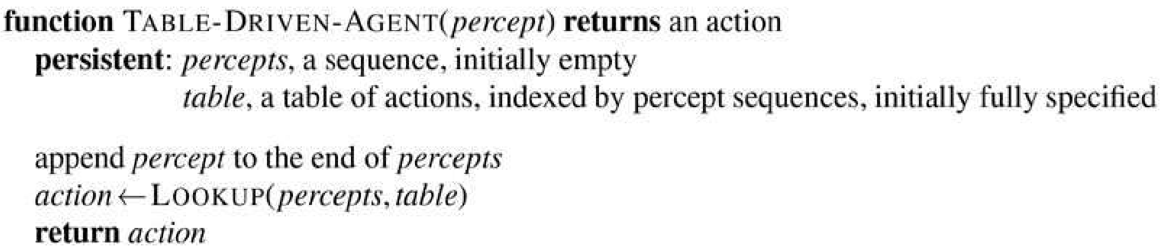
Obviously, the program we choose has to be one that is appropriate for the architecture. If the program is going to recommend actions like *Walk*, the architecture had better have legs. The architecture might be just an ordinary PC, or it might be a robotic car with several onboard computers, cameras,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

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.

Figure 2.7



```
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
```

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 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 $\sum_{t=1}^{T} \left|P\right|^{t}$ 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

[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.

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:

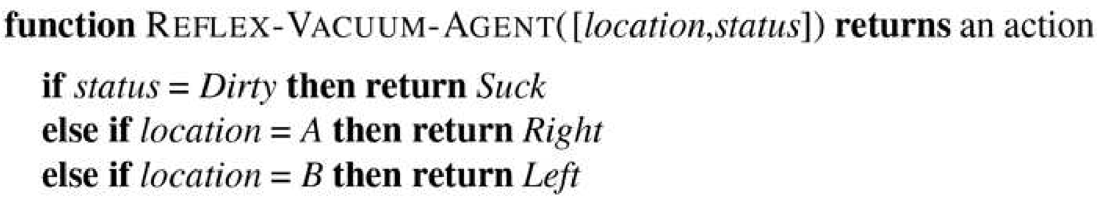
- Simple reflex agents;
- Model-based reflex agents;
- Goal-based agents; and
- Utility-based agents.

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.

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
```

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.

Simple reflex agent

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.*

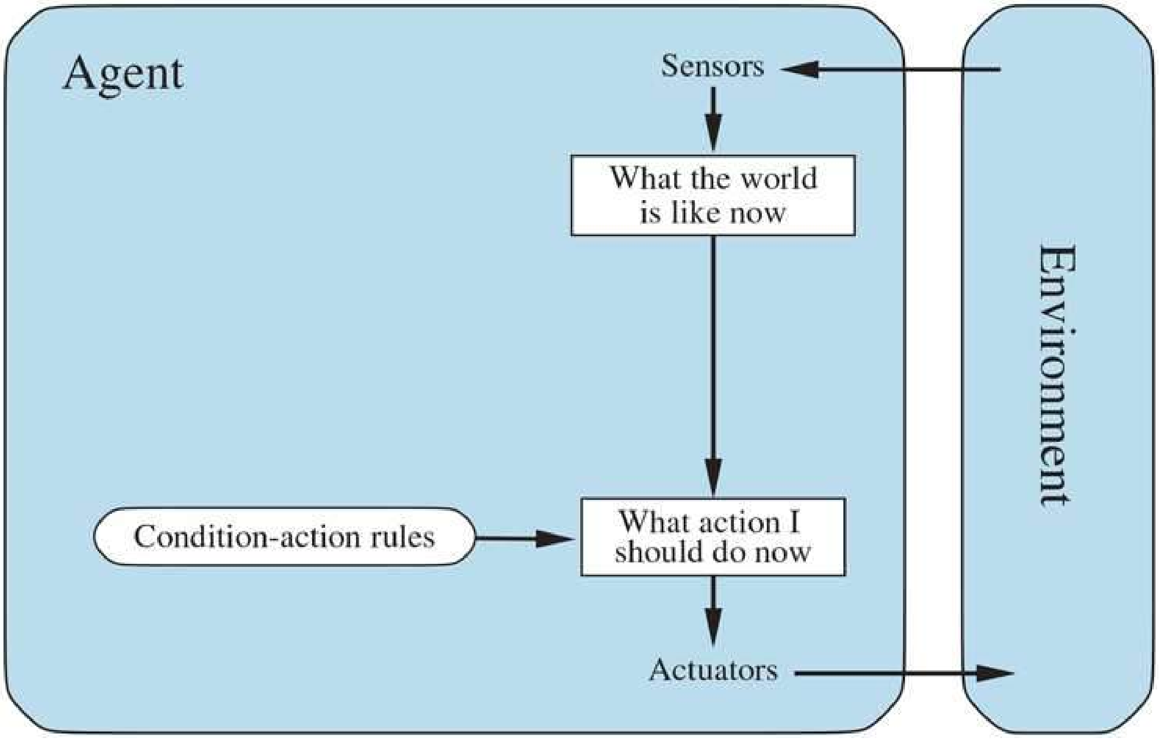
[6] Also called **situation–action rules**, **productions**, or **if–then rules**.

condition–action rule

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.

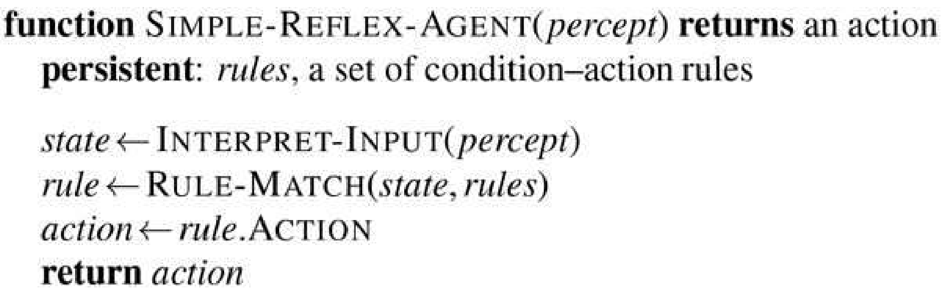
Figure 2.9



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 21).

Figure 2.10



```
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
```

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.

Randomization

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.

Internal state

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

Transition model

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

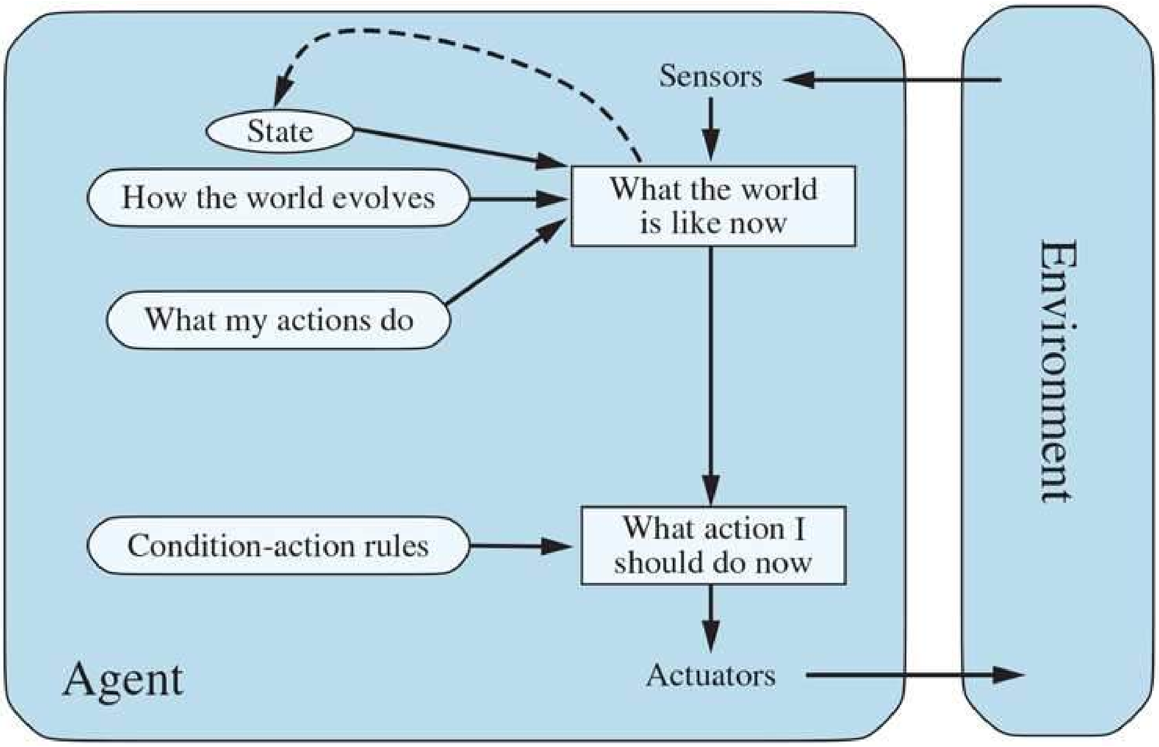
Sensor model

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

Model-based agent

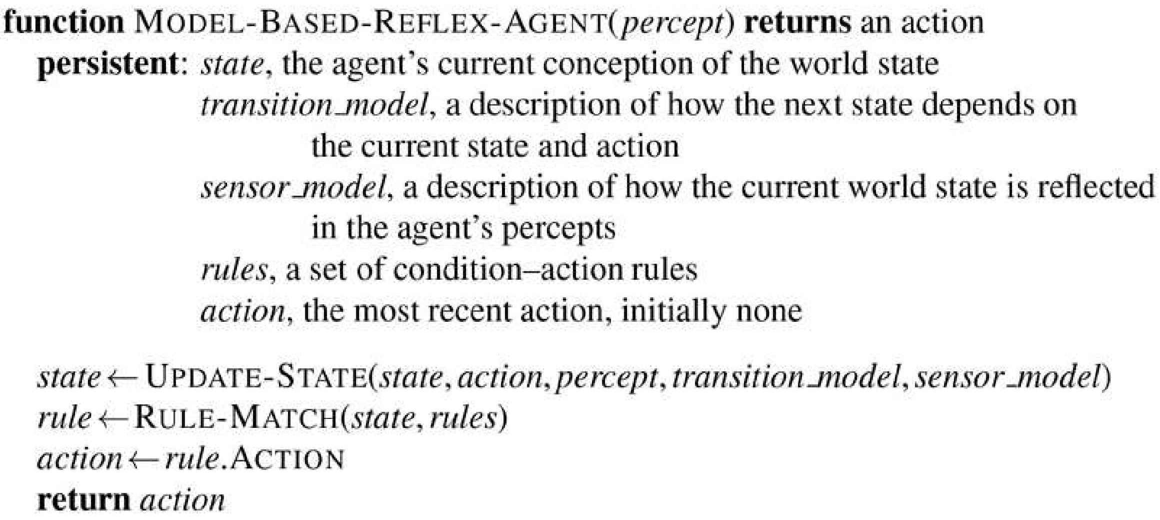
Figure 2.11 gives the structure of the model-based reflex agent with internal state, showing how the current percept is combined with the old internal state to generate the updated description of the current state, based on the agent's model of how the world works. The agent program is shown in Figure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

Figure 2.11



A model-based reflex agent.

Figure 2.12



```
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
```

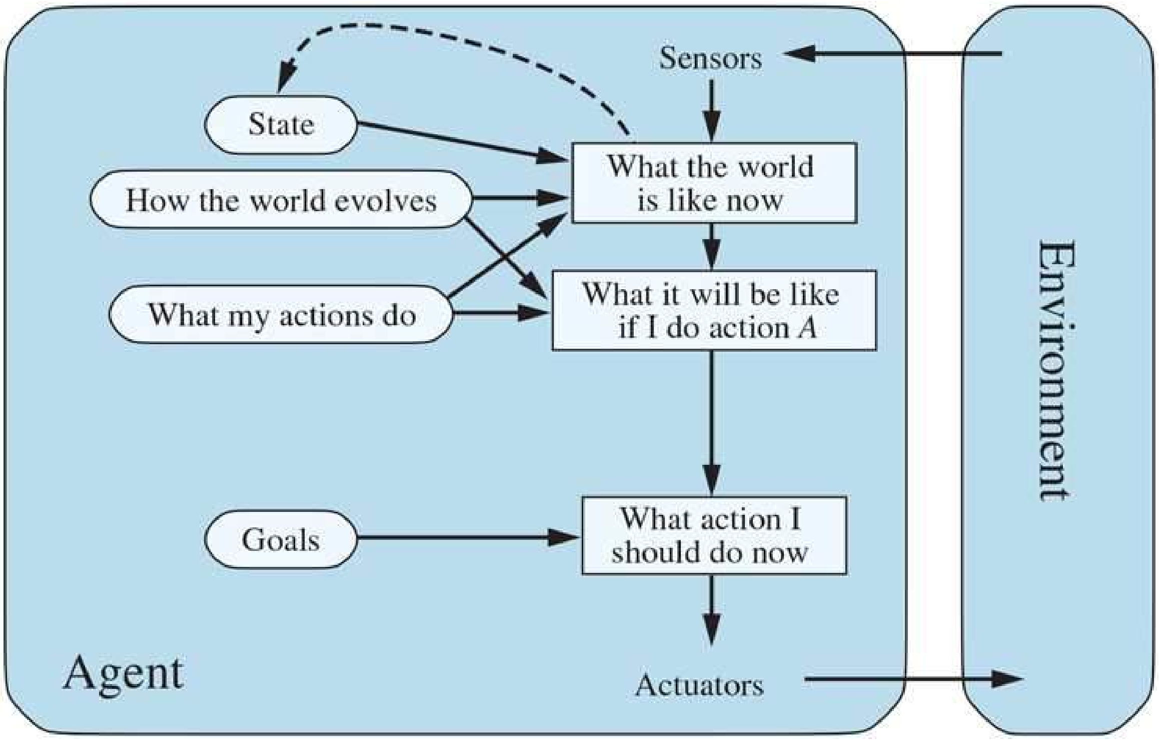
A model-based reflex agent. It keeps track of the current state of the world, using an internal model. It then chooses an action in the same way as the reflex agent.

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.

Goal

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

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

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

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

Utility

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

Utility function

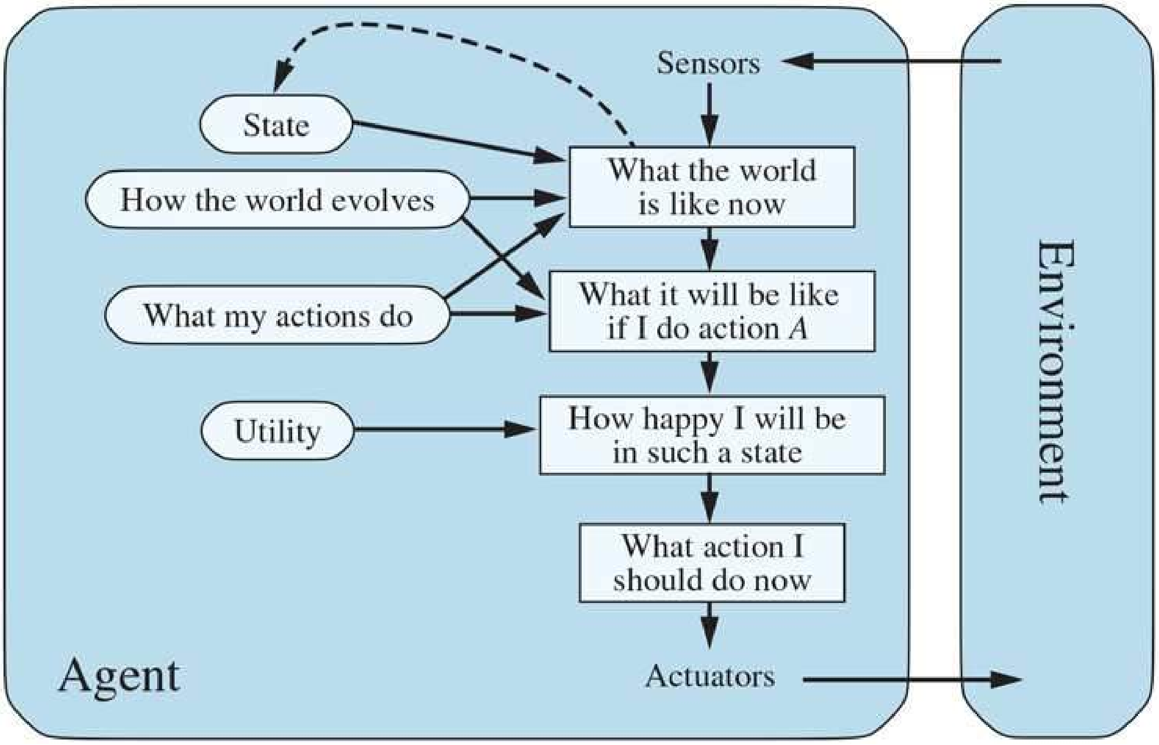
Let us emphasize again that this is not the *only* way to be rational—we have already seen a rational agent program for the vacuum world (Figure 2.8) that has no idea what its utility function is—but, like goal-based agents, a utility-based agent has many advantages in terms of flexibility and learning. Furthermore, in two kinds of cases, goals are inadequate but a utility-based agent can still make rational decisions. First, when there are conflicting goals, only some of which can be achieved (for example, speed and safety), the utility function specifies the appropriate tradeoff. Second, when there are several goals that the agent can aim for, none of which can be achieved with certainty, utility provides a way in which the likelihood of success can be weighed against the importance of the goals.

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

Expected utility

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

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 22 and 26 that a **model-free agent** can learn what action is best in a particular situation without ever learning exactly how that action changes the environment.

Model-free agent

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

2.4.6 Learning agents

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

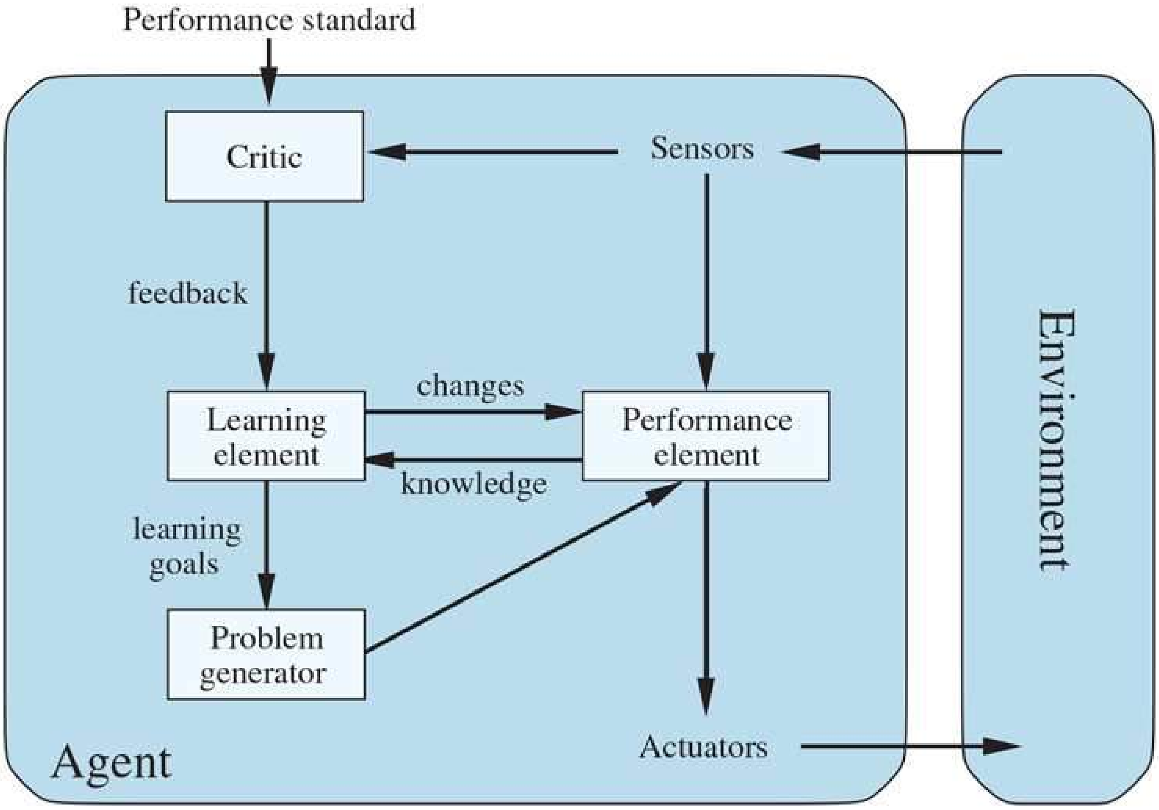
Learning has another advantage, as we noted earlier: it allows the agent to operate in initially unknown environments and to become more competent than its initial knowledge alone might allow. In this section, we briefly introduce the main ideas of learning agents. Throughout the book, we comment on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

Learning element

Performance element

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.

Critic

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

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

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

Problem generator

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

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

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

Reward

Penalty

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

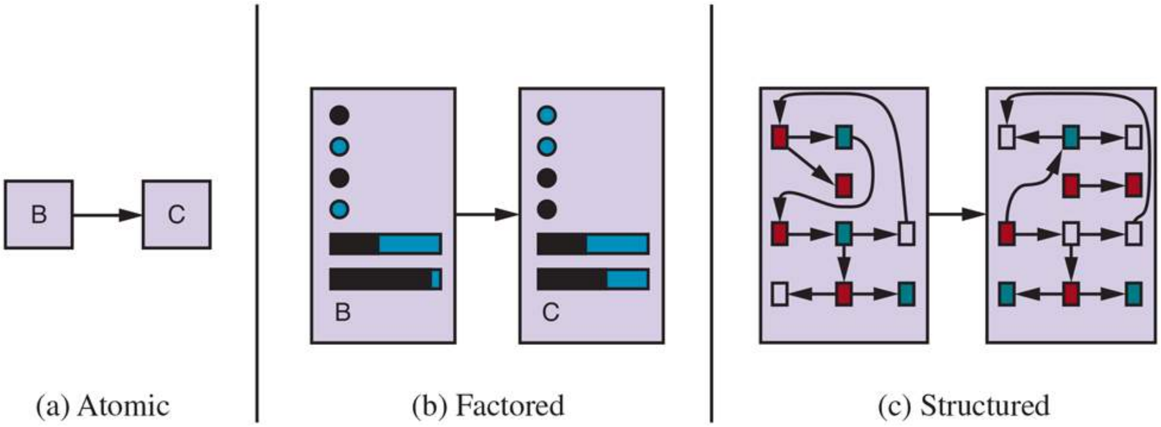
In summary, agents have a variety of components, and those components can be represented in many ways within the agent program, so there appears to be great variety among learning methods. There is, however, a single unifying theme. Learning in intelligent agents can be summarized as a process of modification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

2.4.7 How the components of agent programs work

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

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

Figure 2.16



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

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

Atomic representation

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

Factored representation

Variable

Attribute

Value

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

Structured representation

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.

Expressiveness

Localist representation

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

Distributed representation

Summary

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

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

Bibliographical and Historical Notes

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

Controller

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

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

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

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

Autonomic computing

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

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

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

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

II Problem-solving

Chapter 3
Solving Problems by Searching

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

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

Problem-solving agent

Search

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

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

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

3.1 Problem-Solving Agents

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

[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.

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

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

Goal formulation

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

Problem formulation

- **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.

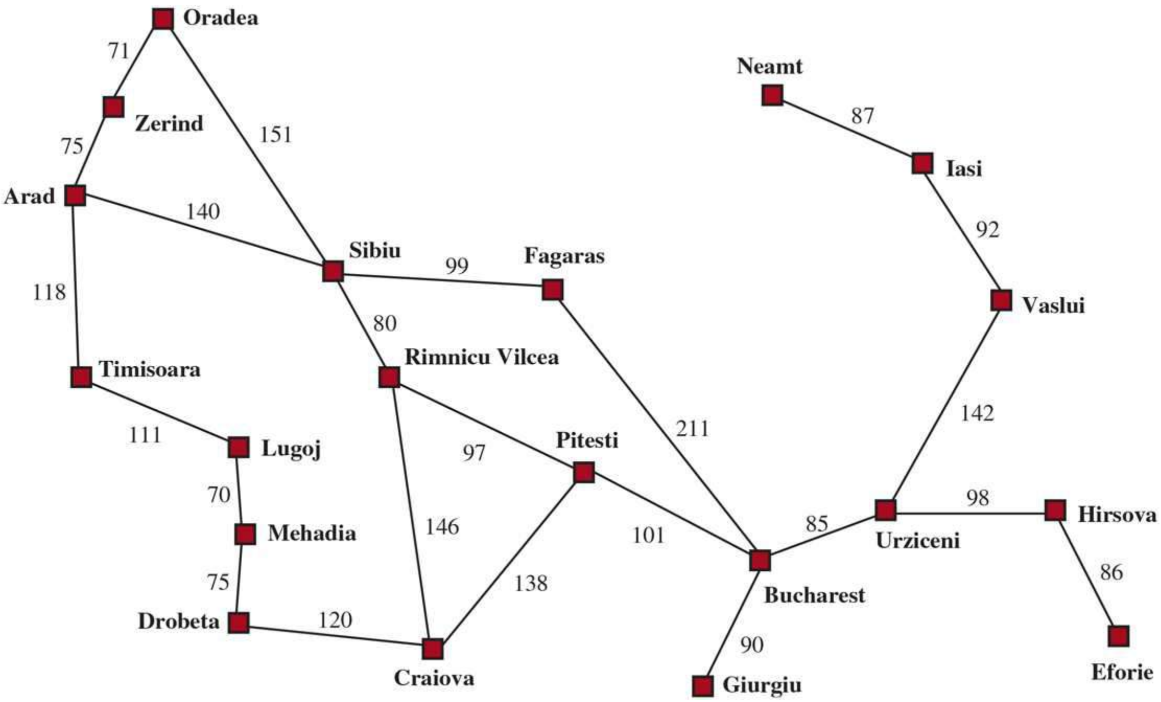
Search

Solution

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

Execution

Figure 3.1



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

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

Open-loop

Closed-loop

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

3.1.1 Search problems and solutions

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

Problem

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

States

State space

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

Initial state

- A set of one or more **goal states**. Sometimes there is one goal state (e.g., *Bucharest*), sometimes there is a small set of alternative goal states, and sometimes the goal is defined by a property that applies to many states (potentially an infinite number). For example, in a vacuum-cleaner world, the goal might be to have no dirt in any location, regardless of any other facts about the state. We can account for all three of these possibilities by specifying an Is-Goal method for a problem. In this chapter we will sometimes say “the goal” for simplicity, but what we say also applies to “any one of the possible goal states.”

Goal states

- The **actions** available to the agent. Given a state s, Actions(s) returns a finite[2] set of actions that can be executed in s. We say that each of these actions is **applicable** in s. An example:

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

Action

Applicable

- 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,

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

Transition model

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

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

Action cost function

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

[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°; the algorithms in this chapter can handle this as long as no more than three consecutive 90° 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.

Path

Optimal solution

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

Graph

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.

Abstraction

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

Level of abstraction

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.

[4] See Section **11.4**.

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.

Standardized problem

Real-world problem

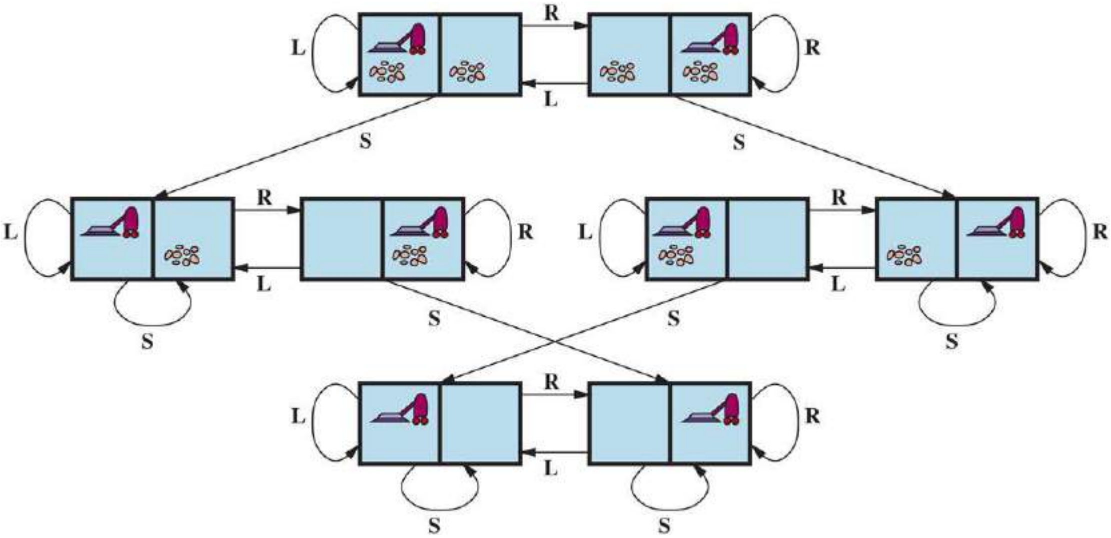
3.2.1 Standardized problems

Grid world

A **grid world** problem is a two-dimensional rectangular array of square cells in which agents can move from cell to cell. Typically the agent can move to any obstacle-free adjacent cell—horizontally or vertically and in some problems diagonally. Cells can contain objects, which the agent can pick up, push, or otherwise act upon; a wall or other impassible obstacle in a cell prevents an agent from moving into that cell. The **vacuum world** from Section 2.1 can be formulated as a grid world problem as follows:

- **STATES:** A state of the world says which objects are in which cells. For the vacuum world, the objects are the agent and any dirt. In the simple two-cell version, the agent can be in either of the two cells, and each call can either contain dirt or not, so there are $2 \cdot 2 \cdot 2 = 8$ states (see Figure 3.2). In general, a vacuum environment with n cells has $n \cdot 2^n$ states.

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 = $Suck$.

- **INITIAL STATE:** Any state can be designated as the initial state.
- **ACTIONS:** In the two-cell world we defined three actions: *Suck*, move *Left*, and move *Right*. In a two-dimensional multi-cell world we need more movement actions. We could add *Upward* and *Downward*, giving us four **absolute** movement actions, or we could switch to **egocentric actions**, defined relative to the viewpoint of the agent—for example, *Forward*, *Backward*, *TurnRight*, and *TurnLeft*.
- **TRANSITION MODEL:** *Suck* removes any dirt from the agent's cell; *Forward* moves the agent ahead one cell in the direction it is facing, unless it hits a wall, in which case the action has no effect. *Backward* moves the agent in the opposite direction, while *TurnRight* and *TurnLeft* change the direction it is facing by 90°.
- **GOAL STATES:** The states in which every cell is clean.
- **ACTION COST:** Each action costs 1.

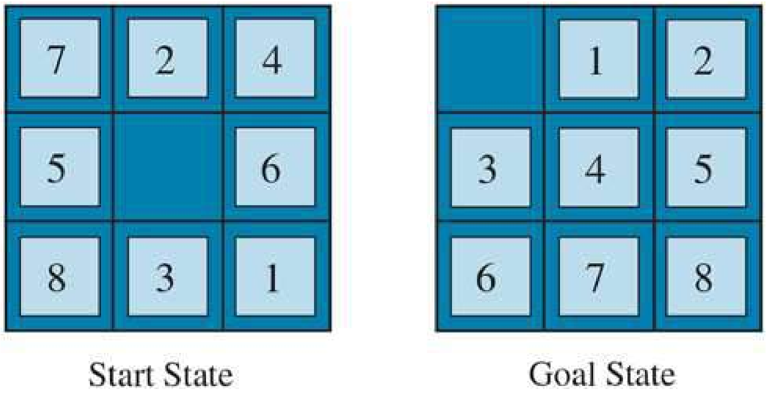
Sokoban puzzle

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

- **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.

Figure 3.3



A typical instance of the 8-puzzle.

- **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.

Sliding-tile puzzle

8-puzzle

15-puzzle

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.

Touring problem

Traveling salesperson problem (TSP)

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.

VLSI layout

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.

Robot navigation

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.

Automatic assembly sequencing

Protein design

3.3 Search Algorithms

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

Search algorithm

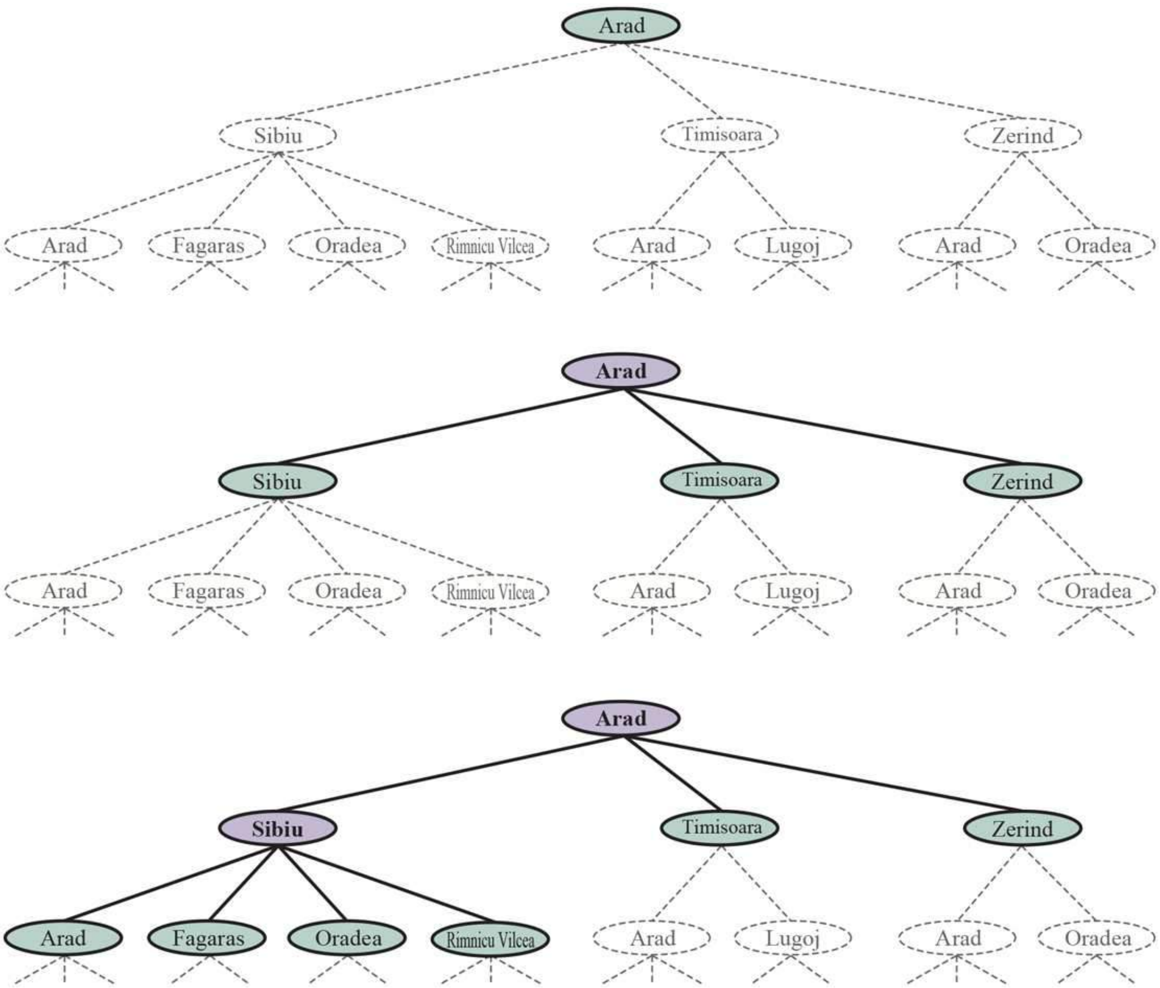
Node

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

Expand

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.

Generating

Child node

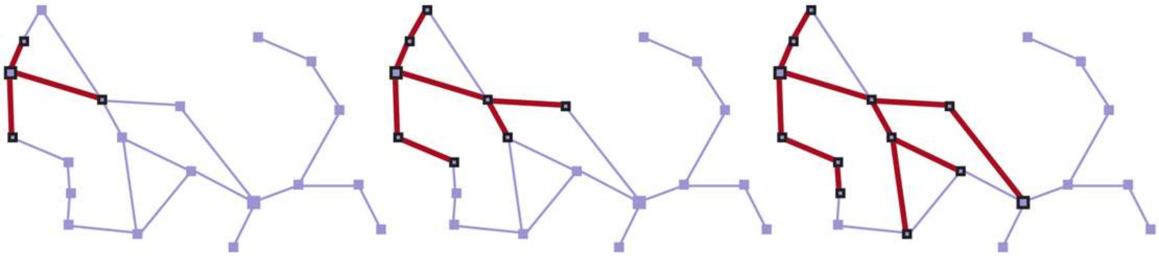
Successor node

Parent node

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

[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*.

Figure 3.5



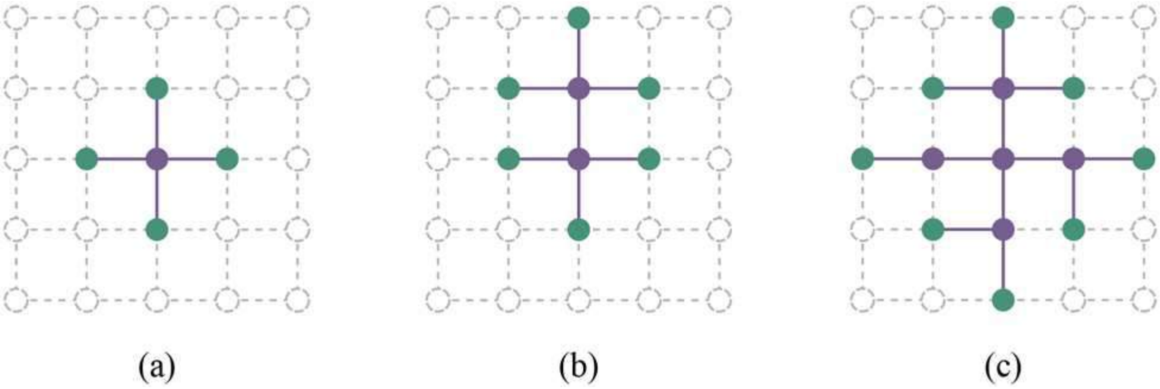
A sequence of search trees generated by a graph search on the Romania problem of Figure 3.1. At each stage, we have expanded every node on the frontier, extending every path with all applicable actions that don’t result in a state that has already been reached. Notice that at the third stage, the topmost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

Frontier

Reached

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

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.

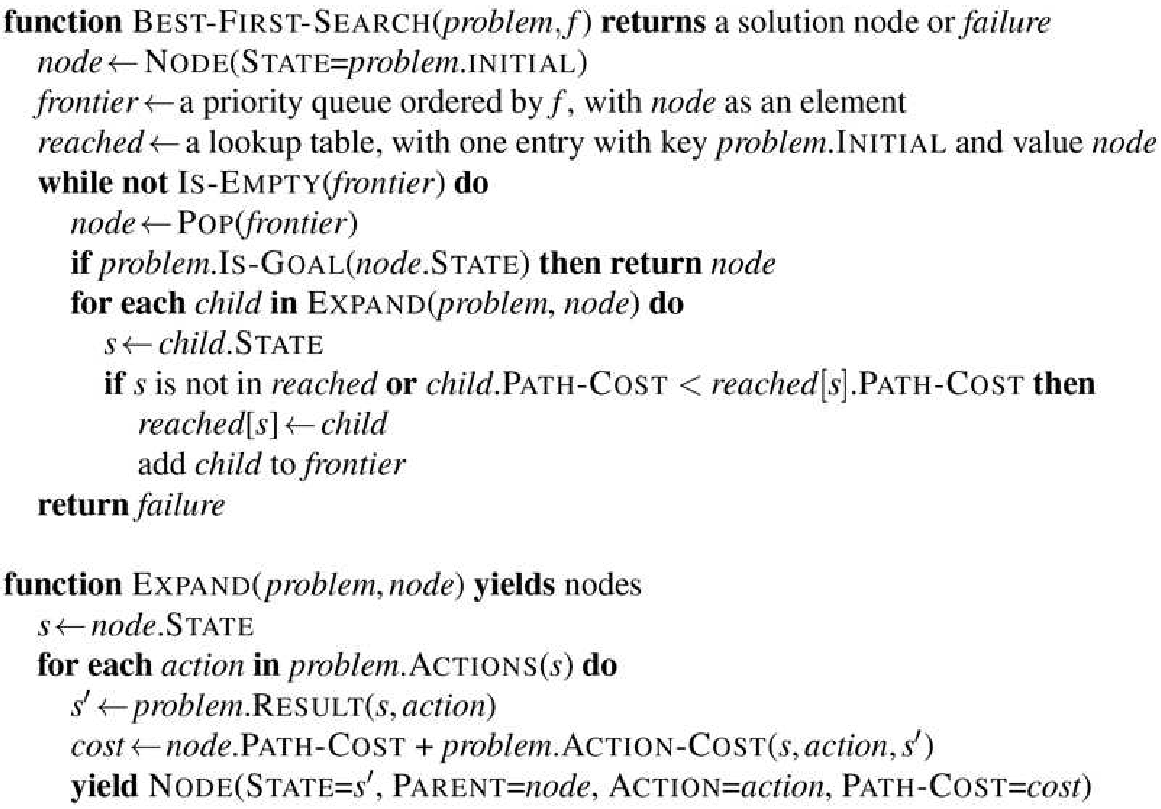
Separator

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.

Figure 3.7



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

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

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.

Best-first search

Evaluation function

3.3.2 Search data structures

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

- *node*.State: the state to which the node corresponds;
- *node*.Parent: the node in the tree that generated this node;
- *node*.Action: the action that was applied to the parent's state to generate this node;
- *node*.Path-Cost: the total cost of the path from the initial state to this node. In mathematical formulas, we use *g(node)* as a synonym for Path-Cost.

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

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.

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.

Priority queue

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

FIFO queue

- 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.

LIFO queue

Stack

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.

Repeated state

Cycle

Loopy path

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

Redundant path

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

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

Graph search

Tree-like search

Second, we can not worry about repeating the past. There are some problem formulations where it is rare or impossible for two paths to reach the same state. An example would be an assembly problem where each action adds a part to an evolving assemblage, and there is an ordering of parts so that it is possible to add A and then B, but not B and then A. For those problems, we could save memory space if we *don't* track reached states and we don't check for redundant paths. We call a search algorithm a **graph search** if it checks for redundant paths and a **tree-like search**[6] if it does not check. The Best-First-Search algorithm in Figure 3.7 is a graph search algorithm; if we remove all references to *reached* we get a tree-like search that uses less memory but will examine redundant paths to the same state, and thus will run slower.

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

3.3.4 Measuring problem-solving performance

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

[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.

- **COMPLETENESS:** Is the algorithm guaranteed to find a solution when there is one, and to correctly report failure when there is not?

Completeness

- **COST OPTIMALITY:** Does it find a solution with the lowest path cost of all solutions?[7]

Cost optimality

- **TIME COMPLEXITY:** How long does it take to find a solution? This can be measured in seconds, or more abstractly by the number of states and actions considered.

Time complexity

- **SPACE COMPLEXITY:** How much memory is needed to perform the search?

Space complexity

To understand completeness, consider a search problem with a single goal. That goal could be anywhere in the state space; therefore a complete algorithm must be capable of systematically exploring every state that is reachable from the initial state. In finite state

[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.

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.

Systematic

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.

Depth

Branching factor

3.4 Uninformed Search Strategies

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

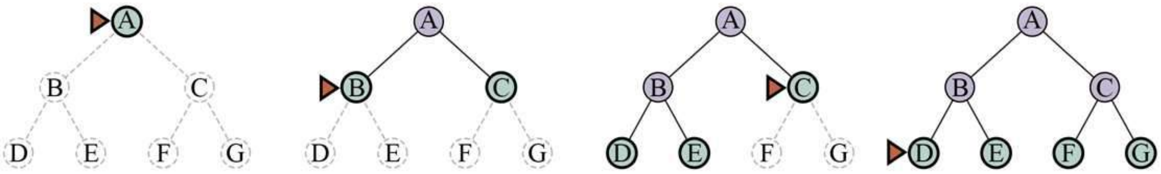
3.4.1 Breadth-first search

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.

Breadth-first search

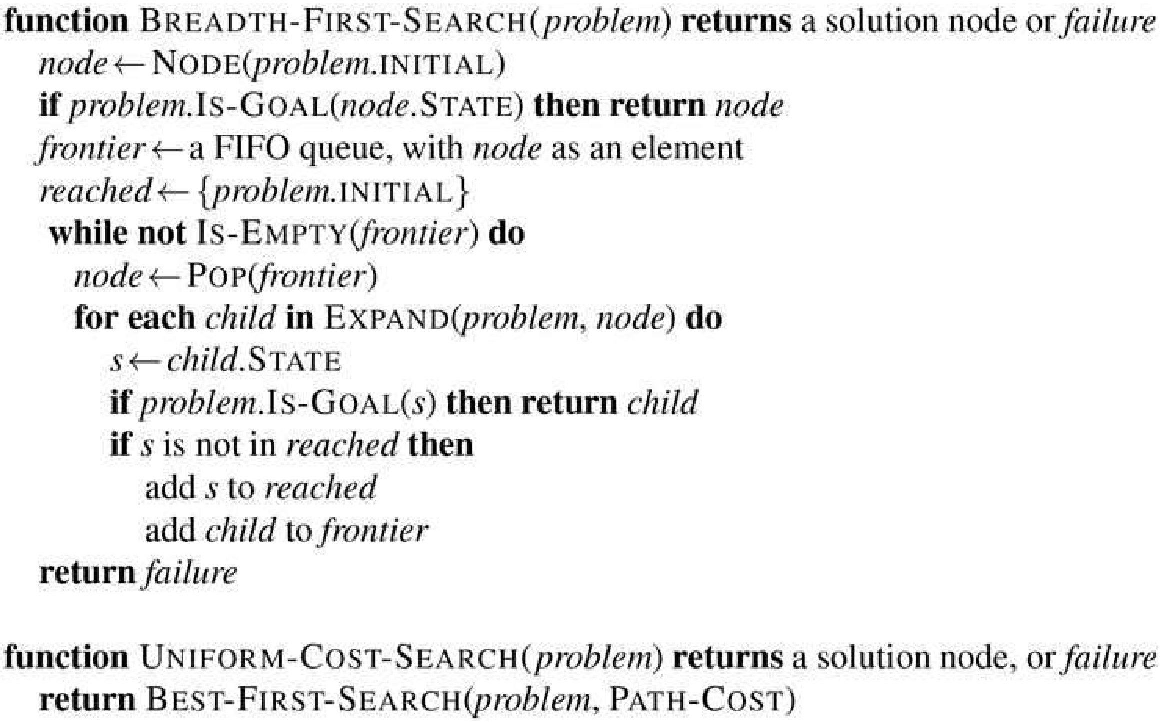
However, we can get additional efficiency with a couple of tricks. A first-in-first-out queue will be faster than a priority queue, and will give us the correct order of nodes: new nodes (which are always deeper than their parents) go to the back of the queue, and old nodes, which are shallower than 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.

Figure 3.9



```
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)
```

Breadth-first search and uniform-cost search algorithms.

Early goal test

Late goal test

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

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

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

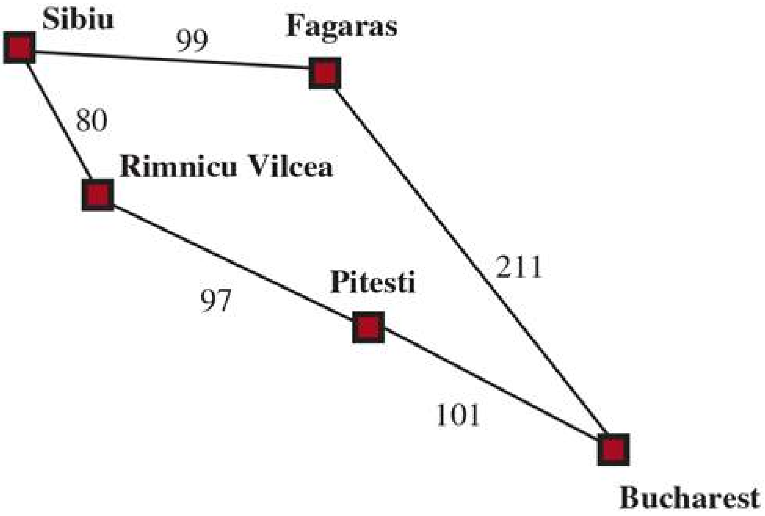
3.4.2 Dijkstra's algorithm or uniform-cost search

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

Uniform-cost search

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.

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

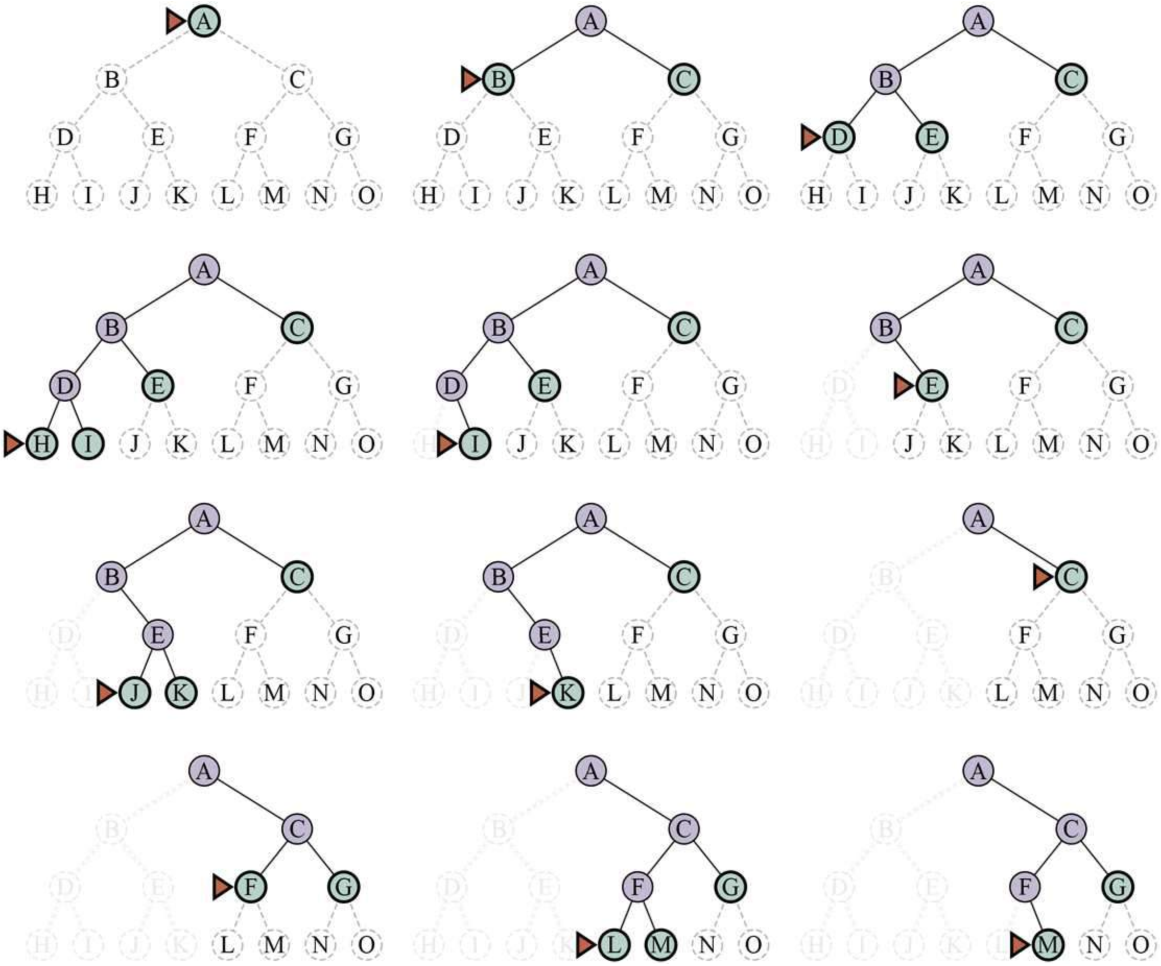
Uniform-cost search is complete and is cost-optimal, because the first solution it finds will have a cost that is at least as low as the cost of any other node in the frontier. Uniform-cost search considers all paths systematically in order of increasing cost, never getting caught going down a single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

3.4.3 Depth-first search and the problem of memory

Depth-first search

Depth-first search always expands the *deepest* node in the frontier first. It could be implemented as a call to BEST-FIRST-SEARCH where the evaluation function f is the negative of the depth. However, it is usually implemented not as a graph search but as a tree-like search that does not keep a table of reached states. The progress of the search is illustrated in Figure 3.11; search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. The search then “backs up” to the next deepest node that still has 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 6), propositional satisfiability (Chapter 7), and logic programming (Chapter 9).

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

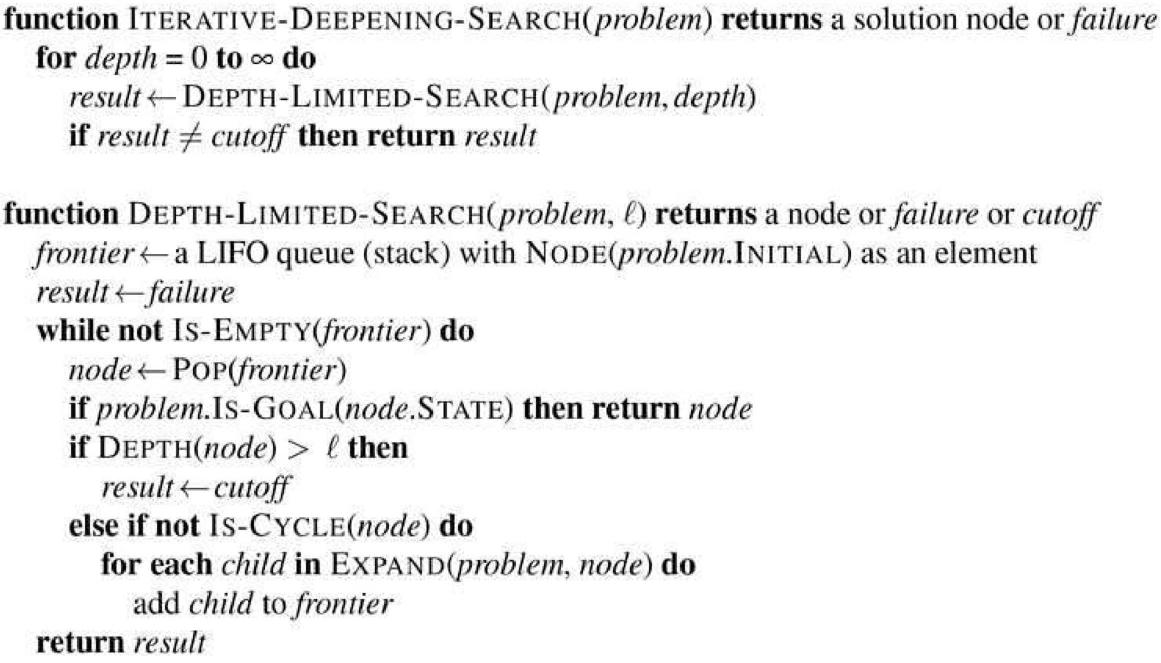
Backtracking search

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

$O(b^{\ell})$ and the space complexity is $O(b\ell)$. Unfortunately, if we make a poor choice for ℓ the algorithm will fail to reach the solution, making it incomplete again.

Figure 3.12



```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution node or failure
  for depth = 0 to $\infty$ do
    result $\leftarrow$ DEPTH-LIMITED-SEARCH(problem, depth)
    if result $\neq$ cutoff then return result

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

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.

Depth-limited search

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

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

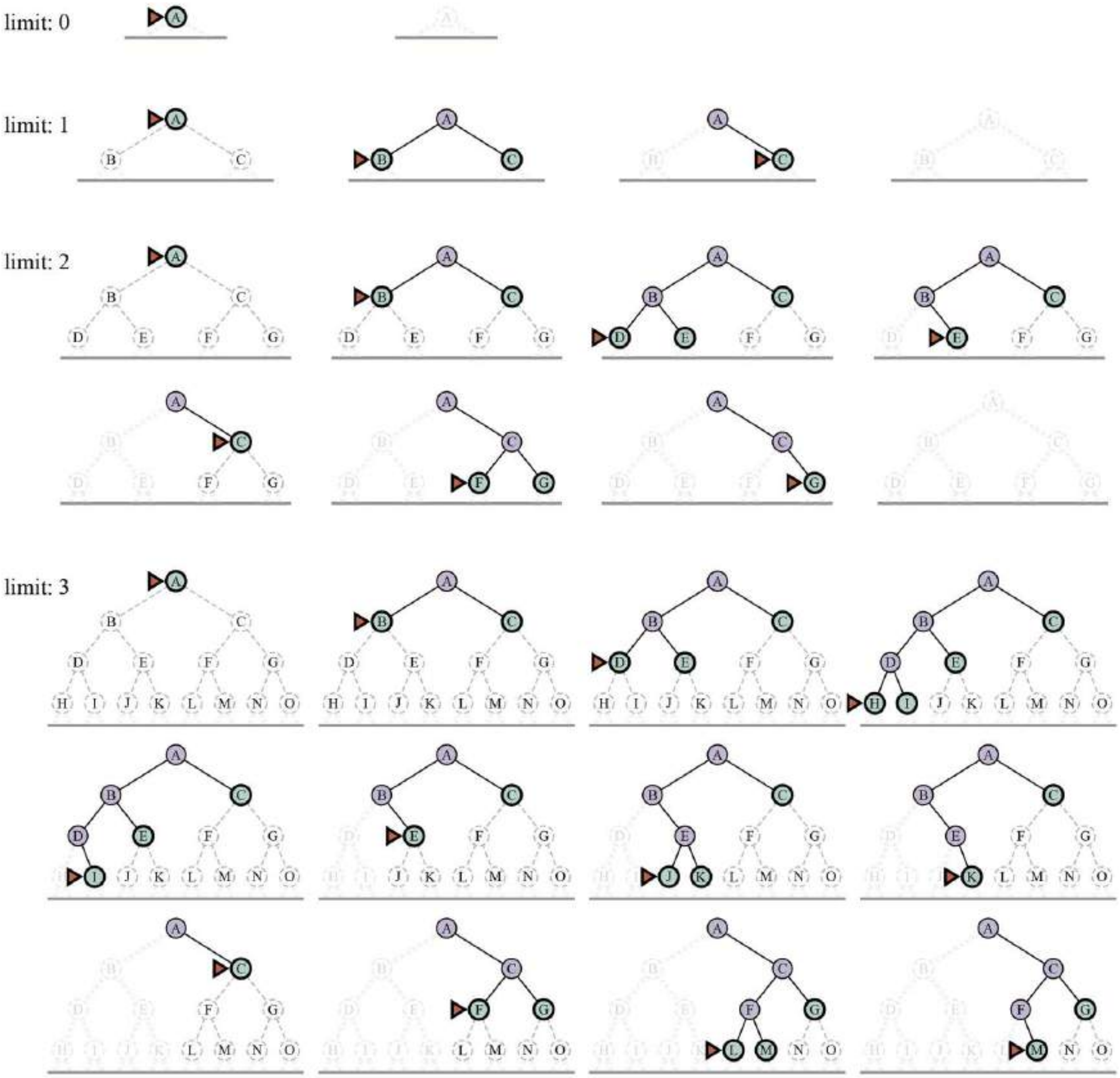
Diameter

Iterative deepening search 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.

Iterative deepening search

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 $\mathbf{M}$ 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

$$N(\mathrm{IDS}) \quad = \quad (d)b^1 + (d-1)b^2 + (d-2)b^3 \ldots + b^d,$$

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

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

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 $b = d = 10$).

Bidirectional search

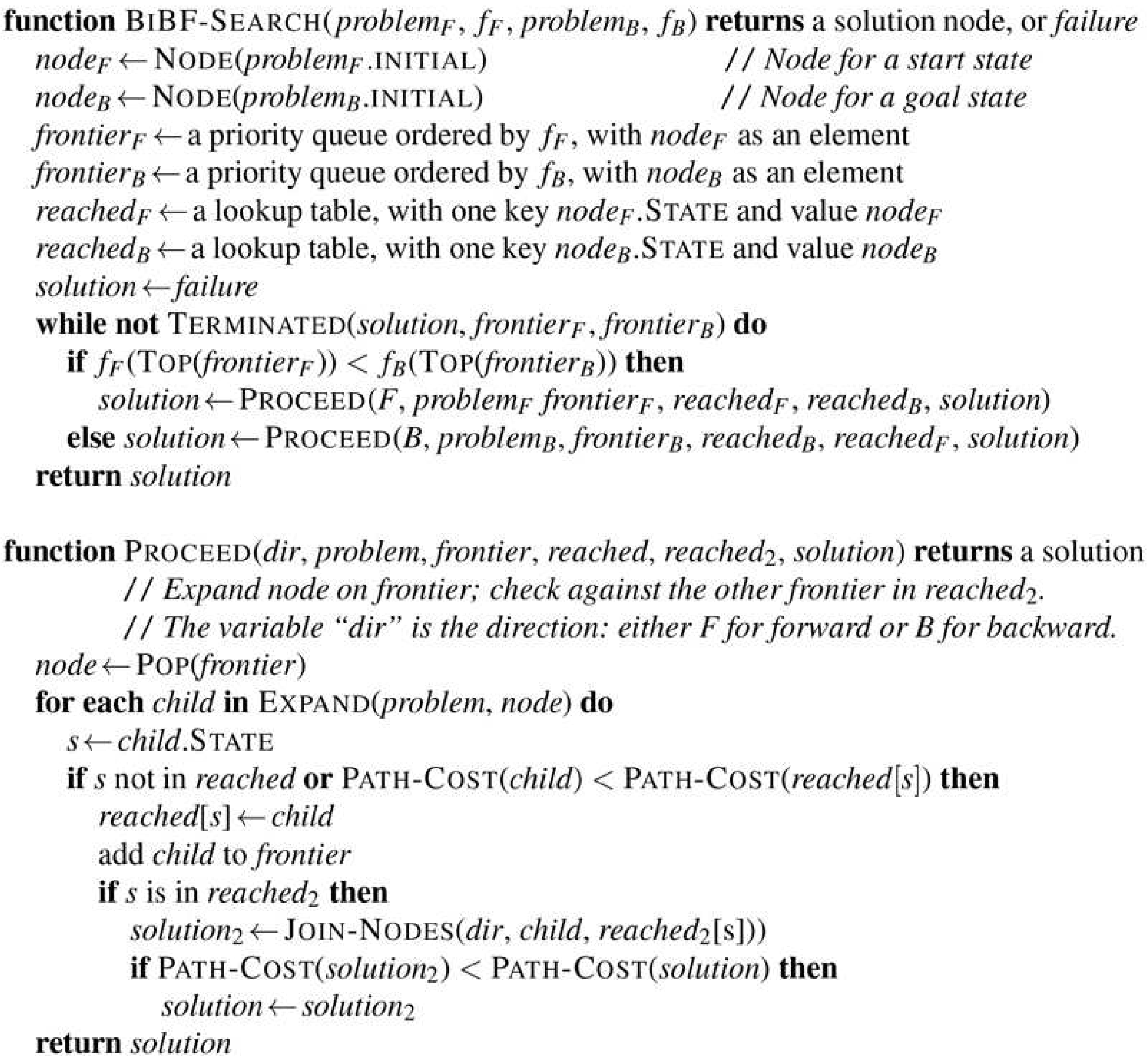
For this to work, we need to keep track of two frontiers and two tables of reached states, and we need to be able to reason backwards: if state s' is a successor of s in the forward direction, then we need to know that s is a successor of s' in the backward direction. We have a solution when the two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

[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.

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.

Figure 3.14



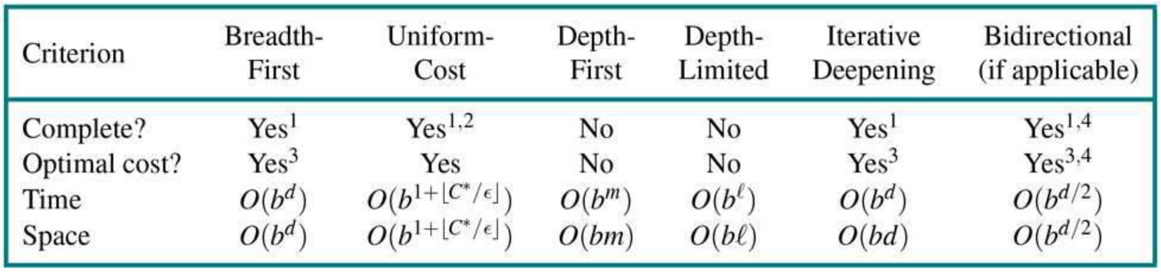
```
function BIBF-SEARCH($problem_F$, $f_F$, $problem_B$, $f_B$) returns a solution node, or failure
  $node_F \leftarrow$ NODE($problem_F$.INITIAL)                         // Node for a start state
  $node_B \leftarrow$ NODE($problem_B$.INITIAL)                         // Node for a goal state
  $frontier_F \leftarrow$ a priority queue ordered by $f_F$, with $node_F$ as an element
  $frontier_B \leftarrow$ a priority queue ordered by $f_B$, with $node_B$ as an element
  $reached_F \leftarrow$ a lookup table, with one key $node_F$.STATE and value $node_F$
  $reached_B \leftarrow$ a lookup table, with one key $node_B$.STATE and value $node_B$
  $solution \leftarrow failure$
  while not TERMINATED($solution$, $frontier_F$, $frontier_B$) do
    if $f_F$(TOP($frontier_F$)) $<$ $f_B$(TOP($frontier_B$)) then
      $solution \leftarrow$ PROCEED($F$, $problem_F$ $frontier_F$, $reached_F$, $reached_B$, $solution$)
    else $solution \leftarrow$ 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.
  $node \leftarrow$ POP($frontier$)
  for each $child$ in EXPAND($problem$, $node$) do
    $s \leftarrow child$.STATE
    if $s$ not in $reached$ or PATH-COST($child$) $<$ PATH-COST($reached[s]$) then
      $reached[s] \leftarrow child$
      add $child$ to $frontier$
      if $s$ is in $reached_2$ then
        $solution_2 \leftarrow$ JOIN-NODES($dir$, $child$, $reached_2$[s]))
        if PATH-COST($solution_2$) $<$ PATH-COST($solution$) then
          $solution \leftarrow solution_2$
  return $solution$
```

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, $|V| + |E|$).

Figure 3.15



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	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

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 $\geq \varepsilon > 0$; [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]

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

Informed search

Heuristic function

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

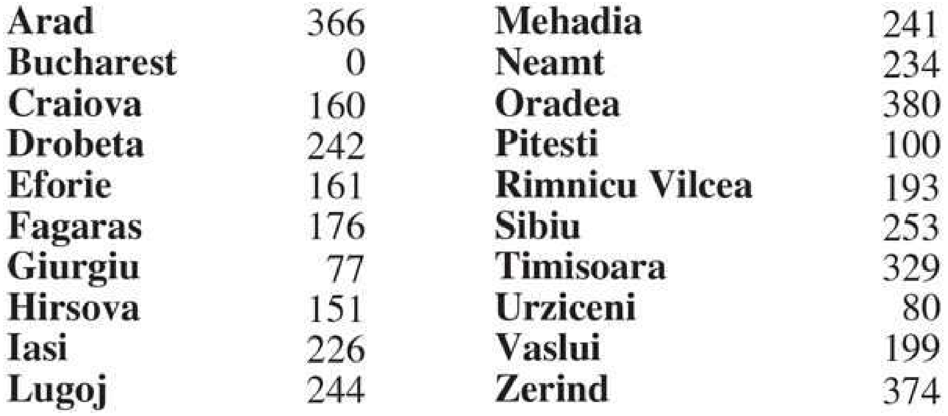
Greedy best-first search is a form of best-first search that expands first the node with the lowest $h(n)$ value—the node that appears to be closest to the goal—on the grounds that this is likely to lead to a solution quickly. So the evaluation function $f(n) = h(n)$.

Greedy best-first search

[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 $g(n)$.

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

Figure 3.16



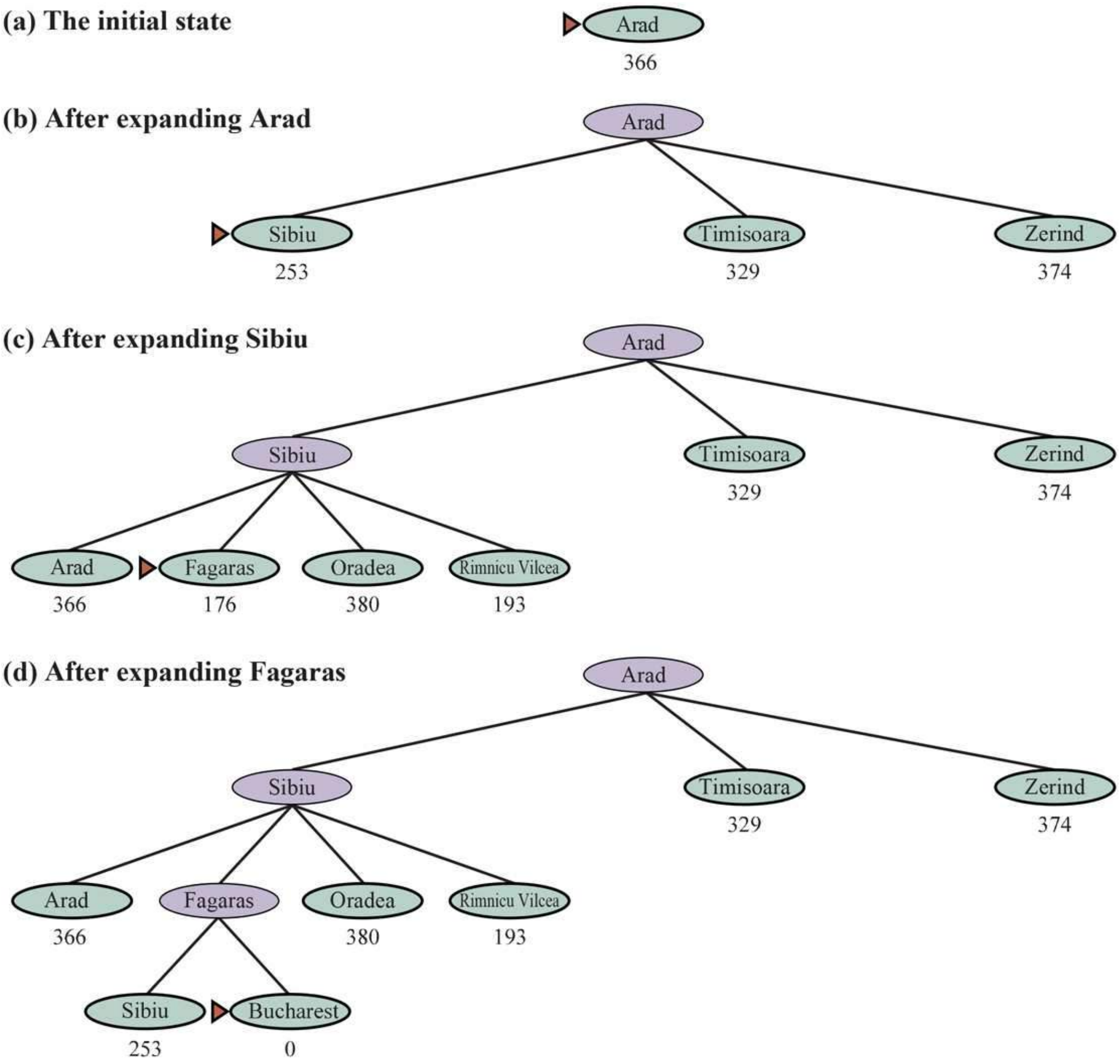
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

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

Straight-line distance

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

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

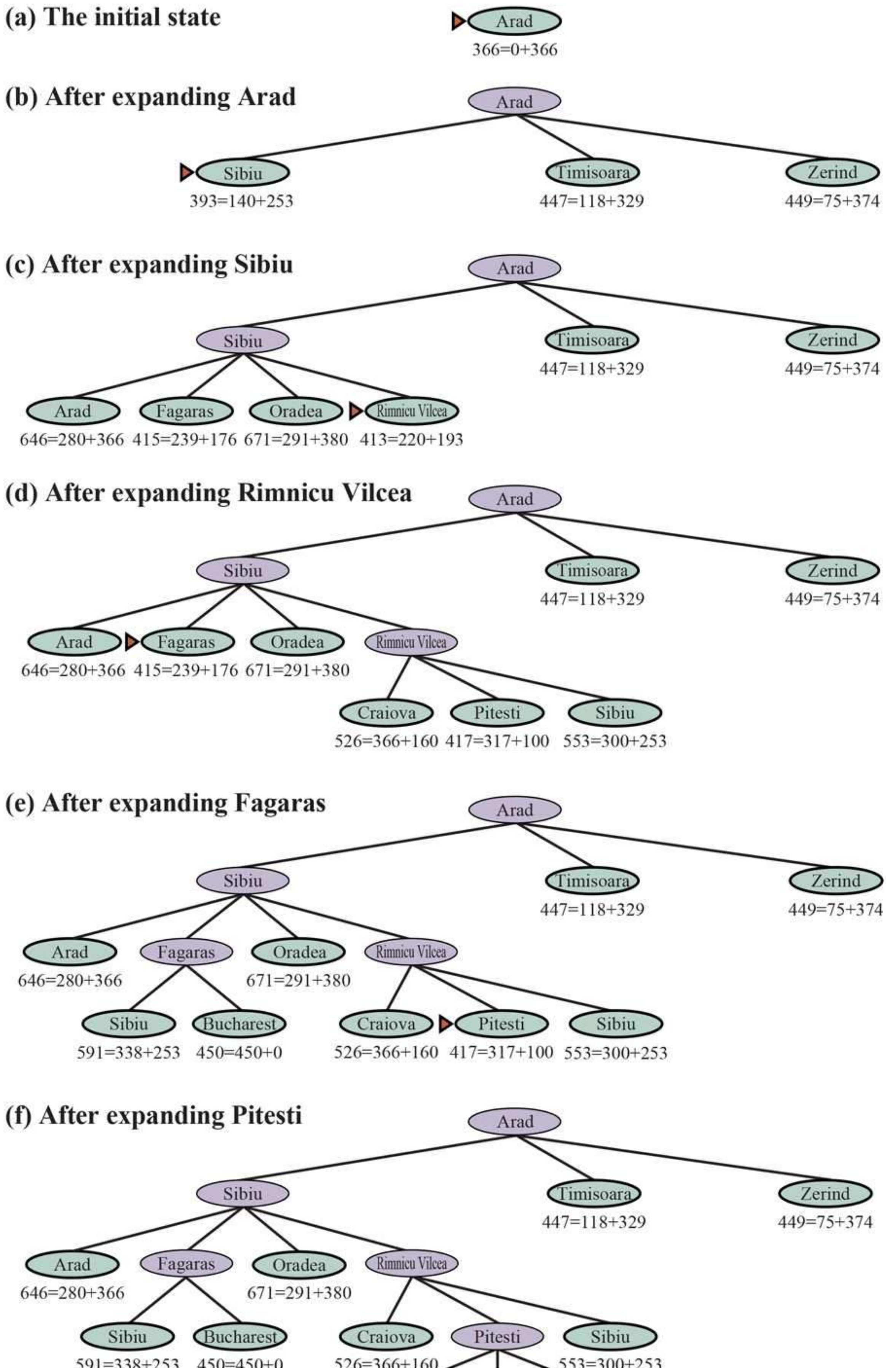
A search*

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.

Figure 3.18



(a) The initial state

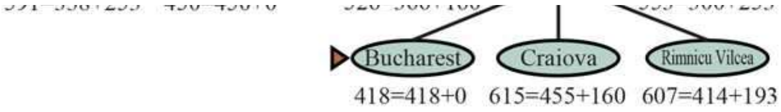
(b) After expanding Arad

(c) After expanding Sibiu

(d) After expanding Rimnicu Vilcea

(e) After expanding Fagaras

(f) After expanding Pitesti



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.

Admissible heuristic

A* search is complete.[11] Whether A* is cost-optimal depends on certain properties of the heuristic. A key property is **admissibility**: an **admissible heuristic** is one that *never overestimates* the cost to reach a goal. (An admissible heuristic is therefore *optimistic*.) With an admissible heuristic, A* is cost-optimal, which we can show with a proof by contradiction. Suppose the optimal path has cost C^*, but the algorithm returns a path with cost $C > C^*$. Then there must be some node n which is on the optimal path and is unexpanded (because if all the nodes on the optimal path had been expanded, then we would have returned that optimal solution). So then, using the notation $g^*(n)$ to mean the cost of the optimal path from the start to n, and $h^*(n)$ to mean the cost of the optimal path from n to the nearest goal, we have:

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

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

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:

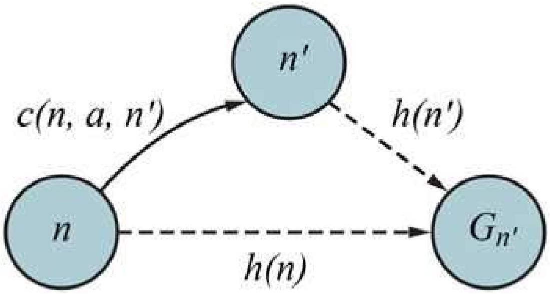
$$h(n) \le c(n, a, n') + h(n').$$

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

Consistency

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 $h(n')$.

Triangle inequality

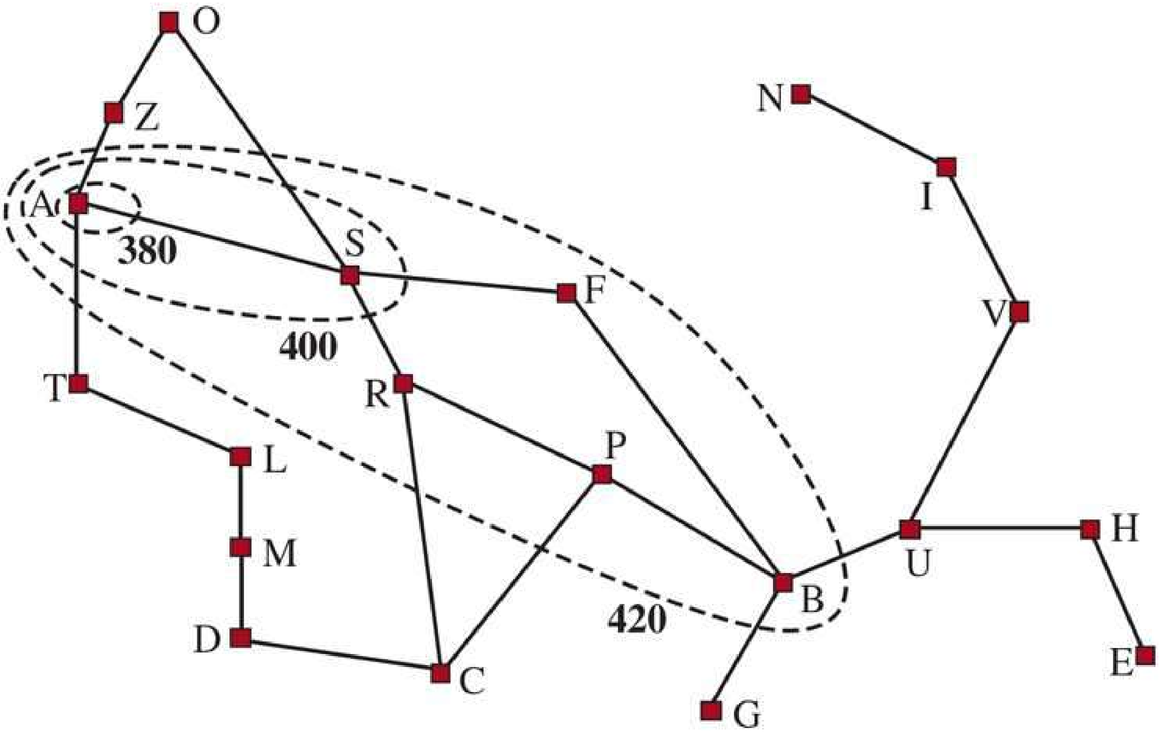
Every consistent heuristic is admissible (but not vice versa), so with a consistent heuristic, A* is cost-optimal. In addition, with a consistent heuristic, the first time we reach a state it will be on an optimal path, so we never have to re-add a state to the frontier, and never have to change an entry 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.

Contour

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

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

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

Monotonic

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

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

- A* 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**.

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.

Optimally efficient

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.

Pruning

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*

Inadmissible heuristic

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.

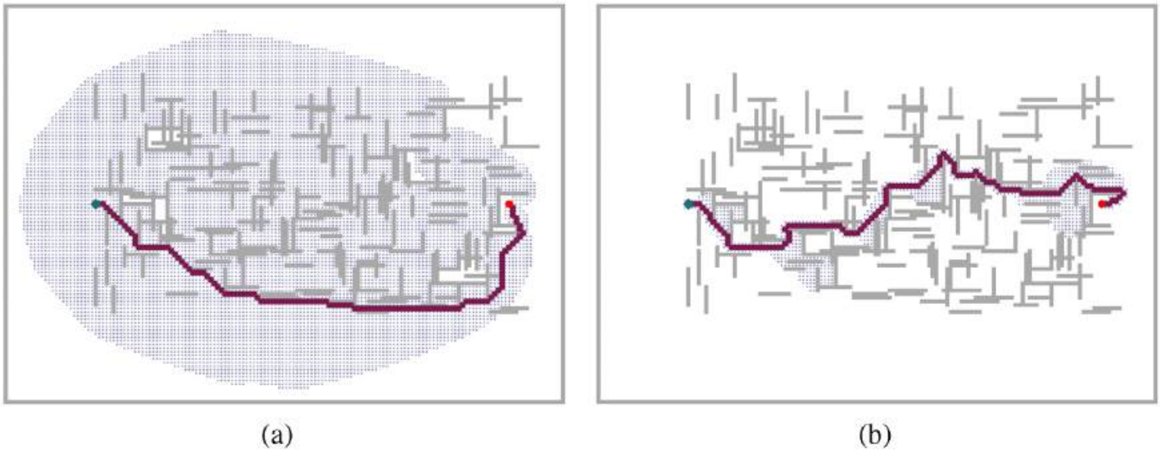
Detour index

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

Weighted A search*

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

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:

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

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.

Bounded suboptimal search

Bounded-cost search

Unbounded-cost search

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

Speedy search

3.5.5 Memory-bounded search

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

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

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

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.

Reference count

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.

Beam search

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

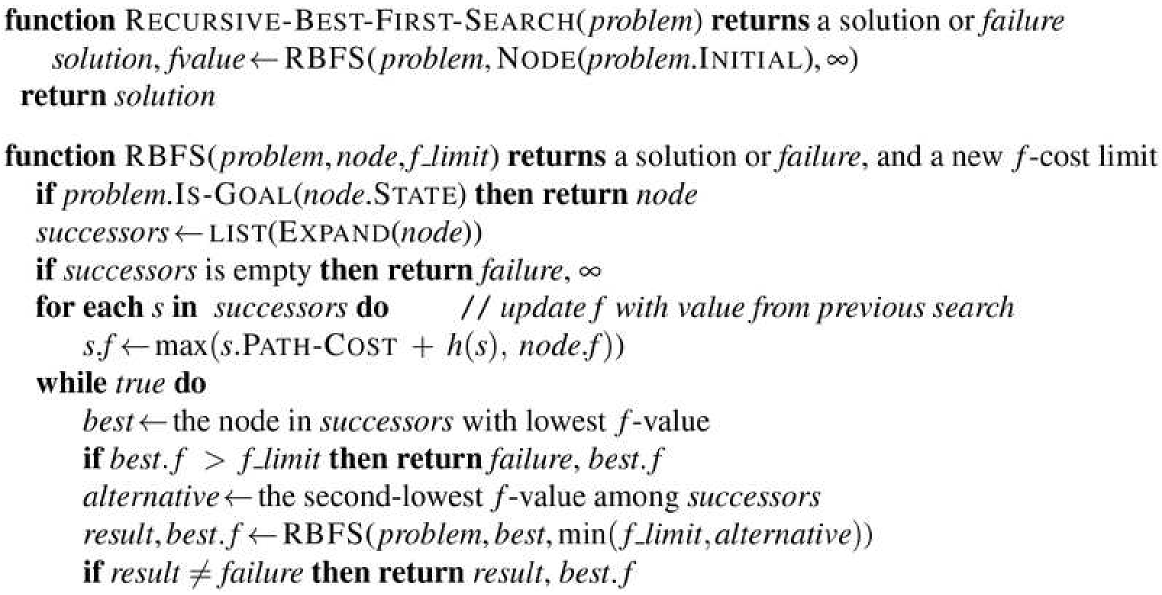
Iterative-deepening A* search (IDA*) is to A* what iterative-deepening search is to depth-first: IDA* gives us the benefits of A* without the requirement to keep all reached states in memory, at a cost of visiting some states multiple times. It is a very important and commonly used algorithm for problems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

Iterative-deepening A search*

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

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

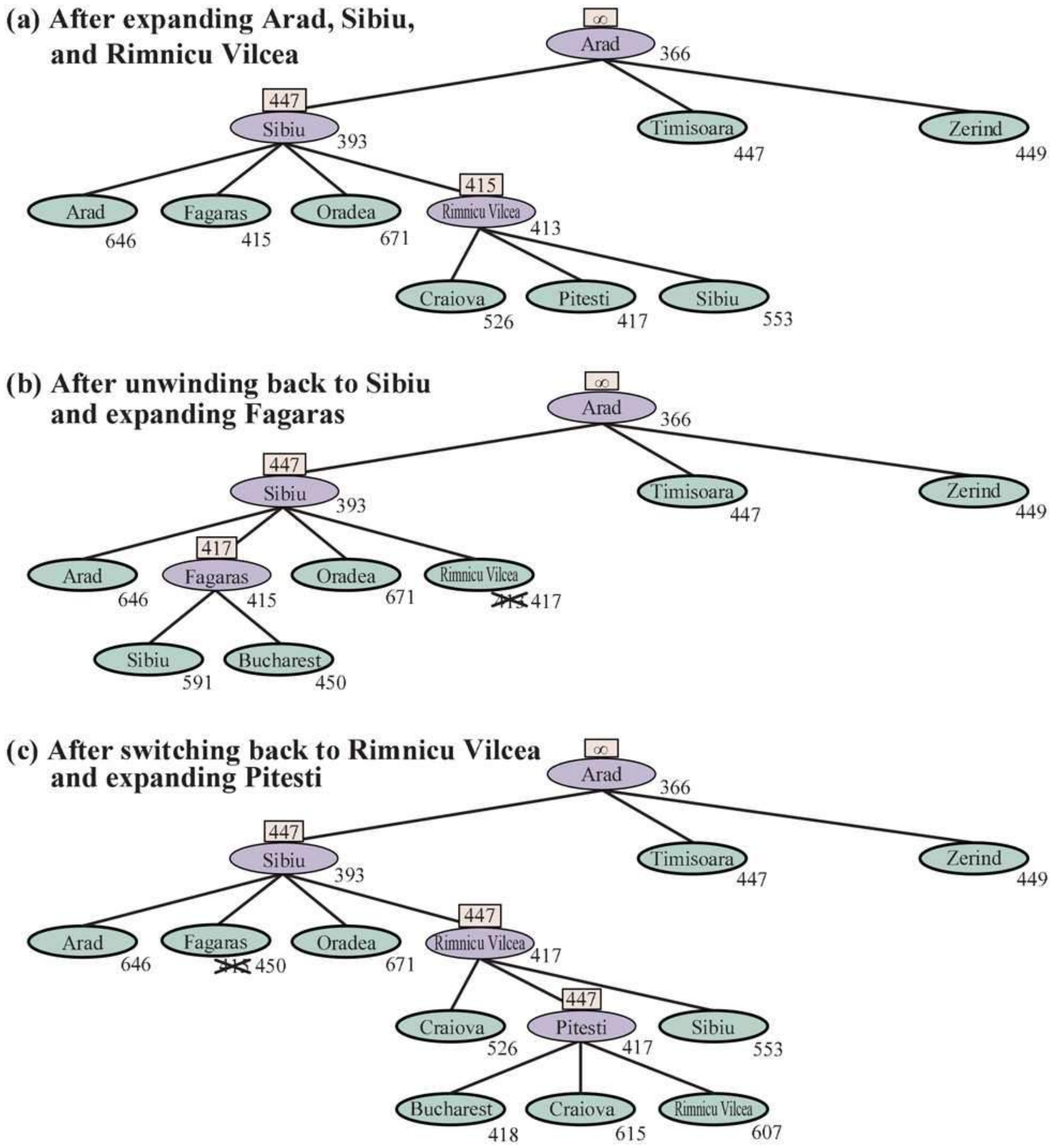
Figure 3.22



```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution or failure
    solution, fvalue $\leftarrow$ RBFS(problem, NODE(problem.INITIAL), $\infty$)
  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 $\leftarrow$ LIST(EXPAND(node))
  if successors is empty then return failure, $\infty$
  for each s in  successors do        // update f with value from previous search
      $s.f \leftarrow \max(s.\text{PATH-COST} + h(s),\ node.f)$
  while true do
      best $\leftarrow$ the node in successors with lowest $f$-value
      if $best.f > f\_limit$ then return failure, $best.f$
      alternative $\leftarrow$ the second-lowest $f$-value among successors
      result, $best.f \leftarrow$ RBFS(problem, best, min(f_limit, alternative))
      if result $\neq$ failure then return result, $best.f$
```

The algorithm for recursive best-first search.

Figure 3.23



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

Recursive best-first search

Backed-up value

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.

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

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

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

*MA**

*SMA**

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

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

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

Thrashing

3.5.6 Bidirectional heuristic search

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

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

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

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

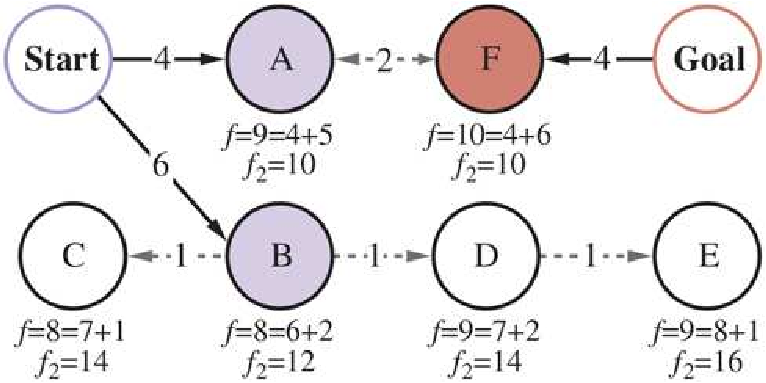
$$lb(m, n) = \max(g_F\ (m) + g_B(n),\ f_F(m),\ f_B(n))$$

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

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

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

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.

Front-to-end

Front-to-front

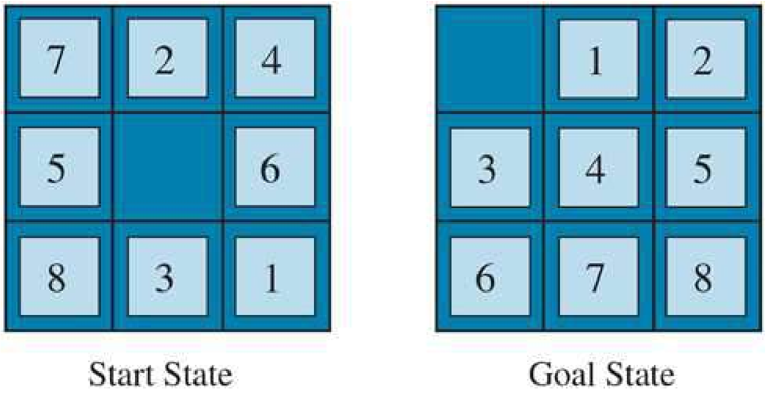
We have described an approach where the h_F heuristic estimates the distance to the goal (or, when the problem has multiple goal states, the distance to the closest goal) and h_B estimates the distance to the start. This is called a **front-to-end** search. An alternative, called **front-to-front** search, 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

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

Manhattan distance

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,

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

Effective branching factor

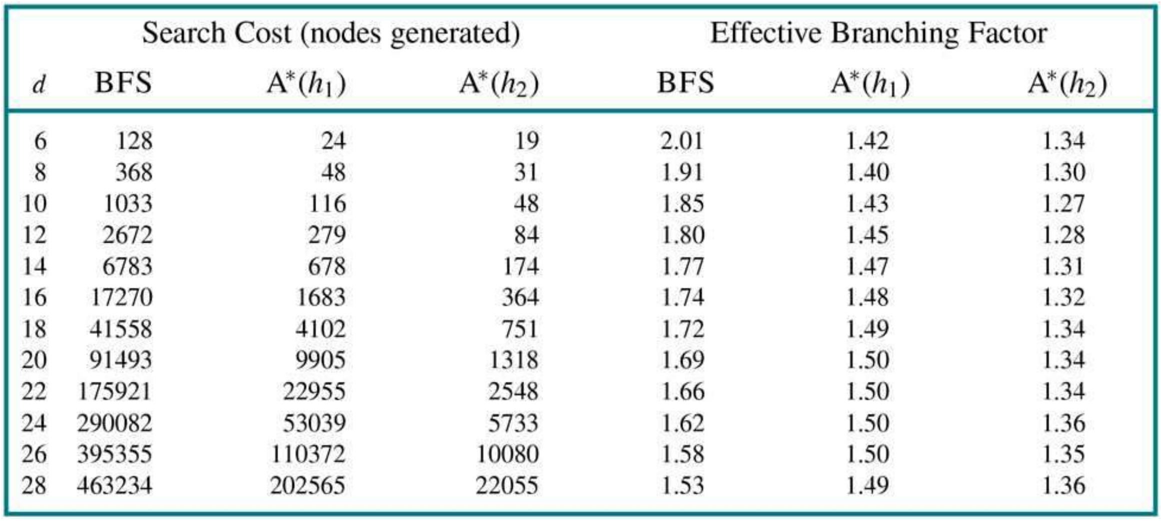
For example, if A* finds a solution at depth 5 using 52 nodes, then the effective branching factor is 1.92. The effective branching factor can vary across problem instances, but usually for a specific domain (such as 8-puzzles) it is fairly constant across all nontrivial problem instances. Therefore, 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.

Effective depth

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.

Figure 3.26



	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

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

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

Domination

3.6.2 Generating heuristics from relaxed problems

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

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

Relaxed problem

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

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

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

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

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

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

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

[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.

If a collection of admissible heuristics $h_1 \dots h_m$ 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

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

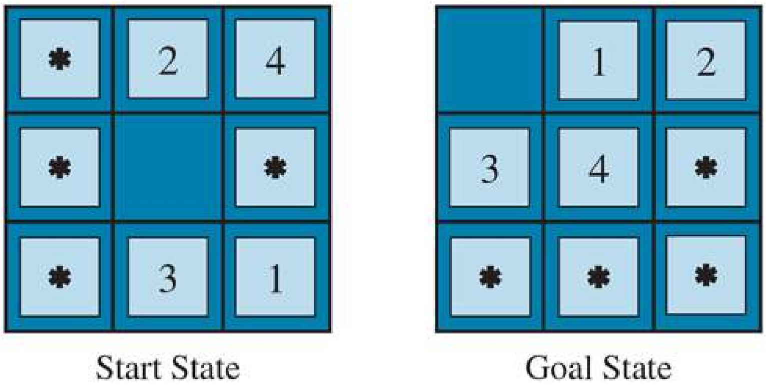
This composite heuristic picks whichever function is most accurate on the node in question. Because the h_i components are admissible, h is admissible (and if h_i are all consistent, h is consistent). Furthermore, h dominates all of its component heuristics. The only drawback is that $h(n)$ takes longer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

3.6.3 Generating heuristics from subproblems: Pattern databases

Subproblem

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

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

Pattern database

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.

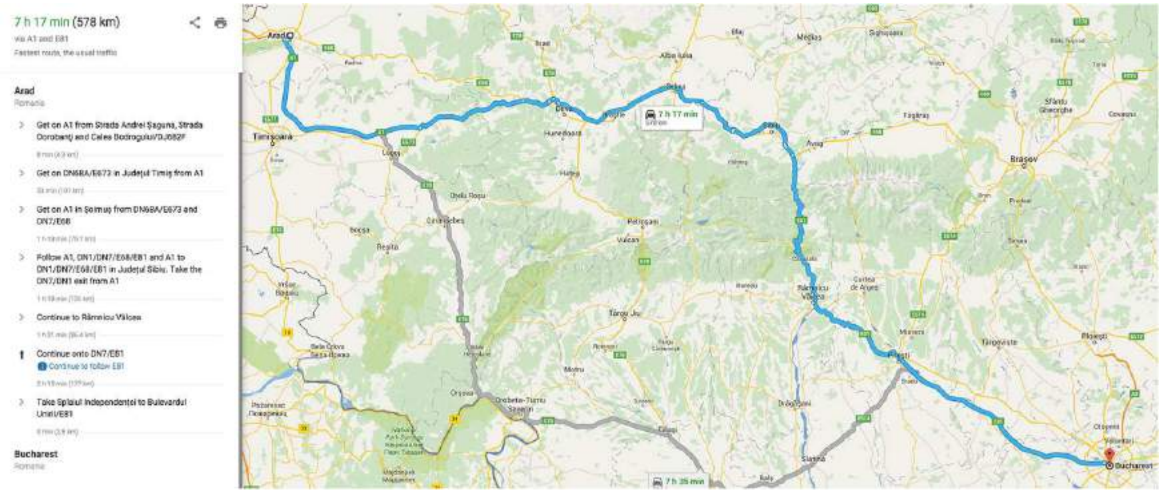
Disjoint pattern databases

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



7 h 17 min (578 km)

via A1 and E81

Fastest route, the usual traffic

Arad

Romania

Get on A1 from Strada Andrei Şaguna, Strada Dorobanţi and Calea Bodrogului/DJ682F

8 min (4.3 km)

Get on DN68A/E673 in Judeţul Timiş from A1

33 min (101 km)

Get on A1 in Şoimuş from DN68A/E673 and DN7/E68

1 h 18 min (75.7 km)

Follow A1, DN1/DN7/E68/E81 and A1 to DN1/DN7/E68/E81 in Judeţul Sibiu. Take the DN7/DN1 exit from A1

1 h 18 min (106 km)

Continue to Râmnicu Vâlcea

1 h 31 min (86.4 km)

Continue onto DN7/E81

Continue to follow E81

2 h 15 min (177 km)

Take Splaiul Independenţei to Bulevardul Unirii/E81

6 min (2.8 km)

Bucharest

Romania

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

Precomputation

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

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:

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

Landmark point

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

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.

Shortcuts

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

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

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.

Differential heuristic

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

Metalevel state space

Object-level state space

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

Metalevel learning

3.6.6 Learning heuristics from experience

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

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:

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

Feature

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

Summary

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

- Before an agent can start searching, a well-defined **problem** must be formulated.
- A problem consists of five parts: the **initial state**, a set of **actions**, a **transition model** describing the results of those actions, a set of **goal states**, and an **action cost function**.
- The environment of the problem is represented by a **state space graph**. A **path** through the state space (a sequence of actions) from the initial state to a goal state is a **solution**.
- Search algorithms generally treat states and actions as **atomic**, without any internal structure (although we introduced features of states when it came time to do learning).
- Search algorithms are judged on the basis of **completeness**, **cost optimality**, **time complexity**, and **space complexity**.
- **Uninformed search** methods have access only to the problem definition. Algorithms build a search tree in an attempt to find a solution. Algorithms differ based on which node they expand first:
  – **BEST-FIRST SEARCH** selects nodes for expansion using to an **evaluation function**.
  – **BREADTH-FIRST SEARCH** expands the shallowest nodes first; it is complete, optimal for unit action costs, but has exponential space complexity.
  – **UNIFORM-COST SEARCH** expands the node with lowest path cost, $g(n)$, and is optimal for general action costs.
  – **DEPTH-FIRST SEARCH** expands the deepest unexpanded node first. It is neither complete nor optimal, but has linear space complexity. **Depth-limited search** adds a depth bound.
  – **ITERATIVE DEEPENING SEARCH** calls depth-first search with increasing depth limits until a goal is found. It is complete when full cycle checking is done, optimal for unit action costs, has time complexity comparable to breadth-first search, and has linear space complexity.

– **BIDIRECTIONAL SEARCH** expands two frontiers, one around the initial state and one around the goal, stopping when the two frontiers meet.

- **Informed search** methods have access to a **heuristic** function $h(n)$ that estimates the cost of a solution from n. They may have access to additional information such as pattern databases with solution costs.

  – **GREEDY BEST-FIRST SEARCH** expands nodes with minimal $h(n)$. It is not optimal but is often efficient.

  – **A* SEARCH** expands nodes with minimal $f(n) = g(n) + h(n)$. A* is complete and optimal, provided that $h(n)$ is admissible. The space complexity of A* is still an issue for many problems.

  – **BIDIRECTIONAL A* SEARCH** is sometimes more efficient than A* itself.

  – **IDA*** (iterative deepening A* search) is an iterative deepening version of A*, and thus adresses the space complexity issue.

  – **RBFS** (recursive best-first search) and **SMA*** (simplified memory-bounded A*) are robust, optimal search algorithms that use limited amounts of memory; given enough time, they can solve problems for which A* runs out of memory.

  – **BEAM SEARCH** puts a limit on the size of the frontier; that makes it incomplete and suboptimal, but it often finds reasonably good solutions and runs faster than complete searches.

  – **WEIGHTED A*** search focuses the search towards a goal, expanding fewer nodes, but sacrificing optimality.

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

Bibliographical and Historical Notes

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Coarse-to-fine search

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

Branch-and-bound

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

Iterative expansion

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*.

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

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

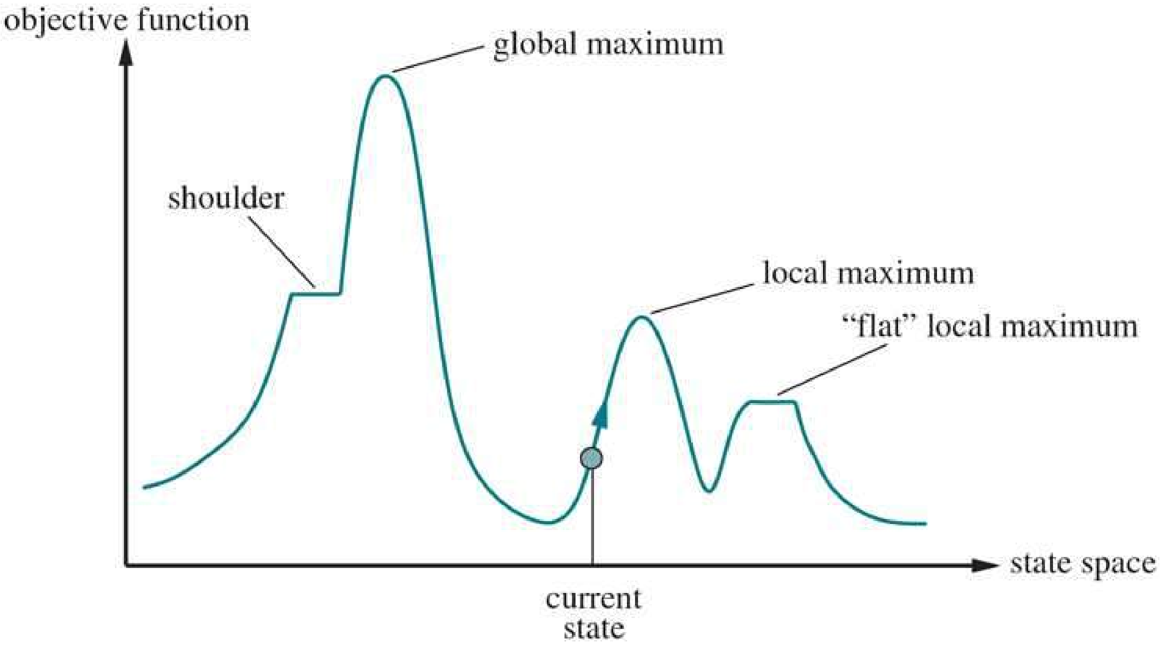
Optimization problem

Objective function

State-space landscape

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

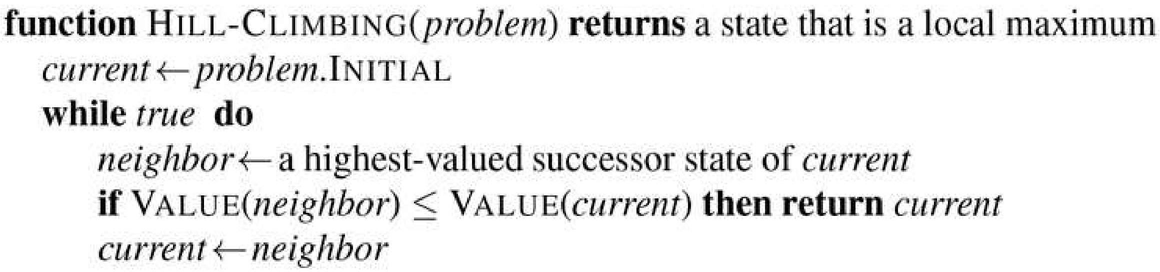
Global maximum

Global minimum

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.

Figure 4.2



```
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
```

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.

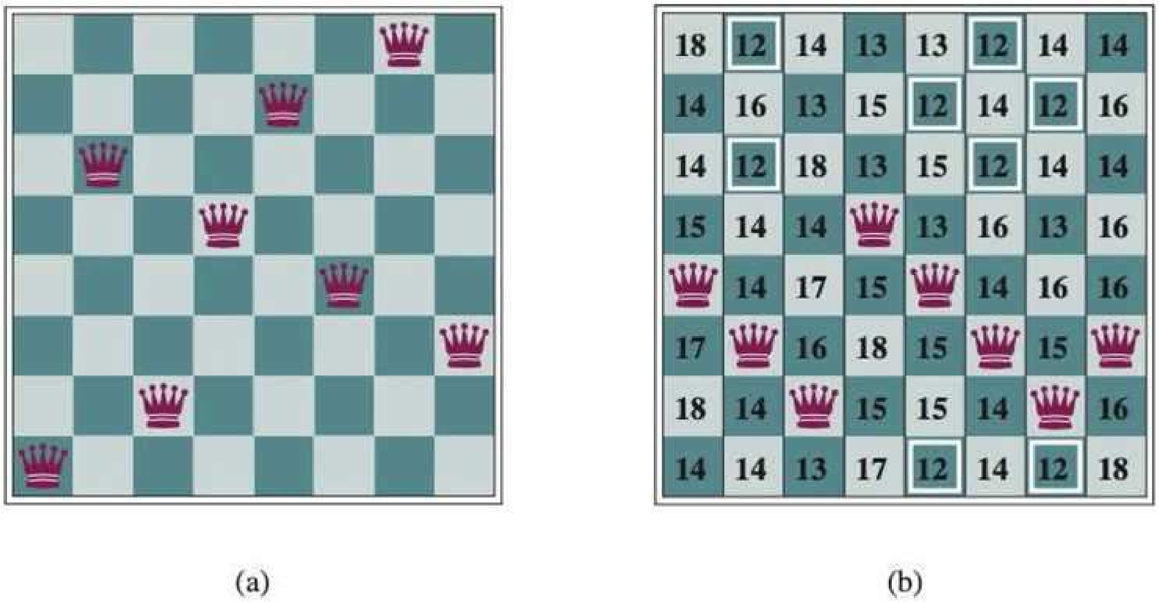
Hill-climbing

Steepest ascent

Complete-state formulation

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

Figure 4.3



18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	♛	13	16	13	16
♛	14	17	15	♛	14	16	16
17	♛	16	18	15	♛	15	♛
18	14	♛	15	15	14	♛	16
14	14	13	17	12	14	12	18

(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.

Greedy local search

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

- **LOCAL 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.

Local maximum

- **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.

Ridge

- **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

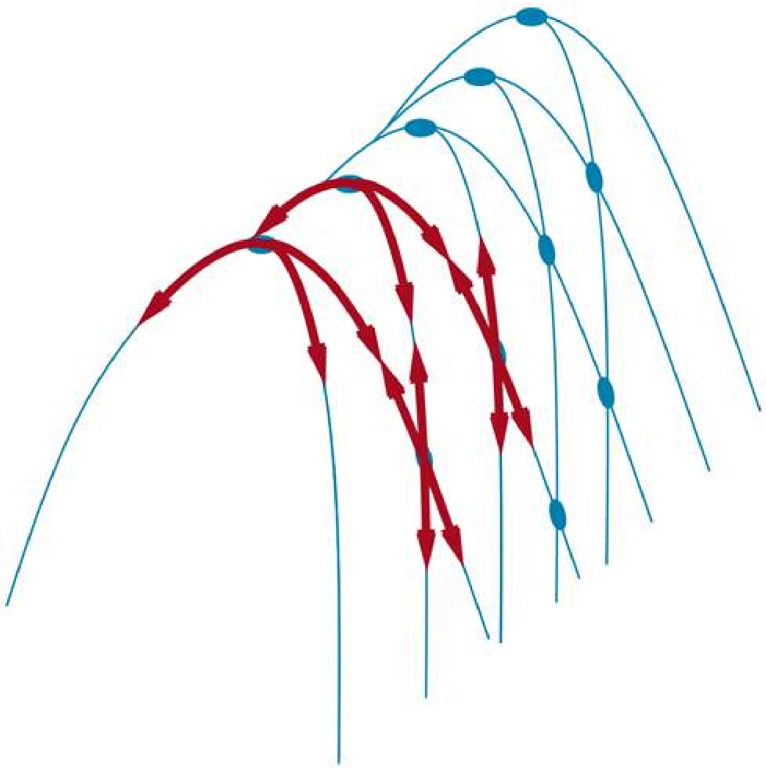


Illustration of why ridges cause difficulties for hill climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point downhill.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

Plateau

Shoulder

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

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

Sideways move

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

Stochastic hill climbing

First-choice hill climbing

Random-restart hill climbing

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

[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.

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.

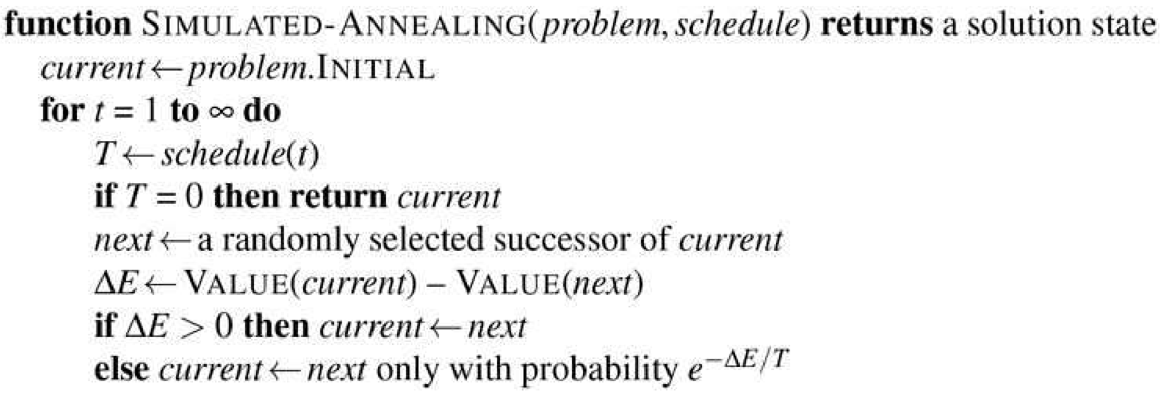
Simulated annealing is such an algorithm. In metallurgy, **annealing** is the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to reach a low-energy crystalline state. To explain simulated annealing, we 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).

Simulated annealing

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.

Figure 4.5



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

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.

Local beam search

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

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

Stochastic beam search

4.1.4 Evolutionary algorithms

Evolutionary algorithms

recombination

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.

Genetic algorithm

Evolution strategies

Genetic programming

- 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.

Selection

- 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.

Crossover point

- 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.

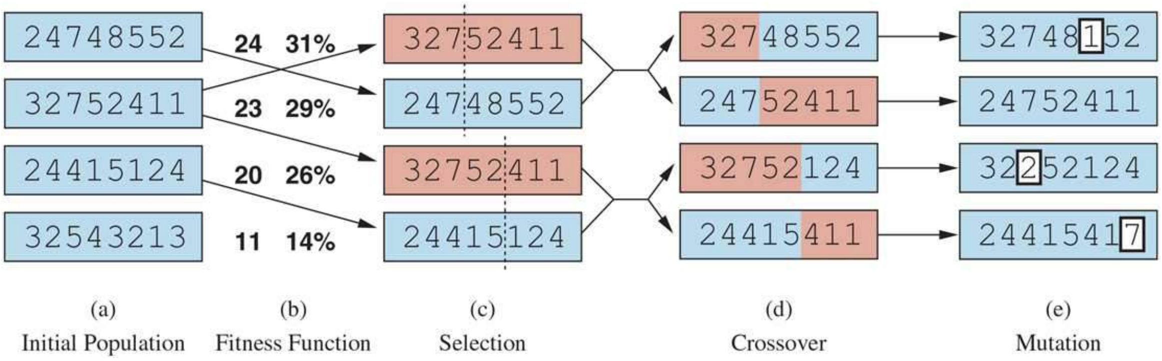
Mutation rate

- 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).

Elitism

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).

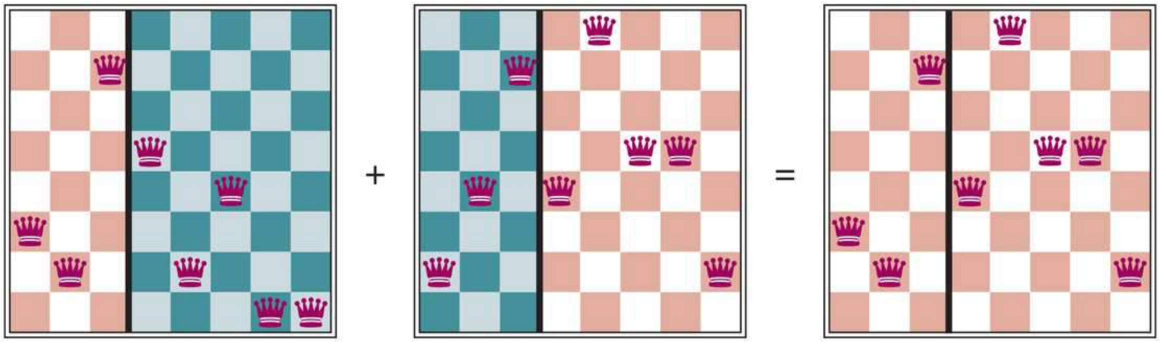
Figure 4.6



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.

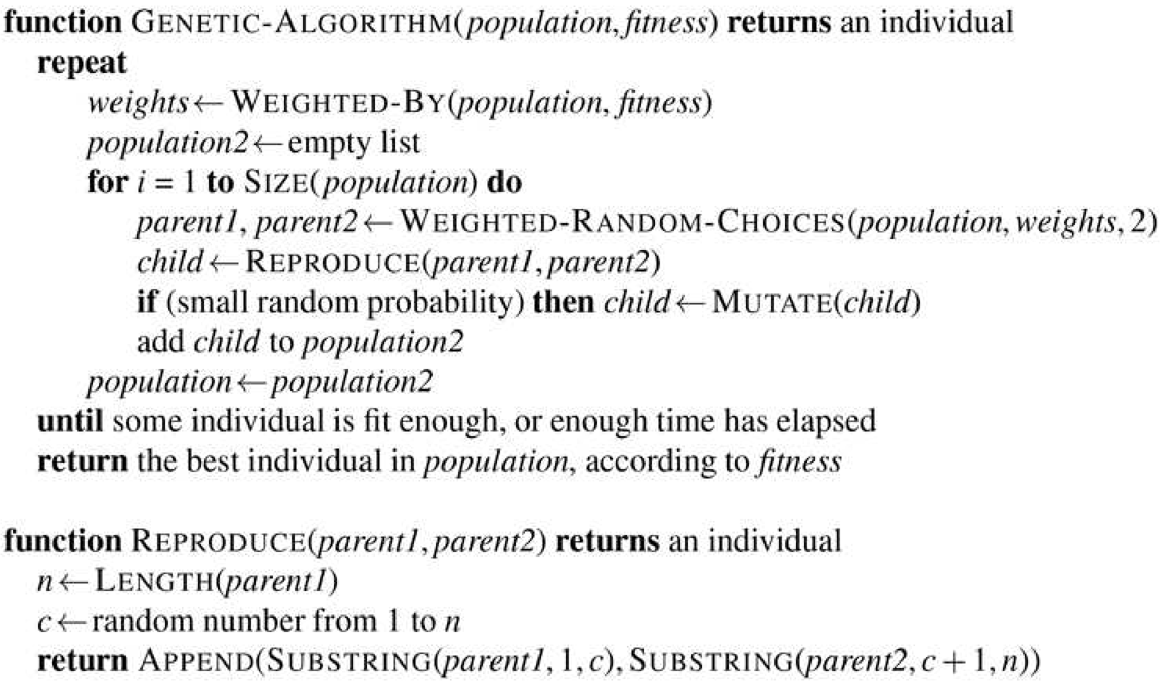
Figure 4.7



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.

Figure 4.8



```
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))
```

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.

Schema

Instance

Evolution and Search

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

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

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

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

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

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.