Artificial Intelligence

A Modern Approach

Second Edition

PRENTICE HALL SERIES
IN ARTIFICIAL INTELLIGENCE
Stuart Russell and Peter Norvig, Editors

FORSYTH & PONCE	*Computer Vision: A Modern Approach*
GRAHAM	ANSI *Common Lisp*
JURAFSKY & MARTIN	*Speech and Language Processing*
NEAPOLITAN	*Learning Bayesian Networks*
RUSSELL & NORVIG	*Artificial Intelligence: A Modern Approach*

Artificial Intelligence

A Modern Approach

Second Edition

Stuart J. Russell and Peter Norvig

Contributing writers:
John F. Canny
Douglas D. Edwards
Jitendra M. Malik
Sebastian Thrun

Pearson Education International

Vice President and Editorial Director, ECS: Marcia J. Horton
Publisher: Alan R. Apt
Associate Editor: Toni Dianne Holm
Editorial Assistant: Patrick Lindner
Vice President and Director of Production and Manufacturing, ESM: David W. Riccardi
Executive Managing Editor: Vince O'Brien
Assistant Managing Editor: Camille Trentacoste
Production Editor: Irwin Zucker
Manufacturing Manager: Trudy Pisciotti
Manufacturing Buyer: Lisa McDowell
Director, Creative Services: Paul Belfanti
Creative Director: Carole Anson
Art Editor: Greg Dulles
Art Director: Heather Scott
Assistant to Art Director: Geoffrey Cassar
Cover Designers: Stuart Russell and Peter Norvig
Cover Image Creation: Stuart Russell and Peter Norvig; Tamara Newnam and Patrice Van Acker
Interior Designer: Stuart Russell and Peter Norvig
Marketing Manager: Pamela Shaffer
Marketing Assistant: Barrie Reinhold

Pearson Education, Inc.,
Upper Saddle River, New Jersey 07458

Printed in the United States of America

10 9 8 7 6 5 4 3

ISBN 0-13-080302-2

Pearson Education Ltd., *London*
Pearson Education Australia Pty. Ltd., *Sydney*
Pearson Education Singapore, Pte. Ltd.
Pearson Education North Asia Ltd., *Hong Kong*
Pearson Education Canada, Inc., *Toronto*
Pearson Educación de Mexico, S.A. de C.V.
Pearson Education—Japan, *Tokyo*
Pearson Education Malaysia, Pte. Ltd.
Pearson Education, Inc., *Upper Saddle River, New Jersey*

For Loy, Gordon, and Lucy — 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; and everything from microelectronic devices to robotic planetary explorers. The book is also big because we go into some depth in presenting results, although we strive to cover only the most central ideas in the main part of each chapter. Pointers are given to further results in the bibliographical notes at the end of each chapter.

The subtitle of this book is "A Modern Approach." The intended meaning of this rather empty phrase is that we have tried to synthesize what is now known into a common framework, rather than trying to explain each subfield of AI in its own historical context. We apologize to those whose subfields are, as a result, less recognizable than they might otherwise have been.

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 production systems, reactive agents, real-time conditional planners, neural networks, and decision-theoretic systems. We explain the role of learning as extending the reach of the designer into unknown environments, and we show how that role constrains agent design, favoring explicit knowledge representation and reasoning. 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 fifty years of AI research and the past two millenia of related work. We have tried to avoid excessive formality in the presentation of these ideas while retaining precision. Wherever appropriate, we have included pseudocode algorithms to make the ideas concrete; our pseudocode is described briefly in Appendix B. Implementations in several programming languages are available on the book's Web site, **aima.cs.berkeley.edu**.

This book is primarily intended for use in an undergraduate course or course sequence. It can also be used in a graduate-level course (perhaps with the addition of some of the primary sources suggested in the bibliographical notes). Because of its comprehensive coverage and large number of detailed algorithms, it is useful as a primary reference volume for AI graduate students and professionals wishing to branch out beyond their own subfield. The only prerequisite is familiarity with basic concepts of computer science (algorithms, data structures, complexity) at a sophomore level. Freshman calculus is useful for understanding neural networks and statistical learning in detail. Some of the required mathematical background is supplied in Appendix A.

Overview of the book

The book is divided into eight parts. Part I, **Artificial Intelligence**, offers a view of the AI enterprise based around the idea of intelligent agents—systems that can decide what to do and then do it. Part II, **Problem Solving**, concentrates on methods for deciding what to do when one needs to think ahead several steps—for example in navigating across a country or playing chess. Part III, **Knowledge and Reasoning**, discusses ways to represent knowledge about the world—how it works, what it is currently like, and what one's actions might do—and how to reason logically with that knowledge. Part IV, **Planning**, then discusses how to use these reasoning methods to decide what to do, particularly by constructing *plans*. Part V, **Uncertain Knowledge and Reasoning**, is analogous to Parts III and IV, but it concentrates on reasoning and decision making in the presence of *uncertainty* about the world, as might be faced, for example, by a system for medical diagnosis and treatment.

Together, Parts II–V describe that part of the intelligent agent responsible for reaching decisions. Part VI, **Learning**, describes methods for generating the knowledge required by these decision-making

components. Part VII, **Communicating, Perceiving, and Acting**, describes ways in which an intelligent agent can perceive its environment so as to know what is going on, whether by vision, touch, hearing, or understanding language, and ways in which it can turn its plans into real actions, either as robot motion or as natural language utterances. Finally, Part VIII, **Conclusions**, analyzes the past and future of AI and the philosophical and ethical implications of artificial intelligence.

Changes from the first edition

Much has changed in AI since the publication of the first edition in 1995, and much has changed in this book. Every chapter has been significantly rewritten to reflect the latest work in the field, to reinterpret old work in a way that is more cohesive with new findings, and to improve the pedagogical flow of ideas. Followers of AI should be encouraged that current techniques are much more practical than those of 1995; for example the planning algorithms in the first edition could generate plans of only dozens of steps, while the algorithms in this edition scale up to tens of thousands of steps. Similar orders-of-magnitude improvements are seen in probabilistic inference, language processing, and other subfields. The following are the most notable changes in the book:

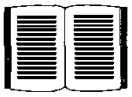
- In Part I, we acknowledge the historical contributions of control theory, game theory, economics, and neuroscience. This helps set the tone for a more integrated coverage of these ideas in subsequent chapters.
- In Part II, online search algorithms are covered and a new chapter on constraint satisfaction has been added. The latter provides a natural connection to the material on logic.
- In Part III, propositional logic, which was presented as a stepping-stone to first-order logic in the first edition, is now presented as a useful representation language in its own right, with fast inference algorithms and circuit-based agent designs. The chapters on first-order logic have been reorganized to present the material more clearly and we have added the Internet shopping domain as an example.
- In Part IV, we include newer planning methods such as GRAPHPLAN and satisfiability-based planning, and we increase coverage of scheduling, conditional planning, hierarchical planning, and multiagent planning.
- In Part V, we have augmented the material on Bayesian networks with new algorithms, such as variable elimination and Markov Chain Monte Carlo, and we have created a new chapter on uncertain temporal reasoning, covering hidden Markov models, Kalman filters, and dynamic Bayesian networks. The coverage of Markov decision processes is deepened, and we add sections on game theory and mechanism design.
- In Part VI, we tie together work in statistical, symbolic, and neural learning and add sections on boosting algorithms, the EM algorithm, instance-based learning, and kernel methods (support vector machines).
- In Part VII, coverage of language processing adds sections on discourse processing and grammar induction, as well as a chapter on probabilistic language models, with applications to information retrieval and machine translation. The coverage of robotics stresses the integration of uncertain sensor data, and the chapter on vision has updated material on object recognition.
- In Part VIII, we introduce a section on the ethical implications of AI.

Using this book

The book has 27 chapters, each requiring about a week's worth of lectures, so working through the whole book requires a two-semester sequence. Alternatively, a course can be tailored to suit the interests of the instructor and student. Through its broad coverage, the book can be used to support such

courses, whether they are short, introductory undergraduate courses or specialized graduate courses on advanced topics. Sample syllabi from the more than 600 universities and colleges that have adopted the first edition are shown on the Web at aima.cs.berkeley.edu, along with suggestions to help you find a sequence appropriate to your needs.

The book includes 385 exercises. Exercises requiring significant programming are marked with a keyboard icon. These exercises can best be solved by taking advantage of the code repository at aima.cs.berkeley.edu. Some of them are large enough to be considered term projects. A number of exercises require some investigation of the literature; these are marked with a book icon.



Throughout the book, important points are marked with a pointing icon. We have included an extensive index of around 10,000 items to make it easy to find things in the book. Wherever a **new term** is first defined, it is also marked in the margin.

NEW TERM

Using the Web site

At the **aima.cs.berkeley.edu** Web site you will find:

- implementations of the algorithms in the book in several programming languages,
- a list of over 600 schools that have used the book, many with links to online course materials,
- an annotated list of over 800 links to sites around the web with useful AI content,
- a chapter by chapter list of supplementary material and links,
- instructions on how to join a discussion group for the book,
- instructions on how to contact the authors with questions or comments,
- instructions on how to report errors in the book, in the likely event that some exist, and
- copies of the figures in the book, along with slides and other material for instructors.

Acknowledgments

Jitendra Malik wrote most of Chapter 24 (on vision). Most of Chapter 25 (on robotics) was written by Sebastian Thrun in this edition and by John Canny in the first edition. Doug Edwards researched the historical notes for the first edition. Tim Huang, Mark Paskin, and Cynthia Bruyns helped with formatting of the diagrams and algorithms. Alan Apt, Sondra Chavez, Toni Holm, Jake Warde, Irwin Zucker, and Camille Trentacoste at Prentice Hall tried their best to keep us on schedule and made many helpful suggestions on the book's design and content.

Stuart would like to thank his parents for their continued support and encouragement and his wife, Loy Sheflott, for her endless patience and boundless wisdom. He hopes that Gordon and Lucy will soon be reading this. RUGS (Russell's Unusual Group of Students) have been unusually helpful.

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

We are indebted to the librarians at Berkeley, Stanford, MIT, and NASA, and to the developers of CiteSeer and Google, who have revolutionized the way we do research.

We can't thank all the people who have used the book and made suggestions, but we would like to acknowledge the especially helpful comments of Eyal Amir, Krzysztof Apt, Ellery Aziel, Jeff Van Baalen, Brian Baker, Don Barker, Tony Barrett, James Newton Bass, Don Beal, Howard Beck, Wolfgang Bibel, John Binder, Larry Bookman, David R. Boxall, Gerhard Brewka, Selmer Bringsjord, Carla Brodley, Chris Brown, Wilhelm Burger, Lauren Burka, Joao Cachopo, Murray Campbell, Norman Carver, Emmanuel Castro, Anil Chakravarthy, Dan Chisarick, Roberto Cipolla, David Cohen, James Coleman, Julie Ann Comparini, Gary Cottrell, Ernest Davis, Rina Dechter, Tom Dietterich, Chuck Dyer, Barbara Engelhardt, Doug Edwards, Kutluhan Erol, Oren Etzioni, Hana Filip, Douglas

Fisher, Jeffrey Forbes, Ken Ford, John Fosler, Alex Franz, Bob Futrelle, Marek Galecki, Stefan Gerberding, Stuart Gill, Sabine Glesner, Seth Golub, Gosta Grahne, Russ Greiner, Eric Grimson, Barbara Grosz, Larry Hall, Steve Hanks, Othar Hansson, Ernst Heinz, Jim Hendler, Christoph Herrmann, Vasant Honavar, Tim Huang, Seth Hutchinson, Joost Jacob, Magnus Johansson, Dan Jurafsky, Leslie Kaelbling, Keiji Kanazawa, Surekha Kasibhatla, Simon Kasif, Henry Kautz, Gernot Kerschbaumer, Richard Kirby, Kevin Knight, Sven Koenig, Daphne Koller, Rich Korf, James Kurien, John Lafferty, Gus Larsson, John Lazzaro, Jon LeBlanc, Jason Leatherman, Frank Lee, Edward Lim, Pierre Louveaux, Don Loveland, Sridhar Mahadevan, Jim Martin, Andy Mayer, David McGrane, Jay Mendelsohn, Brian Milch, Steve Minton, Vibhu Mittal, Leora Morgenstern, Stephen Muggleton, Kevin Murphy, Ron Musick, Sung Myaeng, Lee Naish, Pandu Nayak, Bernhard Nebel, Stuart Nelson, XuanLong Nguyen, Illah Nourbakhsh, Steve Omohundro, David Page, David Palmer, David Parkes, Ron Parr, Mark Paskin, Tony Passera, Michael Pazzani, Wim Pijls, Ira Pohl, Martha Pollack, David Poole, Bruce Porter, Malcolm Pradhan, Bill Pringle, Lorraine Prior, Greg Provan, William Rapaport, Philip Resnik, Francesca Rossi, Jonathan Schaeffer, Richard Scherl, Lars Schuster, Soheil Shams, Stuart Shapiro, Jude Shavlik, Satinder Singh, Daniel Sleator, David Smith, Bryan So, Robert Sproull, Lynn Stein, Larry Stephens, Andreas Stolcke, Paul Stradling, Devika Subramanian, Rich Sutton, Jonathan Tash, Austin Tate, Michael Thielscher, William Thompson, Sebastian Thrun, Eric Tiedemann, Mark Torrance, Randall Upham, Paul Utgoff, Peter van Beek, Hal Varian, Sunil Vemuri, Jim Waldo, Bonnie Webber, Dan Weld, Michael Wellman, Michael Dean White, Kamin Whitehouse, Brian Williams, David Wolfe, Bill Woods, Alden Wright, Richard Yen, Weixiong Zhang, Shlomo Zilberstein, and the anonymous reviewers provided by Prentice Hall.

About the Cover

The cover image was designed by the authors and executed by Lisa Marie Sardegna and Maryann Simmons using SGI Inventor™ and Adobe Photoshop™. The cover depicts the following items from the history of AI:

1. Aristotle's planning algorithm from *De Motu Animalium* (c. 400 B.C.).
2. Ramon Lull's concept generator from *Ars Magna* (c. 1300 A.D.).
3. Charles Babbage's Difference Engine, a prototype for the first universal computer (1848).
4. Gottlob Frege's notation for first-order logic (1789).
5. Lewis Carroll's diagrams for logical reasoning (1886).
6. Sewall Wright's probabilistic network notation (1921).
7. Alan Turing (1912–1954).
8. Shakey the Robot (1969–1973).
9. A modern diagnostic expert system (1993).

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 of computer science, director of the Center for Intelligent Systems, 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 was a 1996 Miller Professor of the University of California and was appointed to a Chancellor's Professorship in 2000. In 1998, he gave the Forsythe Memorial Lectures at Stanford University. He is a Fellow and former Executive Council member of the American Association for Artificial Intelligence. He has published over 100 papers on a wide range of topics in artificial intelligence. His other books include *The Use of Knowledge in Analogy and Induction* and (with Eric Wefald) *Do the Right Thing: Studies in Limited Rationality.*

Peter Norvig is director of Search Quality at Google, Inc. He is a Fellow and Executive Council member of the American Association for Artificial Intelligence. Previously, he was head of the Computational Sciences Division at NASA Ames Research Center, where he oversaw NASA's research and development in artificial intelligence and robotics. Before that he served as chief scientist at Junglee, where he helped develop one of the first Internet information extraction services, and as a senior scientist at Sun Microsystems Laboratories working on intelligent information retrieval. He received a B.S. in applied mathematics from Brown University and a Ph.D. in computer science from the University of California at Berkeley. He has been a professor at the University of Southern California and a research faculty member at Berkeley. He has over 50 publications in computer science including the books *Paradigms of AI Programming: Case Studies in Common Lisp*, *Verbmobil: A Translation System for Face-to-Face Dialog*, and *Intelligent Help Systems for UNIX.*

Summary of Contents

Contents

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 mental capacities are so important to us. For thousands of years, we have tried to understand *how we think*; that is, how a mere handful of stuff can perceive, understand, predict, and manipulate a world far larger and more complicated than itself. The field of **artificial intelligence**, or AI, goes further still: it attempts not just to understand but also to *build* intelligent entities.

ARTIFICIAL INTELLIGENCE

AI is one of the newest sciences. Work started in earnest soon after World War II, and the name itself was coined in 1956. Along with molecular biology, AI is regularly cited as the "field I would most like to be in" by scientists in other disciplines. A student in physics might reasonably feel that all the good ideas have already been taken by Galileo, Newton, Einstein, and the rest. AI, on the other hand, still has openings for several full-time Einsteins.

AI currently encompasses a huge variety of subfields, ranging from general-purpose areas, such as learning and perception to such specific tasks as playing chess, proving mathematical theorems, writing poetry, and diagnosing diseases. AI systematizes and automates intellectual tasks and is therefore potentially relevant to any sphere of human intellectual activity. In this sense, it is truly a universal field.

1.1 WHAT IS AI?

We have claimed that AI is exciting, but we have not said what it *is*. Definitions of artificial intelligence according to eight textbooks are shown in Figure 1.1. These definitions vary along two main dimensions. Roughly, the ones on top are concerned with *thought processes* and *reasoning*, whereas the ones on the bottom address *behavior*. The definitions on the left measure success in terms of fidelity to *human* performance, whereas the ones on the right measure against an *ideal* concept of intelligence, which we will call **rationality**. A system is rational if it does the "right thing," given what it knows.

RATIONALITY

Systems that think like humans	Systems that think rationally
"The exciting new effort to make computers think ... *machines with minds*, in the full and literal sense." (Haugeland, 1985)	"The study of mental faculties through the use of computational models." (Charniak and McDermott, 1985)
"[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning ..." (Bellman, 1978)	"The study of the computations that make it possible to perceive, reason, and act." (Winston, 1992)
Systems that act like humans	**Systems that act rationally**
"The art of creating machines that perform functions that require intelligence when performed by people." (Kurzweil, 1990)	"Computational Intelligence is the study of the design of intelligent agents." (Poole *et al.*, 1998)
"The study of how to make computers do things at which, at the moment, people are better." (Rich and Knight, 1991)	"AI ...is concerned with intelligent behavior in artifacts." (Nilsson, 1998)

Figure 1.1 Some definitions of artificial intelligence, organized into four categories.

Historically, all four approaches to AI have been followed. As one might expect, a tension exists between approaches centered around humans and approaches centered around rationality.[1] A human-centered approach must be an empirical science, involving hypothesis and experimental confirmation. A rationalist approach involves a combination of mathematics and engineering. Each group has both disparaged and helped the other. Let us look at the four approaches in more detail.

Acting humanly: The Turing Test approach

TURING TEST

The **Turing Test**, proposed by Alan Turing (1950), was designed to provide a satisfactory operational definition of intelligence. Rather than proposing a long and perhaps controversial list of qualifications required for intelligence, he suggested a test based on indistinguishability from undeniably intelligent entities—human beings. The computer passes the test if a human interrogator, after posing some written questions, cannot tell whether the written responses come from a person or not. Chapter 26 discusses the details of the test and whether a computer is really intelligent if it passes. For now, we note that programming a computer to pass the test provides plenty to work on. The computer would need to possess the following capabilities:

NATURAL LANGUAGE PROCESSING

- ◇ **natural language processing** to enable it to communicate successfully in English.

[1] We should point out that, by distinguishing between *human* and *rational* behavior, we are not suggesting that humans are necessarily "irrational" in the sense of "emotionally unstable" or "insane." One merely need note that we are not perfect: we are not all chess grandmasters, even those of us who know all the rules of chess; and, unfortunately, not everyone gets an A on the exam. Some systematic errors in human reasoning are cataloged by Kahneman *et al.* (1982).

KNOWLEDGE REPRESENTATION
AUTOMATED REASONING
MACHINE LEARNING

- ◇ **knowledge representation** to store what it knows or hears;
- ◇ **automated reasoning** to use the stored information to answer questions and to draw new conclusions;
- ◇ **machine learning** to adapt to new circumstances and to detect and extrapolate patterns.

TOTAL TURING TEST

Turing's test deliberately avoided direct physical interaction between the interrogator and the computer, because *physical* simulation of a person is unnecessary for intelligence. However, the so-called **total Turing Test** includes a video signal so that the interrogator can test the subject's perceptual abilities, as well as the opportunity for the interrogator to pass physical objects "through the hatch." To pass the total Turing Test, the computer will need

COMPUTER VISION
ROBOTICS

- ◇ **computer vision** to perceive objects, and
- ◇ **robotics** to manipulate objects and move about.

These six disciplines compose most of AI, and Turing deserves credit for designing a test that remains relevant 50 years later. 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 than to duplicate an exemplar. The quest for "artificial flight" succeeded when the Wright brothers and others stopped imitating birds and learned 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."

Thinking humanly: The cognitive modeling approach

If we are going to say that a given program thinks like a human, we must have some way of determining how humans think. We need to get *inside* the actual workings of human minds. There are two ways to do this: through introspection—trying to catch our own thoughts as they go by—and through psychological experiments. 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 and timing behaviors match corresponding human behaviors, 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 to have their program solve problems correctly. They were more concerned with comparing the trace of its reasoning steps to traces of human subjects solving the same problems.

COGNITIVE SCIENCE

The interdisciplinary field of **cognitive science** brings together computer models from AI and experimental techniques from psychology to try to construct precise and testable theories of the workings of the human mind.

Cognitive science is a fascinating field, worthy of an encyclopedia in itself (Wilson and Keil, 1999). We will not attempt to describe what is known of human cognition in this book. 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, and we assume that the reader has access only to 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 continue to fertilize each other, especially in the areas of vision and natural language. Vision in particular has recently made advances via an integrated approach that considers neurophysiological evidence and computational models.

Thinking rationally: The "laws of thought" approach

SYLLOGISMS

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—for example, "Socrates is a man; all men are mortal; therefore, Socrates is mortal." These laws of thought were supposed to govern the operation of the mind; their study initiated the field called **logic**.

LOGIC

Logicians in the 19th century developed a precise notation for statements about all kinds of things in the world and about the relations among them. (Contrast this with ordinary arithmetic notation, which provides mainly for equality and inequality statements about numbers.) By 1965, programs existed that could, in principle, solve *any* solvable problem described in logical notation.[2] The so-called **logicist** tradition within artificial intelligence hopes to build on such programs to create intelligent systems.

LOGICIST

There are two main obstacles to this approach. First, it is not easy to take informal knowledge and state it in the formal terms required by logical notation, particularly when the knowledge is less than 100% certain. Second, there is a big difference between being able to solve a problem "in principle" and doing so in practice. Even problems with just a few dozen facts can exhaust the computational resources of any computer unless it has some guidance as to which reasoning steps to try first. Although both of these obstacles apply to *any* attempt to build computational reasoning systems, they appeared first in the logicist tradition.

Acting rationally: The rational agent approach

AGENT

An **agent** is just something that acts (*agent* comes from the Latin *agere*, to do). But computer agents are expected to have other attributes that distinguish them from mere "programs," such as operating under autonomous control, perceiving their environment, persisting over a prolonged time period, adapting to change, and being capable of taking on another's 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 reason logically to the conclusion that a given action will achieve one's goals and then to act on that conclusion. On the other hand, correct inference is not *all* of rationality, because there are often situations where there is no provably correct thing to do, yet something must still be done. There are also 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.

[2] If there is no solution, the program might never stop looking for one.

All the skills needed for the Turing Test are there to allow rational actions. Thus, we need the ability to represent knowledge and reason with it because this enables us to reach good decisions in a wide variety of situations. We need to be able to generate comprehensible sentences in natural language because saying those sentences helps us get by in a complex society. We need learning not just for erudition, but because having a better idea of how the world works enables us to generate more effective strategies for dealing with it. We need visual perception not just because seeing is fun, but to get a better idea of what an action might achieve—for example, being able to see a tasty morsel helps one to move toward it.

For these reasons, the study of AI as rational-agent design has at least two advantages. 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 than are approaches based on human behavior or human thought because the standard of rationality is clearly defined and completely general. Human behavior, on the other hand, is well-adapted for one specific environment and is the product, in part, of a complicated and largely unknown evolutionary process that still is far from producing perfection. *This book will therefore concentrate on general principles of rational agents and on components for constructing them.* We will see that despite the apparent simplicity with which the problem can be stated, an enormous variety of issues come up when we try to solve it. Chapter 2 outlines some of these issues in more detail.

One important point to keep in mind: We will see before too long that achieving perfect rationality—always doing the right thing—is not feasible in complicated environments. The computational demands are just too high. For most of the book, however, we will adopt the working hypothesis that perfect rationality is a good starting point for analysis. It simplifies the problem and provides the appropriate setting for most of the foundational material in the field. Chapters 6 and 17 deal explicitly with the issue of **limited rationality**—acting appropriately when there is not enough time to do all the computations one might like.

LIMITED RATIONALITY

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 is forced to concentrate on a small number of people, events, and ideas and to ignore 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.

Philosophy (428 B.C.–present)

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

Aristotle (384-322 B.C.) 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. Much later, Ramon Lull (d. 1315) had the idea that useful reasoning could actually be carried out by a mechanical artifact. His "concept wheels" are on the cover of this book. Thomas Hobbes (1588–1679) proposed that reasoning was like numerical computation, that "we add and subtract in our silent thoughts." The automation of computation itself was already well under way; 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), although the Pascaline, built in 1642 by Blaise Pascal (1623–1662), is more famous. Pascal wrote that "the arithmetical machine 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.

Now that we have the idea of a set of rules that can describe the formal, rational part of the mind, the next step is to consider the mind as a physical system. René Descartes (1596–1650) gave the first clear discussion of the distinction between mind and matter and of the problems that arise. One problem with a purely physical conception of the mind is that it 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 toward the center of the earth. Although a strong advocate of the power of reasoning, Descartes was also a proponent of **dualism**. He held that there is a part of the human mind (or soul or spirit) that is outside of nature, exempt from physical laws. Animals, on the other hand, did not possess this dual quality; they could be treated as machines. An alternative to dualism is **materialism**, which holds that the brain's operation according to the laws of physics *constitutes* the mind. Free will is simply the way that the perception of available choices appears to the choice process.

DUALISM

MATERIALISM

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*,[3] is characterized by a dictum of John Locke (1632–1704): "Nothing is in the understanding, which was not first in the senses." David Hume's (1711–1776) *A Treatise of Human Nature* (Hume, 1739) proposed what is now known as the principle of **induction**: that general rules are acquired by exposure to repeated associations between their elements. Building on the work of Ludwig Wittgenstein (1889–1951) and Bertrand Russell (1872–1970), the famous Vienna Circle, led by Rudolf Carnap (1891–1970), 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.[4] The **confirmation theory** of Carnap and Carl Hempel (1905–1997) attempted to understand how knowledge can be acquired from experience. Carnap's book *The Logical Structure of*

EMPIRICISM

INDUCTION

LOGICAL POSITIVISM

OBSERVATION SENTENCES

CONFIRMATION THEORY

[3] An update of Aristotle's *Organon*, or instrument of thought.

[4] In this picture, all meaningful statements can be verified or falsified either by analyzing the meaning of the words or by carrying out experiments. Because this rules out most of metaphysics, as was the intention, logical positivism was unpopular in some circles.

the World (1928) defined an explicit computational procedure for extracting knowledge from elementary experiences. It was probably the first theory of mind as a computational process.

The final element in the philosophical picture of the mind is the connection between knowledge and action. This question is vital to AI, because intelligence requires action as well as reasoning. Moreover, only by understanding how actions are justified can we understand how to build an agent whose actions are justifiable (or rational). Aristotle argued that actions are justified by a logical connection between goals and knowledge of the action's outcome (the last part of this extract also appears on the front cover of this book):

> 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. (Nussbaum, 1978, p. 40)

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 GPS program. We would now call it a regression planning system. (See Chapter 11.)

Goal-based analysis is useful, but does not say what to do when several actions will achieve the goal, or when no action will achieve it completely. Antoine Arnauld (1612–1694) correctly described a quantitative formula for deciding what action to take in cases like this (see Chapter 16). John Stuart Mill's (1806–1873) book *Utilitarianism* (Mill, 1863) promoted the idea of rational decision criteria in all spheres of human activity. The more formal theory of decisions is discussed in the following section.

Mathematics (c. 800–present)

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

Philosophers staked out most of the important ideas of AI, but the leap to a formal science required a level of mathematical formalization in three fundamental areas: logic, computation, and probability.

The idea of formal logic can be traced back to the philosophers of ancient Greece (see Chapter 7), 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 as the most basic knowledge representation system.[5] Alfred Tarski (1902–1983) introduced a theory of reference that shows how to relate the objects in a logic to objects in the real world. The next step was to determine the limits of what could be done with logic and computation.

ALGORITHM

The first nontrivial **algorithm** is thought to be Euclid's algorithm for computing greatest common denominators. The study of algorithms as objects in themselves goes back to al-Khowarazmi, a Persian mathematician of the 9th century, 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. In 1900, David Hilbert (1862–1943) presented a list of 23 problems that he correctly predicted would occupy mathematicians for the bulk of the century. The final problem asks whether there is an algorithm for deciding the truth of any logical proposition involving the natural numbers—the famous *Entscheidungsproblem*, or decision problem. Essentially, Hilbert was asking whether there were fundamental limits to the power of effective proof procedures. In 1930, 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, he showed that real limits do exist.

INCOMPLETENESS THEOREM

His **incompleteness theorem** showed that in any language expressive enough to describe the properties of the natural numbers, there are true statements that are undecidable in the sense that their truth cannot be established by any algorithm.

This fundamental result can also be interpreted as showing that there are some functions on the integers that 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* capable of being computed. This notion is actually slightly problematic, because the notion of a computation or effective procedure really cannot be given a formal definition. However, the Church–Turing thesis, which states that the Turing machine (Turing, 1936) is capable of computing any computable function, is generally accepted as providing a sufficient definition. 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.

INTRACTABILITY

Although undecidability and noncomputability are important to an understanding of computation, the notion of **intractability** has had a much greater impact. 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. Therefore, one should strive to divide

[5] Frege's proposed notation for first-order logic never became popular, for reasons that are apparent immediately from the example on the front cover.

the overall problem of generating intelligent behavior into tractable subproblems rather than intractable ones.

NP-COMPLETENESS

How can one recognize an intractable problem? The theory of **NP-completeness**, pioneered by Steven Cook (1971) and Richard Karp (1972), provides a method. Cook and Karp showed the existence of large classes of canonical combinatorial search and reasoning problems that are NP-complete. 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 will characterize intelligent systems. Put crudely, the world is an *extremely* large problem instance! In recent years, AI has helped explain why some instances of NP-complete problems are hard, yet others are easy (Cheeseman *et al.*, 1991).

PROBABILITY

Besides logic and computation, the third great contribution of mathematics to AI is the theory of **probability**. The Italian Gerolamo Cardano (1501–1576) first framed the idea of probability, describing it in terms of the possible outcomes of gambling events. Probability quickly became an invaluable part of all the quantitative sciences, helping to deal with uncertain measurements and incomplete theories. Pierre Fermat (1601–1665), Blaise Pascal (1623–1662), James Bernoulli (1654–1705), 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 and the resulting field called Bayesian analysis form the basis of most modern approaches to uncertain reasoning in AI systems.

Economics (1776–present)

- How should we make decisions so as to maximize payoff?
- 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 got its start in 1776, when Scottish philosopher Adam Smith (1723–1790) published *An Inquiry into the Nature and Causes of the Wealth of Nations*. While the ancient Greeks and others had made contributions to economic thought, Smith was the first to treat it as a science, using the idea that economies can be thought of as consisting of individual agents maximizing their own economic well-being. Most people think of economics as being about money, but economists will say that they are really studying how people make choices that lead to preferred outcomes. The mathematical treatment of "preferred outcomes" or **utility** was first formalized by Léon Walras (pronounced "Valrasse") (1834-1910) and was improved by Frank Ramsey (1931) and later by John von Neumann and Oskar Morgenstern in their book *The Theory of Games and Economic Behavior* (1944).

DECISION THEORY

Decision theory, which combines probability theory with utility theory, provides a formal and complete framework for 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 act in a random fashion, or at least in a way that appears random to the adversaries.

GAME THEORY

For the most part, economists did not address the third question listed above, namely, 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 civilian applications in complex management decisions. The work of Richard Bellman (1957) formalized a class of sequential decision problems called **Markov decision processes**, which we study in Chapters 17 and 21.

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. Herbert Simon (1916–2001), the pioneering AI researcher, 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). In the 1990s, there has been a resurgence of interest in decision-theoretic techniques for agent systems (Wellman, 1995).

SATISFICING

Neuroscience (1861–present)

- How do brains process information?

NEUROSCIENCE

Neuroscience is the study of the nervous system, particularly the brain. The exact way in which the brain enables thought is one of the great mysteries of science. It has been appreciated for thousands of years that the brain is somehow involved in thought, 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 B.C. 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, the spleen, and the pineal gland.

Paul Broca's (1824–1880) study of aphasia (speech deficit) in brain-damaged patients in 1861 reinvigorated the field and persuaded the medical establishment of the existence of localized areas of the brain responsible for specific cognitive functions. In particular, he showed that speech production was localized to a portion of the left hemisphere now called Broca's area.[7] By that time, it was known that the brain consisted 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 in the brain (see Figure 1.2). This technique

NEURONS

[6] Since then, it has been discovered that some species of dolphins and whales have relatively larger brains. The large size of human brains is now thought to be enabled in part by recent improvements in its cooling system.

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

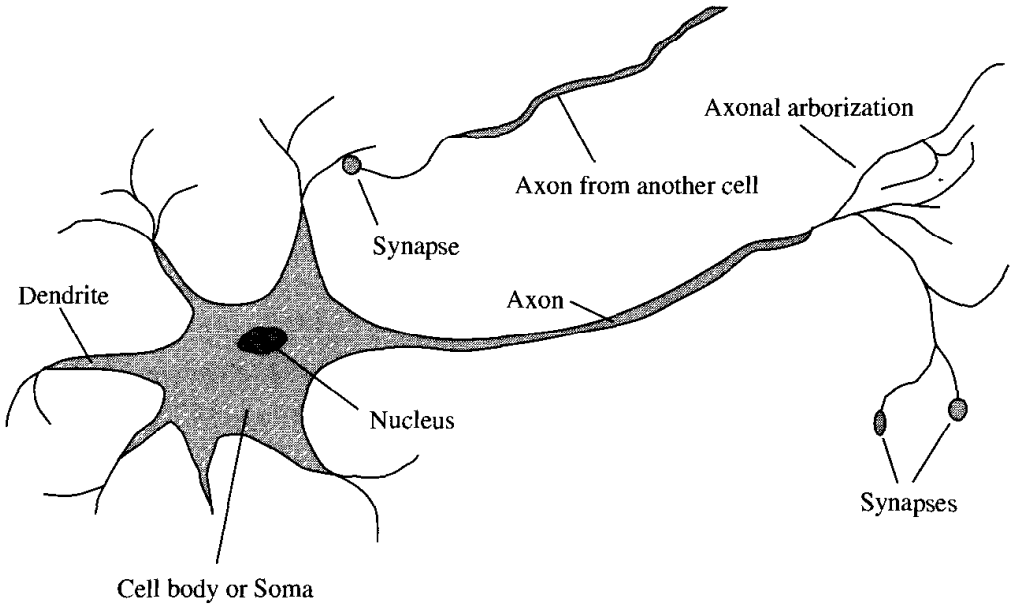


Figure 1.2 The parts of a nerve cell or neuron. Each neuron consists of a cell body, or soma, that contains a cell nucleus. Branching out from the cell body are a number of fibers called dendrites and a single long fiber called the axon. The axon stretches out for a long distance, much longer than the scale in this diagram indicates. Typically they are 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 position and 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, extending the full depth of the cortex, which is about 4 mm in humans. A column contains about 20,000 neurons.

was used by Santiago Ramon y Cajal (1852–1934) in his pioneering studies of the brain's neuronal structures.[8]

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.

The measurement of intact brain activity began in 1929 with the invention by Hans Berger of the electroencephalograph (EEG). The recent development of functional magnetic resonance imaging (fMRI) (Ogawa *et al.*, 1990) 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 recording of

[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 rather antagonistic acceptance speeches.

	Computer	Human Brain
Computational units	1 CPU, 10^8 gates	10^{11} neurons
Storage units	10^{10} bits RAM	10^{11} neurons
	10^{11} bits disk	10^{14} synapses
Cycle time	10^{-9} sec	10^{-3} sec
Bandwidth	10^{10} bits/sec	10^{14} bits/sec
Memory updates/sec	10^9	10^{14}

Figure 1.3 A crude comparison of the raw computational resources available to computers (*circa* 2003) and brains. The computer's numbers have all increased by at least a factor of 10 since the first edition of this book, and are expected to do so again this decade. The brain's numbers have not changed in the last 10,000 years.

neuron activity. Despite these advances, we are still a long way from understanding how any of these cognitive processes actually work.

The truly amazing conclusion is that *a collection of simple cells can lead to thought, action, and consciousness* or, in other words, that *brains cause minds* (Searle, 1992). The only real alternative theory is mysticism: that there is some mystical realm in which minds operate that is beyond physical science.

Brains and digital computers perform quite different tasks and have different properties. Figure 1.3 shows that there are 1000 times more neurons in the typical human brain than there are gates in the CPU of a typical high-end computer. Moore's Law[9] predicts that the CPU's gate count will equal the brain's neuron count around 2020. Of course, little can be inferred from such predictions; moreover, the difference in storage capacity is minor compared to the difference in switching speed and in parallelism. Computer chips can execute an instruction in a nanosecond, whereas neurons are millions of times slower. Brains more than make up for this, however, because all the neurons and synapses are active simultaneously, whereas most current computers have only one or at most a few CPUs. Thus, *even though a computer is a million times faster in raw switching speed, the brain ends up being 100,000 times faster at what it does.*

Psychology (1879–present)

- 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* is even now 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 associa-

[9] Moore's Law says that the number of transistors per square inch doubles every 1 to 1.5 years. Human brain capacity doubles roughly every 2 to 4 million years.

tive 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 an experimenter would ever disconfirm his or her own theories. Biologists studying animal behavior, on the other hand, lacked introspective data and developed an objective methodology, as described by H. S. Jennings (1906) in his influential work *Behavior of the Lower Organisms*. Applying this viewpoint to humans, the **behaviorism** movement, led by John Watson (1878–1958), rejected *any* theory involving mental processes on the grounds that introspection could not provide reliable evidence. Behaviorists insisted on studying only objective measures of the percepts (or *stimulus*) given to an animal and its resulting actions (or *response*). Mental constructs such as knowledge, beliefs, goals, and reasoning steps were dismissed as unscientific "folk psychology." Behaviorism discovered a lot about rats and pigeons, but had less success at understanding humans. Nevertheless, it exerted a strong hold on psychology (especially in the United States) from about 1920 to 1960.

BEHAVIORISM

The view of the brain as an information-processing device, which is a principal characteristic of **cognitive psychology**, can be traced back at least to the works of William James[10] (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 their being made of molecules that have neither. Craik specified the three key steps of a knowledge-based agent: (1) the stimulus must be translated into an internal representation, (2) the representation is manipulated by cognitive processes to derive new internal representations, and (3) these are in turn retranslated back into action. He clearly explained why this was a good design for an agent:

COGNITIVE PSYCHOLOGY

> 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) included some of the first information-processing models of psychological phenomena. 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. (We shall see that this is 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

COGNITIVE SCIENCE

[10] William James was the brother of novelist Henry James. It is said that Henry wrote fiction as if it were psychology and William wrote psychology as if it were fiction.

address the psychology of memory, language, and logical thinking, respectively. It is now a common view among psychologists that "a cognitive theory should be like a computer program" (Anderson, 1980), that is, it should describe a detailed information-processing mechanism whereby some cognitive function might be implemented.

Computer engineering (1940–present)

- How can we build an efficient computer?

For artificial intelligence to succeed, we need two things: intelligence and an artifact. The computer has been the artifact of choice. 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,[11] built in 1940 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.[12] 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 John Eckert, that proved to be the most influential forerunner of modern computers.

In the half-century since then, each generation of computer hardware has brought an increase in speed and capacity and a decrease in price. Performance doubles every 18 months or so, with a decade or two to go at this rate of increase. After that, we will need molecular engineering or some other new technology.

Of course, there were calculating devices before the electronic computer. The earliest automated machines, dating from the 17th century, were discussed on page 6. The first *programmable* machine was a loom devised in 1805 by Joseph Marie Jacquard (1752–1834) that used punched cards to store instructions for the pattern to be woven. In the mid-19th century, Charles Babbage (1792–1871) designed two machines, neither of which he completed. The "Difference Engine," which appears on the cover of this book, was intended to compute mathematical tables for engineering and scientific projects. It was finally built and shown to work in 1991 at the Science Museum in London (Swade, 1993). Babbage's "Analytical Engine" was far more ambitious: it included addressable memory, stored programs, and conditional jumps and was the first artifact capable of universal computation. Babbage's colleague Ada Lovelace, daughter of the poet Lord Byron, was perhaps the world's first programmer. (The programming language Ada is named after her.) She wrote programs for the unfinished Analytical Engine and even speculated that the machine could play chess or compose music.

[11] Heath Robinson was a cartoonist famous for his depictions of whimsical and absurdly complicated contraptions for everyday tasks such as buttering toast.

[12] In the postwar period, Turing wanted to use these computers for AI research—for example, one of the first chess programs (Turing *et al.*, 1953). His efforts were blocked by the British government.

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, dynamic, and object-oriented programming.

Control theory and Cybernetics (1948–present)

- How can artifacts operate under their own control?

Ktesibios of Alexandria (c. 250 B.C.) built the first self-controlling machine: a water clock with a regulator that kept the flow of water running through it at a constant, predictable pace. 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. The mathematical theory of stable feedback systems was developed in the 19th century.

CONTROL THEORY

The central figure in the creation of what is now called **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 conferences that explored the new mathematical and computational models of cognition and influenced many other researchers in the behavioral sciences. Wiener's book *Cybernetics* (1948) became a bestseller and awoke the public to the possibility of artificially intelligent machines.

CYBERNETICS

OBJECTIVE FUNCTION

Modern control theory, especially the branch known as stochastic optimal control, has as its goal the design of systems that maximize an **objective function** over time. This roughly matches our view of AI: designing systems that behave optimally. Why, then, are AI and control theory two different fields, especially given 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; furthermore, exact analysis is typically feasible only for *linear* systems. AI was founded in part as a way to escape from the limitations of the mathematics of control theory in the 1950s. The tools of logical inference and computation allowed AI researchers to consider some problems such as language, vision, and planning, that fell completely outside the control theorist's purview.

Linguistics (1957–present)

- 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 Noam Chomsky, who had just published a book on his own theory, *Syntactic Structures*. Chomsky showed how the behaviorist theory did not address the notion of creativity in language—it did not explain how a child could understand and make up sentences that he or she had never heard before. Chomsky's theory—based on syntactic models going back to the Indian linguist Panini (c. 350 B.C.)—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 soon 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

With the background material behind us, we are ready to cover the development of AI itself.

The gestation of artificial intelligence (1943–1955)

The first work that is now generally recognized as AI was done by Warren McCulloch and Walter Pitts (1943). 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 net structures. McCulloch and Pitts also suggested that suitably defined networks could learn. Donald Hebb (1949) demonstrated a simple updating rule for modifying the connection strengths between neurons. His rule, now called **Hebbian learning**, remains an influential model to this day.

Two graduate students in the Princeton mathematics department, Marvin Minsky and Dean Edmonds, built the first neural network computer in 1951. 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. Minsky's 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." Minsky was later to prove influential theorems showing the limitations of neural network research.

There were a number of early examples of work that can be characterized as AI, but it was Alan Turing who first articulated a complete vision of AI in his 1950 article "Computing Machinery and Intelligence." Therein, he introduced the Turing test, machine learning, genetic algorithms, and reinforcement learning.

The birth of artificial intelligence (1956)

Princeton was home to another influential figure in AI, John McCarthy. After graduation, McCarthy moved to Dartmouth College, which was to become the official birthplace of the field. McCarthy 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 Trenchard More from Princeton, Arthur Samuel from IBM, and Ray Solomonoff and Oliver Selfridge from MIT.

Two researchers from Carnegie Tech,[13] Allen Newell and Herbert Simon, rather stole the show. Although the others had ideas and in some cases programs for particular applications such as checkers, Newell and Simon already had a reasoning program, the Logic Theorist (LT), about which Simon claimed, "We have invented a computer program capable of thinking non-numerically, and thereby solved the venerable mind–body problem."[14] 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 Simon showed him that the program 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.

The Dartmouth workshop did not lead to any new breakthroughs, but it did introduce all the major figures to each other. For the next 20 years, the field would be dominated by these people and their students and colleagues at MIT, CMU, Stanford, and IBM. Perhaps the longest-lasting thing to come out of the workshop was an agreement to adopt McCarthy's new name for the field: **artificial intelligence**. Perhaps "computational rationality" would have been better, but "AI" has stuck.

Looking at the proposal for the Dartmouth workshop (McCarthy *et al.*, 1955), we can see why it was necessary for AI to become a separate field. Why couldn't all the work done

[13] Now Carnegie Mellon University (CMU).

[14] 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.

in AI have taken place under the name of control theory, or operations research, or decision theory, which, after all, have objectives similar to those of AI? Or why isn't AI a branch of mathematics? The first answer is that AI from the start embraced the idea of duplicating human faculties like creativity, self-improvement, and language use. None of the other fields were addressing these issues. The second answer is methodology. AI is the only one of these fields that is clearly a branch of computer science (although operations research does share an emphasis on computer simulations), and AI is the only field to attempt to build machines that will function autonomously in complex, changing environments.

Early enthusiasm, great expectations (1952–1969)

The early years of AI were full of successes—in a limited way. Given the primitive computers and programming tools of the time, and the fact that only a few years earlier computers were seen as things that could do arithmetic and no more, it was astonishing whenever a computer did anything remotely clever. The intellectual establishment, by and large, preferred to believe that "a machine can never do X." (See Chapter 26 for a long list of X's gathered by Turing.) AI researchers naturally responded by demonstrating one X after another. John McCarthy referred to this period as the "Look, Ma, no hands!" era.

Newell and Simon's early success was followed up with the General Problem Solver, or GPS. Unlike Logic Theorist, 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. Starting in 1952, Arthur Samuel wrote a series of programs for checkers (draughts) that eventually learned to play at a strong amateur level. Along the way, he 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 February 1956, creating a very 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. Chapter 6 covers game playing, and Chapter 21 describes and expands on the learning techniques used by Samuel.

John McCarthy moved from Dartmouth to MIT and there made three crucial contributions in one historic year: 1958. In MIT AI Lab Memo No. 1, McCarthy defined the high-level language **Lisp**, which was to become the dominant AI programming language. Lisp is the

LISP

second-oldest major high-level language in current use, one year younger than FORTRAN. With Lisp, McCarthy had the tool he needed, but access to scarce and expensive computing resources was also a serious problem. In response, he and others at MIT invented time sharing. Also in 1958, McCarthy published a paper entitled *Programs with Common Sense*, in which he described the Advice Taker, a hypothetical program that can be seen as the first complete AI system. Like the Logic Theorist and Geometry Theorem Prover, McCarthy's program was designed to use knowledge to search for solutions to problems. But unlike the others, it was to embody general knowledge of the world. For example, he showed how some simple axioms would enable the program to generate a plan to drive to the airport to catch a plane. The program was also designed so that it could 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 of the way an agent's actions affect the world and to be able to manipulate these representations with deductive processes. It is remarkable how much of the 1958 paper remains relevant even today.

1958 also marked the year that Marvin Minsky moved to MIT. His initial collaboration with McCarthy did not last, however. McCarthy stressed representation and reasoning in formal logic, whereas Minsky was more interested in getting programs to work and eventually developed an anti-logical 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 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 new Stanford Research Institute (SRI). The latter project, discussed further in Chapter 25, was the first to demonstrate the complete integration of logical reasoning and physical activity.

MICROWORLDS

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 (1963a) 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, such as the one in Figure 1.4. 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 was 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 1.5. 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

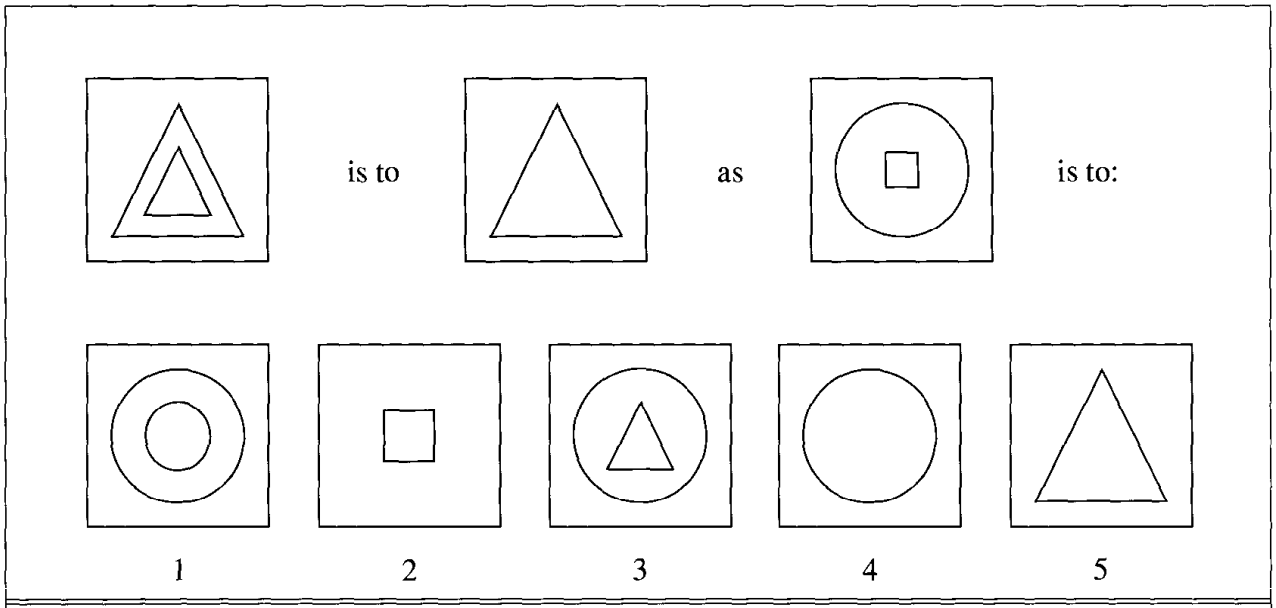


Figure 1.4 An example problem solved by Evans's ANALOGY program.

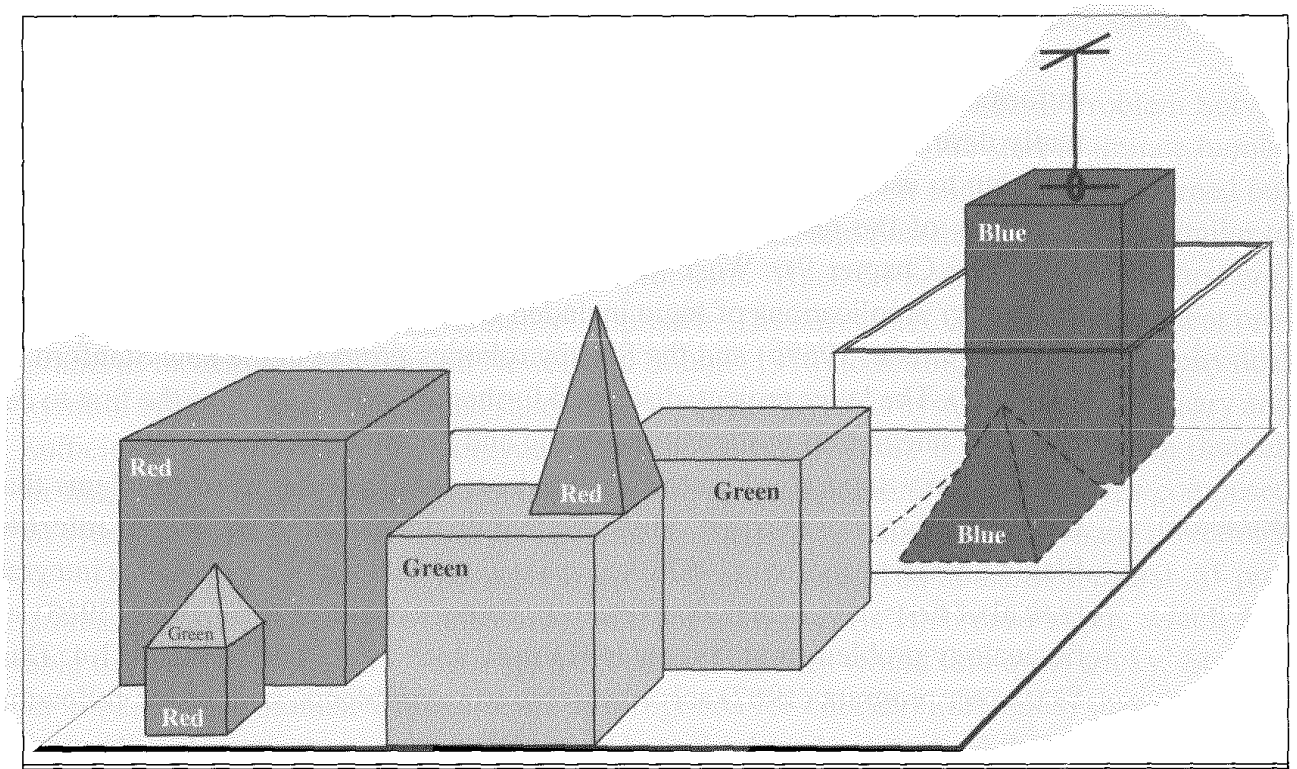


Figure 1.5 A scene from the blocks world. SHRDLU (Winograd, 1972) has just completed the command, "Find a block which is taller than the one you are holding and put it in the box."

of Terry Winograd (1972), and the planner of Scott Fahlman (1974).

Early work building on the neural networks of McCulloch and Pitts also flourished. The work of Winograd and 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**. Rosenblatt proved the **perceptron convergence theorem**, showing that his learning algorithm could adjust the connection strengths of a perceptron to match any input data, provided such a match existed. These topics are covered in Chapter 20.

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.

Terms such as "visible future" can be interpreted in various ways, but Simon also made a more concrete prediction: 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 over-confidence was due to the promising performance of early AI systems on simple examples. In almost all cases, however, these early systems turned out to fail miserably when tried out on wider selections of problems and on more difficult problems.

The first kind of difficulty arose because most early programs contained little or no knowledge of their subject matter; they succeeded by means of simple syntactic manipulations. A typical story occurred in early machine translation efforts, which were generously funded by the U.S. National Research Council in an attempt to speed up the translation of Russian scientific papers in the wake of the Sputnik launch in 1957. It was thought initially that simple syntactic transformations based on the grammars of Russian and English, and word replacement using an electronic dictionary, would suffice to preserve the exact meanings of sentences. The fact is that translation requires general knowledge of the subject matter in order to resolve ambiguity and establish the content of the sentence. The famous re-translation of "the spirit is willing but the flesh is weak" as "the vodka is good but the meat is rotten" illustrates the difficulties encountered. In 1966, a report by an advisory committee found that "there has been no machine translation of general scientific text, and none is in immediate prospect." All U.S. government funding for academic translation projects was canceled. Today, machine translation is an imperfect but widely used tool for technical, commercial, government, and Internet documents.

The second kind of difficulty was the intractability of many of the problems that AI was attempting to solve. Most of the early AI programs solved problems 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.*

MACHINE EVOLUTION

The illusion of unlimited computational power was not confined to problem-solving programs. Early experiments in **machine evolution** (now called **genetic algorithms**) (Friedberg, 1958; Friedberg *et al.*, 1959) were based on the undoubtedly correct belief that by making an appropriate series of small mutations to a machine code program, one can generate a program with good performance for any particular simple 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. Modern genetic algorithms use better representations and have shown more success.

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 for multilayer networks that were to cause an enormous resurgence in neural-net research in the late 1980s were actually discovered first in 1969 (Bryson and Ho, 1969).

Knowledge-based systems: The key to power? (1969–1979)

WEAK METHODS

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

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

For example, the mass spectrum might contain a peak at $m = 15$, corresponding to the mass of a methyl (CH_3) fragment.

The naive version of the program generated all possible structures consistent with the formula, and then predicted what mass spectrum would be observed for each, comparing this with the actual spectrum. As one might expect, this is intractable for decent-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** there are two peaks at x_1 and x_2 such that
> (a) $x_1 + x_2 = M + 28$ (M is the mass of the whole molecule);
> (b) $x_1 - 28$ is a high peak;
> (c) $x_2 - 28$ is a high peak;
> (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. DENDRAL was powerful because

> All the relevant theoretical knowledge to solve these problems has been mapped over from its general form in the [spectrum prediction component] ("first principles") to efficient special forms ("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. Later systems also incorporated the main theme of McCarthy's Advice Taker approach—the clean separation of the knowledge (in the form of rules) from the reasoning component.

With this lesson in mind, 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 of human expertise. The next major effort was in the area of medical diagnosis. Feigenbaum, Buchanan, and Dr. Edward Shortliffe developed MYCIN to diagnose 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, who in turn acquired them from textbooks, other experts, and direct experience of cases. 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.

EXPERT SYSTEMS

The importance of domain knowledge was also apparent in the area of understanding natural language. Although Winograd's SHRDLU system for understanding natural language had engendered a good deal of excitement, its dependence on syntactic analysis caused some of the same problems as occurred in the early machine translation work. It was able to overcome ambiguity and understand pronoun references, but this was mainly because it was

designed specifically for one area—the blocks world. Several researchers, including Eugene Charniak, a fellow graduate student of Winograd's at MIT, suggested that robust language understanding would require general knowledge about the world and a general method for using that knowledge.

At Yale, the linguist-turned-AI-researcher Roger Schank emphasized this point, 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; Dyer, 1983) 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 problems included representing stereotypical situations (Cullingford, 1981), describing human memory organization (Rieger, 1976; Kolodner, 1983), and understanding plans and goals (Wilensky, 1983).

The widespread growth of applications to real-world problems caused a concurrent increase in the demands for workable knowledge representation schemes. A large number of different representation and reasoning languages were developed. Some were based on logic—for example, the Prolog language became popular in Europe, 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

AI becomes an industry (1980–present)

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. Du Pont had 100 in use and 500 in development, saving an estimated $10 million a year. Nearly every major U.S. corporation had its own AI group and was either using or investigating expert systems.

In 1981, the Japanese announced the "Fifth Generation" project, a 10-year plan to build intelligent computers running Prolog. In response the United States formed the Microelectronics and Computer Technology Corporation (MCC) as a research consortium designed to assure national competitiveness. In both cases, AI was part of a broad effort, including chip design and human-interface research. However, the AI components of MCC and the Fifth Generation projects never met their ambitious goals. In Britain, the Alvey report reinstated the funding that was cut by the Lighthill report.[15]

Overall, the AI industry boomed from a few million dollars in 1980 to billions of dollars in 1988. Soon after that came a period called the "AI Winter," in which many companies suffered as they failed to deliver on extravagant promises.

[15] To save embarrassment, a new field called IKBS (Intelligent Knowledge-Based Systems) was invented because Artificial Intelligence had been officially canceled.

The return of neural networks (1986–present)

Although computer science had largely abandoned the field of neural networks in the late 1970s, work continued in other fields. Physicists such as John Hopfield (1982) used techniques from statistical mechanics to analyze the storage and optimization properties of networks, treating collections of nodes like collections of atoms. Psychologists including David Rumelhart and Geoff Hinton continued the study of neural-net models of memory. As we discuss in Chapter 20, the real impetus came in the mid-1980s when at least four different groups reinvented the back-propagation learning algorithm first found in 1969 by Bryson and Ho. The algorithm was applied to many learning problems in computer science and psychology, and the widespread dissemination of the results in the collection *Parallel Distributed Processing* (Rumelhart and McClelland, 1986) caused great excitement.

CONNECTIONIST

These so-called **connectionist** models of intelligent systems 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 (Smolensky, 1988). It might seem obvious that at some level humans manipulate symbols—in fact, Terrence Deacon's book *The Symbolic Species* (1997) suggests that this is the *defining characteristic* of humans, but the most ardent connectionists questioned whether symbol manipulation had any real explanatory role in detailed models of cognition. This question remains unanswered, but the current view is that connectionist and symbolic approaches are complementary, not competing.

AI becomes a science (1987–present)

Recent years have seen a revolution in both the content and the methodology of work in artificial intelligence.[16] It is now more common to build on existing theories than to propose brand new ones, to base claims on rigorous theorems or hard experimental evidence rather than on intuition, and to show relevance to real-world applications rather than toy examples.

AI was founded in part as a rebellion against the limitations of existing fields like control theory and statistics, but now it is embracing those fields. As David McAllester (1998) put it,

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

In terms of methodology, AI has finally come firmly under the scientific method. To be accepted, hypotheses must be subjected to rigorous empirical experiments, and the results must

[16] 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. Whether that stability will be disrupted by a new scruffy idea is another question.

be analyzed statistically for their importance (Cohen, 1995). Through the use of the Internet and shared repositories of test data and code, it is now possible to replicate experiments.

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 were demonstrated on only a few specially selected examples. In recent years, approaches based on **hidden Markov models** (HMMs) have come to dominate the area. Two aspects of HMMs are relevant. First, they are based on a rigorous mathematical theory. This has 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 the HMMs have been improving their scores steadily. Speech technology and the related field of handwritten character recognition are already making the transition to widespread industrial and consumer applications.

Neural networks also fit this trend. Much of the work on neural nets in the 1980s was done in an attempt to scope out what could be done and to learn how neural nets differ from "traditional" techniques. Using improved methodology and theoretical frameworks, the field arrived at an understanding in which neural nets can now be compared with corresponding techniques from statistics, pattern recognition, and machine learning, and the most promising technique can be applied to each application. As a result of these developments, so-called

DATA MINING

data mining technology has spawned a vigorous new industry.

Judea Pearl's (1988) *Probabilistic Reasoning in Intelligent Systems* led to a new acceptance of probability and decision theory in AI, following a resurgence of interest epitomized by Peter Cheeseman's (1985) article "In Defense of Probability." The **Bayesian network** formalism was invented to allow efficient representation of, and rigorous reasoning with, uncertain knowledge. This approach largely overcomes many problems of the probabilistic reasoning systems of the 1960s and 1970s; it now dominates AI research on uncertain reasoning and expert systems. The approach allows for learning from experience, and it combines the best of classical AI and neural nets. Work by Judea Pearl (1982a) and by Eric Horvitz and David Heckerman (Horvitz and Heckerman, 1986; Horvitz *et al.*, 1986) promoted the idea of *normative* expert systems: ones that act rationally according to the laws of decision theory and do not try to imitate the thought steps of human experts. The Windows™ operating system includes several normative diagnostic expert systems for correcting problems. Chapters 13 to 16 cover this area.

Similar gentle revolutions have occurred in robotics, computer vision, and knowledge representation. A better understanding of the problems and their complexity properties, combined with increased mathematical sophistication, has led to workable research agendas and robust methods. In many cases, formalization and specialization have also led to fragmentation: topics such as vision and robotics are increasingly isolated from "mainstream" AI work. The unifying view of AI as rational agent design is one that can bring unity back to these disparate fields.

The emergence of intelligent agents (1995–present)

Perhaps encouraged by the progress in solving the subproblems of AI, researchers have also started to look at the "whole agent" problem again. The work of Allen Newell, John Laird, and Paul Rosenbloom on SOAR (Newell, 1990; Laird *et al.*, 1987) is the best-known example of a complete agent architecture. The so-called situated movement aims to understand the workings of agents embedded in real environments with continuous sensory inputs. One of the most important environments for intelligent agents is the Internet. AI systems have become so common in web-based applications that the "-bot" suffix has entered everyday language. Moreover, AI technologies underlie many Internet tools, such as search engines, recommender systems, and Web site construction systems.

Besides the first edition of this text (Russell and Norvig, 1995), other recent texts have also adopted the agent perspective (Poole *et al.*, 1998; Nilsson, 1998). One consequence of trying to build complete agents is the realization that the previously isolated subfields of AI might need to be reorganized somewhat when their results are to be tied together. In particular, it is now widely appreciated that sensory systems (vision, sonar, speech recognition, etc.) cannot deliver perfectly reliable information about the environment. Hence, reasoning and planning systems must be able to handle uncertainty. A second major consequence of the agent perspective is that AI has been drawn into much closer contact with other fields, such as control theory and economics, that also deal with agents.

1.4 THE STATE OF THE ART

What can AI do today? A concise answer is difficult, because there are so many activities in so many subfields. Here we sample a few applications; others appear throughout the book.

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 it monitored the operation of the spacecraft as the plans were executed—detecting, diagnosing, and recovering from problems as they occurred.

Game playing: IBM's Deep Blue became the first computer program to defeat the world champion in a chess match when it bested Garry Kasparov by a score of 3.5 to 2.5 in an exhibition match (Goodman and Keene, 1997). Kasparov said that he felt a "new kind of intelligence" across the board from him. *Newsweek* magazine described the match as "The brain's last stand." The value of IBM's stock increased by $18 billion.

Autonomous control: The ALVINN computer vision system was trained to steer a car to keep it following a lane. It was placed in CMU's NAVLAB computer-controlled minivan and used to navigate across the United States—for 2850 miles it was in control of steering the vehicle 98% of the time. A human took over the other 2%, mostly at exit ramps. NAVLAB has video cameras that transmit road images to ALVINN, which then computes the best direction to steer, based on experience from previous training runs.

Diagnosis: Medical diagnosis programs based on probabilistic analysis have been able to perform at the level of an expert physician in several areas of medicine. Heckerman (1991) describes a case where a leading expert on lymph-node pathology scoffs at a program's diagnosis of an especially difficult case. The creators of the program suggest he ask the computer for an explanation of the diagnosis. The machine points out the major factors influencing its decision and explains the subtle interaction of several of the symptoms in this case. Eventually, the expert agrees with the program.

Logistics Planning: 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, and conflict resolution among all parameters. The AI planning techniques allowed a plan to be generated in hours that would have taken weeks with older methods. The Defense Advanced Research Project Agency (DARPA) stated that this single application more than paid back DARPA's 30-year investment in AI.

Robotics: Many surgeons now use robot assistants in microsurgery. HipNav (DiGioia *et al.*, 1996) is a system that uses computer vision techniques to create a three-dimensional model of a patient's internal anatomy and then uses robotic control to guide the insertion of a hip replacement prosthesis.

Language understanding and problem solving: PROVERB (Littman *et al.*, 1999) is a computer program that solves crossword puzzles better than most humans, using constraints on possible word fillers, a large database of past puzzles, and a variety of information sources including dictionaries and online databases such as a list of movies and the actors that appear in them. For example, it determines that the clue "Nice Story" can be solved by "ETAGE" because its database includes the clue/solution pair "Story in France/ETAGE" and because it recognizes that the patterns "Nice X" and "X in France" often have the same solution. The program does not know that Nice is a city in France, but it can solve the puzzle.

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 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 think of AI differently. Two important questions to ask are: Are you concerned with thinking or behavior? Do you want to model humans or work from an ideal standard?
- In this book, we adopt the view that intelligence is concerned mainly with **rational action**. Ideally, an **intelligent agent** takes the best possible action in a situation. We will study the problem of building agents that are intelligent in this sense.

- Philosophers (going back to 400 B.C.) made AI conceivable by considering the ideas 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 outcome to the decision-maker.
- 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 artifacts that make AI applications possible. AI programs tend to be large, and they could not work without the great advances in speed and memory that the computer industry has provided.
- 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 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 advanced more rapidly in the past decade because of greater use of the scientific method in experimenting with and comparing approaches.
- Recent progress in understanding the theoretical basis for intelligence has gone hand in hand with improvements in the capabilities of real systems. The subfields of AI have become more integrated, and AI has found common ground with other disciplines.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

The methodological status of artificial intelligence is investigated in *The Sciences of the Artificial*, by Herb Simon (1981), which discusses research areas concerned with complex artifacts. It explains how AI can be viewed as both science and mathematics. Cohen (1995) gives an overview of experimental methodology within AI. Ford and Hayes (1995) give an opinionated view of the usefulness of the Turing Test.

Artificial Intelligence: The Very Idea, by John Haugeland (1985) gives a readable account of the philosophical and practical problems of AI. Cognitive science is well described by several recent texts (Johnson-Laird, 1988; Stillings *et al.*, 1995; Thagard, 1996) and by the *Encyclopedia of the Cognitive Sciences* (Wilson and Keil, 1999). Baker (1989) covers the syntactic part of modern linguistics, and Chierchia and McConnell-Ginet (1990) cover semantics. Jurafsky and Martin (2000) cover computational linguistics.

Early AI is described in Feigenbaum and Feldman's *Computers and Thought* (1963), Minsky's *Semantic Information Processing* (1968), and the *Machine Intelligence* series edited by Donald Michie. A large number of influential papers have been anthologized by Webber

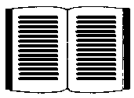
and Nilsson (1981) and by Luger (1995). Early papers on neural networks are collected in *Neurocomputing* (Anderson and Rosenfeld, 1988). The *Encyclopedia of AI* (Shapiro, 1992) contains survey articles on almost every topic in AI. These articles usually provide a good entry point into the research literature on each topic.

The most recent work appears in the proceedings of the major AI conferences: the biennial International Joint Conference on AI (IJCAI), the annual European Conference on AI (ECAI), and the National Conference on AI, more often known as AAAI, after its sponsoring organization. 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 electronic *Journal of Artificial Intelligence Research*. There are also many conferences and journals devoted to specific areas, which we cover in the appropriate chapters. The main professional societies for AI are the American Association for Artificial Intelligence (AAAI), the ACM Special Interest Group in Artificial Intelligence (SIGART), and the Society for Artificial Intelligence and Simulation of Behaviour (AISB). AAAI's *AI Magazine* contains many topical and tutorial articles, and its website, aaai.org, contains news and background information.

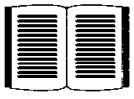
EXERCISES

These exercises are intended to stimulate discussion, and some might be set as term projects. Alternatively, preliminary attempts can be made now, and these attempts can be reviewed after the completion of the book.

1.1 Define in your own words: (a) intelligence, (b) artificial intelligence, (c) agent.



1.2 Read Turing's original paper on AI (Turing, 1950). In the paper, he discusses several potential objections to his proposed enterprise and his test for intelligence. Which objections still carry some weight? Are his refutations valid? Can you think of new objections arising from developments since he wrote the paper? In the paper, he predicts that, by the year 2000, a computer will have a 30% chance of passing a five-minute Turing Test with an unskilled interrogator. What chance do you think a computer would have today? In another 50 years?

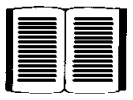


1.3 Every year the Loebner prize is awarded to the program that comes closest to passing a version of the Turing test. Research and report on the latest winner of the Loebner prize. What techniques does it use? How does it advance the state of the art in AI?

1.4 There are well-known classes of problems that are intractably difficult for computers, and other classes that are provably undecidable. Does this mean that AI is impossible?

1.5 Suppose we extend Evans's ANALOGY program so that it can score 200 on a standard IQ test. Would we then have a program more intelligent than a human? Explain.

1.6 How could introspection—reporting on one's inner thoughts—be inaccurate? Could I be wrong about what I'm thinking? Discuss.



1.7 Examine the AI literature to discover whether the following tasks can currently be solved by computers:

a. Playing a decent game of table tennis (ping-pong).
b. Driving in the center of Cairo.
c. Buying a week's worth of groceries at the market.
d. Buying a week's worth of groceries on the web.
e. Playing a decent game of bridge at a competitive level.
f. Discovering and proving new mathematical theorems.
g. Writing an intentionally funny story.
h. Giving competent legal advice in a specialized area of law.
i. Translating spoken English into spoken Swedish in real time.
j. Performing a complex surgical operation.

For the currently infeasible tasks, try to find out what the difficulties are and predict when, if ever, they will be overcome.

1.8 Some authors have claimed that perception and motor skills are the most important part of intelligence, and that "higher level" capacities are necessarily parasitic—simple add-ons to these underlying facilities. Certainly, most of evolution and a large part of the brain have been devoted to perception and motor skills, whereas AI has found tasks such as game playing and logical inference to be easier, in many ways, than perceiving and acting in the real world. Do you think that AI's traditional focus on higher-level cognitive abilities is misplaced?

1.9 Why would evolution tend to result in systems that act rationally? What goals are such systems designed to achieve?

1.10 Are reflex actions (such as moving your hand away from a hot stove) rational? Are they intelligent?

1.11 "Surely computers cannot be intelligent—they can do only what their programmers tell them." Is the latter statement true, and does it imply the former?

1.12 "Surely animals cannot be intelligent—they can do only what their genes tell them." Is the latter statement true, and does it imply the former?

1.13 "Surely animals, humans, and computers cannot be intelligent—they can do only what their constituent atoms are told to do by the laws of physics." Is the latter statement true, and does it imply the former?

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 will 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 will be fleshed out in the rest of the book.

2.1 AGENTS AND ENVIRONMENTS

ENVIRONMENT
SENSOR
ACTUATOR

An **agent** is anything that can be viewed as perceiving its **environment** through **sensors** and acting upon that environment through **actuators**. This simple idea is illustrated in Figure 2.1. A human agent has eyes, ears, and other organs for sensors and hands, legs, mouth, and other body parts for actuators. A robotic agent might have cameras and infrared range finders for sensors and various motors for actuators. A software agent receives keystrokes, file contents, and network packets as sensory inputs and acts on the environment by displaying on the screen, writing files, and sending network packets. We will make the general assumption that every agent can perceive its own actions (but not always the effects).

PERCEPT
PERCEPT SEQUENCE

We use the term **percept** to refer to the agent's perceptual inputs at any given instant. 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 the entire percept sequence observed to date.* If we can specify the agent's choice of action for every possible

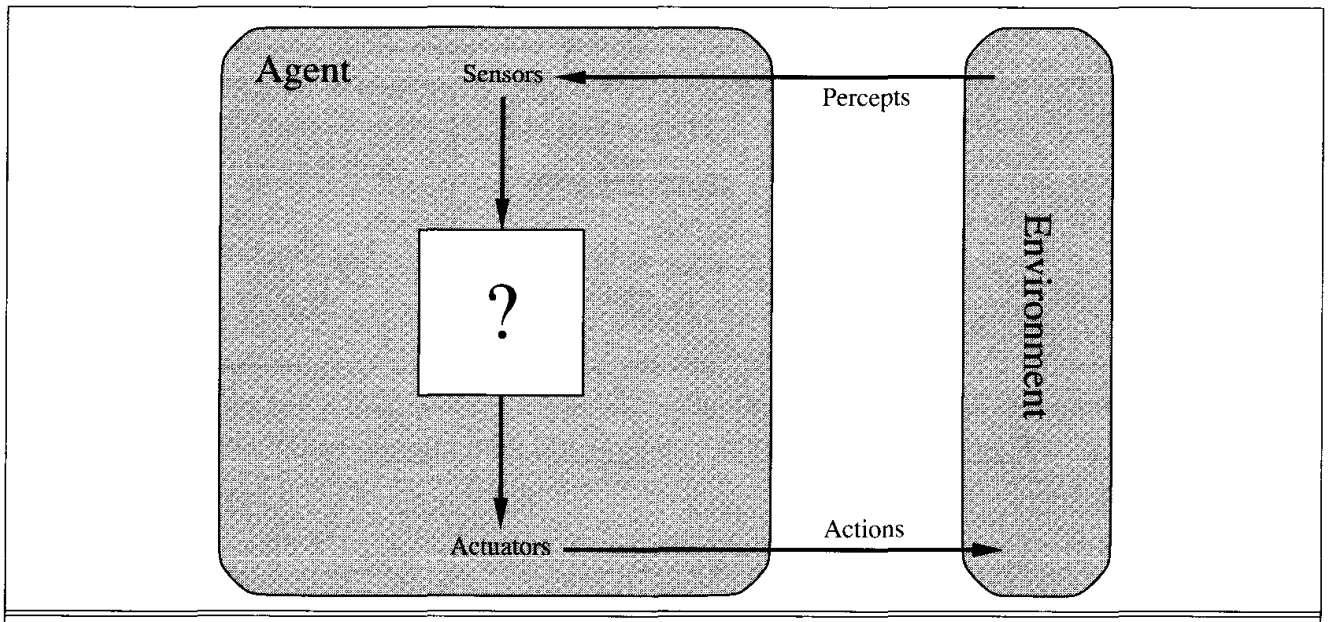


Figure 2.1 Agents interact with environments through sensors and actuators.

percept sequence, then we have said more or less everything there is to say about the agent. Mathematically speaking, we say that an agent's behavior is described by the **agent function** that maps any given percept sequence to an action.

AGENT FUNCTION

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 on the agent architecture.

AGENT PROGRAM

To illustrate these ideas, we will use a very simple example—the vacuum-cleaner world shown in Figure 2.2. This world is so simple that we can describe everything that happens; it's also a made-up world, so we can invent many variations. This particular world has just two locations: squares A and B. The vacuum agent perceives which square it is in and whether there is dirt in the square. It can choose to move left, move right, suck up the dirt, or do nothing. 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. A simple agent program for this agent function is given later in the chapter, in Figure 2.8.

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*

[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'll see later in this chapter that it can be very intelligent.

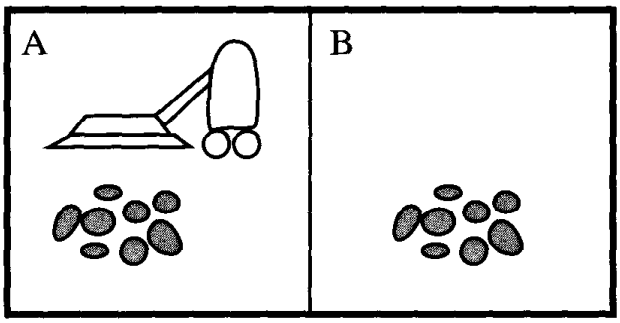


Figure 2.2 A vacuum-cleaner world with just two locations.

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

Figure 2.3 Partial tabulation of a simple agent function for the vacuum-cleaner world shown in Figure 2.2.

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 will remark 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.

2.2 Good Behavior: The Concept of Rationality

RATIONAL AGENT

A **rational agent** is one that does the right thing—conceptually speaking, every entry in the table for the agent function is filled out correctly. Obviously, doing the right thing is better than doing the wrong thing, but what does it mean to do the right thing? As a first approximation, we will say that the right action is the one that will cause the agent to be

most successful. Therefore, we will need some way to measure success. Together with the description of the environment and the sensors and actuators of the agent, this will provide a complete specification of the task facing the agent. Given this, we can define more precisely what it means to be rational.

Performance measures

PERFORMANCE MEASURE

A **performance measure** embodies the criterion for success of an agent's behavior. 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. Obviously, there is not one fixed measure suitable for all agents. We could ask the agent for a subjective opinion of how happy it is with its own performance, but some agents would be unable to answer, and others would delude themselves.[2] Therefore, we will insist on an objective performance measure, typically one imposed by the designer who is constructing the agent.

Consider 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 in the environment, rather than according to how one thinks the agent should behave.*

The selection of a performance measure is not always easy. 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 will leave these questions as an exercise for the diligent reader.

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.

[2] Human agents in particular are notorious for "sour grapes"—believing they did not really want something after not getting it, as in, "Oh well, never mind, I didn't want that stupid Nobel prize anyway."

DEFINITION OF A RATIONAL AGENT

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

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

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

- The performance measure awards one point for each clean square at each time step, over a "lifetime" of 1000 time steps.
- The "geography" of the environment is known *a priori* (Figure 2.2) but the dirt distribution and the initial location of the agent are not. Clean squares stay clean and sucking cleans the current square. The *Left* and *Right* actions move the agent left and right except when this would take the agent outside the environment, in which case the agent remains where it is.
- The only available actions are *Left*, *Right*, *Suck*, and *NoOp* (do nothing).
- The agent correctly perceives its location and whether that location contains dirt.

We claim that *under these circumstances* the agent is indeed rational; its expected performance is at least as high as any other agent's. Exercise 2.4 asks you to prove this.

One can see easily that the same agent would be irrational under different circumstances. For example, once all the dirt is cleaned up it will oscillate needlessly back and forth; if the performance measure includes a penalty of one point for each movement left or right, 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 rather than stick to squares A and B. Exercise 2.4 asks you to design agents for these cases.

Omniscience, learning, and autonomy

OMNISCIENCE

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

This example shows that rationality is not the same as perfection. Rationality maximizes *expected* performance, while perfection maximizes *actual* performance. Retreating from a requirement of perfection is not just a question of being fair to agents. The point is

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

that if we expect an agent to do what turns out to be the best action after the fact, 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

EXPLORATION

LEARNING

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*. In such cases, the agent need not perceive or learn; it simply acts correctly. Of course, such agents are very 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 on 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 back to the "drag" step of its plan, and will continue the plan without modification, 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.

Successful agents split the task of computing the agent function into three different periods: when the agent is being designed, some of the computation is done by its designers; when it is deliberating on its next action, the agent does more computation; and as it learns from experience, it does even more computation to decide how to modify its behavior.

AUTONOMY

To the extent that an agent relies on the prior knowledge of its designer rather than on its own percepts, 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 foresee where and when additional dirt will appear will do better than one that does not. As a practical matter, one seldom requires complete autonomy from the start: when the agent has had little or no experience, it

would have to act randomly unless the designer gave some assistance. So, just as evolution provides animals with enough built-in reflexes so that they can survive long enough to learn for themselves, it would be reasonable to provide an artificial intelligent agent with some initial knowledge as well as an ability to learn. After sufficient experience of its environment, the behavior of a rational agent can become effectively *independent* of its prior knowledge. Hence, the incorporation of learning allows one to design a single rational agent that will succeed in a vast variety of environments.

2.3 THE NATURE OF ENVIRONMENTS

TASK 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 flavor of the task environment directly affects the appropriate design for the agent program.

Specifying the task environment

PEAS

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

The vacuum world was a simple example; let us consider a more complex problem: an automated taxi driver. We will use this example throughout the rest of the chapter. We should point out, before the reader becomes alarmed, that a fully automated taxi is currently somewhat beyond the capabilities of existing technology. (See page 27 for a description of an existing driving robot, or look at recent proceedings of the conferences on Intelligent Transportation Systems.) The full driving task is extremely *open-ended*. There is no limit to the novel combinations of circumstances that can arise—another reason we chose it as a focus for discussion. Figure 2.4 summarizes the PEAS description for the taxi's task environment. We discuss each element in more detail in the following paragraphs.

First, what is the **performance measure** to which we would like our automated driver to aspire? Desirable qualities include getting to the correct destination; minimizing fuel consumption and wear and tear; minimizing the trip time and/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 there will be tradeoffs involved.

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,

Agent Type	Performance Measure	Environment	Actuators	Sensors
Taxi driver	Safe, fast, legal, comfortable trip, maximize profits	Roads, other traffic, pedestrians, customers	Steering, accelerator, brake, signal, horn, display	Cameras, sonar, speedometer, GPS, odometer, accelerometer, engine sensors, keyboard

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

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** available to an automated taxi will be more or less the same as 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.

To achieve its goals in the driving environment, the taxi will need to know where it is, what else is on the road, and how fast it is going. Its basic **sensors** should therefore include one or more controllable TV cameras, the speedometer, and the odometer. To control the vehicle properly, especially on curves, it should have an accelerometer; it will also need to know the mechanical state of the vehicle, so it will need the usual array of engine and electrical system sensors. It might have instruments that are not available to the average human driver: a satellite global positioning system (GPS) to give it accurate position information with respect to an electronic map, and infrared or sonar sensors to detect distances to other cars and obstacles. Finally, it will need a keyboard or microphone 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 2.5. It may come as a surprise to some readers that we include in our list of agent types some programs that operate in the entirely artificial environment defined by keyboard input and character output on a screen. "Surely," one might say, "this is not a real environment, is it?" In fact, what matters is not the distinction between "real" and "artificial" environments, but the complexity of the relationship among the behavior of the agent, the percept sequence generated by the environment, and the performance measure. Some "real" environments are actually quite simple. For example, a robot designed to inspect parts as they come by on a conveyor belt can make use of a number of simplifying assumptions: that the lighting is always just so, that the only thing on the conveyer belt will be parts of a kind that it knows about, and that there are only two actions (accept or reject).

Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	Healthy patient, minimize costs, lawsuits	Patient, hospital, staff	Display questions, tests, diagnoses, treatments, referrals	Keyboard entry of symptoms, findings, patient's answers
Satellite image analysis system	Correct image categorization	Downlink from orbiting satellite	Display categorization of scene	Color pixel arrays
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts; bins	Jointed arm and hand	Camera, joint angle sensors
Refinery controller	Maximize purity, yield, safety	Refinery, operators	Valves, pumps, heaters, displays	Temperature, pressure, chemical sensors
Interactive English tutor	Maximize student's score on test	Set of students, testing agency	Display exercises, suggestions, corrections	Keyboard entry

Figure 2.5 Examples of agent types and their PEAS descriptions.

SOFTWARE AGENTS
SOFTBOTS

In contrast, some **software agents** (or software robots or **softbots**) exist in rich, unlimited domains. Imagine a softbot designed to fly a flight simulator for a large commercial airplane. The simulator is a very detailed, complex environment including other aircraft and ground operations, and the software agent must choose from a wide variety of actions in real time. Or imagine a softbot designed to scan Internet news sources and show the interesting items to its customers. To do well, it will need some natural language processing abilities, it will need to learn what each customer is interested in, and it will need to change its plans dynamically—for example, when the connection for one news source goes down or when a new one comes online. The Internet is an environment whose complexity rivals that of the physical world and whose inhabitants include many artificial agents.

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

◇ **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.[4] 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.

DETERMINISTIC
STOCHASTIC

◇ **Deterministic** vs. **stochastic.**
If the next state of the environment is completely determined by the current state and the action executed by the agent, then we say the environment is deterministic; otherwise, it is stochastic. 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 stochastic. This is particularly true if the environment is complex, making it hard to keep track of all the unobserved aspects. Thus, it is often better to think of an environment as deterministic or stochastic *from the point of view of the agent.* Taxi driving is clearly stochastic in this sense, because one can never predict the behavior of traffic exactly; moreover, one's tires blow out and one's engine seizes up without warning. The vacuum world as we described it is deterministic, but variations can include stochastic elements such as randomly appearing dirt and an unreliable suction mechanism (Exercise 2.12). If the environment is deterministic except for the actions of other agents, we say that the environment is **strategic.**

STRATEGIC

EPISODIC
SEQUENTIAL

◇ **Episodic** vs. **sequential.**[5]
In an episodic task environment, the agent's experience is divided into atomic episodes. Each episode consists of the agent perceiving and then performing a single action. Crucially, the next episode does not depend on the actions taken in previous episodes. In episodic environments, the choice of action in each episode depends only on the episode itself. 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

[4] The first edition of this book used the terms **accessible** and **inaccessible** instead of **fully** and **partially observable**; **nondeterministic** instead of **stochastic**; and **nonepisodic** instead of **sequential**. The new terminology is more consistent with established usage.

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

part is defective. In sequential environments, on the other hand, the current decision could affect all future decisions. Chess and taxi driving are sequential: in both cases, short-term actions can have long-term consequences. Episodic environments are much simpler than sequential environments because the agent does not need to think ahead.

STATIC
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
CONTINUOUS

◇ **Discrete** vs. **continuous**.
The discrete/continuous distinction can be applied to the *state* of the environment, to the way *time* is handled, and to the *percepts* and *actions* of the agent. For example, a discrete-state environment such as a chess game has a finite number of distinct states. 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.

SINGLE AGENT
MULTIAGENT

◇ **Single agent** vs. **multiagent**.
The distinction between single-agent and multiagent environments may seem simple enough. For example, an agent solving a crossword puzzle by itself is clearly in a single-agent environment, whereas an agent playing chess is in a two-agent environment. There are, however, 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 a stochastically behaving object, analogous to waves at the beach or leaves blowing in the wind? The key distinction is whether B's behavior is best described as maximizing a performance measure whose value depends on agent A's behavior. For example, in chess, the opponent entity B is trying to maximize its performance measure, which, by the rules of chess, minimizes agent A's performance measure. Thus, chess is a **competitive** multiagent environment. In the taxi-driving environment, on the other hand, avoiding collisions maximizes the performance measure of all agents, so it is a partially **cooperative** multiagent environment. It is also partially competitive because, for example, only one car can occupy a parking space. The agent-design problems arising in multiagent environments are often

COMPETITIVE
COOPERATIVE

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

Figure 2.6 Examples of task environments and their characteristics.

quite different from those in single-agent environments; for example, **communication** often emerges as a rational behavior in multiagent environments; in some partially observable competitive environments, **stochastic behavior** is rational because it avoids the pitfalls of predictability.

As one might expect, the hardest case is *partially observable*, *stochastic*, *sequential*, *dynamic*, *continuous*, and *multiagent*. It also turns out that most real situations are so complex that whether they are *really* deterministic is a moot point. For practical purposes, they must be treated as stochastic. Taxi driving is hard in all these senses.

Figure 2.6 lists the properties of a number of familiar environments. Note that the answers are not always cut and dried. For example, we have listed chess as fully observable; strictly speaking, this is false because certain rules about castling, *en passant* capture, and draws by repetition require remembering some facts about the game history that are not observable as part of the board state. These exceptions to observability are of course minor compared to those faced by the taxi driver, the English tutor, or the medical diagnosis system.

Some other answers in the table depend on how the task environment is defined. 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, and so on. Also, many environments are episodic at higher levels than the agent's individual actions. For example, a chess tournament consists of a sequence of games; each game is an episode, because (by and large) the contribution of the moves in one game to the agent's overall performance is not affected by the moves in its previous game. On the other hand, decision making within a single game is certainly sequential.

The code repository associated with this book (aima.cs.berkeley.edu) includes implementations of a number of environments, together with a general-purpose environment simulator that places one or more agents in a simulated environment, observes their behavior over time, and evaluates them according to a given performance measure. Such 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. If we designed the agent for a single scenario, we might be able to take advantage of specific properties of the particular case but might not identify a good design for driving in general. For this reason, the code repository also includes an **environment generator** for each environment class that selects particular environments (with certain likelihoods) in which to run the agent. For example, the vacuum environment generator initializes the dirt pattern and agent location randomly. We are then interested in the agent's average performance over the environment class. A rational agent for a given environment class maximizes this average performance. Exercises 2.7 to 2.12 take you through the process of developing an environment class and evaluating various agents therein.

ENVIRONMENT CLASS

ENVIRONMENT GENERATOR

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 will have to bite the bullet and talk about how the insides work. The job of AI is to design the **agent program** that implements the agent function mapping percepts to actions. We assume this program will run on some sort of computing device with physical sensors and actuators—we call this the **architecture**:

AGENT PROGRAM

ARCHITECTURE

$$agent = architecture + program\ .$$

Obviously, the program we choose has to be one that is appropriate for the architecture. If the program is going to recommend actions like *Walk*, the architecture had better have legs. The architecture might be just an ordinary PC, or it might be a robotic car with several onboard computers, cameras, and other sensors. In general, the architecture makes the percepts from the sensors available to the program, runs the program, and feeds the program's action choices to the actuators as they are generated. Most of this book is about designing agent programs, although Chapters 24 and 25 deal directly with the sensors and actuators.

Agent programs

The agent programs that we will 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.[6] Notice the difference between the agent program, which takes the current percept as input, and the agent function, which takes the entire percept history. The agent program takes just the current

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

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

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

Figure 2.7 The TABLE-DRIVEN-AGENT program is invoked for each new percept and returns an action each time. It keeps track of the percept sequence using its own private data structure.

percept as input because nothing more is available from the environment; if the agent's actions depend on the entire percept sequence, the agent will have to remember the percepts.

We will describe the agent programs via 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 represents explicitly the agent function that the agent program embodies. To build a rational agent in this way, we as designers must construct a table that contains the appropriate action for every possible percept sequence.

It is instructive to consider why the table-driven approach to agent construction is doomed to failure. Let $\mathcal{P}$ be the set of possible percepts and let T be the lifetime of the agent (the total number of percepts it will receive). The lookup table will contain $\sum_{t=1}^{T} |\mathcal{P}|^t$ entries. Consider the automated taxi: the visual input from a single camera comes in at the rate of roughly 27 megabytes per second (30 frames per second, 640×480 pixels with 24 bits of color information). This gives a lookup table with over $10^{250,000,000,000}$ entries for an hour's driving. Even the lookup table for chess—a tiny, well-behaved fragment of the real world—would have at least 10^{150} entries. The daunting size of these tables (the number of atoms in the observable universe is less than 10^{80}) means that (a) no physical agent in this universe will have the space to store the table, (b) the designer would not have time to create the table, (c) no agent could ever learn all the right table entries from its experience, and (d) even if the environment is simple enough to yield a feasible table size, the designer still has no guidance about how to fill in the table entries.

Despite all this, TABLE-DRIVEN-AGENT *does* do what we want: 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 small amount of code rather than from a large number of table entries. 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.

```
function REFLEX-VACUUM-AGENT([location,status]) returns an action

  if status = Dirty then return Suck
  else if location = A then return Right
  else if location = B then return Left
```

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

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

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

We then explain in general terms how to convert all these into *learning agents*.

Simple reflex agents

SIMPLE REFLEX AGENT

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

Notice that the vacuum agent program is very small indeed compared to the corresponding table. The most obvious reduction comes from ignoring the percept history, which cuts down the number of possibilities 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.

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**,[7] written as

CONDITION–ACTION RULE

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

Humans also have many such connections, some of which are learned responses (as for driving) and some of which are innate reflexes (such as blinking when something approaches the eye). In the course of the book, we will see 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–

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

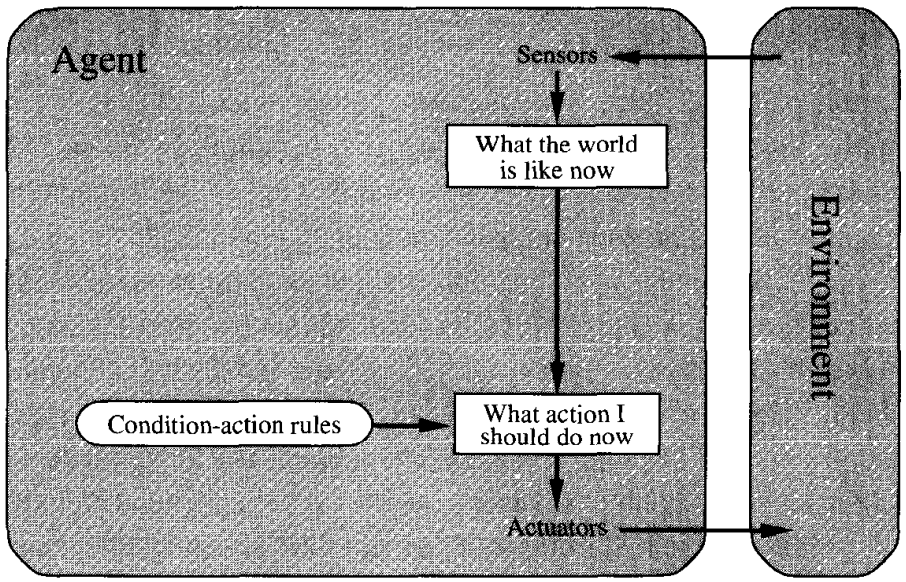


Figure 2.9 Schematic diagram of a simple reflex agent.

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

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

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

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.) 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. The agent program, which is also very simple, 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; actual implementations can be as simple as a collection of logic gates implementing a Boolean circuit.

Simple reflex agents have the admirable property of being simple, but they turn out to be of very limited intelligence. The agent in Figure 2.10 will work *only if the correct decision can be made on the basis of only 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—the current video image—if the car in front has a centrally mounted 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. 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 (for ever) if it happens to start in square A, and moving *Right* fails (for ever) if it happens to start in square B. Infinite loops are often unavoidable for simple reflex agents operating in partially observable environments.

RANDOMIZATION

Escape from infinite loops is possible if the agent can **randomize** its actions. For example, if the vacuum agent perceives $[Clean]$, it might flip a coin to choose between *Left* and *Right*. 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, it will clean it and the cleaning task will be complete. Hence, a randomized simple reflex agent might outperform a deterministic simple reflex agent.

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

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.

INTERNAL STATE

Updating this internal state information as time goes by requires two kinds of knowledge to be encoded in the agent program. First, we need some information about how the world evolves independently of the agent—for example, that an overtaking car generally will be closer behind than it was a moment ago. Second, we need some information about how the agent's own actions affect the world—for example, that when the agent turns the steering wheel clockwise, the car turns to the right or that after driving for five minutes northbound on the freeway one is usually about five miles north of where one was five minutes ago. This knowledge about "how the world works"—whether implemented in simple Boolean circuits or in complete scientific theories—is called a **model** of the world. An agent that uses such a model is called a **model-based agent**.

MODEL-BASED AGENT

Figure 2.11 gives the structure of the reflex agent with internal state, showing how the current percept is combined with the old internal state to generate the updated description

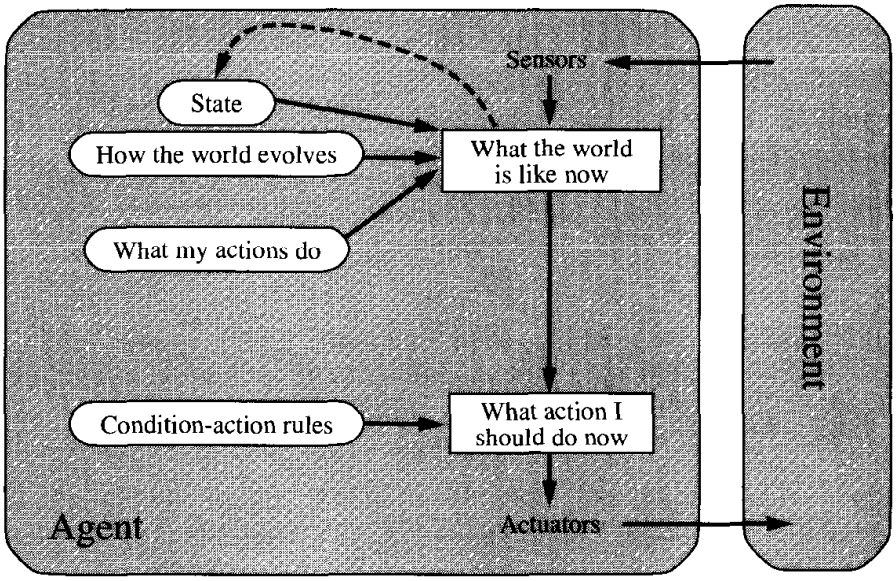


Figure 2.11 A model-based reflex agent.

```
function REFLEX-AGENT-WITH-STATE(percept) returns an action
  static: state, a description of the current world state
          rules, a set of condition–action rules
          action, the most recent action, initially none

  state ← UPDATE-STATE(state, action, percept)
  rule ← RULE-MATCH(state, rules)
  action ← RULE-ACTION[rule]
  return action
```

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

of the current state. 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. As well as interpreting the new percept in the light of existing knowledge about the state, it uses information about how the world evolves to keep track of the unseen parts of the world, and also must know about what the agent's actions do to the state of the world. Detailed examples appear in Chapters 10 and 17.

Goal-based agents

Knowing 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 the passenger's destination. The agent

GOAL

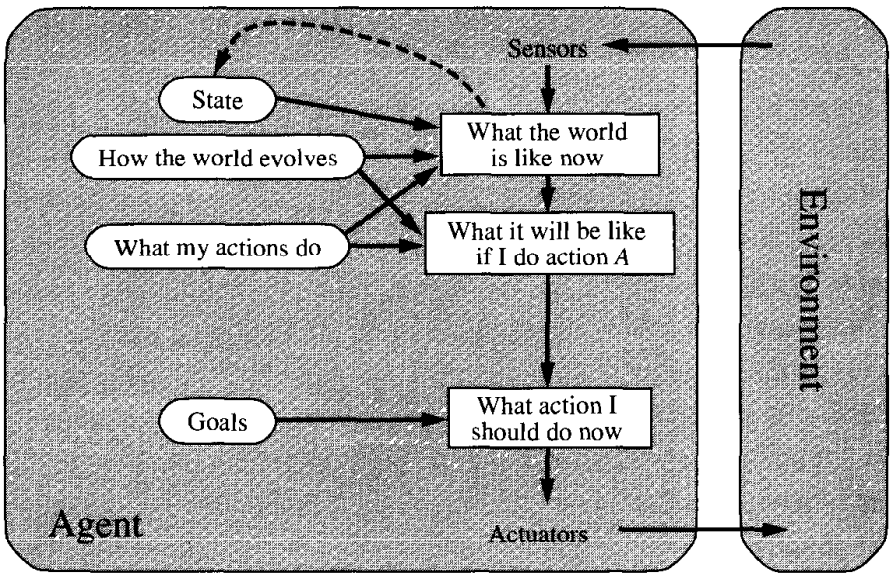


Figure 2.13 A model-based, goal-based agent. It keeps track of the world state as well as a set of goals it is trying to achieve, and chooses an action that will (eventually) lead to the achievement of its goals.

program can combine this with information about the results of possible actions (the same information as was used to update internal state in the reflex agent) in order to choose actions that achieve the goal. Figure 2.13 shows the goal-based agent's structure.

Sometimes goal-based action selection is straightforward, when goal satisfaction results immediately from a single action. Sometimes it will be more tricky, when the agent has to consider long sequences of twists and turns to find a way to achieve the goal. **Search** (Chapters 3 to 6) and **planning** (Chapters 11 and 12) 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. A goal-based agent, in principle, could reason that if the car in front has its brake lights on, it will slow down. Given the way the world usually evolves, the only action that will achieve the goal of not hitting other cars is to brake.

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. If it starts to rain, the agent can update its knowledge of how effectively its brakes will operate; this will automatically cause all of the relevant behaviors to be altered to suit the new conditions. For the reflex agent, on the other hand, we would have to rewrite many condition–action rules. The goal-based agent's behavior can easily be changed to go to a different location. 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.

Utility-based agents

Goals alone are not really enough to generate high-quality behavior in most environments. For example, there are many action sequences that 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, whereas a more general performance measure should allow a comparison of different world states according to exactly how happy they would make the agent if they could be achieved. Because "happy" does not sound very scientific, the customary terminology is to say that if one world state is preferred to another, then it has higher **utility** for the agent.[8]

UTILITY

UTILITY FUNCTION

A **utility function** maps a state (or a sequence of states) onto a real number, which describes the associated degree of happiness. A complete specification of the utility function allows rational decisions in two kinds of cases where goals are inadequate. 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 up against the importance of the goals.

In Chapter 16, we will 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 therefore can make rational decisions, and it can do so via a general-purpose algorithm that does not depend on the specific utility function being maximized. In this way, the "global" definition of rationality—designating as rational those agent functions that have the highest performance—is turned into a "local" constraint on rational-agent designs that can be expressed in a simple program.

The utility-based agent structure appears in Figure 2.14. Utility-based agent programs appear in Part V, where we design decision making agents that must handle the uncertainty inherent in partially observable environments.

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. 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. In almost every chapter of the book, we will comment on opportunities and methods for learning in particular kinds of agents. Part VI goes into much more depth on the various learning algorithms themselves.

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

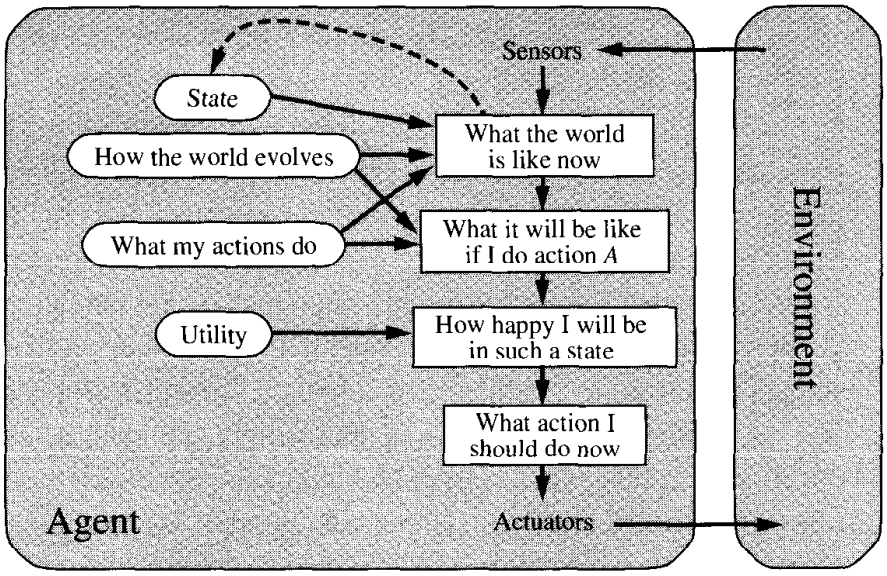


Figure 2.14 A model-based, utility-based agent. It uses a model of the world, along with a utility function that measures its preferences among states of the world. Then it chooses the action that leads to the best expected utility, where expected utility is computed by averaging over all possible outcome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

LEARNING ELEMENT
PERFORMANCE ELEMENT
CRITIC

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.

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 need to do this once it has learned how?" Given an agent design, 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.

PROBLEM GENERATOR

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. The point is that 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

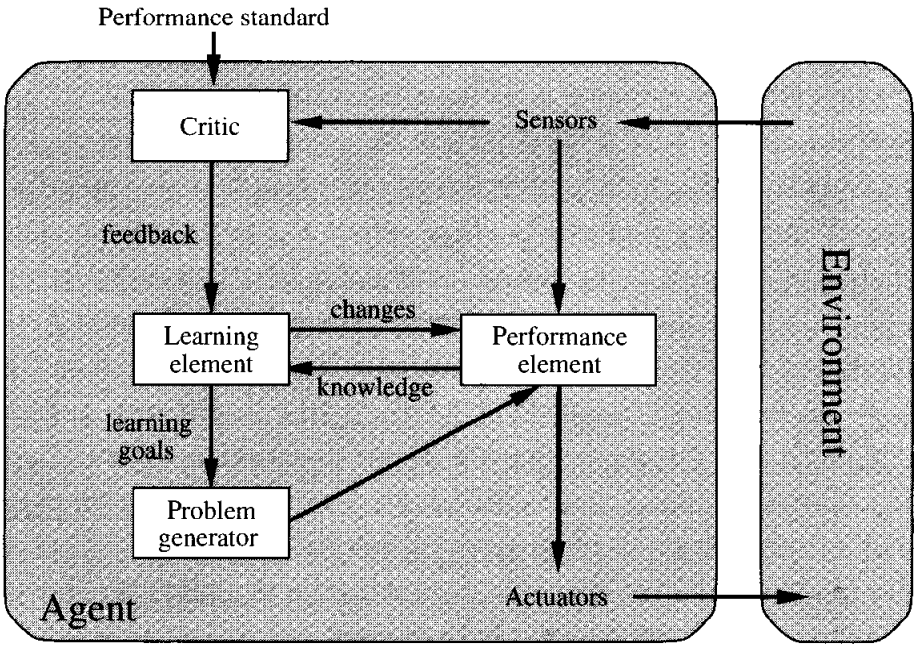


Figure 2.15 A general model of learning agents.

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, nor to modify the brains of unfortunate passers-by. His aim was to modify his own brain, by identifying a better theory of the motion of objects.

To make the overall design more concrete, let us return to the automated taxi example. The performance element consists of whatever collection of knowledge and procedures the taxi has for selecting its driving actions. The taxi goes out on the road and drives, using this performance element. The critic observes the world and passes information along to the learning element. For example, after the taxi makes a quick left turn across three lanes of traffic, the critic observes the shocking language used by other drivers. From this experience, the learning element is able to formulate a rule saying this was a bad action, and the performance element is modified by installing the new rule. The problem generator might identify certain areas of behavior in need of improvement and suggest experiments, such as trying out the brakes on different road surfaces under different conditions.

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 "How the world evolves," and observation of the results of its actions can allow the agent to learn "What my actions do." For example, if the taxi exerts a certain braking pressure when driving on a wet road, then it will soon find out how much deceleration is actually achieved. Clearly, these two learning tasks are more difficult if the environment is only partially observable.

The forms of learning in the preceding paragraph do not need to access the external performance standard—in a sense, the standard is the universal one of making predictions

that agree with experiment. The situation is slightly more complex for a utility-based agent that wishes to learn utility information. 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. This issue is discussed further in Chapter 21.

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.

2.5 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, deterministic or stochastic, episodic or sequential, static or dynamic, discrete or continuous, and single-agent or multiagent.
- 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 (1958) influential paper "Programs with Common Sense." The fields of robotics and control theory are, by their very nature, concerned principally with the construction of 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 in the field and is a central theme in recent texts (Poole *et al.*, 1998; Nilsson, 1998).

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, 1999) handle fully observable, deterministic environments; stochastic optimal control (Kumar and Varaiya, 1986) handles partially observable, stochastic environments; and hybrid control (Henzinger and Sastry, 1998) 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 will discuss in Chapter 17.

Reflex agents were the primary model for psychological behaviorists such as Skinner (1953), who attempted to reduce the psychology of organisms strictly to input/output or stimulus/response mappings. The advance from behaviorism to functionalism in psychology, which was at least partly driven by the application of the computer metaphor to agents (Putnam, 1960; Lewis, 1966), introduced the internal state of the agent into the picture. Most work in AI views the idea of pure reflex agents with state as too simple to provide much leverage, but work by Rosenschein (1985) and Brooks (1986) questioned this assumption (see Chapter 25). In recent years, a great deal of work has gone into finding efficient algorithms for keeping track of complex environments (Hamscher *et al.*, 1992). The Remote Agent program that controlled the Deep Space One spacecraft (described on page 27) is a particularly impressive example (Muscettola *et al.*, 1998; Jonsson *et al.*, 2000).

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 goal-based view 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 theory of agents developed by Bratman (1987). This theory has been influential both in natural language understanding and multiagent systems.

Horvitz *et al.* (1988) specifically suggest the use of rationality conceived as the maximization of expected utility as a basis for AI. 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 Part V).

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 Part VI.

Interest in agents and in agent design has risen rapidly in recent years, partly because of the growth of the Internet and the perceived need for automated and mobile **softbots** (Etzioni and Weld, 1994). Relevant papers are collected in *Readings in Agents* (Huhns and Singh, 1998) and *Foundations of Rational Agency* (Wooldridge and Rao, 1999). *Multiagent Systems* (Weiss, 1999) provides a solid foundation for many aspects of agent design. Conferences devoted to agents include the International Conference on Autonomous Agents, the International Workshop on Agent Theories, Architectures, and Languages, and the International Conference on Multiagent Systems. Finally, *Dung Beetle Ecology* (Hanski and Cambefort, 1991) provides a wealth of interesting information on the behavior of dung beetles.

EXERCISES

2.1 Define in your own words the following terms: agent, agent function, agent program, rationality, autonomy, reflex agent, model-based agent, goal-based agent, utility-based agent, learning agent.

2.2 Both the performance measure and the utility function measure how well an agent is doing. Explain the difference between the two.

2.3 This exercise explores the differences between agent functions and agent programs.

a. Can there be more than one agent program that implements a given agent function? Give an example, or show why one is not possible.

b. Are there agent functions that cannot be implemented by any agent program?

c. Given a fixed machine architecture, does each agent program implement exactly one agent function?

d. Given an architecture with n bits of storage, how many different possible agent programs are there?

2.4 Let us examine the rationality of various vacuum-cleaner agent functions.

a. Show that the simple vacuum-cleaner agent function described in Figure 2.3 is indeed rational under the assumptions listed on page 36.

b. Describe a rational agent function for the modified performance measure that deducts one point for each movement. Does the corresponding agent program require internal state?

c. Discuss possible agent designs for the cases in which clean squares can become dirty and the geography of the environment is unknown. Does it make sense for the agent to learn from its experience in these cases? If so, what should it learn?

2.5 For each of the following agents, develop a PEAS description of the task environment:

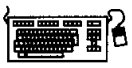
a. Robot soccer player;

b. Internet book-shopping agent;

c. Autonomous Mars rover;

d. Mathematician's theorem-proving assistant.

2.6 For each of the agent types listed in Exercise 2.5, characterize the environment according to the properties given in Section 2.3, and select a suitable agent design.



The following exercises all concern the implementation of environments and agents for the vacuum-cleaner world.

2.7 Implement a performance-measuring environment simulator for the vacuum-cleaner world depicted in Figure 2.2 and specified on page 36. Your implementation should be modular, so that the sensors, actuators, and environment characteristics (size, shape, dirt placement, etc.) can be changed easily. (*Note:* for some choices of programming language and operating system there are already implementations in the online code repository.)

2.8 Implement a simple reflex agent for the vacuum environment in Exercise 2.7. Run the environment simulator with this agent for all possible initial dirt configurations and agent locations. Record the agent's performance score for each configuration and its overall average score.

2.9 Consider a modified version of the vacuum environment in Exercise 2.7, in which the agent is penalized one point for each movement.

a. Can a simple reflex agent be perfectly rational for this environment? Explain.

b. What about a reflex agent with state? Design such an agent.

c. How do your answers to **a** and **b** change if the agent's percepts give it the clean/dirty status of every square in the environment?

2.10 Consider a modified version of the vacuum environment in Exercise 2.7, in which the geography of the environment—its extent, boundaries, and obstacles—is unknown, as is the initial dirt configuration. (The agent can go *Up* and *Down* as well as *Left* and *Right*.)

a. Can a simple reflex agent be perfectly rational for this environment? Explain.

b. Can a simple reflex agent with a *randomized* agent function outperform a simple reflex agent? Design such an agent and measure its performance on several environments.

c. Can you design an environment in which your randomized agent will perform very poorly? Show your results.

d. Can a reflex agent with state outperform a simple reflex agent? Design such an agent and measure its performance on several environments. Can you design a rational agent of this type?

2.11 Repeat Exercise 2.10 for the case in which the location sensor is replaced with a "bump" sensor that detects the agent's attempts to move into an obstacle or to cross the boundaries of the environment. Suppose the bump sensor stops working; how should the agent behave?

2.12 The vacuum environments in the preceding exercises have all been deterministic. Discuss possible agent programs for each of the following stochastic versions:

a. Murphy's law: twenty-five percent of the time, the *Suck* action fails to clean the floor if it is dirty and deposits dirt onto the floor if the floor is clean. How is your agent program affected if the dirt sensor gives the wrong answer 10% of the time?

b. Small children: At each time step, each clean square has a 10% chance of becoming dirty. Can you come up with a rational agent design for this case?

3 SOLVING PROBLEMS BY SEARCHING

In which we see how an agent can find a sequence of actions that achieves its goals, when no single action will do.

The simplest agents discussed in Chapter 2 were the reflex agents, which base their actions on a direct mapping from states to actions. Such agents cannot operate well in environments for which this mapping would be too large to store and would take too long to learn. Goal-based agents, on the other hand, can succeed by considering future actions and the desirability of their outcomes.

PROBLEM-SOLVING AGENT

This chapter describes one kind of goal-based agent called a **problem-solving agent**. Problem-solving agents decide what to do by finding sequences of actions that lead to desirable states. We start by defining precisely the elements that constitute a "problem" and its "solution," and give several examples to illustrate these definitions. We then describe several general-purpose search algorithms that can be used to solve these problems and compare the advantages of each algorithm. The algorithms are **uninformed**, in the sense that they are given no information about the problem other than its definition. Chapter 4 deals with **informed** search algorithms, ones that have some idea of where to look for solutions.

This chapter uses concepts from the analysis of algorithms. Readers unfamiliar with the concepts of asymptotic complexity (that is, $O()$ notation) and NP-completeness should consult Appendix A.

3.1 PROBLEM-SOLVING AGENTS

Intelligent agents are supposed to maximize their performance measure. As we mentioned in Chapter 2, achieving this is sometimes simplified if the agent can adopt a **goal** and aim at satisfying it. Let us first look at why and how an agent might do this.

Imagine an agent in the city of Arad, Romania, enjoying a touring holiday. The agent's performance measure contains many factors: it wants to improve its suntan, improve its Romanian, take in the sights, enjoy the nightlife (such as it is), avoid hangovers, and so on. The decision problem is a complex one involving many tradeoffs and careful reading of guidebooks. Now, suppose the agent has a nonrefundable ticket to fly out of Bucharest the follow-

ing day. In that case, it makes sense for the agent to adopt the **goal** of getting to Bucharest. Courses of action that don't reach Bucharest on time can be rejected without further consideration and the agent's decision problem is greatly simplified. Goals help organize behavior by limiting the objectives that the agent is trying to achieve. **Goal formulation**, based on the current situation and the agent's performance measure, is the first step in problem solving.

GOAL FORMULATION

We will consider a goal to be a set of world states—exactly those states in which the goal is satisfied. The agent's task is to find out which sequence of actions will get it to a goal state. Before it can do this, it needs to decide what sorts of actions and states to consider. If it were to try to consider actions at the level of "move the left foot forward an inch" 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, because at that level of detail there is too much uncertainty in the world and there would be too many steps in a solution. **Problem formulation** is the process of deciding what actions and states to consider, given a goal. We will discuss this process in more detail later. For now, let us assume that the agent will consider actions at the level of driving from one major town to another. The states it will consider therefore correspond to being in a particular town.[1]

PROBLEM FORMULATION

Our agent has now adopted the goal of driving to Bucharest, and is considering where to go from Arad. There are three roads out of Arad, one toward Sibiu, one to Timisoara, and one to Zerind. None of these achieves the goal, so unless the agent is very familiar with the geography of Romania, it will not know which road to follow.[2] In other words, the agent will not know which of its possible actions is best, because it does not know enough about the state that results from taking each action. If the agent has no additional knowledge, then it is stuck. The best it can do is choose one of the actions at random.

But suppose the agent has a map of Romania, either on paper or in its memory. The point of a map is to provide the agent with information about the states it might get itself into, and the actions it can take. The agent can use this information to consider subsequent stages of a hypothetical journey via each of the three towns, trying to find a journey that eventually gets to Bucharest. Once it has found a path on the map from Arad to Bucharest, it can achieve its goal by carrying out the driving actions that correspond to the legs of the journey. In general, *an agent with several immediate options of unknown value can decide what to do by first examining different possible* sequences *of actions that lead to states of known value, and then choosing the best sequence.*

SEARCH

SOLUTION

EXECUTION

This process of looking for such a sequence is called **search**. A search algorithm takes a problem as input and returns a **solution** in the form of an action sequence. Once a solution is found, the actions it recommends can be carried out. This is called the **execution** phase. Thus, we have a simple "formulate, search, execute" design for the agent, as shown in Figure 3.1. After formulating a goal and a problem to solve, the agent calls a search procedure to solve it. It then uses the solution to guide its actions, doing whatever the solution recommends as

[1] Notice that each of these "states" actually corresponds to a large *set* of world states, because a real world state specifies every aspect of reality. It is important to keep in mind the distinction between states in problem solving and world states.

[2] 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.

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  inputs: percept, a percept
  static: seq, an action sequence, initially empty
          state, some description of the current world state
          goal, a goal, initially null
          problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then do
     goal ← FORMULATE-GOAL(state)
     problem ← FORMULATE-PROBLEM(state, goal)
     seq ← SEARCH(problem)
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

Figure 3.1 A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over. Note that when it is executing the sequence it ignores its percepts: it assumes that the solution it has found will always work.

the next thing to do—typically, the first action of the sequence—and then removing that step from the sequence. Once the solution has been executed, the agent will formulate a new goal.

We first describe the process of problem formulation, and then devote the bulk of the chapter to various algorithms for the SEARCH function. We will not discuss the workings of the UPDATE-STATE and FORMULATE-GOAL functions further in this chapter.

Before plunging into the details, let us pause briefly to see where problem-solving agents fit into the discussion of agents and environments in Chapter 2. The agent design in Figure 3.1 assumes that the environment is **static**, because formulating and solving the problem is done without paying attention to any changes that might be occurring in the environment. The agent design also assumes that the initial state is known; knowing it is easiest if the environment is **observable**. The idea of enumerating "alternative courses of action" assumes that the environment can be viewed as **discrete**. Finally, and most importantly, the agent design assumes that the environment is **deterministic**. Solutions to problems are single sequences of actions, so they cannot handle any unexpected events; moreover, solutions are executed without paying attention to the percepts! An agent that carries out its plans with its eyes closed, so to speak, must be quite certain of what is going on. (Control theorists call this an **open-loop** system, because ignoring the percepts breaks the loop between agent and environment.) All these assumptions mean that we are dealing with the easiest kinds of environments, which is one reason this chapter comes early on in the book. Section 3.6 takes a brief look at what happens when we relax the assumptions of observability and determinism. Chapters 12 and 17 go into much greater depth.

OPEN-LOOP

Well-defined problems and solutions

PROBLEM

A **problem** can be defined formally by four components:

INITIAL STATE

- The **initial state** that the agent starts in. For example, the initial state for our agent in Romania might be described as $In(Arad)$.

SUCCESSOR FUNCTION

- A description of the possible **actions** available to the agent. The most common formulation[3] uses a **successor function**. Given a particular state x, SUCCESSOR-FN(x) returns a set of $\langle action, successor \rangle$ ordered pairs, where each action is one of the legal actions in state x and each successor is a state that can be reached from x by applying the action. For example, from the state $In(Arad)$, the successor function for the Romania problem would return

  $$\{\langle Go(Sibiu), In(Sibiu)\rangle, \langle Go(Timisoara), In(Timisoara)\rangle, \langle Go(Zerind), In(Zerind)\rangle\}$$

STATE SPACE

  Together, the initial state and successor function implicitly define the **state space** of the problem—the set of all states reachable from the initial state. The state space forms a graph in which the nodes are states and the arcs between nodes are actions. (The map of Romania shown in Figure 3.2 can be interpreted as a state space graph if we view each road as standing for two driving actions, one in each direction.) A **path** in the state space is a sequence of states connected by a sequence of actions.

PATH

GOAL TEST

- The **goal test**, which determines whether a given state is a goal state. Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them. The agent's goal in Romania is the singleton set $\{In(Bucharest)\}$. Sometimes the goal is specified by an abstract property rather than an explicitly enumerated set of states. For example, in chess, the goal is to reach a state called "checkmate," where the opponent's king is under attack and can't escape.

PATH COST

- A **path cost** function that assigns a numeric cost to each path. The problem-solving agent chooses a cost function that reflects its own performance measure. For the agent trying to get to Bucharest, time is of the essence, so the cost of a path might be its length in kilometers. In this chapter, we assume that the cost of a path can be described as the sum of the costs of the individual actions along the path. The **step cost** of taking action a to go from state x to state y is denoted by $c(x, a, y)$. The step costs for Romania are shown in Figure 3.2 as route distances. We will assume that step costs are nonnegative.[4]

STEP COST

The preceding elements define a problem and can be gathered together into a single data structure that is given as input to a problem-solving algorithm. A **solution** to a problem is a path from the initial state to a goal state. Solution quality is measured by the path cost function, and an **optimal solution** has the lowest path cost among all solutions.

OPTIMAL SOLUTION

Formulating problems

In the preceding section we proposed a formulation of the problem of getting to Bucharest in terms of the initial state, successor function, goal test, and path cost. This formulation seems

[3] An alternative formulation uses a set of **operators** that can be applied to a state to generate successors.

[4] The implications of negative costs are explored in Exercise 3.17.

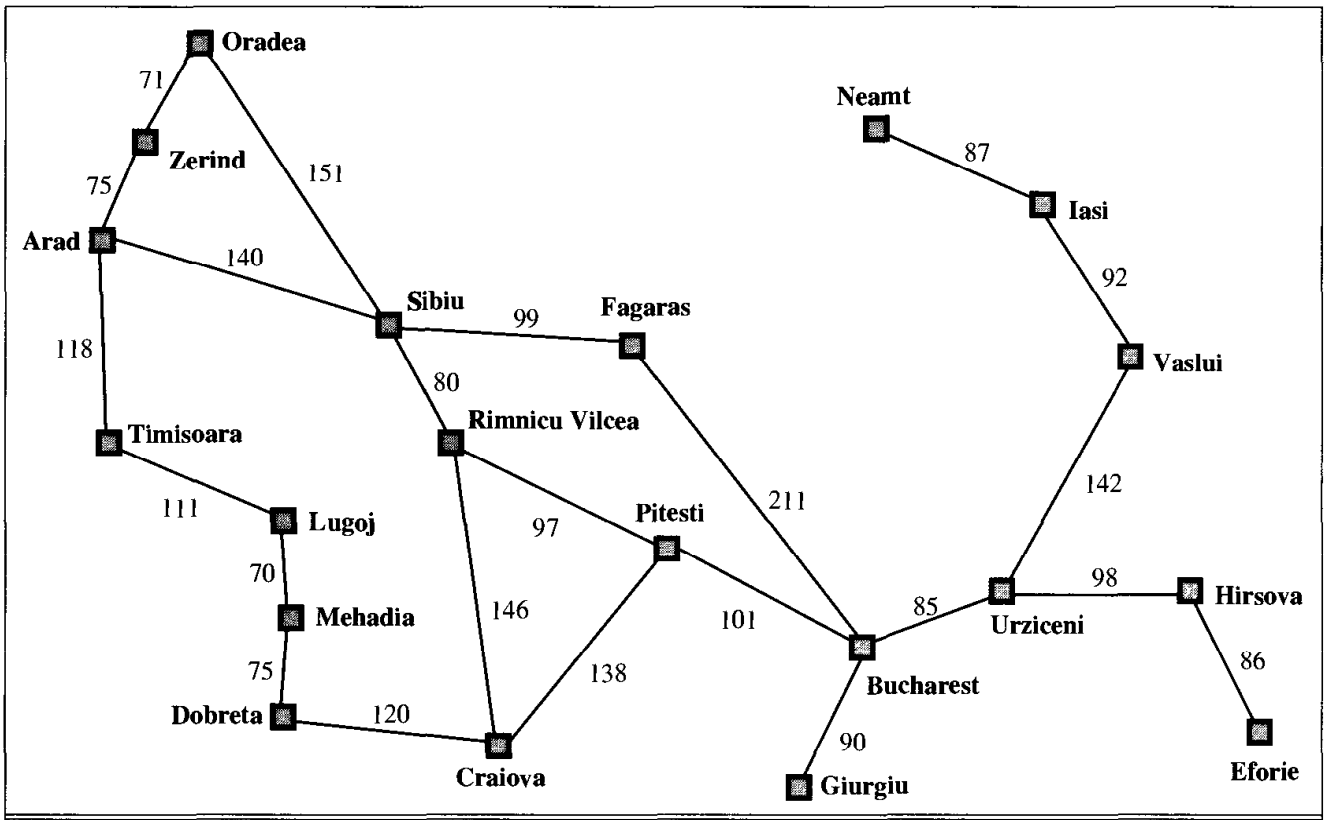


Figure 3.2 A simplified road map of part of Romania.

reasonable, yet it omits a great many aspects of the real world. Compare the simple state description we have chosen, *In(Arad)*, to an actual cross-country trip, where the state of the world includes so many things: the traveling companions, what is on the radio, the scenery out of the window, whether there are any law enforcement officers nearby, how far it is to the next rest stop, the condition of the road, the weather, and so on. All these considerations are left out of our state descriptions because they are irrelevant to the problem of finding a route to Bucharest. The process of removing detail from a representation is called **abstraction**.

ABSTRACTION

In addition to abstracting the state description, we must abstract the actions themselves. A driving action has many effects. Besides changing the location of the vehicle and its occupants, it takes up time, consumes fuel, generates pollution, and changes the agent (as they say, travel is broadening). In our formulation, we take into account only the change in location. Also, there are many actions that we will omit altogether: turning on the radio, looking out of the window, slowing down for law enforcement officers, and so on. And of course, we don't specify actions at the level of "turn steering wheel to the left by three degrees."

Can we be more precise about defining the appropriate level of abstraction? Think of the abstract states and actions we have chosen as corresponding to large sets of detailed world states and detailed action sequences. Now consider a solution to the abstract problem: for example, the path from Arad to Sibiu to Rimnicu Vilcea to Pitesti to Bucharest. This abstract solution corresponds to a large number of more detailed paths. For example, we could drive with the radio on between Sibiu and Rimnicu Vilcea, and then switch it off for the rest of the trip. The abstraction is *valid* if we can expand 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. The abstraction is *useful* if carrying out each of the actions in the solution is easier than the original problem; in this case they are easy enough that they can be carried out without further search or planning by an average driving agent. The choice of a good abstraction thus involves removing as much detail as possible while retaining validity and ensuring that the abstract actions are easy to carry out. Were it not for the ability to construct useful abstractions, intelligent agents would be completely swamped by the real world.

3.2 EXAMPLE PROBLEMS

The problem-solving approach has been applied to a vast array of task environments. We list some of the best known here, distinguishing between *toy* and *real-world* problems. A **toy problem** is intended to illustrate or exercise various problem-solving methods. It can be given a concise, exact description. This means that it can be used easily by different researchers to compare the performance of algorithms. A **real-world problem** is one whose solutions people actually care about. They tend not to have a single agreed-upon description, but we will attempt to give the general flavor of their formulations.

TOY PROBLEM

REAL-WORLD PROBLEM

Toy problems

The first example we will examine is the **vacuum world** first introduced in Chapter 2. (See Figure 2.2.) This can be formulated as a problem as follows:

- ◇ **States**: The agent is in one of two locations, each of which might or might not contain dirt. Thus there are $2 \times 2^2 = 8$ possible world states.
- ◇ **Initial state**: Any state can be designated as the initial state.
- ◇ **Successor function**: This generates the legal states that result from trying the three actions (*Left*, *Right*, and *Suck*). The complete state space is shown in Figure 3.3.
- ◇ **Goal test**: This checks whether all the squares are clean.
- ◇ **Path cost**: Each step costs 1, so the path cost is the number of steps in the path.

Compared with the real world, this toy problem has discrete locations, discrete dirt, reliable cleaning, and it never gets messed up once cleaned. (In Section 3.6, we will relax these assumptions.) One important thing to note is that the state is determined by both the agent location and the dirt locations. A larger environment with n locations has $n\ 2^n$ states.

8-PUZZLE

The **8-puzzle**, an instance of which is shown in Figure 3.4, consists of a 3×3 board with eight numbered tiles and a blank space. A tile adjacent to the blank space can slide into the space. The object is to reach a specified goal state, such as the one shown on the right of the figure. The standard formulation is as follows:

- ◇ **States**: A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
- ◇ **Initial state**: Any state can be designated as the initial state. Note that any given goal can be reached from exactly half of the possible initial states (Exercise 3.4).

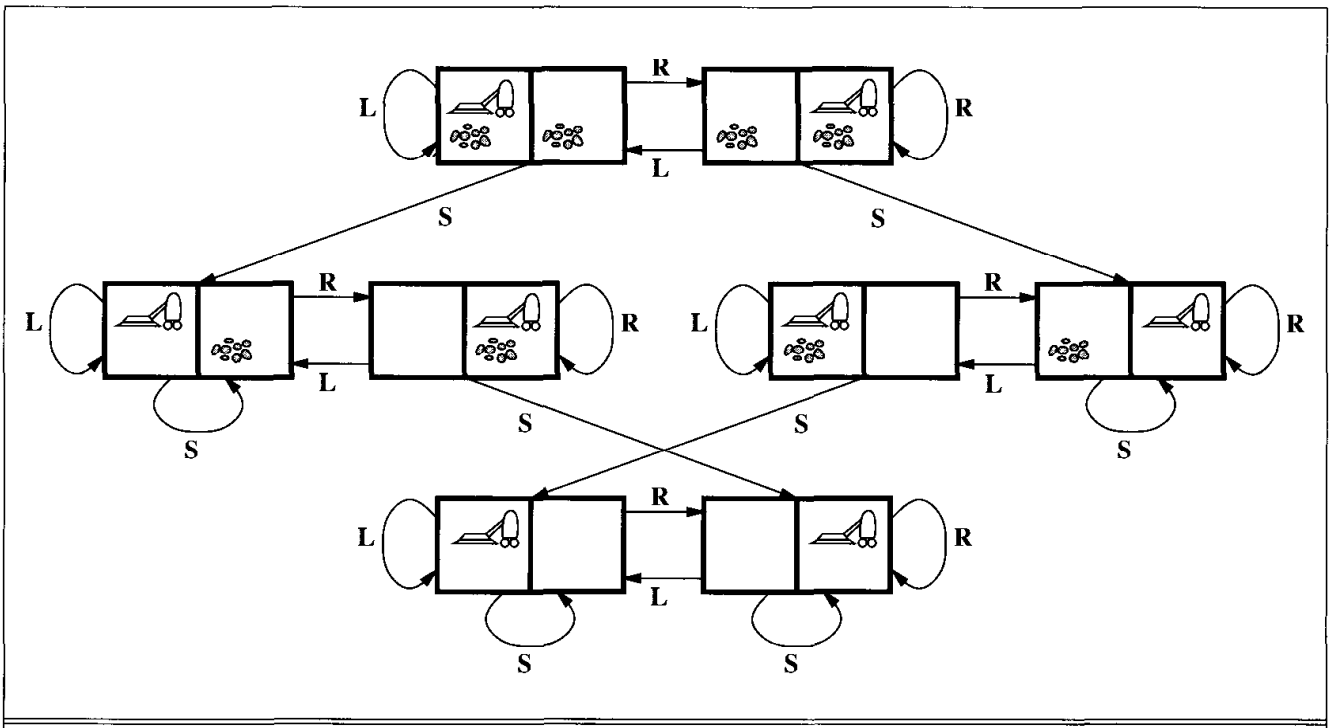


Figure 3.3 The state space for the vacuum world. Arcs denote actions: L = *Left*, R = *Right*, S = *Suck*.

◇ **Successor function**: This generates the legal states that result from trying the four actions (blank moves *Left*, *Right*, *Up*, or *Down*).

◇ **Goal test**: This checks whether the state matches the goal configuration shown in Figure 3.4. (Other goal configurations are possible.)

◇ **Path cost**: Each step costs 1, so the path cost is the number of steps in the path.

What abstractions have we included here? The actions are abstracted to their beginning and final states, ignoring the intermediate locations where the block is sliding. We've abstracted away actions such as shaking the board when pieces get stuck, or extracting the pieces with a knife and putting them back again. We're left with a description of the rules of the puzzle, avoiding all the details of physical manipulations.

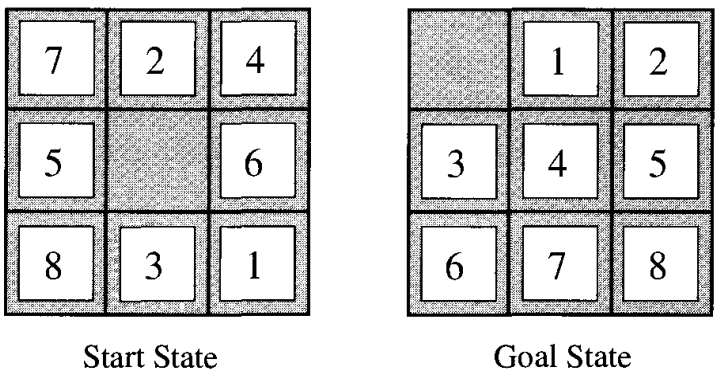


Figure 3.4 A typical instance of the 8-puzzle.

SLIDING-BLOCK PUZZLES

The 8-puzzle belongs to the family of **sliding-block puzzles**, which are often used as test problems for new search algorithms in AI. This general class is known to be NP-complete, so one does not expect to find methods significantly better in the worst case than the search algorithms described in this chapter and the next. The 8-puzzle has $9!/2 = 181,440$ reachable states and is easily solved. The 15-puzzle (on a 4×4 board) has around 1.3 trillion states, and random instances can be solved optimally in a few milliseconds by the best search algorithms. The 24-puzzle (on a 5×5 board) has around 10^{25} states, and random instances are still quite difficult to solve optimally with current machines and algorithms.

8-QUEENS PROBLEM

The goal of the **8-queens problem** is to place eight queens on a chessboard such that no queen attacks any other. (A queen attacks any piece in the same row, column or diagonal.) Figure 3.5 shows an attempted solution that fails: the queen in the rightmost column is attacked by the queen at the top left.

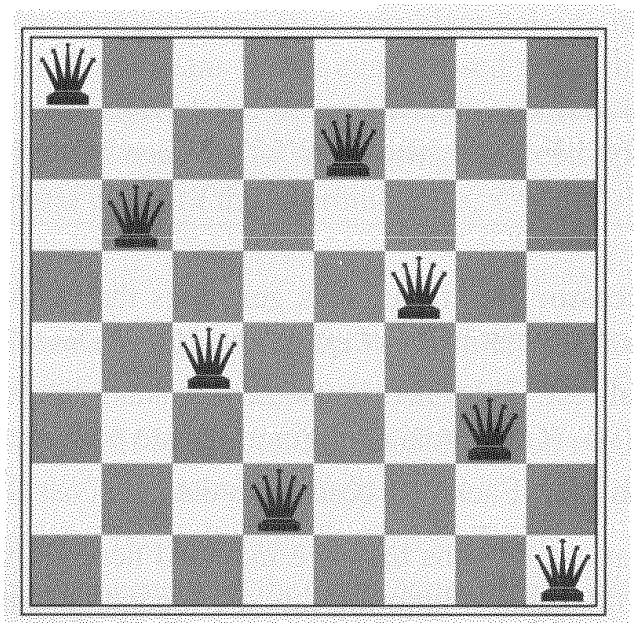


Figure 3.5 Almost a solution to the 8-queens problem. (Solution is left as an exercise.)

Although efficient special-purpose algorithms exist for this problem and the whole n-queens family, it remains an interesting test problem for search algorithms. There are two main kinds of formulation. An **incremental formulation** involves operators that *augment* the state description, starting with an empty state; for the 8-queens problem, this means that each action adds a queen to the state. A **complete-state formulation** starts with all 8 queens on the board and moves them around. In either case, the path cost is of no interest because only the final state counts. The first incremental formulation one might try is the following:

INCREMENTAL FORMULATION

COMPLETE-STATE FORMULATION

- ◇ **States**: Any arrangement of 0 to 8 queens on the board is a state.
- ◇ **Initial state**: No queens on the board.
- ◇ **Successor function**: Add a queen to any empty square.
- ◇ **Goal test**: 8 queens are on the board, none attacked.

In this formulation, we have $64 \cdot 63 \cdots 57 \approx 3 \times 10^{14}$ possible sequences to investigate. A better formulation would prohibit placing a queen in any square that is already attacked:

- ◇ **States**: Arrangements of n queens ($0 \leq n \leq 8$), one per column in the leftmost n columns, with no queen attacking another are states.
- ◇ **Successor function**: Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.

This formulation reduces the 8-queens state space from 3×10^{14} to just 2,057, and solutions are easy to find. On the other hand, for 100 queens the initial formulation has roughly 10^{400} states whereas the improved formulation has about 10^{52} states (Exercise 3.5). This is a huge reduction, but the improved state space is still too big for the algorithms in this chapter to handle. Chapter 4 describes the complete-state formulation and Chapter 5 gives a simple algorithm that makes even the million-queens problem easy to solve.

Real-world problems

ROUTE-FINDING PROBLEM

We have already seen how the **route-finding problem** is defined in terms of specified locations and transitions along links between them. Route-finding algorithms are used in a variety of applications, such as routing in computer networks, military operations planning, and airline travel planning systems. These problems are typically complex to specify. Consider a simplified example of an airline travel problem specified as follows:

- ◇ **States**: Each is represented by a location (e.g., an airport) and the current time.
- ◇ **Initial state**: This is specified by the problem.
- ◇ **Successor function**: This returns the states resulting from taking any scheduled flight (perhaps further specified by seat class and location), leaving later than the current time plus the within-airport transit time, from the current airport to another.
- ◇ **Goal test**: Are we at the destination by some prespecified time?
- ◇ **Path cost**: This depends on monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards, and so on.

Commercial travel advice systems use a problem formulation of this kind, with many additional complications to handle the byzantine fare structures that airlines impose. Any seasoned traveller knows, however, that not all air travel goes according to plan. A really good system should include contingency plans—such as backup reservations on alternate flights—to the extent that these are justified by the cost and likelihood of failure of the original plan.

TOURING PROBLEMS

Touring problems are closely related to route-finding problems, but with an important difference. Consider, for example, the problem, "Visit every city in Figure 3.2 at least once, starting and ending in Bucharest." As with route finding, the actions correspond to trips between adjacent cities. The state space, however, is quite different. Each state must include not just the current location but also the *set of cities the agent has visited.* So the initial state would be "In Bucharest; visited {Bucharest}," a typical intermediate state would be "In Vaslui; visited {Bucharest,Urziceni,Vaslui}," and the goal test would check whether the agent is in Bucharest and all 20 cities have been visited.

TRAVELING SALESPERSON PROBLEM

The **traveling salesperson problem** (TSP) is a touring problem in which each city must be visited exactly once. The aim is to find the *shortest* tour. The problem is known to be NP-hard, but an enormous amount of effort has been expended to improve the capabilities of TSP algorithms. In addition to planning trips for traveling salespersons, these algorithms have been used for tasks such as planning movements of automatic circuit-board drills and of stocking machines on shop floors.

VLSI LAYOUT

A **VLSI layout** problem requires positioning millions of components and connections on a chip to minimize area, minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield. The layout problem comes after the logical design phase, and is usually split into two parts: **cell layout** and **channel routing**. In cell layout, the primitive components of the circuit are grouped into cells, each of which performs some recognized function. Each cell has a fixed footprint (size and shape) and requires a certain number of connections to each of the other cells. The aim is to place the cells on the chip so that they do not overlap and so that there is room for the connecting wires to be placed between the cells. Channel routing finds a specific route for each wire through the gaps between the cells. These search problems are extremely complex, but definitely worth solving. In Chapter 4, we will see some algorithms capable of solving them.

ROBOT NAVIGATION

Robot navigation is a generalization of the route-finding problem described earlier. Rather than a discrete set of routes, a robot can move in a continuous space with (in principle) an infinite set of possible actions and states. For a circular robot moving on a flat surface, the space is essentially two-dimensional. When the robot has arms and legs or wheels that must also be controlled, the search space becomes many-dimensional. Advanced techniques are required just to make the search space finite. We examine some of these methods in Chapter 25. In addition to the complexity of the problem, real robots must also deal with errors in their sensor readings and motor controls.

AUTOMATIC ASSEMBLY SEQUENCING

Automatic assembly sequencing of complex objects by a robot was first demonstrated by FREDDY (Michie, 1972). Progress since then has been slow but sure, to the point where the assembly of intricate objects such as electric motors is economically feasible. 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 a step in the sequence for feasibility is a difficult geometrical search problem closely related to robot navigation. Thus, the generation of legal successors is the expensive part of assembly sequencing. Any practical algorithm must avoid exploring all but a tiny fraction of the state space. Another important assembly problem is **protein design**, in which the goal is to find a sequence of amino acids that will fold into a three-dimensional protein with the right properties to cure some disease.

PROTEIN DESIGN

INTERNET SEARCHING

In recent years there has been increased demand for software robots that perform **Internet searching**, looking for answers to questions, for related information, or for shopping deals. This is a good application for search techniques, because it is easy to conceptualize the Internet as a graph of nodes (pages) connected by links. A full description of Internet search is deferred until Chapter 10.

3.3 SEARCHING FOR SOLUTIONS

Having formulated some problems, we now need to solve them. This is done by a search through the state space. This chapter deals with search techniques that use an explicit **search tree** that is generated by the initial state and the successor function that together define the state space. In general, we may have a search *graph* rather than a search *tree*, when the same state can be reached from multiple paths. We defer consideration of this important complication until Section 3.5.

SEARCH TREE

Figure 3.6 shows some of the expansions in the search tree for finding a route from Arad to Bucharest. The root of the search tree is a **search node** corresponding to the initial state, *In(Arad)*. The first step is to test whether this is a goal state. Clearly it is not, but it is important to check so that we can solve trick problems like "starting in Arad, get to Arad." Because this is not a goal state, we need to consider some other states. This is done by **expanding** the current state; that is, applying the successor function to the current state, thereby **generating** a new set of states. In this case, we get three new states: *In(Sibiu)*, *In(Timisoara)*, and *In(Zerind)*. Now we must choose which of these three possibilities to consider further.

SEARCH NODE

EXPANDING

GENERATING

This is the essence of search—following up one option now and putting the others aside for later, in case the first choice does not lead to a solution. Suppose we choose Sibiu first. We check to see whether it is a goal state (it is not) and then expand it to get *In(Arad)*, *In(Fagaras)*, *In(Oradea)*, and *In(RimnicuVilcea)*. We can then choose any of these four, or go back and choose Timisoara or Zerind. We continue choosing, testing, and expanding until either a solution is found or there are no more states to be expanded. The choice of which state to expand is determined by the **search strategy**. The general tree-search algorithm is described informally in Figure 3.7.

SEARCH STRATEGY

It is important to distinguish between the state space and the search tree. For the route finding problem, there are only 20 states in the state space, one for each city. But there are an infinite number of paths in this state space, so the search tree has an infinite number of nodes. For example, the three paths Arad–Sibiu, Arad–Sibiu–Arad, Arad–Sibiu–Arad–Sibiu are the first three of an infinite sequence of paths. (Obviously, a good search algorithm avoids following such repeated paths; Section 3.5 shows how.)

There are many ways to represent nodes, but we will assume that a node is a data structure with five components:

- STATE: the state in the state space to which the node corresponds;
- PARENT-NODE: the node in the search tree that generated this node;
- ACTION: the action that was applied to the parent to generate the node;
- PATH-COST: the cost, traditionally denoted by $g(n)$, of the path from the initial state to the node, as indicated by the parent pointers; and
- DEPTH: the number of steps along the path from the initial state.

It is important to remember the distinction between nodes and states. A node is a bookkeeping data structure used to represent the search tree. A state corresponds to a configuration of the

world. Thus, nodes are on particular paths, as defined by PARENT-NODE pointers, whereas states are not. Furthermore, two different nodes can contain the same world state, if that state is generated via two different search paths. The node data structure is depicted in Figure 3.8.

We also need to represent the collection of nodes that have been generated but not yet expanded—this collection is called the **fringe**. Each element of the fringe is a **leaf node**, that

FRINGE

LEAF NODE

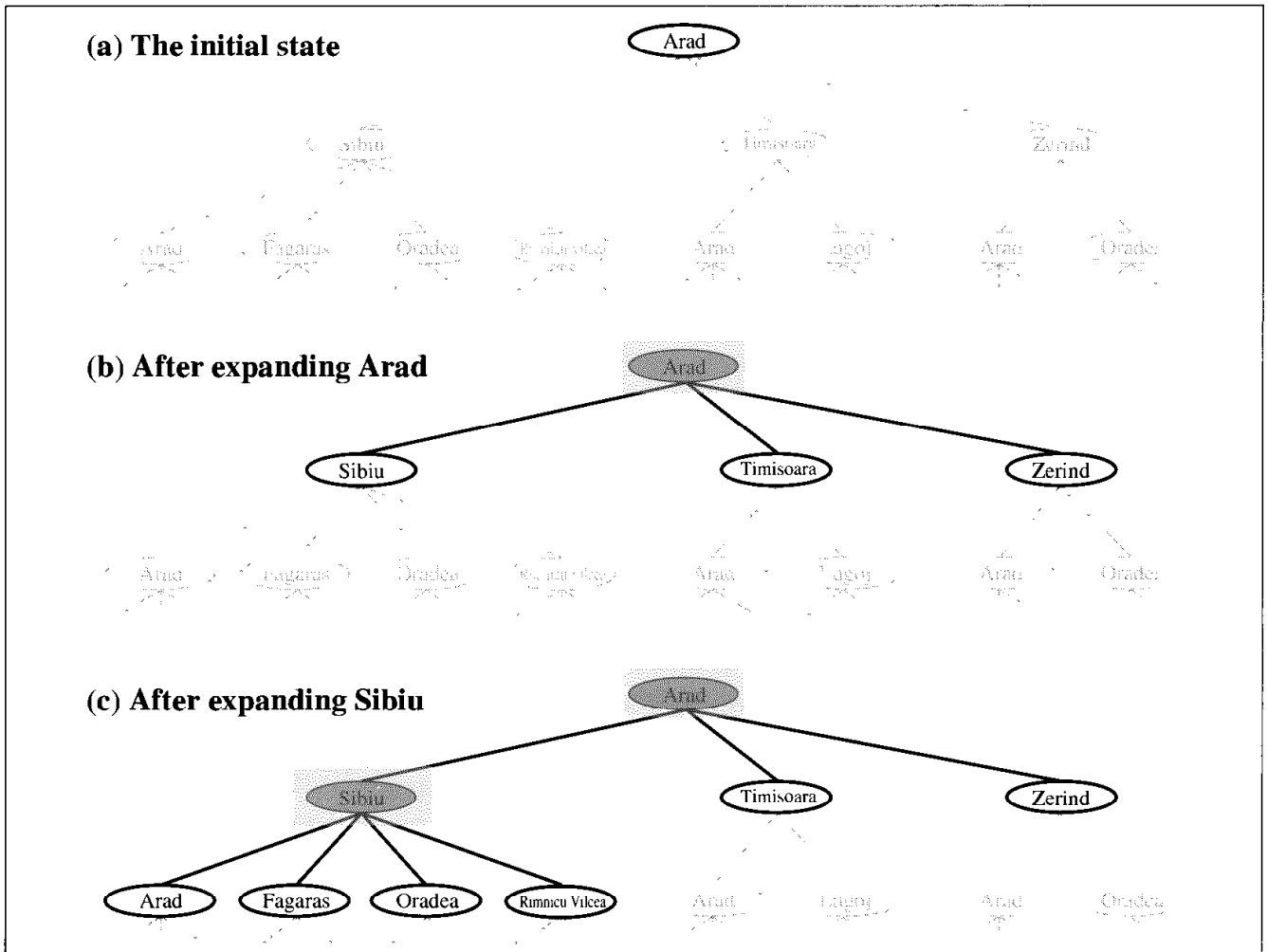


Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

function TREE-SEARCH(*problem*, *strategy*) **returns** a solution, or failure
  initialize the search tree using the initial state of *problem*
  loop do
    if there are no candidates for expansion **then return** failure
    choose a leaf node for expansion according to *strategy*
    if the node contains a goal state **then return** the corresponding solution
    else expand the node and add the resulting nodes to the search tree

Figure 3.7 An informal description of the general tree-search algorithm.

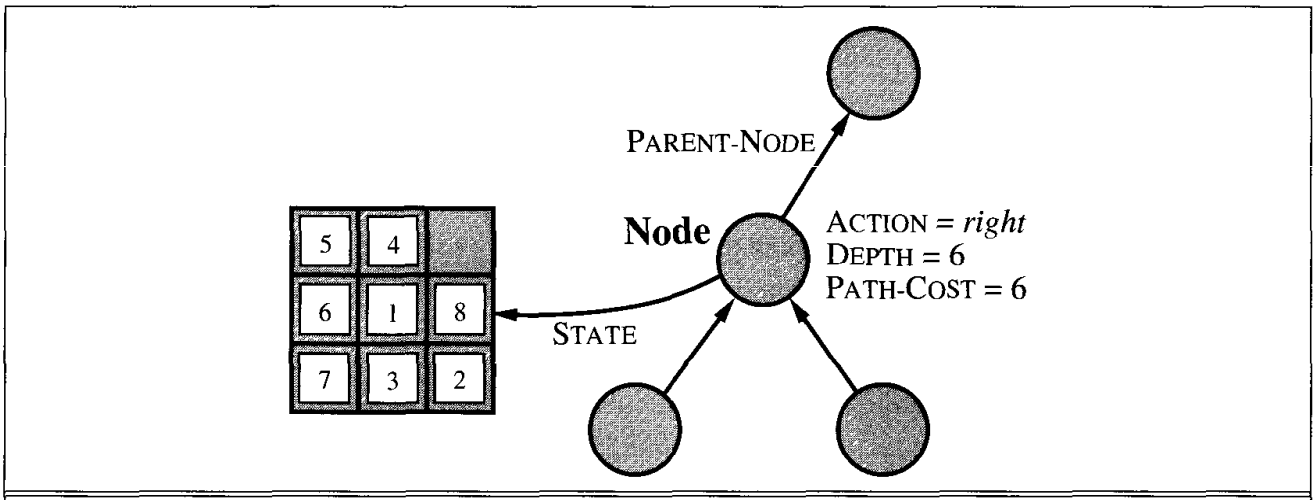


Figure 3.8 Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

is, a node with no successors in the tree. In Figure 3.6, the fringe of each tree consists of those nodes with bold outlines. The simplest representation of the fringe would be a set of nodes. The search strategy then would be a function that selects the next node to be expanded from this set. Although this is conceptually straightforward, it could be computationally expensive, because the strategy function might have to look at every element of the set to choose the best one. Therefore, we will assume that the collection of nodes is implemented as a **queue**. The operations on a queue are as follows:

QUEUE

- MAKE-QUEUE(*element*, ...) creates a queue with the given element(s).
- EMPTY?(*queue*) returns true only if there are no more elements in the queue.
- FIRST(*queue*) returns the first element of the queue.
- REMOVE-FIRST(*queue*) returns FIRST(*queue*) and removes it from the queue.
- INSERT(*element*, *queue*) inserts an element into the queue and returns the resulting queue. (We will see that different types of queues insert elements in different orders.)
- INSERT-ALL(*elements*, *queue*) inserts a set of elements into the queue and returns the resulting queue.

With these definitions, we can write the more formal version of the general tree-search algorithm shown in Figure 3.9.

Measuring problem-solving performance

The output of a problem-solving algorithm is either *failure* or a solution. (Some algorithms might get stuck in an infinite loop and never return an output.) We will evaluate an algorithm's performance in four ways:

COMPLETENESS
OPTIMALITY
TIME COMPLEXITY
SPACE COMPLEXITY

- ◇ **Completeness**: Is the algorithm guaranteed to find a solution when there is one?
- ◇ **Optimality**: Does the strategy find the optimal solution, as defined on page 62?
- ◇ **Time complexity**: How long does it take to find a solution?
- ◇ **Space complexity**: How much memory is needed to perform the search?

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure

  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if EMPTY?(fringe) then return failure
    node ← REMOVE-FIRST(fringe)
    if GOAL-TEST[problem] applied to STATE[node] succeeds
      then return SOLUTION(node)
    fringe ← INSERT-ALL(EXPAND(node, problem), fringe)

function EXPAND(node, problem) returns a set of nodes

  successors ← the empty set
  for each ⟨action, result⟩ in SUCCESSOR-FN[problem](STATE[node]) do
    s ← a new NODE
    STATE[s] ← result
    PARENT-NODE[s] ← node
    ACTION[s] ← action
    PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
    DEPTH[s] ← DEPTH[node] + 1
    add s to successors
  return successors
```

Figure 3.9 The general tree-search algorithm. (Note that the *fringe* argument must be an empty queue, and the type of the queue will affect the order of the search.) The SOLUTION function returns the sequence of actions obtained by following parent pointers back to the root.

Time and space complexity are always 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, because the graph is viewed as an explicit data structure that is input to the search program. (The map of Romania is an example of this.) In AI, where the graph is represented implicitly by the initial state and successor function and is frequently infinite, complexity is expressed in terms of three quantities: b, the **branching factor** or maximum number of successors of any node; d, the depth of the shallowest goal node; and m, the maximum length of any path in the state space.

BRANCHING FACTOR

Time is often measured in terms of the number of nodes generated[5] during the search, and space in terms of the maximum number of nodes stored in memory.

SEARCH COST

TOTAL COST

To assess the effectiveness of a search algorithm, we can consider just the **search cost**—which typically depends on the time complexity but can also include a term for memory usage—or we can use the **total cost**, which combines the search cost and the path cost of the solution found. For the problem of finding a route from Arad to Bucharest, the search cost

[5] Some texts measure time in terms of the number of node *expansions* instead. The two measures differ by at most a factor of b. It seems to us that the execution time of a node expansion increases with the number of nodes generated in that expansion.

is the amount of time taken by the search and the solution cost is the total length of the path in kilometers. Thus, to compute the total cost, we have to add kilometers and milliseconds. There is no "official exchange rate" between the two, but it might be reasonable in this case to convert kilometers into milliseconds by using an estimate of the car's average speed (because time is what the agent cares about). This enables the agent to find an optimal tradeoff point at which further computation to find a shorter path becomes counterproductive. The more general problem of tradeoffs between different goods will be taken up in Chapter 16.

3.4 UNINFORMED SEARCH STRATEGIES

UNINFORMED SEARCH

This section covers five search strategies that come under the heading of **uninformed search** (also called **blind search**). The term means that they have no additional information about states beyond that provided in the problem definition. All they can do is generate successors and distinguish a goal state from a nongoal state. Strategies that know whether one non-

INFORMED SEARCH
HEURISTIC SEARCH

goal state is "more promising" than another are called **informed search** or **heuristic search** strategies; they will be covered in Chapter 4. All search strategies are distinguished by the *order* in which nodes are expanded.

Breadth-first search

BREADTH-FIRST SEARCH

Breadth-first search is a simple strategy 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. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.

Breadth-first search can be implemented by calling TREE-SEARCH with an empty fringe that is a first-in-first-out (FIFO) queue, assuring that the nodes that are visited first will be expanded first. In other words, calling TREE-SEARCH(*problem*,FIFO-QUEUE()) results in a breadth-first search. The FIFO queue puts all newly generated successors at the end of the queue, which means that shallow nodes are expanded before deeper nodes. Figure 3.10 shows the progress of the search on a simple binary tree.

We will evaluate breadth-first search using the four criteria from the previous section. We can easily see that it is *complete*—if the shallowest goal node is at some finite depth d, breadth-first search will eventually find it after expanding all shallower nodes (provided the branching factor b is finite). The *shallowest* goal node is not necessarily the *optimal* one; technically, breadth-first search is optimal if the path cost is a nondecreasing function of the depth of the node. (For example, when all actions have the same cost.)

So far, the news about breadth-first search has been good. To see why it is not always the strategy of choice, we have to consider the amount of time and memory it takes to complete a search. To do this, we consider a hypothetical state space where every state has b successors. The root of the search tree generates b nodes at the first level, 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. In the worst

case, we would expand all but the last node at level d (since the goal itself is not expanded), generating $b^{d+1} - b$ nodes at level $d + 1$. Then the total number of nodes generated is

$$b + b^2 + b^3 + \cdots + b^d + (b^{d+1} - b) = O(b^{d+1}) .$$

Every node that is generated must remain in memory, because it is either part of the fringe or is an ancestor of a fringe node. The space complexity is, therefore, the same as the time complexity (plus one node for the root).

Those who do complexity analysis are worried (or excited, if they like a challenge) by exponential complexity bounds such as $O(b^{d+1})$. Figure 3.11 shows why. It lists the time and memory required for a breadth-first search with branching factor $b = 10$, for various values of the solution depth d. The table assumes that 10,000 nodes can be generated per second and that a node requires 1000 bytes of storage. Many search problems fit roughly within these assumptions (give or take a factor of 100) when run on a modern personal computer.

There are two lessons to be learned from Figure 3.11. First, *the memory requirements are a bigger problem for breadth-first search than is the execution time.* 31 hours would not be too long to wait for the solution to an important problem of depth 8, but few computers have the terabyte of main memory it would take. Fortunately, there are other search strategies that require less memory.

The second lesson is that the time requirements are still a major factor. If your problem has a solution at depth 12, then (given our assumptions) it will take 35 years for breadth-first search (or indeed any uninformed search) to find it. In general, *exponential-complexity search problems cannot be solved by uninformed methods for any but the smallest instances.*

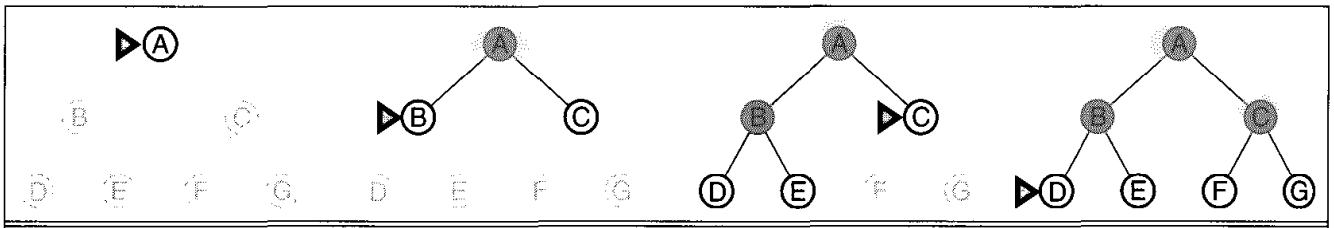


Figure 3.10 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

Depth	Nodes	Time	Memory
2	1100	.11 seconds	1 megabyte
4	111,100	11 seconds	106 megabytes
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabytes
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3,523 years	1 exabyte

Figure 3.11 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 10,000 nodes/second; 1000 bytes/node.

Uniform-cost search

UNIFORM-COST SEARCH

Breadth-first search is optimal when all step costs are equal, because it always expands the *shallowest* unexpanded node. By a simple extension, we can find an algorithm that is optimal with any step cost function. Instead of expanding the shallowest node, **uniform-cost search** expands the node n with the *lowest path cost*. Note that if all step costs are equal, this is identical to breadth-first search.

Uniform-cost search does not care about the *number* of steps a path has, but only about their total cost. Therefore, it will get stuck in an infinite loop if it ever expands a node that has a zero-cost action leading back to the same state (for example, a $NoOp$ action). We can guarantee completeness provided the cost of every step is greater than or equal to some small positive constant ϵ. This condition is also sufficient to ensure *optimality*. It means that the cost of a path always increases as we go along the path. From this property, it is easy to see that the algorithm expands nodes in order of increasing path cost. Therefore, the first goal node selected for expansion is the optimal solution. (Remember that TREE-SEARCH applies the goal test only to the nodes that are selected for expansion.) We recommend trying the algorithm out to find the shortest path to Bucharest.

Uniform-cost search is guided by path costs rather than depths, so its complexity cannot easily be characterized in terms of b and d. Instead, let C^* be the cost of the optimal solution, and assume that every action costs at least ϵ. 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, and often does, explore large trees of small steps before exploring paths involving large and perhaps useful steps. When all step costs are equal, of course, $b^{1+\lfloor C^*/\epsilon \rfloor}$ is just b^d.

Depth-first search

DEPTH-FIRST SEARCH

Depth-first search always expands the *deepest* node in the current fringe of the search tree. The progress of the search is illustrated in Figure 3.12. The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As those nodes are expanded, they are dropped from the fringe, so then the search "backs up" to the next shallowest node that still has unexplored successors.

This strategy can be implemented by TREE-SEARCH with a last-in-first-out (LIFO) queue, also known as a stack. As an alternative to the TREE-SEARCH implementation, it is common to implement depth-first search with a recursive function that calls itself on each of its children in turn. (A recursive depth-first algorithm incorporating a depth limit is shown in Figure 3.13.)

Depth-first search has very modest memory requirements. It needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path. Once a node has been expanded, it can be removed from memory as soon as all its descendants have been fully explored. (See Figure 3.12.) For a state space with branching factor b and maximum depth m, depth-first search requires storage of only $bm + 1$ nodes. Using the same assumptions as Figure 3.11, and assuming that nodes at the same depth as the goal node have no successors, we find that depth-first search would require 118 kilobytes instead of 10 petabytes at depth $d = 12$, a factor of 10 billion times less space.

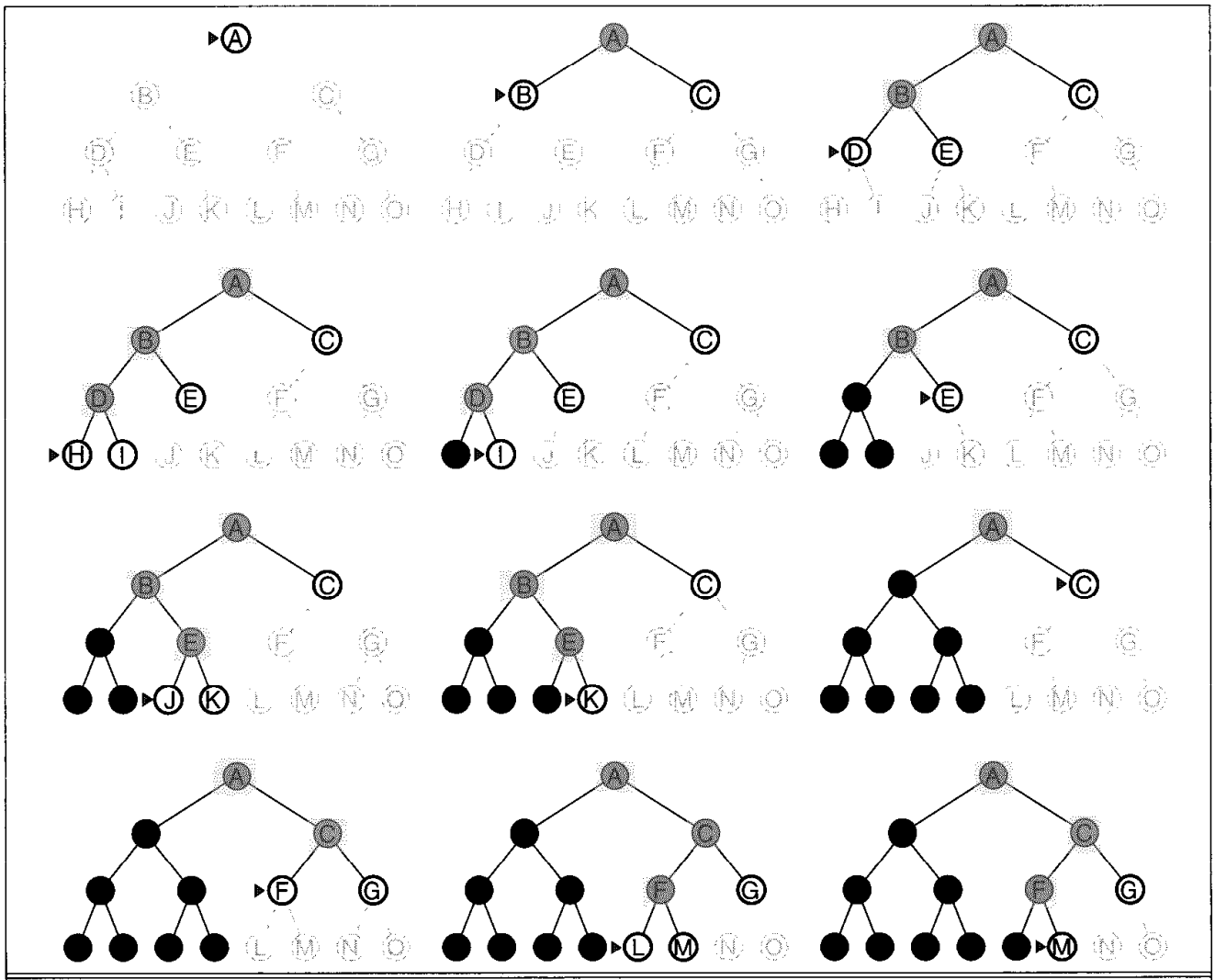


Figure 3.12 Depth-first search on a binary tree. Nodes that have been expanded and have no descendants in the fringe can be removed from memory; these are shown in black. Nodes at depth 3 are assumed to have no successors and M is the only goal node.

BACKTRACKING SEARCH

A variant of depth-first search called **backtracking search** uses still less memory. 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 this way, only $O(m)$ memory is needed rather than $O(bm)$. Backtracking search facilitates yet another memory-saving (and time-saving) trick: the idea of generating a successor by *modifying* the current state description directly rather than copying it first. This reduces the memory requirements to just one state description and $O(m)$ actions. For this to work, we must be able to undo each modification when we go back to generate the next successor. For problems with large state descriptions, such as robotic assembly, these techniques are critical to success.

The drawback of depth-first search is that it can make a wrong choice and get stuck going down a very long (or even infinite) path when a different choice would lead to a solution near the root of the search tree. For example, in Figure 3.12, depth-first search will explore the entire left subtree even if node C is a goal node. If node J were also a goal node, then depth-first search would return it as a solution; hence, depth-first search is not optimal. If

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  cutoff_occurred? ← false
  if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
  else if DEPTH[node] = limit then return cutoff
  else for each successor in EXPAND(node, problem) do
      result ← RECURSIVE-DLS(successor, problem, limit)
      if result = cutoff then cutoff_occurred? ← true
      else if result ≠ failure then return result
  if cutoff_occurred? then return cutoff else return failure
```

Figure 3.13 A recursive implementation of depth-limited search.

the left subtree were of unbounded depth but contained no solutions, depth-first search would never terminate; hence, it is not complete. In the worst case, depth-first search will generate all of the $O(b^m)$ nodes in the search tree, where m is the maximum depth of any node. Note that m can be much larger than d (the depth of the shallowest solution), and is infinite if the tree is unbounded.

Depth-limited search

The problem of unbounded trees can be alleviated by supplying depth-first search with a predetermined depth limit ℓ. That is, nodes at depth ℓ are treated as if they have no successors.

DEPTH-LIMITED SEARCH

This approach is called **depth-limited search**. The depth limit solves the infinite-path problem. Unfortunately, it also introduces an additional source of incompleteness if we choose $\ell < d$, that is, the shallowest goal is beyond the depth limit. (This is not unlikely when d is unknown.) Depth-limited search will also be nonoptimal if we choose $\ell > d$. Its time complexity is $O(b^\ell)$ and its space complexity is $O(b\ell)$. Depth-first search can be viewed as a special case of depth-limited search with $\ell = \infty$.

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

DIAMETER

Depth-limited search can be implemented as a simple modification to the general tree-search algorithm or to the recursive depth-first search algorithm. We show the pseudocode for recursive depth-limited search in Figure 3.13. Notice that depth-limited search can terminate with two kinds of failure: the standard *failure* value indicates no solution; the *cutoff* value indicates no solution within the depth limit.

Iterative deepening depth-first search

ITERATIVE DEEPENING SEARCH

Iterative deepening search (or iterative deepening depth-first search) is a general strategy, often used in combination with depth-first search, that finds the best depth limit. It does this by gradually increasing the limit—first 0, then 1, then 2, and so on—until a goal is found. This will occur when the depth limit reaches d, the depth of the shallowest goal node. The algorithm is shown in Figure 3.14. Iterative deepening combines the benefits of depth-first and breadth-first search. Like depth-first search, its memory requirements are very modest: $O(bd)$ to be precise. Like breadth-first search, it is complete when the branching factor is finite and optimal when the path cost is a nondecreasing function of the depth of the node. Figure 3.15 shows four iterations of ITERATIVE-DEEPENING-SEARCH on a binary search tree, where the solution is found on the fourth iteration.

Iterative deepening search may seem wasteful, because states are generated multiple times. It turns out this is not very costly. The reason is that in a search tree with the same (or nearly the same) branching factor at each level, most of the nodes are in the bottom level, so it does not matter much that the upper levels are generated multiple times. 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 is

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

which gives a time complexity of $O(b^d)$. We can compare this to the nodes generated by a breadth-first search:

$$N(\text{BFS}) = b + b^2 + \cdots + b^d + (b^{d+1} - b) .$$

Notice that breadth-first search generates some nodes at depth $d+1$, whereas iterative deepening does not. The result is that iterative deepening is actually *faster* than breadth-first search, despite the repeated generation of states. For example, if $b = 10$ and $d = 5$, the numbers are

$$\begin{aligned} N(\text{IDS}) &= 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450 \\ N(\text{BFS}) &= 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100 . \end{aligned}$$

In general, iterative deepening is the preferred uninformed search method when there is a large search space and the depth of the solution is not known.

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution, or failure
  inputs: *problem*, a problem

  for *depth* $\leftarrow$ 0 **to** ∞ **do**
    result $\leftarrow$ DEPTH-LIMITED-SEARCH(*problem*, *depth*)
    if *result* $\neq$ cutoff **then return** *result*

Figure 3.14 The iterative deepening search algorithm, which repeatedly applies depth-limited search with increasing limits. It terminates when a solution is found or if the depth-limited search returns *failure*, meaning that no solution exists.

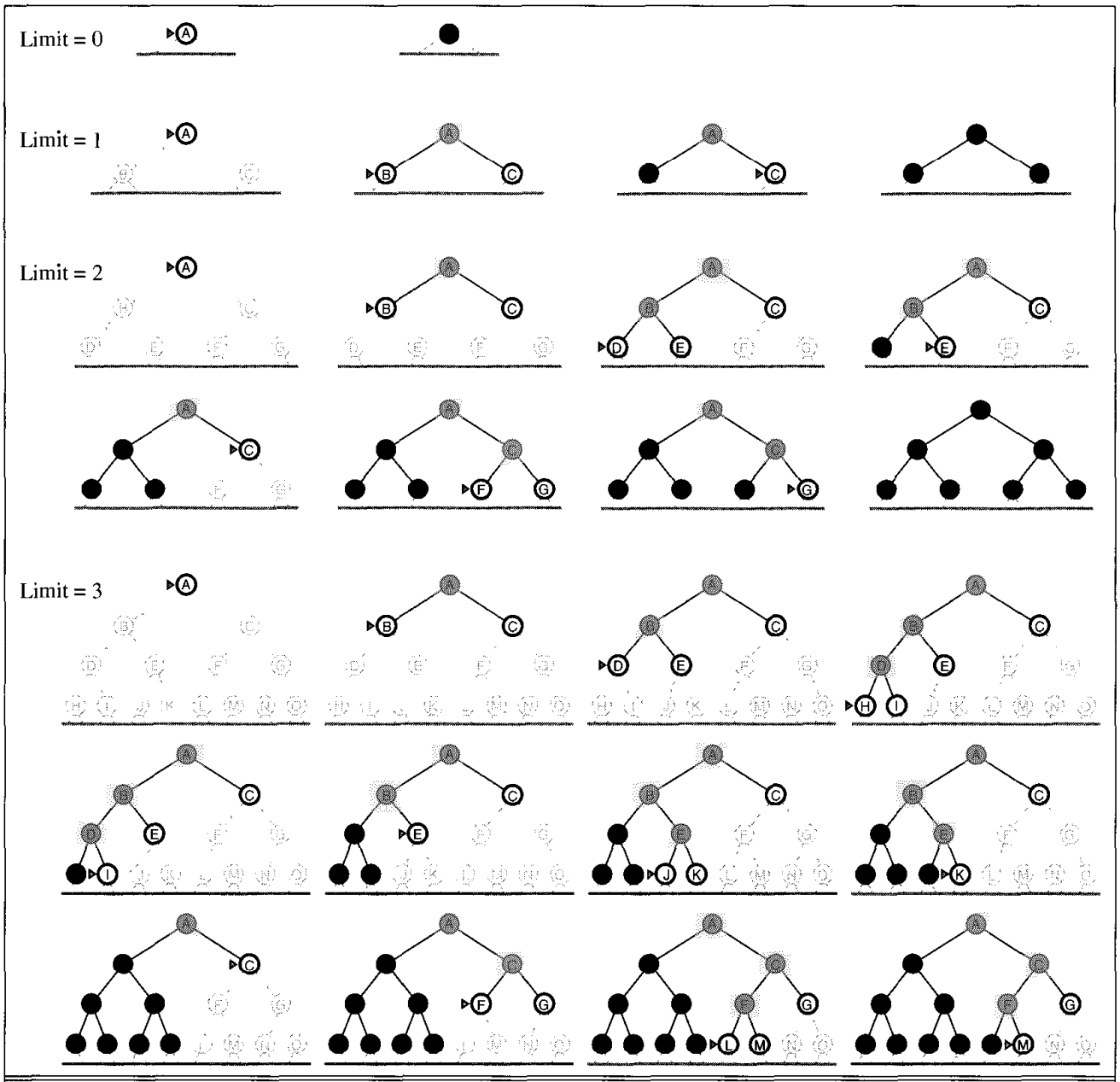


Figure 3.15 Four iterations of iterative deepening search on a binary tree.

Iterative deepening search is analogous to breadth-first search in that it explores a complete layer of new nodes at each iteration before going on to the next layer. It would seem worthwhile to develop an iterative analog to uniform-cost search, inheriting the latter algorithm's optimality guarantees while avoiding its memory requirements. The idea is to use increasing path-cost limits instead of increasing depth limits. The resulting algorithm, called **iterative lengthening search**, is explored in Exercise 3.11. It turns out, unfortunately, that iterative lengthening incurs substantial overhead compared to uniform-cost search.

ITERATIVE LENGTHENING SEARCH

Bidirectional search

The idea behind bidirectional search is to run two simultaneous searches—one forward from the initial state and the other backward from the goal, stopping when the two searches meet

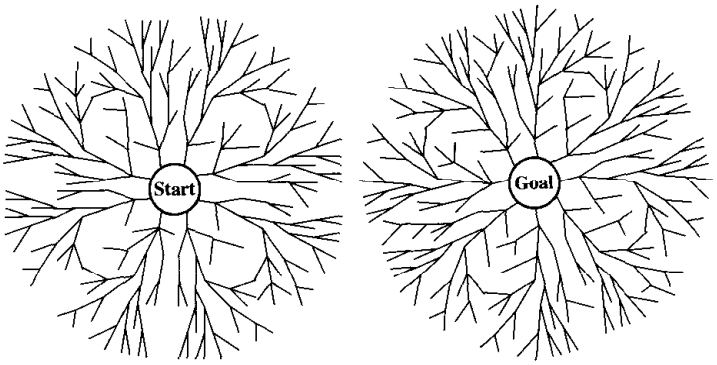


Figure 3.16 A schematic view of a bidirectional search that is about to succeed, when a branch from the start node meets a branch from the goal node.

in the middle (Figure 3.16). The motivation is that $b^{d/2} + b^{d/2}$ is much less than b^d, or in the figure, the area of the two small circles is less than the area of one big circle centered on the start and reaching to the goal.

Bidirectional search is implemented by having one or both of the searches check each node before it is expanded to see if it is in the fringe of the other search tree; if so, a solution has been found. For example, if a problem has solution depth $d = 6$, and each direction runs breadth-first search one node at a time, then in the worst case the two searches meet when each has expanded all but one of the nodes at depth 3. For $b = 10$, this means a total of 22,200 node generations, compared with 11,111,100 for a standard breadth-first search. Checking a node for membership in the other search tree can be done in constant time with a hash table, so the time complexity of bidirectional search is $O(b^{d/2})$. At least one of the search trees must be kept in memory so that the membership check can be done, hence the space complexity is also $O(b^{d/2})$. This space requirement is the most significant weakness of bidirectional search. The algorithm is complete and optimal (for uniform step costs) if both searches are breadth-first; other combinations may sacrifice completeness, optimality, or both.

PREDECESSORS

The reduction in time complexity makes bidirectional search attractive, but how do we search backwards? This is not as easy as it sounds. Let the **predecessors** of a node n, $Pred(n)$, be all those nodes that have n as a successor. Bidirectional search requires that $Pred(n)$ be efficiently computable. The easiest case is when all the actions in the state space are reversible, so that $Pred(n) = Succ(n)$. Other cases may require substantial ingenuity.

Consider the question of what we mean by "the goal" in searching "backward from the goal." For the 8-puzzle and for finding a route in Romania, there is just one goal state, so the backward search is very much like the forward search. If there are several *explicitly listed* goal states—for example, the two dirt-free goal states in Figure 3.3—then we can construct a new dummy goal state whose immediate predecessors are all the actual goal states. Alternatively, some redundant node generations can be avoided by viewing the set of goal states as a single state, each of whose predecessors is also a set of states—specifically, the set of states having a corresponding successor in the set of goal states. (See also Section 3.6.)

The most difficult case for bidirectional search is when the goal test gives only an implicit description of some possibly large set of goal states—for example, all the states satisfy-

ing the "checkmate" goal test in chess. A backward search would need to construct compact descriptions of "all states that lead to checkmate by move m_1" and so on; and those descriptions would have to be tested against the states generated by the forward search. There is no general way to do this efficiently.

Comparing uninformed search strategies

Figure 3.17 compares search strategies in terms of the four evaluation criteria set forth in Section 3.4.

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

Figure 3.17 Evaluation of search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

3.5 Avoiding Repeated States

Up to this point, we have all but ignored one of the most important complications to the search process: the possibility of wasting time by expanding states that have already been encountered and expanded before. For some problems, this possibility never comes up; the state space is a tree and there is only one path to each state. The efficient formulation of the 8-queens problem (where each new queen is placed in the leftmost empty column) is efficient in large part because of this—each state can be reached only through one path. If we formulate the 8-queens problem so that a queen can be placed in any column, then each state with n queens can be reached by $n!$ different paths.

For some problems, repeated states are unavoidable. This includes all problems where the actions are reversible, such as route-finding problems and sliding-blocks puzzles. The search trees for these problems are infinite, but if we prune some of the repeated states, we can cut the search tree down to finite size, generating only the portion of the tree that spans the state-space graph. Considering just the search tree up to a fixed depth, it is easy to find cases where eliminating repeated states yields an exponential reduction in search cost. In the extreme case, a state space of size $d + 1$ (Figure 3.18(a)) becomes a tree with 2^d leaves (Figure 3.18(b)). A more realistic example is the **rectangular grid** as illustrated in Figure 3.18(c). On a grid, each state has four successors, so the search tree including repeated

RECTANGULAR GRID

states has 4^d leaves; but there are only about $2d^2$ distinct states within d steps of any given state. For $d = 20$, this means about a trillion nodes but only about 800 distinct states.

Repeated states, then, can cause a solvable problem to become unsolvable if the algorithm does not detect them. Detection usually means comparing the node about to be expanded to those that have been expanded already; if a match is found, then the algorithm has discovered two paths to the same state and can discard one of them.

For depth-first search, the only nodes in memory are those on the path from the root to the current node. Comparing those nodes to the current node allows the algorithm to detect looping paths that can be discarded immediately. This is fine for ensuring that finite state spaces do not become infinite search trees because of loops; unfortunately, it does not avoid the exponential proliferation of nonlooping paths in problems such as those in Figure 3.18. The only way to avoid these is to keep more nodes in memory. There is a fundamental tradeoff between space and time. *Algorithms that forget their history are doomed to repeat it.*

If an algorithm remembers every state that it has visited, then it can be viewed as exploring the state-space graph directly. We can modify the general TREE-SEARCH algorithm to include a data structure called the **closed list**, which stores every expanded node. (The fringe of unexpanded nodes is sometimes called the **open list**.) If the current node matches a node on the closed list, it is discarded instead of being expanded. The new algorithm is called GRAPH-SEARCH (Figure 3.19). On problems with many repeated states, GRAPH-SEARCH is much more efficient than TREE-SEARCH. Its worst-case time and space requirements are proportional to the size of the state space. This may be much smaller than $O(b^d)$.

CLOSED LIST

OPEN LIST

Optimality for graph search is a tricky issue. We said earlier that when a repeated state is detected, the algorithm has found two paths to the same state. The GRAPH-SEARCH algorithm in Figure 3.19 always discards the *newly discovered* path; obviously, if the newly discovered path is shorter than the original one, GRAPH-SEARCH could miss an optimal solution. Fortunately, we can show (Exercise 3.12) that this cannot happen when using either

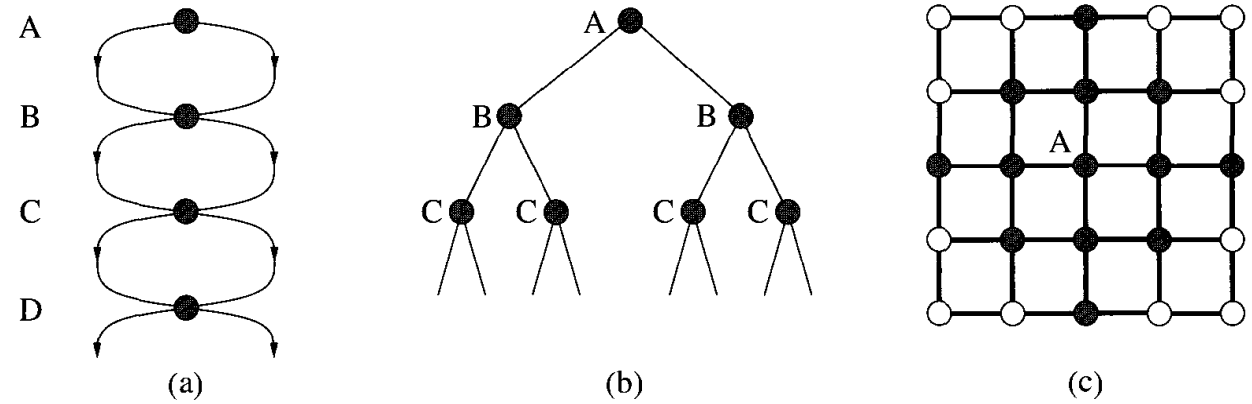


Figure 3.18 State spaces that generate an exponentially larger search tree. (a) A state space in which there are two possible actions leading from A to B, two from B to C, and so on. The state space contains $d + 1$ states, where d is the maximum depth. (b) The corresponding search tree, which has 2^d branches corresponding to the 2^d paths through the space. (c) A rectangular grid space. States within 2 steps of the initial state (A) are shown in gray.

```
function GRAPH-SEARCH(problem, fringe) returns a solution, or failure

  closed ← an empty set
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if EMPTY?(fringe) then return failure
    node ← REMOVE-FIRST(fringe)
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    if STATE[node] is not in closed then
      add STATE[node] to closed
      fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
```

Figure 3.19 The general graph-search algorithm. The set *closed* can be implemented with a hash table to allow efficient checking for repeated states. This algorithm assumes that the first path to a state s is the cheapest (see text).

uniform-cost search or breadth-first search with constant step costs; hence, these two optimal tree-search strategies are also optimal graph-search strategies. Iterative deepening search, on the other hand, uses depth-first expansion and can easily follow a suboptimal path to a node before finding the optimal one. Hence, iterative deepening graph search needs to check whether a newly discovered path to a node is better than the original one, and if so, it might need to revise the depths and path costs of that node's descendants.

Note that the use of a closed list means that depth-first search and iterative deepening search no longer have linear space requirements. Because the GRAPH-SEARCH algorithm keeps every node in memory, some searches are infeasible because of memory limitations.

3.6 SEARCHING WITH PARTIAL INFORMATION

In Section 3.3 we assumed that the environment is fully observable and deterministic and that the agent knows what the effects of each action are. Therefore, the agent can calculate exactly which state results from any sequence of actions and always knows which state it is in. Its percepts provide no new information after each action. What happens when knowledge of the states or actions is incomplete? We find that different types of incompleteness lead to three distinct problem types:

1. **Sensorless problems** (also called **conformant problems**): If the agent has no sensors at all, then (as far as it knows) it could be in one of several possible initial states, and each action might therefore lead to one of several possible successor states.
2. **Contingency problems**: If the environment is partially observable or if actions are uncertain, then the agent's percepts provide *new* information after each action. Each possible percept defines a contingency that must be planned for. A problem is called **adversarial** if the uncertainty is caused by the actions of another agent.

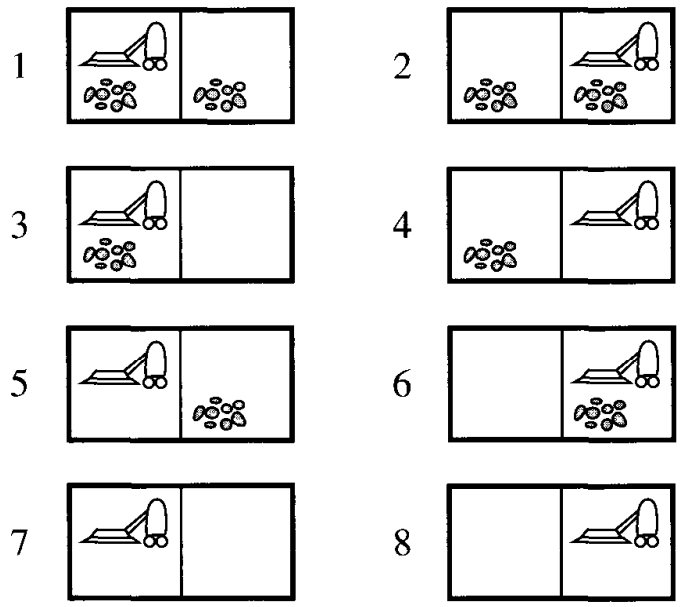


Figure 3.20 The eight possible states of the vacuum world.

3. **Exploration problems**: When the states and actions of the environment are unknown, the agent must act to discover them. Exploration problems can be viewed as an extreme case of contingency problems.

As an example, we will use the vacuum world environment. Recall that the state space has eight states, as shown in Figure 3.20. There are three actions—*Left*, *Right*, and *Suck*—and the goal is to clean up all the dirt (states 7 and 8). If the environment is observable, deterministic, and completely known, then the problem is trivially solvable by any of the algorithms we have described. For example, if the initial state is 5, then the action sequence [*Right*,*Suck*] will reach a goal state, 8. The remainder of this section deals with the sensorless and contingency versions of the problem. Exploration problems are covered in Section 4.5, adversarial problems in Chapter 6.

Sensorless problems

Suppose that the vacuum agent knows all the effects of its actions, but has no sensors. Then it knows only that its initial state is one of the set $\{1, 2, 3, 4, 5, 6, 7, 8\}$. One might suppose that the agent's predicament is hopeless, but in fact it can do quite well. Because it knows what its actions do, it can, for example, calculate that the action *Right* will cause it to be in one of the states $\{2, 4, 6, 8\}$, and the action sequence [*Right*,*Suck*] will always end up in one of the states $\{4, 8\}$. Finally, the sequence [*Right*,*Suck*,*Left*,*Suck*] is guaranteed to reach the goal state 7 no matter what the start state. We say that the agent can **coerce** the world into state 7, even when it doesn't know where it started. To summarize: when the world is not fully observable, the agent must reason about *sets* of states that it might get to, rather than single states. We call each such set of states a **belief state**, representing the agent's current belief about the possible physical states it might be in. (In a fully observable environment, each belief state contains one physical state.)

COERCION

BELIEF STATE

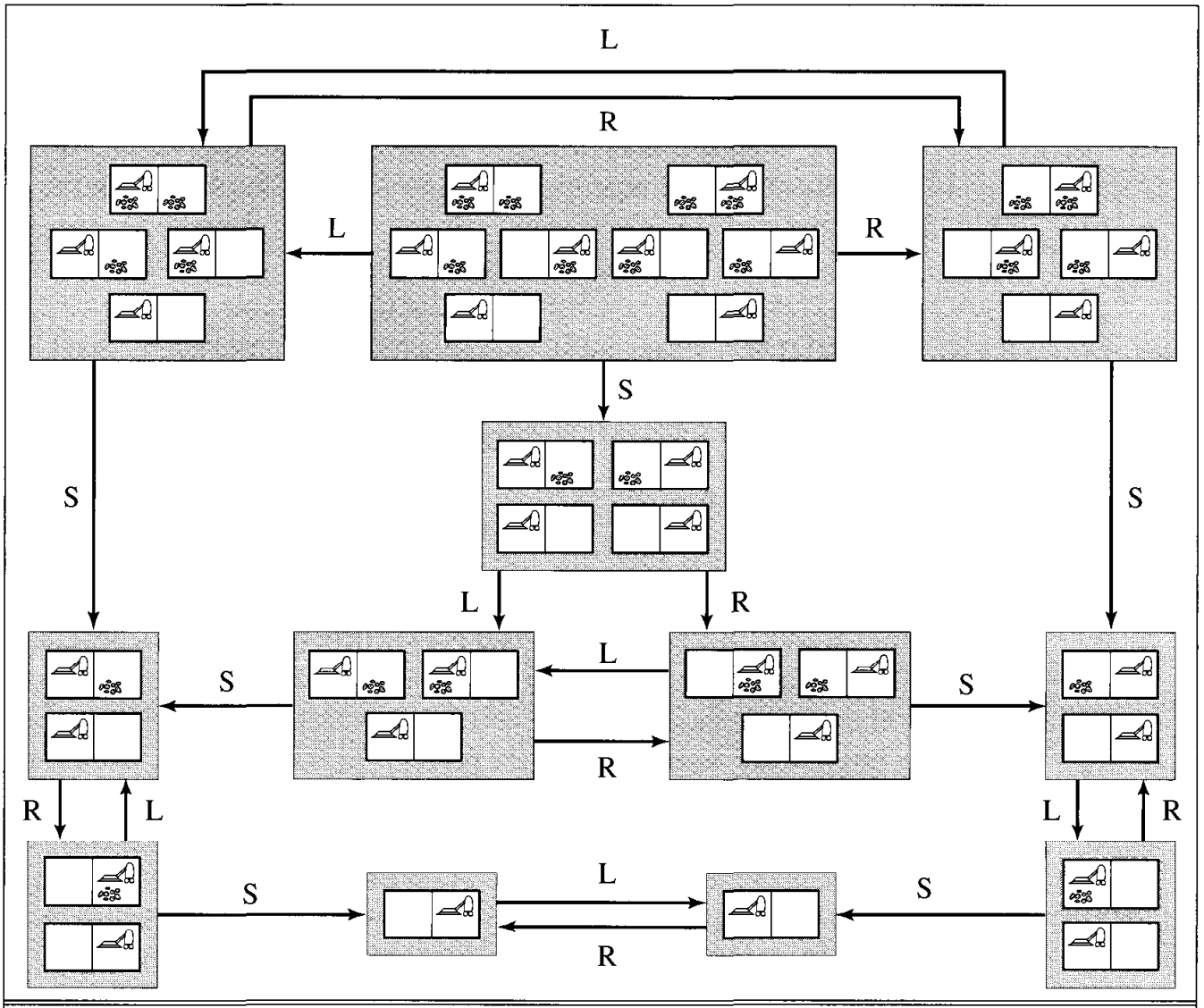


Figure 3.21 The reachable portion of the belief state space for the deterministic, sensorless vacuum world. Each shaded box corresponds to a single belief state. At any given point, the agent is in a particular belief state but does not know which physical state it is in. The initial belief state (complete ignorance) is the top center box. Actions are represented by labeled arcs. Self-loops are omitted for clarity.

To solve sensorless problems, we search in the space of belief states rather than physical states. The initial state is a belief state, and each action maps from a belief state to another belief state. An action is applied to a belief state by unioning the results of applying the action to each physical state in the belief state. A path now connects several belief states, and a solution is now a path that leads to a belief state, *all of whose members* are goal states. Figure 3.21 shows the reachable belief-state space for the deterministic, sensorless vacuum world. There are only 12 reachable belief states, but the entire belief state space contains every possible set of physical states, i.e., $2^8 = 256$ belief states. In general, if the physical state space has S states, the belief state space has 2^S belief states.

Our discussion of sensorless problems so far has assumed deterministic actions, but the analysis is essentially unchanged if the environment is nondeterministic—that is, if actions may have several possible outcomes. The reason is that, in the absence of sensors, the agent

has no way to tell which outcome actually occurred, so the various possible outcomes are just additional physical states in the successor belief state. For example, suppose the environment obeys Murphy's Law: the so-called *Suck* action *sometimes* deposits dirt on the carpet *but only if there is no dirt there already*.[6] Then, if *Suck* is applied in physical state 4 (see Figure 3.20), there are two possible outcomes: states 2 and 4. Applied to the initial belief state, $\{1, 2, 3, 4, 5, 6, 7, 8\}$, *Suck* now leads to the belief state that is the union of the outcome sets for the eight physical states. Calculating this, we find that the new belief state is $\{1, 2, 3, 4, 5, 6, 7, 8\}$. So, for a sensorless agent in the Murphy's Law world, the *Suck* action leaves the belief state unchanged! In fact, the problem is unsolvable. (See Exercise 3.18.) Intuitively, the reason is that the agent cannot tell whether the current square is dirty and hence cannot tell whether the *Suck* action will clean it up or create more dirt.

Contingency problems

CONTINGENCY PROBLEM

When the environment is such that the agent can obtain new information from its sensors after acting, the agent faces a **contingency problem**. The solution to a contingency problem often takes the form of a *tree*, where each branch may be selected depending on the percepts received up to that point in the tree. For example, suppose that the agent is in the Murphy's Law world and that it has a position sensor and a local dirt sensor, but no sensor capable of detecting dirt in other squares. Thus, the percept $[L, Dirty]$ means that the agent is in one of the states $\{1, 3\}$. The agent might formulate the action sequence [*Suck*, *Right*, *Suck*]. Sucking would change the state to one of $\{5, 7\}$, and moving right would then change the state to one of $\{6, 8\}$. Executing the final *Suck* action in state 6 takes us to state 8, a goal, but executing it in state 8 might take us back to state 6 (by Murphy's Law), in which case the plan fails.

By examining the belief-state space for this version of the problem, it can easily be determined that no fixed action sequence guarantees a solution to this problem. There is, however, a solution if we don't insist on a *fixed* action sequence:

[*Suck*, *Right*, ***if*** *[R,Dirty]* ***then*** *Suck*] .

This extends the space of solutions to include the possibility of selecting actions based on contingencies arising during execution. Many problems in the real, physical world are contingency problems, because exact prediction is impossible. For this reason, many people keep their eyes open while walking around or driving.

Contingency problems *sometimes* allow purely sequential solutions. For example, consider a *fully observable* Murphy's Law world. Contingencies arise if the agent performs a *Suck* action in a clean square, because dirt might or might not be deposited in the square. As long as the agent never does this, no contingencies arise and there is a sequential solution from every initial state (Exercise 3.18).

The algorithms for contingency problems are more complex than the standard search algorithms in this chapter; they are covered in Chapter 12. Contingency problems also lend themselves to a somewhat different agent design, in which the agent can act *before* it has found a guaranteed plan. This is useful because rather than considering in advance every

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

possible contingency that *might* arise during execution, it is often better to start acting and see which contingencies *do* arise. The agent can then continue to solve the problem, taking into account the additional information. This type of **interleaving** of search and execution is also useful for exploration problems (see Section 4.5) and for game playing (see Chapter 6).

INTERLEAVING

3.7 SUMMARY

This chapter has introduced methods that an agent can use to select actions in environments that are deterministic, observable, static, and completely known. In such cases, the agent can construct sequences of actions that achieve its goals; this process is called **search**.

- Before an agent can start searching for solutions, it must formulate a **goal** and then use the goal to formulate a **problem**.
- A problem consists of four parts: the **initial state**, a set of **actions**, a **goal test** function, and a **path cost** function. The environment of the problem is represented by a **state space**. A **path** through the state space from the initial state to a goal state is a **solution**.
- A single, general TREE-SEARCH algorithm can be used to solve any problem; specific variants of the algorithm embody different strategies.
- Search algorithms are judged on the basis of **completeness**, **optimality**, **time complexity**, and **space complexity**. Complexity depends on b, the branching factor in the state space, and d, the depth of the shallowest solution.
- **Breadth-first search** selects the shallowest unexpanded node in the search tree for expansion. It is complete, optimal for unit step costs, and has time and space complexity of $O(b^d)$. The space complexity makes it impractical in most cases. **Uniform-cost search** is similar to breadth-first search but expands the node with lowest path cost, $g(n)$. It is complete and optimal if the cost of each step exceeds some positive bound ϵ.
- **Depth-first search** selects the deepest unexpanded node in the search tree for expansion. It is neither complete nor optimal, and has time complexity of $O(b^m)$ and space complexity of $O(bm)$, where m is the maximum depth of any path in the state space.
- **Depth-limited search** imposes a fixed depth limit on a depth-first search.
- **Iterative deepening search** calls depth-limited search with increasing limits until a goal is found. It is complete, optimal for unit step costs, and has time complexity of $O(b^d)$ and space complexity of $O(bd)$.
- **Bidirectional search** can enormously reduce time complexity, but it is not always applicable and may require too much space.
- When the state space is a graph rather than a tree, it can pay off to check for repeated states in the search tree. The GRAPH-SEARCH algorithm eliminates all duplicate states.
- When the environment is partially observable, the agent can apply search algorithms in the space of **belief states**, or sets of possible states that the agent might be in. In some cases, a single solution sequence can be constructed; in other cases, the agent needs a **contingency plan** to handle unknown circumstances that may arise.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Most of the state-space search problems analyzed in this chapter have a long history in the literature and are less trivial than they might seem. The missionaries and cannibals problem used in Exercise 3.9 was analyzed in detail by Amarel (1968). It had been considered earlier in AI by Simon and Newell (1961), and in operations research by Bellman and Dreyfus (1962). Studies such as these and Newell and Simon's work on the Logic Theorist (1957) and GPS (1961) led to the establishment of search algorithms as the primary weapons in the armory of 1960s AI researchers and to the establishment of problem solving as the canonical AI task. Unfortunately, very little work was done on the automation of the problem formulation step. A more recent treatment of problem representation and abstraction, including AI programs that themselves perform these tasks (in part), is in Knoblock (1990).

The 8-puzzle is a smaller cousin of the 15-puzzle, which was invented by the famous American game designer Sam Loyd (1959) in the 1870s. The 15-puzzle quickly achieved immense popularity in the United States, comparable to the more recent sensation caused by Rubik's Cube. It also quickly attracted the attention of 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. But this would not have weighed with the editors to induce them to insert articles upon such a subject in the *American Journal of Mathematics*, but for the fact that ..." (there follows a summary of the mathematical interest of the 15-puzzle). An exhaustive analysis of the 8-puzzle was carried out with computer aid by Schofield (1967). Ratner and Warmuth (1986) showed that the general $n \times n$ version of the 15-puzzle belongs to the class of NP-complete problems.

The 8-queens problem was first published anonymously in the German chess magazine *Schach* in 1848; it was later attributed to one Max Bezzel. It was republished in 1850 and at that time drew the attention of the eminent mathematician Carl Friedrich Gauss, who attempted to enumerate all possible solutions, but found only 72. Nauck published all 92 solutions later in 1850. Netto (1901) generalized the problem to n queens, and Abramson and Yung (1989) found an $O(n)$ algorithm.

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 (personal communication) has shown that airline ticket pricing and restrictions have become so convoluted that the problem of selecting an optimal flight is formally *undecidable*. The traveling-salesperson problem is a standard combinatorial problem in theoretical computer science (Lawler, 1985; Lawler *et al.*, 1992). Karp (1972) proved the TSP 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 Shahookar and Mazumder (1991), and many layout optimization papers appear in VLSI journals. Robotic navigation and assembly problems are discussed in Chapter 25.

Uninformed search algorithms for problem solving are a central topic of classical computer science (Horowitz and Sahni, 1978) and operations research (Dreyfus, 1969); Deo and Pang (1984) and Gallo and Pallottino (1988) give more recent surveys. Breadth-first search was formulated for solving mazes by Moore (1959). The method of **dynamic programming** (Bellman and Dreyfus, 1962), which systematically records solutions for all subproblems of increasing lengths, can be seen as a form of breadth-first search on graphs. The two-point shortest-path algorithm of Dijkstra (1959) is the origin of uniform-cost search.

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, but the application to shortest path graph search is due to Korf (1985a). Bidirectional search, which was introduced by Pohl (1969, 1971), can also be very effective in some cases.

Partially observable and nondeterministic environments have not been studied in great depth within the problem-solving approach. Some efficiency issues in belief-state search have been investigated by Genesereth and Nourbakhsh (1993). Koenig and Simmons (1998) studied robot navigation from an unknown initial position, and Erdmann and Mason (1988) studied the problem of robotic manipulation without sensors, using a continuous form of belief-state search. Contingency search has been studied within the planning subfield. (See Chapter 12.) For the most part, planning and acting with uncertain information have been handled using the tools of probability and decision theory (see Chapter 17).

The textbooks by Nilsson (1971, 1980) are good general sources of information about classical search algorithms. A comprehensive and more up-to-date survey can be found in Korf (1988). Papers about new search algorithms—which, remarkably, continue to be discovered—appear in journals such as *Artificial Intelligence*.

EXERCISES

3.1 Define in your own words the following terms: state, state space, search tree, search node, goal, action, successor function, and branching factor.

3.2 Explain why problem formulation must follow goal formulation.

3.3 Suppose that LEGAL-ACTIONS(s) denotes the set of actions that are legal in state s, and RESULT(a, s) denotes the state that results from performing a legal action a in state s. Define SUCCESSOR-FN in terms of LEGAL-ACTIONS and RESULT, and *vice versa*.

3.4 Show that the 8-puzzle states are divided into two disjoint sets, such that no state in one set can be transformed into a state in the other set by any number of moves. (*Hint:* See Berlekamp *et al.* (1982).) Devise a procedure that will tell you which class a given state is in, and explain why this is a good thing to have for generating random states.

3.5 Consider the n-queens problem using the "efficient" incremental formulation given on page 67. Explain why the state space size is at least $\sqrt[3]{n!}$ and estimate the largest n for which exhaustive exploration is feasible. (*Hint*: Derive a lower bound on the branching factor by considering the maximum number of squares that a queen can attack in any column.)

3.6 Does a finite state space always lead to a finite search tree? How about a finite state space that is a tree? Can you be more precise about what types of state spaces always lead to finite search trees? (Adapted from Bender, 1996.)

3.7 Give the initial state, goal test, successor function, and cost function for each of the following. Choose a formulation that is precise enough to be implemented.

a. You have to color a planar map using only four colors, in such a way that no two adjacent regions have the same color.

b. A 3-foot-tall monkey is in a room where some bananas are suspended from the 8-foot ceiling. He would like to get the bananas. The room contains two stackable, movable, climbable 3-foot-high crates.

c. You have a program that outputs the message "illegal input record" when fed a certain file of input records. You know that processing of each record is independent of the other records. You want to discover what record is illegal.

d. You have three jugs, measuring 12 gallons, 8 gallons, and 3 gallons, and a water faucet. You can fill the jugs up or empty them out from one to another or onto the ground. You need to measure out exactly one gallon.

3.8 Consider a state space where the start state is number 1 and the successor function for state n returns two states, numbers $2n$ and $2n + 1$.

a. Draw the portion of the state space for states 1 to 15.

b. Suppose the goal state is 11. List the order in which nodes will be visited for breadth-first search, depth-limited search with limit 3, and iterative deepening search.

c. Would bidirectional search be appropriate for this problem? If so, describe in detail how it would work.

d. What is the branching factor in each direction of the bidirectional search?

e. Does the answer to (c) suggest a reformulation of the problem that would allow you to solve the problem of getting from state 1 to a given goal state with almost no search?

3.9 The **missionaries and cannibals** problem is usually stated as follows. Three missionaries and three cannibals are on one side of a river, along with a boat that can hold one or two people. Find a way to get everyone to the other side, without ever leaving a group of missionaries in one place outnumbered by the cannibals in that place. This problem is famous in AI because it was the subject of the first paper that approached problem formulation from an analytical viewpoint (Amarel, 1968).

a. Formulate the problem precisely, making only those distinctions necessary to ensure a valid solution. Draw a diagram of the complete state space.

b. Implement and solve the problem optimally using an appropriate search algorithm. Is it a good idea to check for repeated states?

c. Why do you think people have a hard time solving this puzzle, given that the state space is so simple?

3.10 Implement two versions of the successor function for the 8-puzzle: one that generates all the successors at once by copying and editing the 8-puzzle data structure, and one that generates one new successor each time it is called and works by modifying the parent state directly (and undoing the modifications as needed). Write versions of iterative deepening depth-first search that use these functions and compare their performance.

3.11 On page 79, we mentioned **iterative lengthening search**, an iterative analog of uniform cost search. The idea is to use increasing limits on path cost. If a node is generated whose path cost exceeds the current limit, it is immediately discarded. For each new iteration, the limit is set to the lowest path cost of any node discarded in the previous iteration.

a. Show that this algorithm is optimal for general path costs.

b. Consider a uniform tree with branching factor b, solution depth d, and unit step costs. How many iterations will iterative lengthening require?

c. Now consider step costs drawn from the continuous range $[0, 1]$ with a minimum positive cost ϵ. How many iterations are required in the worst case?

d. Implement the algorithm and apply it to instances of the 8-puzzle and traveling salesperson problems. Compare the algorithm's performance to that of uniform-cost search, and comment on your results.

3.12 Prove that uniform-cost search and breadth-first search with constant step costs are optimal when used with the GRAPH-SEARCH algorithm. Show a state space with constant step costs in which GRAPH-SEARCH using iterative deepening finds a suboptimal solution.

3.13 Describe a state space in which iterative deepening search performs much worse than depth-first search (for example, $O(n^2)$ vs. $O(n)$).

3.14 Write a program that will take as input two Web page URLs and find a path of links from one to the other. What is an appropriate search strategy? Is bidirectional search a good idea? Could a search engine be used to implement a predecessor function?

3.15 Consider the problem of finding the shortest path between two points on a plane that has convex polygonal obstacles as shown in Figure 3.22. This is an idealization of the problem that a robot has to solve to navigate its way around a crowded environment.

a. Suppose the state space consists of all positions (x, y) in the plane. How many states are there? How many paths are there to the goal?

b. Explain briefly why the shortest path from one polygon vertex to any other in the scene must consist of straight-line segments joining some of the vertices of the polygons. Define a good state space now. How large is this state space?

c. Define the necessary functions to implement the search problem, including a successor function that takes a vertex as input and returns the set of vertices that can be reached in a straight line from the given vertex. (Do not forget the neighbors on the same polygon.) Use the straight-line distance for the heuristic function.

d. Apply one or more of the algorithms in this chapter to solve a range of problems in the domain, and comment on their performance.

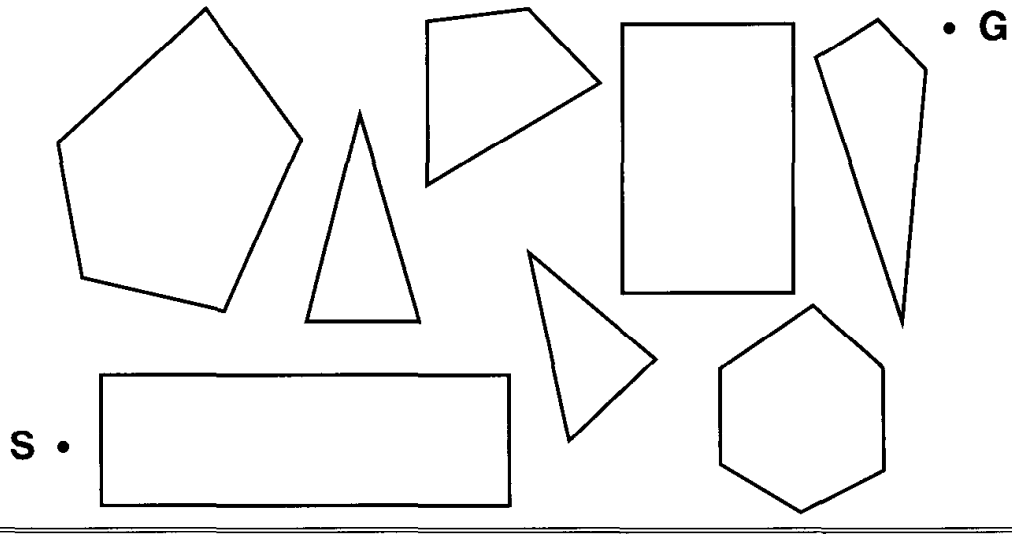


Figure 3.22 A scene with polygonal obstacles.

3.16 We can turn the navigation problem in Exercise 3.15 into an environment as follows:

- The percept will be a list of the positions, *relative to the agent*, of the visible vertices. The percept does *not* include the position of the robot! The robot must learn its own position from the map; for now, you can assume that each location has a different "view."
- Each action will be a vector describing a straight-line path to follow. If the path is unobstructed, the action succeeds; otherwise, the robot stops at the point where its path first intersects an obstacle. If the agent returns a zero motion vector and is at the goal (which is fixed and known), then the environment should teleport the agent to a *random location* (not inside an obstacle).
- The performance measure charges the agent 1 point for each unit of distance traversed and awards 1000 points each time the goal is reached.

a. Implement this environment and a problem-solving agent for it. The agent will need to formulate a new problem after each teleportation, which will involve discovering its current location.

b. Document your agent's performance (by having the agent generate suitable commentary as it moves around) and report its performance over 100 episodes.

c. Modify the environment so that 30% of the time the agent ends up at an unintended destination (chosen randomly from the other visible vertices if any, otherwise no move at all). This is a crude model of the motion errors of a real robot. Modify the agent so that when such an error is detected, it finds out where it is and then constructs a plan to get back to where it was and resume the old plan. Remember that sometimes getting back to where it was might also fail! Show an example of the agent successfully overcoming two successive motion errors and still reaching the goal.

d. Now try two different recovery schemes after an error: (1) Head for the closest vertex on the original route; and (2) replan a route to the goal from the new location. Compare the performance of the three recovery schemes. Would the inclusion of search costs affect the comparison?

e. Now suppose that there are locations from which the view is identical. (For example, suppose the world is a grid with square obstacles.) What kind of problem does the agent now face? What do solutions look like?

3.17 On page 62, we said that we would not consider problems with negative path costs. In this exercise, we explore this in more depth.

a. Suppose that actions can have arbitrarily large negative costs; explain why this possibility would force any optimal algorithm to explore the entire state space.

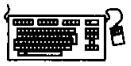
b. Does it help if we insist that step costs must be greater than or equal to some negative constant c? Consider both trees and graphs.

c. Suppose that there is a set of operators that form a loop, so that executing the set in some order results in no net change to the state. If all of these operators have negative cost, what does this imply about the optimal behavior for an agent in such an environment?

d. One can easily imagine operators with high negative cost, even in domains such as route finding. For example, some stretches of road might have such beautiful scenery as to far outweigh the normal costs in terms of time and fuel. Explain, in precise terms, within the context of state-space search, why humans do not drive round scenic loops indefinitely, and explain how to define the state space and operators for route finding so that artificial agents can also avoid looping.

e. Can you think of a real domain in which step costs are such as to cause looping?

3.18 Consider the sensorless, two-location vacuum world under Murphy's Law. Draw the belief state space reachable from the initial belief state $\{1, 2, 3, 4, 5, 6, 7, 8\}$, and explain why the problem is unsolvable. Show also that if the world is fully observable then there is a solution sequence for each possible initial state.



3.19 Consider the vacuum-world problem defined in Figure 2.2.

a. Which of the algorithms defined in this chapter would be appropriate for this problem? Should the algorithm check for repeated states?

b. Apply your chosen algorithm to compute an optimal sequence of actions for a 3×3 world whose initial state has dirt in the three top squares and the agent in the center.

c. Construct a search agent for the vacuum world, and evaluate its performance in a set of 3×3 worlds with probability 0.2 of dirt in each square. Include the search cost as well as path cost in the performance measure, using a reasonable exchange rate.

d. Compare your best search agent with a simple randomized reflex agent that sucks if there is dirt and otherwise moves randomly.

e. Consider what would happen if the world were enlarged to $n \times n$. How does the performance of the search agent and of the reflex agent vary with n?

4 INFORMED SEARCH AND EXPLORATION

In which we see how information about the state space can prevent algorithms from blundering about in the dark.

Chapter 3 showed that uninformed search strategies can find solutions to problems by systematically generating new states and testing them against the goal. Unfortunately, these strategies are incredibly inefficient in most cases. This chapter shows how an informed search strategy—one that uses problem-specific knowledge—can find solutions more efficiently. Section 4.1 describes informed versions of the algorithms in Chapter 3, and Section 4.2 explains how the necessary problem-specific information can be obtained. Sections 4.3 and 4.4 cover algorithms that perform purely **local search** in the state space, evaluating and modifying one or more current states rather than systematically exploring paths from an initial state. These algorithms are suitable for problems in which the path cost is irrelevant and all that matters is the solution state itself. The family of local-search algorithms includes methods inspired by statistical physics (**simulated annealing**) and evolutionary biology (**genetic algorithms**). Finally, Section 4.5 investigates **online search**, in which the agent is faced with a state space that is completely unknown.

4.1 INFORMED (HEURISTIC) SEARCH STRATEGIES

INFORMED SEARCH

This section shows how an **informed search** strategy—one that uses problem-specific knowledge beyond the definition of the problem itself—can find solutions more efficiently than an uninformed strategy.

BEST-FIRST SEARCH

The general approach we will consider is called **best-first search**. Best-first search is an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an **evaluation function**, $f(n)$. Traditionally, the node with the *lowest* evaluation is selected for expansion, because the evaluation measures distance to the goal. Best-first search can be implemented within our general search framework via a priority queue, a data structure that will maintain the fringe in ascending order of f-values.

EVALUATION FUNCTION

The name "best-first search" is a venerable but inaccurate one. After all, if we could *really* expand the best node first, it would not be a search at all; it would be a straight march to

the goal. All we can do is choose the node that *appears* to be best according to the evaluation function. If the evaluation function is exactly accurate, then this will indeed be the best node; in reality, the evaluation function will sometimes be off, and can lead the search astray. Nevertheless, we will stick with the name "best-first search," because "seemingly-best-first search" is a little awkward.

There is a whole family of BEST-FIRST-SEARCH algorithms with different evaluation functions.[1] A key component of these algorithms is a **heuristic function**,[2] denoted $h(n)$:

HEURISTIC FUNCTION

$h(n) =$ estimated cost of the cheapest path from node n to a goal node.

For example, in Romania, one might estimate the cost of the cheapest path from Arad to Bucharest via the straight-line distance from Arad to Bucharest.

Heuristic functions are the most common form in which additional knowledge of the problem is imparted to the search algorithm. We will study heuristics in more depth in Section 4.2. For now, we will consider them to be arbitrary problem-specific functions, with one constraint: if n is a goal node, then $h(n) = 0$. The remainder of this section covers two ways to use heuristic information to guide search.

Greedy best-first search

GREEDY BEST-FIRST SEARCH

Greedy best-first search[3] tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function: $f(n) = h(n)$.

STRAIGHT-LINE DISTANCE

Let us see how this works for route-finding problems in Romania, using the **straight-line distance** heuristic, which we will call h_{SLD}. If the goal is Bucharest, we will need to know the straight-line distances to Bucharest, which are shown in Figure 4.1. For example, $h_{SLD}(In(Arad)) = 366$. Notice that the values of h_{SLD} cannot be computed from the problem description itself. Moreover, it takes a certain amount of experience to know that h_{SLD} is correlated with actual road distances and is, therefore, a useful heuristic.

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

Figure 4.1 Values of h_{SLD}—straight-line distances to Bucharest.

[1] Exercise 4.3 asks you to show that this family includes several familiar uninformed algorithms.

[2] A heuristic function $h(n)$ takes a *node* as input, but it depends only on the *state* at that node.

[3] Our first edition called this **greedy search**; other authors have called it **best-first search**. Our more general usage of the latter term follows Pearl (1984).

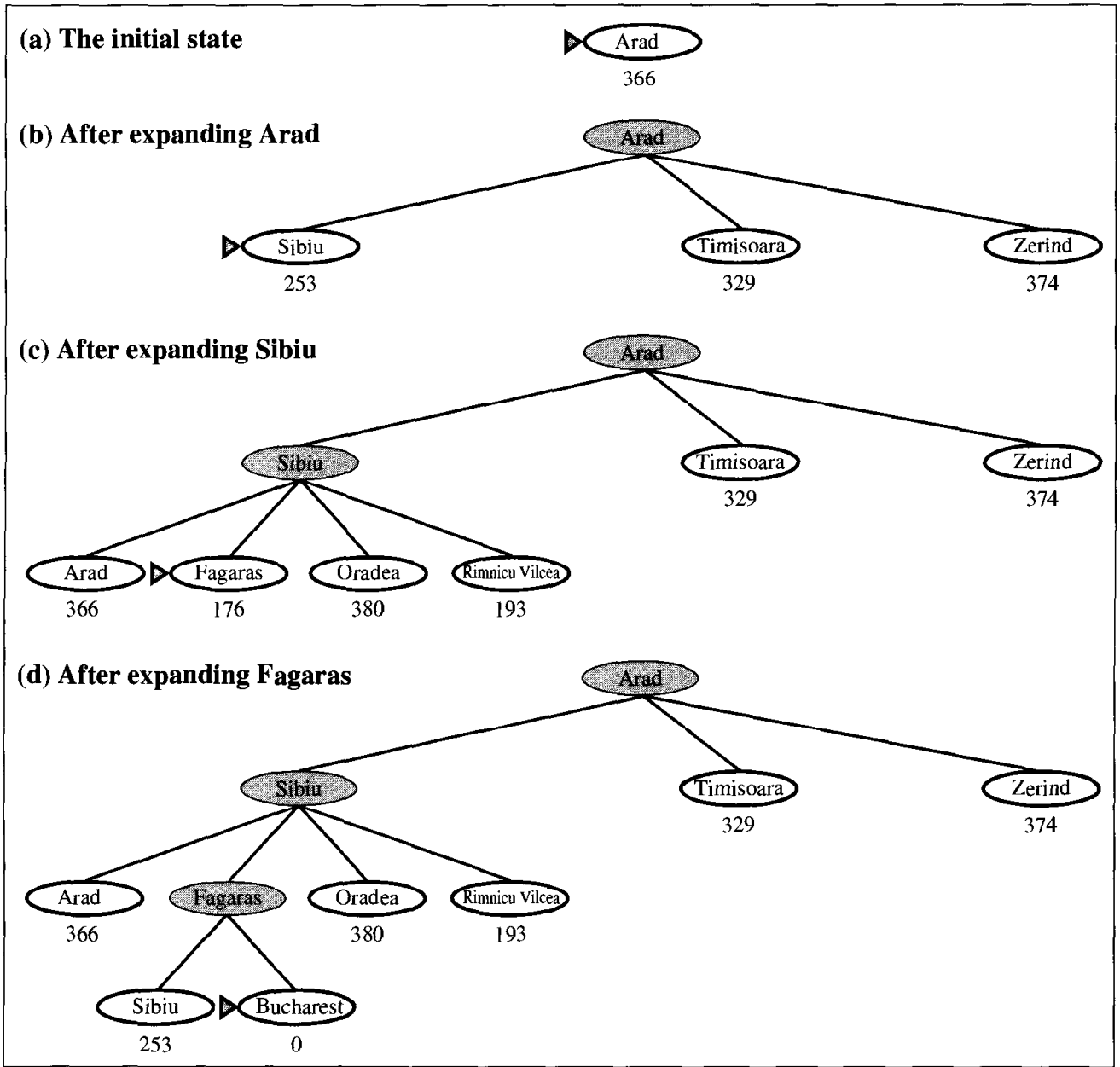


Figure 4.2 Stages in a greedy best-first search for Bucharest using the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

Figure 4.2 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 it is closer to Bucharest than either Zerind or Timisoara. The next node to be expanded will be Fagaras, because it is closest. 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; hence, its search cost is minimal. It is not optimal, however: the path via Sibiu and Fagaras to Bucharest is 32 kilometers longer than the path through Rimnicu Vilcea and Pitesti. This shows why the algorithm is called "greedy"—at each step it tries to get as close to the goal as it can.

Minimizing $h(n)$ is susceptible to false starts. Consider the problem of getting from Iasi to Fagaras. The heuristic suggests that Neamt be expanded first, because it is closest

to Fagaras, but it is a dead end. The solution is to go first to Vaslui—a step that is actually farther from the goal according to the heuristic—and then to continue to Urziceni, Bucharest, and Fagaras. In this case, then, the heuristic causes unnecessary nodes to be expanded. Furthermore, if we are not careful to detect repeated states, the solution will never be found—the search will oscillate between Neamt and Iasi.

Greedy best-first search resembles depth-first search in the way it prefers to follow a single path all the way to the goal, but will back up when it hits a dead end. It suffers from the same defects as depth-first search—it is not optimal, and it is incomplete (because it can start down an infinite path and never return to try other possibilities). The worst-case time and space complexity is $O(b^m)$, where m is the maximum depth of the search space. With a good heuristic function, however, the complexity can be reduced substantially. The amount of the reduction depends on the particular problem and on the quality of the heuristic.

A* search: Minimizing the total estimated solution cost

A* SEARCH

The most widely-known form of best-first search is called **A* search** (pronounced "A-star search"). It evaluates nodes by combining $g(n)$, the cost to reach the node, and $h(n)$, the cost to get from the node to the goal:

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

Since $g(n)$ gives the path cost from the start node to node n, and $h(n)$ is the estimated cost of the cheapest path from n to the goal, we have

$$f(n) = \text{ estimated cost of the cheapest solution through } n .$$

Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of $g(n) + h(n)$. It turns out that this strategy is more than just reasonable: provided that the heuristic function $h(n)$ satisfies certain conditions, A* search is both complete and optimal.

The optimality of A* is straightforward to analyze if it is used with TREE-SEARCH.

ADMISSIBLE HEURISTIC

In this case, A* is optimal if $h(n)$ is an **admissible heuristic**—that is, provided that $h(n)$ *never overestimates* the cost to reach the goal. Admissible heuristics are by nature optimistic, because they think the cost of solving the problem is less than it actually is. Since $g(n)$ is the exact cost to reach n, we have as immediate consequence that $f(n)$ never overestimates the true cost of a solution through n.

An obvious example of an admissible heuristic is the straight-line distance h_{SLD} that we used in getting to Bucharest. Straight-line distance is admissible because the shortest path between any two points is a straight line, so the straight line cannot be an overestimate. In Figure 4.3, we show the progress of an A* tree search for Bucharest. The values of g are computed from the step costs in Figure 3.2, and the values of h_{SLD} are given in Figure 4.1. Notice in particular that Bucharest first appears on the fringe at step (e), but it is not selected for expansion because its f-cost (450) is higher than that of Pitesti (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. From this example, we can extract a general proof that *A* using* TREE-SEARCH *is optimal if* $h(n)$ *is admissible.* Suppose a

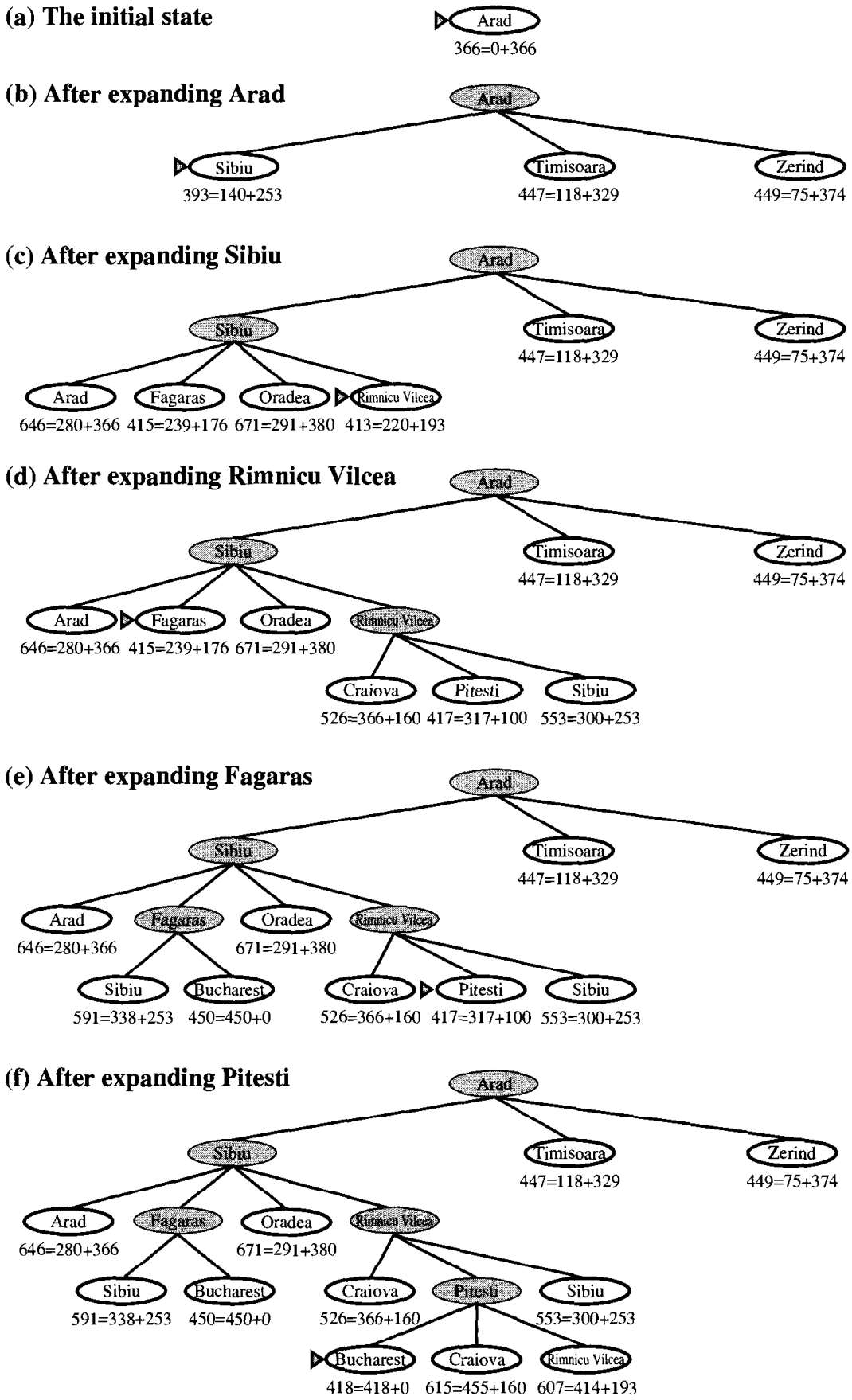


Figure 4.3 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 4.1.

suboptimal goal node G_2 appears on the fringe, and let the cost of the optimal solution be C^*. Then, because G_2 is suboptimal and because $h(G_2) = 0$ (true for any goal node), we know

$$f(G_2) = g(G_2) + h(G_2) = g(G_2) > C^* .$$

Now consider a fringe node n that is on an optimal solution path—for example, Pitesti in the example of the preceding paragraph. (There must always be such a node if a solution exists.) If $h(n)$ does not overestimate the cost of completing the solution path, then we know that

$$f(n) = g(n) + h(n) \leq C^* .$$

Now we have shown that $f(n) \leq C^* < f(G_2)$, so G_2 will not be expanded and A* must return an optimal solution.

If we use the GRAPH-SEARCH algorithm of Figure 3.19 instead of TREE-SEARCH, then this proof breaks down. Suboptimal solutions can be returned because GRAPH-SEARCH can discard the optimal path to a repeated state if it is not the first one generated. (See Exercise 4.4.) There are two ways to fix this problem. The first solution is to extend GRAPH-SEARCH so that it discards the more expensive of any two paths found to the same node. (See the discussion in Section 3.5.) The extra bookkeeping is messy, but it does guarantee optimality. The second solution is to ensure that the optimal path to any repeated state is always the first one followed—as is the case with uniform-cost search. This property holds if we impose an extra requirement on $h(n)$, namely the requirement of **consistency** (also called **monotonicity**). A heuristic $h(n)$ is consistent if, for every node n and every successor n' of n generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n':

CONSISTENCY
MONOTONICITY

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

TRIANGLE INEQUALITY

This is a form of the general **triangle inequality**, which stipulates that each side of a triangle cannot be longer than the sum of the other two sides. Here, the triangle is formed by n, n', and the goal closest to n. It is fairly easy to show (Exercise 4.7) that every consistent heuristic is also admissible. The most important consequence of consistency is the following: *A* using* GRAPH-SEARCH *is optimal if $h(n)$ is consistent.*

Although consistency is a stricter requirement than admissibility, one has to work quite hard to concoct heuristics that are admissible but not consistent. All the admissible heuristics we discuss in this chapter are also consistent. Consider, for example, h_{SLD}. We know that the general triangle inequality is satisfied when each side is measured by the straight-line distance, and that the straight-line distance between n and n' is no greater than $c(n, a, n')$. Hence, h_{SLD} is a consistent heuristic.

Another important consequence of consistency is the following: *If $h(n)$ is consistent, then the values of $f(n)$ along any path are nondecreasing.* The proof follows directly from the definition of consistency. Suppose n' is a successor of n; then $g(n') = g(n) + c(n, a, n')$ for some a, and we have

$$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n) .$$

It follows that the sequence of nodes expanded by A* using GRAPH-SEARCH is in nondecreasing order of $f(n)$. Hence, the first goal node selected for expansion must be an optimal solution, since all later nodes will be at least as expensive.

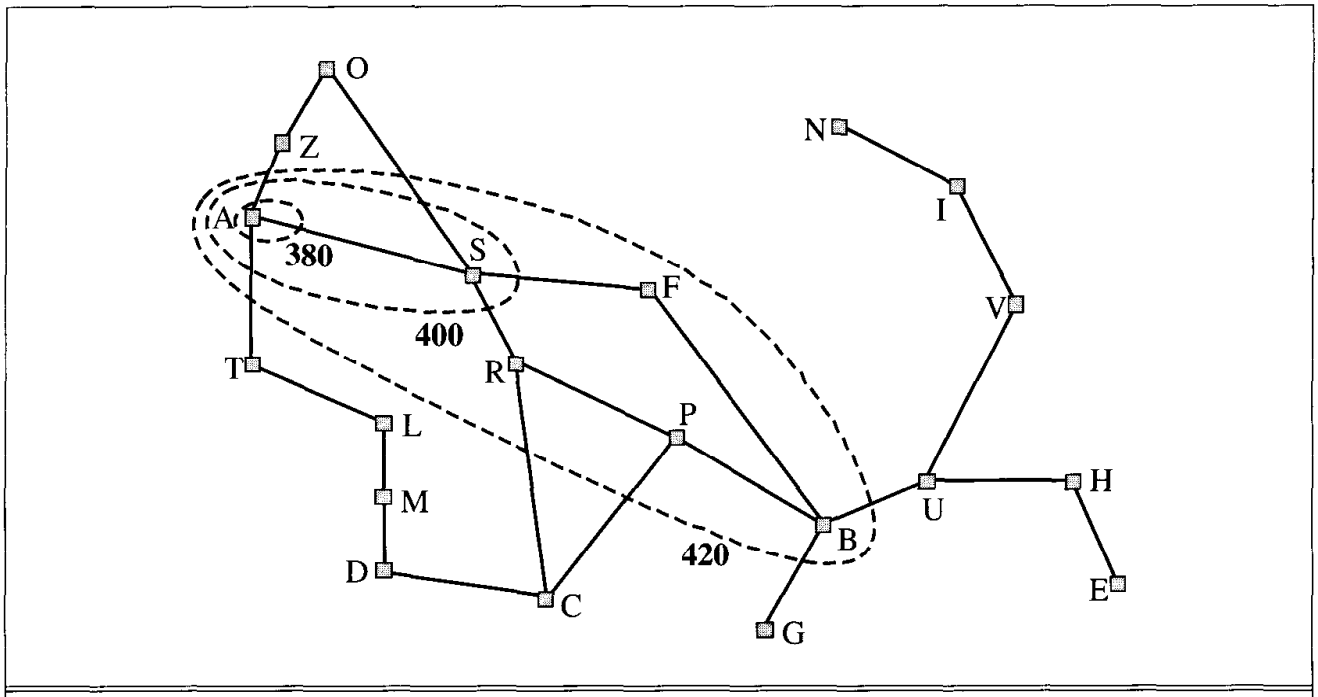


Figure 4.4 Map of Romania showing contours at $f = 380$, $f = 400$ and $f = 420$, with Arad as the start state. Nodes inside a given contour have f-costs less than or equal to the contour value.

CONTOURS

The fact that f-costs are nondecreasing along any path also means that we can draw **contours** in the state space, just like the contours in a topographic map. Figure 4.4 shows an example. Inside the contour labeled 400, all nodes have $f(n)$ less than or equal to 400, and so on. Then, because A* expands the fringe 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.

With uniform-cost search (A* search using $h(n) = 0$), the bands will be "circular" around the start state. With more accurate heuristics, the bands will stretch toward the goal state and become more narrowly focused around the optimal path. If C^* is the cost of the optimal solution path, then we can say the following:

- A* expands all nodes with $f(n) < C^*$.
- A* might then expand some of the nodes right on the "goal contour" (where $f(n) = C*$) before selecting a goal node.

Intuitively, it is obvious that the first solution found must be an optimal one, because goal nodes in all subsequent contours will have higher f-cost, and thus higher g-cost (because all goal nodes have $h(n) = 0$). Intuitively, it is also obvious that A* search is complete. As we add bands of increasing f, we must eventually reach a band where f is equal to the cost of the path to a goal state.[4]

PRUNING

Notice that A* expands no nodes with $f(n) > C^*$—for example, Timisoara is not expanded in Figure 4.3 even though it is a child of the root. We say that the subtree below Timisoara is **pruned**; because h_{SLD} is admissible, the algorithm can safely ignore this subtree

[4] Completeness requires that there be only finitely many nodes with cost less than or equal to C^*, a condition that is true if all step costs exceed some finite ϵ and if b is finite.

while still guaranteeing optimality. The concept of pruning—eliminating possibilities from consideration without having to examine them—is important for many areas of AI.

OPTIMALLY EFFICIENT

One final observation is that among optimal algorithms of this type—algorithms that extend search paths from the root—A* is **optimally efficient** for any given heuristic function. That is, no other optimal algorithm is guaranteed to expand fewer nodes than A* (except possibly through tie-breaking among nodes with $f(n) = C^*$). This is because any algorithm that *does not* expand all nodes with $f(n) < C^*$ runs the risk of missing the optimal solution.

That A* search is complete, 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 most problems, the number of nodes within the goal contour search space is still exponential in the length of the solution. Although the proof of the result is beyond the scope of this book, it has been shown that exponential growth will occur unless the error in the heuristic function grows no faster than the logarithm of the actual path cost. In mathematical notation, the condition for subexponential growth is that

$$|h(n) - h^*(n)| \leq O(\log h^*(n)) ,$$

where $h^*(n)$ is the *true* cost of getting from n to the goal. For almost all heuristics in practical use, the error is at least proportional to the path cost, and the resulting exponential growth eventually overtakes any computer. For this reason, it is often impractical to insist on finding an optimal solution. One can use variants of A* that find suboptimal solutions quickly, or one can sometimes design heuristics that are more accurate, but not strictly admissible. In any case, the use of a good heuristic still provides enormous savings compared to the use of an uninformed search. In Section 4.2, we will look at the question of designing good heuristics.

Computation time is not, however, A*'s main drawback. Because it keeps all generated nodes in memory (as do all GRAPH-SEARCH algorithms), A* usually runs out of space long before it runs out of time. For this reason, A* is not practical for many large-scale problems. Recently developed algorithms have overcome the space problem without sacrificing optimality or completeness, at a small cost in execution time. These are discussed next.

Memory-bounded heuristic search

The simplest way to reduce memory requirements for A* is to adapt the idea of iterative deepening to the heuristic search context, resulting in the iterative-deepening A* (IDA*) algorithm. The main difference between IDA* and standard iterative deepening is that the cutoff used is the f-cost ($g + h$) rather than the depth; at each iteration, the cutoff value is the smallest f-cost of any node that exceeded the cutoff on the previous iteration. IDA* is practical for many problems with unit step costs and avoids the substantial overhead associated with keeping a sorted queue of nodes. Unfortunately, it suffers from the same difficulties with real-valued costs as does the iterative version of uniform-cost search described in Exercise 3.11. This section briefly examines two more recent memory-bounded algorithms, called RBFS and MA*.

RECURSIVE BEST-FIRST SEARCH

Recursive best-first search (RBFS) is a simple recursive algorithm that attempts to mimic the operation of standard best-first search, but using only linear space. The algorithm is shown in Figure 4.5. Its structure is similar to that of a recursive depth-first search, but rather

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
    RBFS(problem, MAKE-NODE(INITIAL-STATE[problem]), ∞)

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
    if GOAL-TEST[problem](STATE[node]) then return node
    successors ← EXPAND(node, problem)
    if successors is empty then return failure, ∞
    for each s in successors do
        f[s] ← max(g(s) + h(s), f[node])
    repeat
        best ← the lowest f-value node in successors
        if f[best] > f_limit then return failure, f[best]
        alternative ← the second-lowest f-value among successors
        result, f[best] ← RBFS(problem, best, min(f_limit, alternative))
        if result ≠ failure then return result
```

Figure 4.5 The algorithm for recursive best-first search.

than continuing indefinitely down the current path, it keeps 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 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 4.6 shows how RBFS reaches Bucharest.

RBFS is somewhat more efficient than IDA*, but still suffers from excessive node regeneration. In the example in Figure 4.6, RBFS first 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, there is a good chance that its f-value will increase—h is usually less optimistic for nodes closer to the goal. When this happens, particularly in large search spaces, 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.

Like A*, RBFS is an optimal algorithm if the heuristic function $h(n)$ is admissible. Its space complexity is $O(bd)$, 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. Both IDA* and RBFS are subject to the potentially exponential increase in complexity associated with searching on graphs (see Section 3.5), because they cannot check for repeated states other than those on the current path. Thus, they may explore the same state many times.

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,

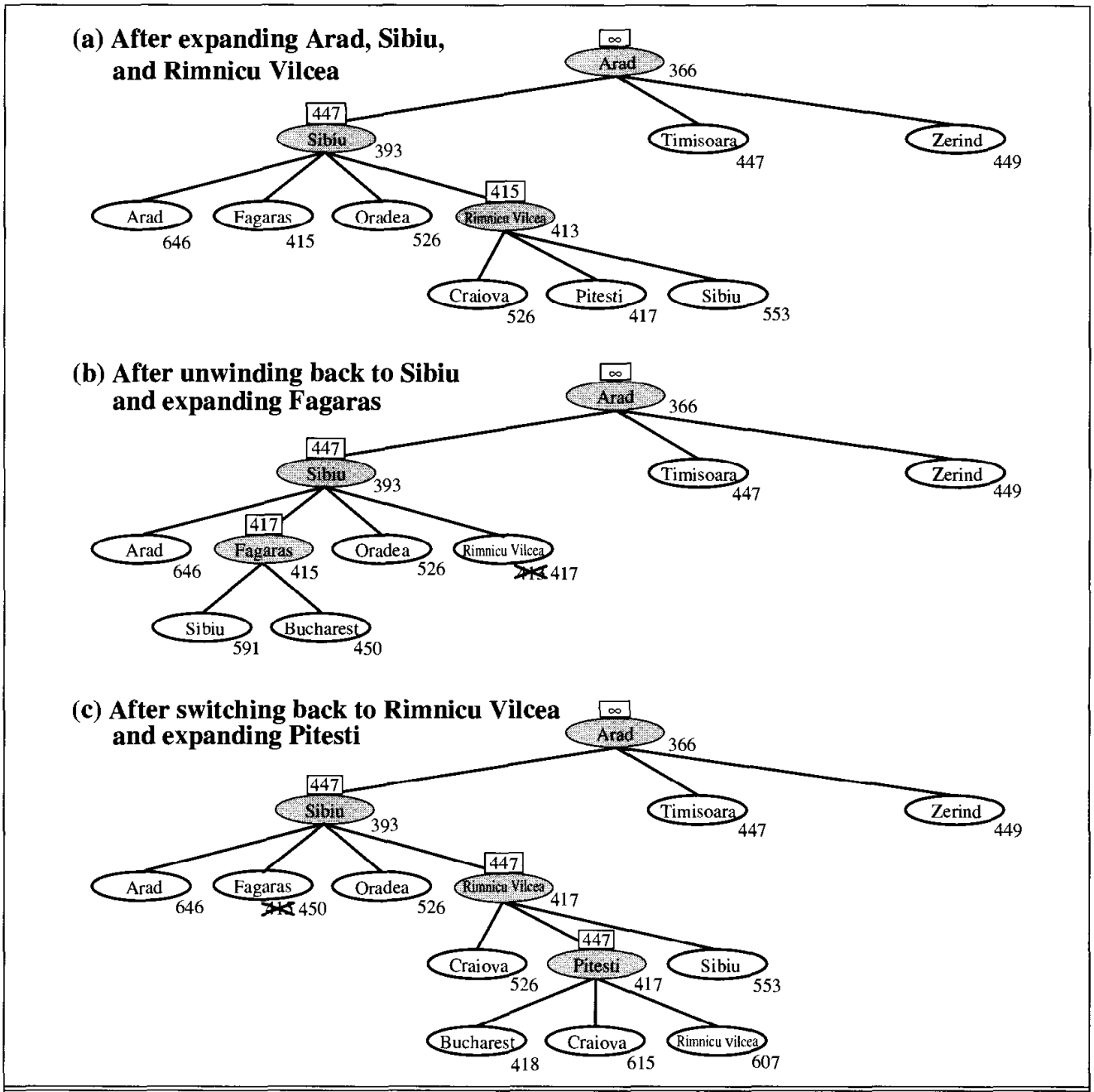


Figure 4.6 Stages in an RBFS search for the shortest route to Bucharest. The f-limit value for each recursive call is shown on top of each current node. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras). (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.

but it uses only $O(bd)$ memory: even if more memory were available, RBFS has no way to make use of it.

It seems sensible, therefore, to use all available memory. Two algorithms that do this are **MA*** (memory-bounded A*) and **SMA*** (simplified MA*). We will describe SMA*, which

MA*

SMA*

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

The complete algorithm is too complicated to reproduce here,[5] but 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? Then the algorithm might select the same node for deletion and expansion. SMA* solves this problem by expanding the *newest* best leaf and deleting the *oldest* worst leaf. These can be the same node only if there is only one leaf; 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* might well be the best general-purpose algorithm for finding optimal solutions, particularly when the state space is a graph, step costs are not uniform, and node generation is expensive compared to the additional overhead of maintaining the open and closed lists.

On very hard problems, however, it will often be the case that SMA* is forced to switch back and forth continually between a set of 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 there is no theory to explain 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

Learning to search better

We have presented several fixed strategies—breadth-first, greedy best-first, and so on—that have been designed 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 **object-level state space** such as Romania. For example, the internal state of the A* algorithm consists of the current search tree. Each action in the metalevel state

METALEVEL STATE SPACE

OBJECT-LEVEL STATE SPACE

[5] A rough sketch appeared in the first edition of this book.

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 4.3, 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 4.3 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

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

4.2 HEURISTIC FUNCTIONS

In this section, we will look at heuristics for the 8-puzzle, in order to shed light on the nature of heuristics in general.

The 8-puzzle was one of the earliest heuristic search problems. 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 4.7).

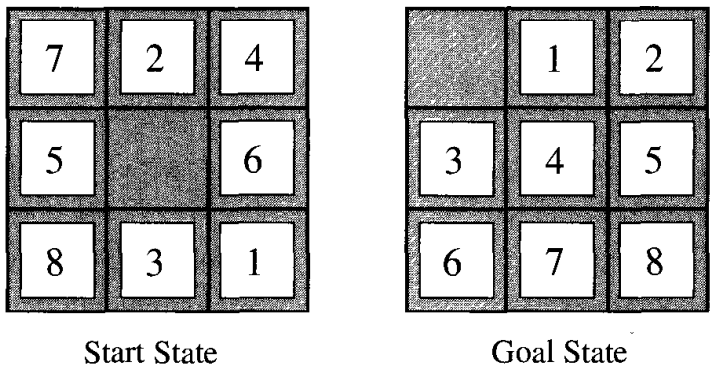


Figure 4.7 A typical instance of the 8-puzzle. The solution is 26 steps long.

The average solution cost for a randomly generated 8-puzzle instance is about 22 steps. The branching factor is about 3. (When the empty tile is in the middle, there are four possible moves; when it is in a corner there are two; and when it is along an edge there are three.) This means that an exhaustive search to depth 22 would look at about $3^{22} \approx 3.1 \times 10^{10}$ states. By keeping track of repeated states, we could cut this down by a factor of about 170,000, because there are only $9!/2 = 181,440$ distinct states that are reachable. (See Exercise 3.4.) This is a manageable number, but the corresponding number for the 15-puzzle is roughly 10^{13}, so the next order of business is to find a good heuristic function. If we want to find the shortest solutions by using A*, we need a heuristic function that never overestimates the number of steps to the goal. 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. For Figure 4.7, all of the eight tiles are out of position, so the start state would have $h_1 = 8$. h_1 is an admissible heuristic, because it is clear that any tile that is out of place must be moved at least once.
- h_2 = the sum of the distances of the tiles from their goal positions. Because tiles cannot move along diagonals, the distance we will count is the sum of the horizontal and vertical distances. This is 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 give a Manhattan distance of

MANHATTAN DISTANCE

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

As we would hope, neither of these overestimates the true solution cost, which is 26.

The effect of heuristic accuracy on performance

EFFECTIVE BRANCHING FACTOR

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

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

For example, if A* finds a solution at depth 5 using 52 nodes, then the effective branching factor is 1.92. The effective branching factor can vary across problem instances, but usually it is fairly constant for sufficiently hard problems. 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.

To test the heuristic functions h_1 and h_2, we generated 1200 random problems with solution lengths from 2 to 24 (100 for each even number) and solved them with iterative deepening search and with A* tree search using both h_1 and h_2. Figure 4.8 gives the average number of nodes expanded by each strategy and the effective branching factor. The results suggest that h_2 is better than h_1, and is far better than using iterative deepening search. On our solutions with length 14, A* with h_2 is 30,000 times more efficient than uninformed iterative deepening search.

One might ask whether h_2 is *always* better than h_1. The answer is 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 possibly for some nodes with $f(n) = C^*$). The argument is simple. Recall the observation on page 100 that every node with $f(n) < C^*$ will surely be expanded. This is the same as saying that every node with $h(n) < C^* - g(n)$ will surely be expanded. 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 will also surely be expanded with h_1, and h_1 might also cause other nodes to be expanded as well. Hence, it is always better to use a heuristic function with higher values, provided it does not overestimate and that the computation time for the heuristic is not too large.

DOMINATION

	Search Cost			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Figure 4.8 Comparison of the search costs and effective branching factors for the ITERATIVE-DEEPENING-SEARCH and A* algorithms with h_1, h_2. Data are averaged over 100 instances of the 8-puzzle, for various solution lengths.

Inventing admissible heuristic functions

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 number of steps in 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 number of steps in the shortest solution. A problem with fewer restrictions on the actions is called a **relaxed problem**. *The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem.* The heuristic is admissible because the optimal solution in the original problem is, by definition, also a solution in the relaxed problem and therefore must be at least as expensive as the optimal solution in the relaxed problem. Because the derived heuristic is an exact cost for the relaxed problem, it must obey the triangle inequality and is therefore **consistent** (see page 99).

RELAXED PROBLEM

If a problem definition is written down in a formal language, it is possible to construct relaxed problems automatically.[6] For example, if the 8-puzzle actions are described as

A tile can move from square A to square B if
A is horizontally or vertically adjacent to B **and** B is blank,

[6] In Chapters 8 and 11, we will 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 will use English.

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

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

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 4.9. 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 step. 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.[7]

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 better than any preexisting heuristic and found the first useful heuristic for the famous Rubik's cube puzzle.

One problem with generating new heuristic functions is that one often fails to get one "clearly best" heuristic. If a collection of admissible heuristics $h_1 \ldots h_m$ is available for a problem, and none of them dominates any of the others, which should we choose? As it turns out, we need not make a choice. We can have the best of all worlds, by defining

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

This composite heuristic uses whichever function is most accurate on the node in question. Because the component heuristics are admissible, h is admissible; it is also easy to prove that h is consistent. Furthermore, h dominates all of its component heuristics.

SUBPROBLEM

Admissible heuristics can also be derived from the solution cost of a **subproblem** of a given problem. For example, Figure 4.9 shows a subproblem of the 8-puzzle instance in Figure 4.7. The subproblem involves getting tiles 1, 2, 3, 4 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 substantially more accurate than Manhattan distance in some cases.

PATTERN DATABASES

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. (Notice that the locations of the other four tiles are irrelevant for the purposes of solving the subproblem, but moves of those tiles do count towards the cost.) Then, we compute an admissible heuristic h_{DB} for each complete state encountered during a search simply by looking up the corresponding subproblem configuration in the database. The database itself is constructed by searching backwards from the goal state and recording the cost of each new pattern encountered; the expense of this search is amortized over many subsequent problem instances.

[7] Note that a perfect heuristic can be obtained simply by allowing h to run a full breadth-first search "on the sly." Thus, there is a tradeoff between accuracy and computation time for heuristic functions.

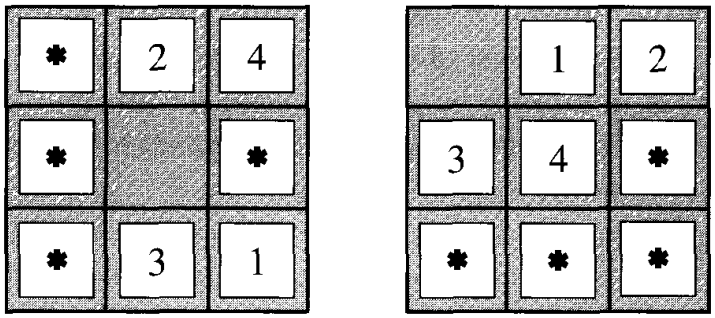


Figure 4.9 A subproblem of the 8-puzzle instance given in Figure 4.7. The task is to get tiles 1, 2, 3, and 4 into their correct positions, without worrying about what happens to the other tiles.

The choice of 1-2-3-4 is fairly arbitrary; we could also construct databases for 5-6-7-8, and 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.

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? 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 it is easy to see that 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**. Using 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 using Manhattan distance. For 24-puzzles, a speedup of roughly a million can be obtained.

DISJOINT PATTERN DATABASES

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. For a problem such as Rubik's cube, this kind of subdivision cannot be done because each move affects 8 or 9 of the 26 cubies. Currently, it is not clear how to define disjoint databases for such problems.

Learning heuristics from experience

A heuristic function $h(n)$ is supposed to estimate the cost of a solution beginning from the state at node n. How could an agent construct such a function? One solution was given in the preceding section—namely, to devise relaxed problems for which an optimal solution can be found easily. Another solution is to learn from experience. "Experience" here means solving lots of 8-puzzles, for instance. Each optimal solution to an 8-puzzle problem provides ex-

amples from which $h(n)$ can be learned. Each example consists of a state from the solution path and the actual cost of the solution from that point. From these examples, an **inductive learning** algorithm can be used to construct a function $h(n)$ that can (with luck) predict solution costs for other states that arise during search. Techniques for doing just this using neural nets, decision trees, and other methods are demonstrated in Chapter 18. (The reinforcement learning methods described in Chapter 21 are also applicable.)

FEATURES

Inductive learning methods work best when supplied with **features** of a state that are relevant to its evaluation, 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 a 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. Given these data, the value of x_1 can be used to predict $h(n)$. Of course, we can use several features. A second feature $x_2(n)$ might be "number of pairs of adjacent tiles that are also adjacent in the goal state." How should $x_1(n)$ and $x_2(n)$ be combined to predict $h(n)$? A common approach is to use a linear combination:

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

The constants c_1 and c_2 are adjusted to give the best fit to the actual data on solution costs. Presumably, c_1 should be positive and c_2 should be negative.

4.3 Local Search Algorithms and Optimization Problems

The search algorithms that we have seen so far are designed to explore search spaces systematically. This systematicity is achieved by keeping one or more paths in memory and by recording which alternatives have been explored at each point along the path and which have not. When a goal is found, the *path* to that goal also constitutes a *solution* to the problem.

In many problems, however, the path to the goal is irrelevant. For example, in the 8-queens problem (see page 66), what matters is the final configuration of queens, not the order in which they are added. This class of problems includes many important applications such as integrated-circuit design, factory-floor layout, job-shop scheduling, automatic programming, telecommunications network optimization, vehicle routing, and portfolio management.

LOCAL SEARCH
CURRENT STATE

If the path to the goal does not matter, we might consider a different class of algorithms, ones that do not worry about paths at all. **Local search** algorithms operate using a single **current state** (rather than multiple paths) and generally move only to neighbors of that state. Typically, the paths followed by the search are not retained. Although local search algorithms are not systematic, they have two key advantages: (1) they use very little memory—usually a constant amount; and (2) they can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.

OPTIMIZATION PROBLEMS
OBJECTIVE FUNCTION

In addition to finding goals, local search algorithms are useful for solving pure **optimization problems**, in which the aim is to find the best state according to an **objective function**. Many optimization problems do not fit the "standard" search model introduced in

Chapter 3. For example, nature provides an objective function—reproductive fitness—that Darwinian evolution could be seen as attempting to optimize, but there is no "goal test" and no "path cost" for this problem.

STATE SPACE LANDSCAPE

To understand local search, we will find it very useful to consider the **state space landscape** (as in Figure 4.10). A landscape has both "location" (defined by the state) and "elevation" (defined by the value of the heuristic cost function or objective function). If elevation corresponds to cost, then the aim is to find the lowest valley—a **global minimum**; if elevation corresponds to an objective function, then the aim is to find the highest peak—a **global maximum**. (You can convert from one to the other just by inserting a minus sign.) Local search algorithms explore this landscape. A **complete** local search algorithm always finds a goal if one exists; an **optimal** algorithm always finds a global minimum/maximum.

GLOBAL MINIMUM

GLOBAL MAXIMUM

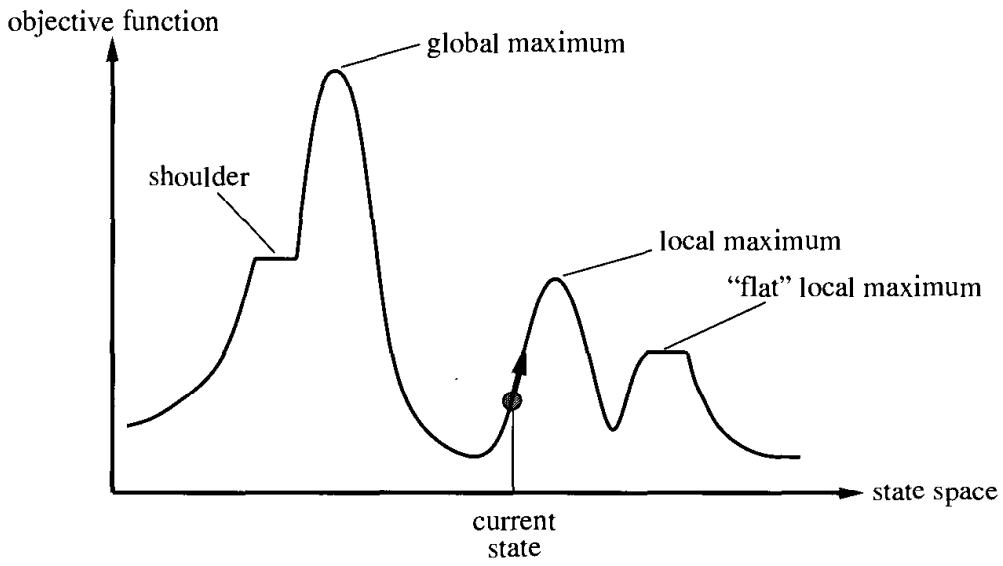


Figure 4.10 A one-dimensional state space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

Hill-climbing search

HILL-CLIMBING

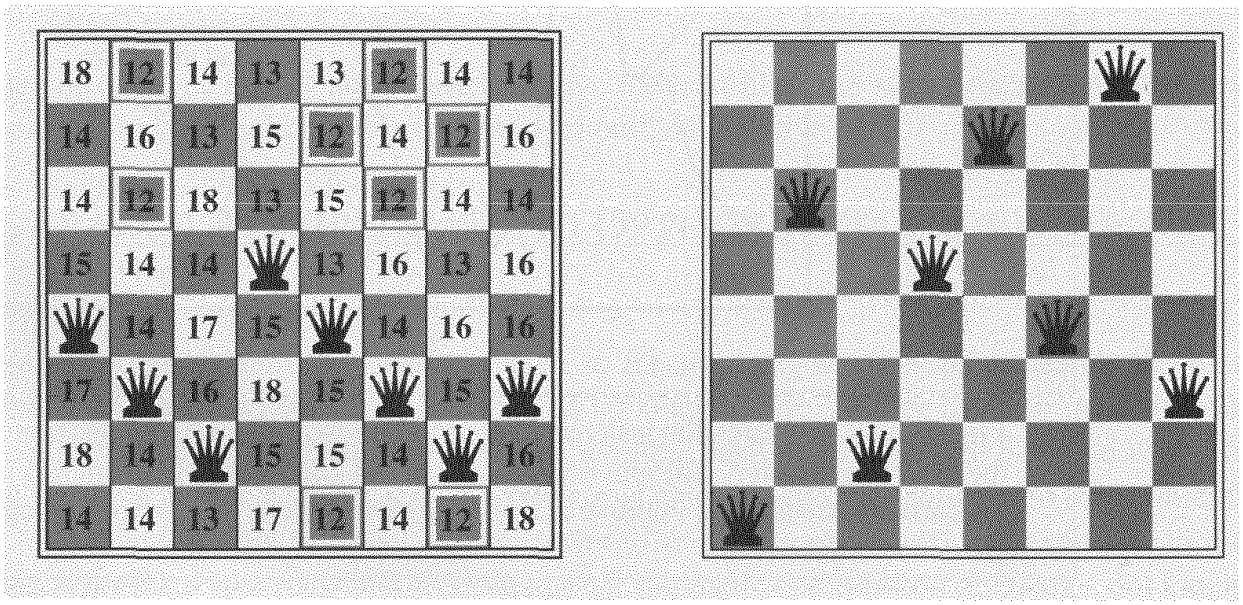
The **hill-climbing** search algorithm is shown in Figure 4.11. It is simply a loop that continually moves in the direction of increasing value—that is, uphill. It terminates when it reaches a "peak" where no neighbor has a higher value. The algorithm does not maintain a search tree, so the current node data structure need only record the state and its objective function 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.

To illustrate hill-climbing, we will use the **8-queens problem** introduced on page 66. Local-search algorithms typically use a **complete-state formulation**, where each state has 8 queens on the board, one per column. The successor function returns all possible states generated by moving a single queen to another square in the same column (so each state has

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  inputs: problem, a problem
  local variables: current, a node
                   neighbor, a node

  current ← MAKE-NODE(INITIAL-STATE[problem])
  loop do
    neighbor ← a highest-valued successor of current
    if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
    current ← neighbor
```

Figure 4.11 The hill-climbing search algorithm (**steepest ascent** version), which is the most basic local search technique. At each step the current node is replaced by the best neighbor; in this version, that means the neighbor with the highest VALUE, but if a heuristic cost estimate h is used, we would find the neighbor with the lowest h.



(a) (b)

Figure 4.12 (a) An 8-queens state with heuristic cost estimate $h = 17$, showing the value of h for each possible successor obtained by moving a queen within its column. The best moves are marked. (b) A local minimum in the 8-queens state space; the state has $h = 1$ but every successor has a higher cost.

$8 \times 7 = 56$ successors). The heuristic cost function h is the number of pairs of queens that are attacking each other, either directly or indirectly. The global minimum of this function is zero, which occurs only at perfect solutions. Figure 4.12(a) shows a state with $h = 17$. The figure also shows the values of all its successors, with the best successors having $h = 12$. Hill-climbing algorithms typically choose randomly among the set of best successors, if there is more than one.

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 often makes very rapid progress towards a solution, because it is usually quite easy to improve a bad state. For example, from the state in Figure 4.12(a), it takes just five steps to reach the state in Figure 4.12(b), which has $h = 1$ and is very nearly a solution. Unfortunatcly, hill climbing often gets stuck for 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 upwards towards the peak, but will then be stuck with nowhere else to go. Figure 4.10 illustrates the problem schematically. More concretely, the state in Figure 4.12(b) is in fact a local maximum (i.e., a local minimum for the cost h); every move of a single queen makes the situation worse.
- ◇ **Ridges**: a ridge is shown in Figure 4.13. Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.
- ◇ **Plateaux**: a plateau is an area of the state space landscape where the evaluation function is flat. It can be a flat local maximum, from which no uphill exit exists, or a **shoulder**, from which it is possible to make progress. (See Figure 4.10.) A hill-climbing search might be unable to find its way off the 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. 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.

The algorithm in Figure 4.11 halts if it reaches a plateau where the best successor has the same value as the current state. Might it not be a good idea to keep going—to allow a **sideways move** in the hope that the plateau is really a shoulder, as shown in Figure 4.10? The answer is usually yes, but we must take care. If we always allow sideways moves when there are no uphill moves, an infinite loop will occur whenever the algorithm reaches a flat local maximum that is not a shoulder. One common solution is to put a limit on the number of consecutive sideways moves allowed. For example, we could allow up to, say, 100 consecutive sideways moves in the 8-queens problem. 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

STOCHASTIC HILL CLIMBING

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. Exercise 4.16 asks you to investigate.

FIRST-CHOICE HILL CLIMBING

The hill-climbing algorithms described so far are incomplete—they often fail to find a goal when one exists because they can get stuck on local maxima. **Random-restart hill**

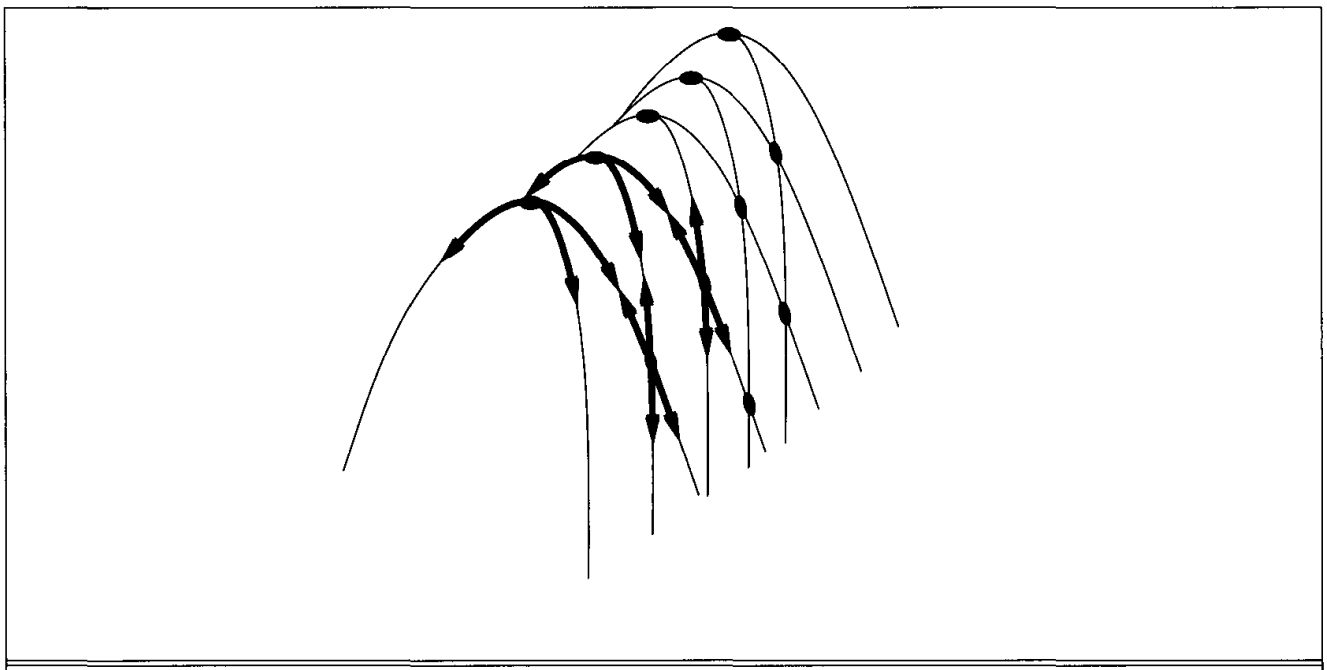


Figure 4.13 Illustration of why ridges cause difficulties for hill-climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point downhill.

RANDOM-RESTART HILL CLIMBING

climbing adopts the well known adage, "If at first you don't succeed, try, try again." It conducts a series of hill-climbing searches from randomly generated initial states,[8] stopping when a goal is found. It is complete with probability approaching 1, for the trivial reason that 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. 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 under a minute.[9]

The success of hill climbing depends very much on the shape of the state-space landscape: if there are few local maxima and plateaux, 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 family of porcupines on a flat floor, with miniature porcupines living on the tip of each porcupine needle, *ad infinitum*. NP-hard problems 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.

[8] Generating a *random* state from an implicitly specified state space can be a hard problem in itself.

[9] Luby *et al.* (1993) prove that it is best, in some cases, to restart a randomized search algorithm after a particular, fixed amount of time and that this can be *much* more efficient than letting each search continue indefinitely. Disallowing or limiting the number of sideways moves is an example of this.

Simulated annealing search

A hill-climbing algorithm that *never* makes "downhill" moves towards states with lower value (or higher cost) is guaranteed to be incomplete, because it can get stuck on a local maximum. In contrast, a purely random walk—that is, moving to a successor chosen uniformly at random from the set of successors—is complete, but extremely inefficient. Therefore, it seems reasonable to try to combine hill climbing with a random walk in some 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 coalesce into a low-energy crystalline state. To understand simulated annealing, let's 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 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. 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

GRADIENT DESCENT

The innermost loop of the simulated-annealing algorithm (Figure 4.14) is quite 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 temperature is high, and they become more unlikely as T decreases. One can prove that if the *schedule* lowers T slowly enough, the algorithm will find a global optimum with probability approaching 1.

Simulated annealing was first used extensively to solve VLSI layout problems in the early 1980s. It has been applied widely to factory scheduling and other large-scale optimization tasks. In Exercise 4.16, you are asked to compare its performance to that of random-restart hill climbing on the n-queens puzzle.

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[10] 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

[10] Local beam search is an adaptation of **beam search**, which is a path-based algorithm.

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"
  local variables: current, a node
                   next, a node
                   T, a "temperature" controlling the probability of downward steps

  current ← MAKE-NODE(INITIAL-STATE[problem])
  for t ← 1 to ∞ do
      T ← schedule[t]
      if T = 0 then return current
      next ← a randomly selected successor of current
      ΔE ← VALUE[next] – VALUE[current]
      if ΔE > 0 then current ← next
      else current ← next only with probability e^(ΔE/T)
```

Figure 4.14 The simulated annealing search algorithm, a version of stochastic hill climbing where some downhill moves are allowed. Downhill moves are accepted readily early in the annealing schedule and then less often as time goes on. The *schedule* input determines the value of T as a function of time.

the others. *In a local beam search, useful information is passed among the k parallel search threads.* For example, if one state generates several good successors and the other $k-1$ states all generate bad successors, then the effect is that the first state says 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.

In its simplest form, local beam search can suffer from a lack of diversity among the k states—they can quickly become concentrated in a small region of the state space, making the search little more than an expensive version of hill climbing. A variant called **stochastic beam search**, analogous to stochastic hill climbing, helps to alleviate this problem. Instead of choosing the best k from the the pool of candidate successors, stochastic beam search chooses k successors at random, with the probability of choosing a given successor being an increasing function of its value. Stochastic beam search bears some resemblance to the process of natural selection, whereby the "successors" (offspring) of a "state" (organism) populate the next generation according to its "value" (fitness).

STOCHASTIC BEAM SEARCH

Genetic algorithms

GENETIC ALGORITHM

A **genetic algorithm** (or GA) is a variant of stochastic beam search in which successor states are generated by combining *two* parent states, rather than by modifying a single state. The analogy to natural selection is the same as in stochastic beam search, except now we are dealing with sexual rather than asexual reproduction.

Like beam search, GAs begin with a set of k randomly generated states, called the **population**. Each state, or **individual**, is represented as a string over a finite alphabet—most

POPULATION

INDIVIDUAL

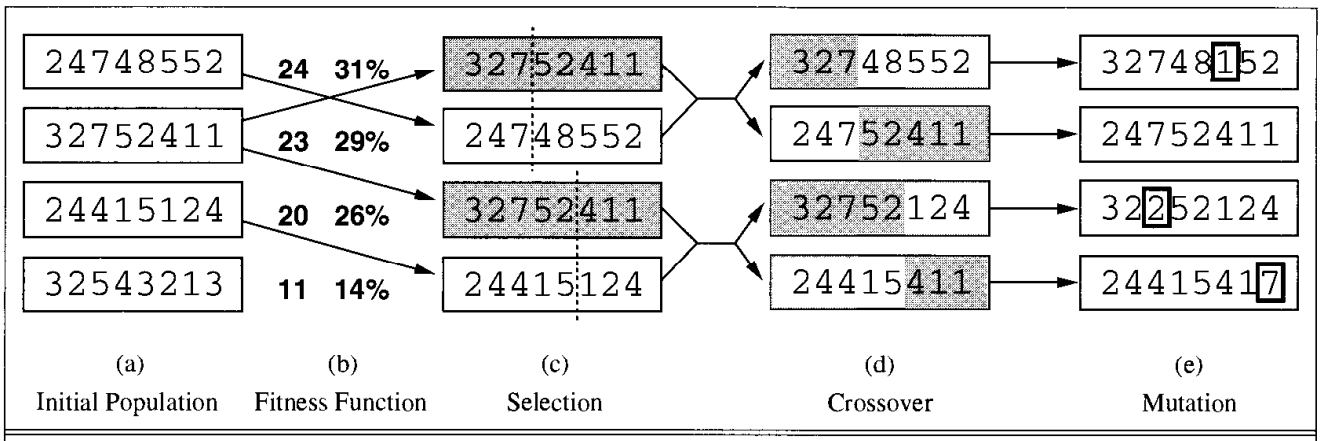


Figure 4.15 The genetic algorithm. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

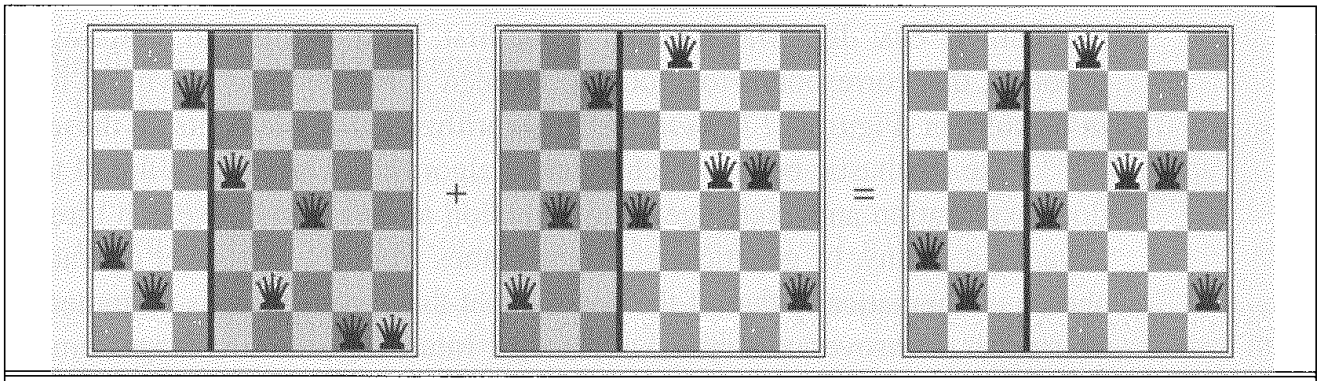


Figure 4.16 The 8-queens states corresponding to the first two parents in Figure 4.15(c) and the first offspring in Figure 4.15(d). The shaded columns are lost in the crossover step and the unshaded columns are retained.

commonly, a string of 0s and 1s. For example, an 8-queens state must specify the positions of 8 queens, each in a column of 8 squares, and so requires $8 \times \log_2 8 = 24$ bits. Alternatively, the state could be represented as 8 digits, each in the range from 1 to 8. (We will see later that the two encodings behave differently.) Figure 4.15(a) shows a population of four 8-digit strings representing 8-queens states.

FITNESS FUNCTION

The production of the next generation of states is shown in Figure 4.15(b)–(e). In (b), each state is rated by the evaluation function or (in GA terminology) the **fitness function**. A fitness function should return higher values for better states, so, for the 8-queens problem we use the number of *nonattacking* pairs of queens, which has a value of 28 for a solution. The values of the four states are 24, 23, 20, and 11. In this particular variant of the genetic algorithm, the probability of being chosen for reproducing is directly proportional to the fitness score, and the percentages are shown next to the raw scores.

In (c), a random choice of two pairs is selected for reproduction, in accordance with the probabilities in (b). Notice that one individual is selected twice and one not at all.[11] For each

[11] There are many variants of this selection rule. The method of **culling**, in which all individuals below a given threshold are discarded, can be shown to converge faster than the random version (Baum *et al.*, 1995).

CROSSOVER

pair to be mated, a **crossover** point is randomly chosen from the positions in the string. In Figure 4.15 the crossover points are after the third digit in the first pair and after the fifth digit in the second pair.[12]

In (d), the offspring themselves are created by crossing over the parent strings at the crossover point. For example, the first child of the first pair gets the first three digits from the first parent and the remaining digits from the second parent, whereas the second child gets the first three digits from the second parent and the rest from the first parent. The 8-queens states involved in this reproduction step are shown in Figure 4.16. The example illustrates the fact that, when two parent states are quite different, the crossover operation can produce a state that is a long way from either parent state. It is often the case that the population is quite diverse early on in the process, so crossover (like simulated annealing) frequently takes large steps in the state space early in the search process and smaller steps later on when most individuals are quite similar.

MUTATION

Finally, in (e), each location 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. Figure 4.17 describes an algorithm that implements all these steps.

Like stochastic beam search, genetic algorithms combine an uphill tendency with random exploration and exchange of information among parallel search threads. The primary advantage, if any, of genetic algorithms comes from the crossover operation. Yet it can be shown mathematically that, if the positions of the genetic code is permuted initially in a random order, crossover conveys no advantage. Intuitively, the advantage comes from the ability of crossover to combine large blocks of letters that have evolved independently to perform useful functions, thus raising the level of granularity at which the search operates. 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 blocks to construct a solution.

SCHEMA

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 within the population will grow over time. 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.

[12] It is here that the encoding matters. If a 24-bit encoding is used instead of 8 digits, then the crossover point has a 2/3 chance of being in the middle of a digit, which results in an essentially arbitrary mutation of that digit.

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
    new_population ← empty set
    loop for i from 1 to SIZE(population) do
      x ← RANDOM-SELECTION(population, FITNESS-FN)
      y ← RANDOM-SELECTION(population, FITNESS-FN)
      child ← REPRODUCE(x, y)
      if (small random probability) then child ← MUTATE(child)
      add child to new_population
    population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN

function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x)
  c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

Figure 4.17 A genetic algorithm. The algorithm is the same as the one diagrammed in Figure 4.15, with one variation: in this more popular version, each mating of two parents produces only one offspring, not two.

In practice, genetic algorithms have had a widespread impact on optimization problems, such as circuit layout and job-shop scheduling. At present, it is not clear whether the appeal of genetic algorithms arises from their performance or from their æsthetically pleasing origins in the theory of evolution. Much work remains to be done to identify the conditions under which genetic algorithms perform well.

4.4 Local Search in Continuous Spaces

In Chapter 2, we explained the distinction between discrete and continuous environments, pointing out that most real-world environments are continuous. Yet none of the algorithms we have described can handle continuous state spaces—the successor function would in most cases return infinitely many states! This section provides a *very brief* introduction to some local search techniques for finding optimal solutions in continuous spaces. The literature on this topic is vast; many of the basic techniques originated in the 17th century, after the development of calculus by Newton and Leibniz.[13] We will find uses for these techniques at

[13] A basic knowledge of multivariate calculus and vector arithmetic is useful when one is reading this section.

EVOLUTION AND SEARCH

The theory of **evolution** was developed in Charles Darwin's *On the Origin of Species by Means of Natural Selection* (1859). The central idea is simple: variations (known as **mutations**) 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 using what he called artificial fertilization. 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* organisms 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 in 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 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 might well seem to be an inefficient mechanism, having generated blindly some 10^{45} or so organisms without improving its search heuristics one iota. Fifty years before Darwin, however, the otherwise great French naturalist Jean Lamarck (1809) proposed a theory of evolution whereby traits *acquired by adaptation during an organism's lifetime* would be passed on to its offspring. Such a process would be effective, but does not seem to occur in nature. Much later, James Baldwin (1896) proposed a superficially similar theory: that behavior learned during an organism's lifetime could accelerate the rate of evolution. Unlike Lamarck's, Baldwin's theory is entirely consistent with Darwinian evolution, because it relies on selection pressures operating on individuals that have found local optima among the set of possible behaviors allowed by their genetic makeup. Modern computer simulations confirm that the "Baldwin effect" is real, provided that "ordinary" evolution can create organisms whose internal performance measure is somehow correlated with actual fitness.

several places in the book, including the chapters on learning, vision, and robotics. In short, anything that deals with the real world.

Let us begin with an example. Suppose we want to place three new airports anywhere in Romania, such that the sum of squared distances from each city on the map (Figure 3.2) to its nearest airport is minimized. Then the state space is defined by the coordinates of the airports: (x_1, y_1), (x_2, y_2), and (x_3, y_3). This is a *six-dimensional* space; we also say that states are defined by six **variables**. (In general, states are defined by an n-dimensional vector of variables, $\mathbf{x}$.) Moving around in this space corresponds to moving one or more of the airports on the map. The objective function $f(x_1, y_1, x_2, y_2, x_3, y_3)$ is relatively easy to compute for any particular state once we compute the closest cities, but rather tricky to write down in general.

One way to avoid continuous problems is simply to **discretize** the neighborhood of each state. For example, we can move only one airport at a time in either the x or y direction by a fixed amount $\pm\delta$. With 6 variables, this gives 12 possible successors for each state. We can then apply any of the local search algorithms described previously. One can also apply stochastic hill climbing and simulated annealing directly, without discretizing the space. These algorithms choose successors randomly, which can be done by generating random vectors of length δ.

GRADIENT

There are many methods that attempt to use the **gradient** of the landscape to find a maximum. The gradient of the objective function is a vector ∇f that gives the magnitude and direction of the steepest slope. For our problem, we have

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

In some cases, we can find a maximum by solving the equation $\nabla f = 0$. (This could be done, for example, if we were placing just one airport; the solution is the arithmetic mean of all the cities' coordinates.) In many cases, however, this equation cannot be solved in closed form. For example, with three airports, the expression for the gradient depends on what cities are closest to each airport in the current state. This means we can compute the gradient *locally* but not *globally*. Even so, we can still perform steepest-ascent hill climbing by updating the current state via the formula

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

where α is a small constant. In other cases, the objective function might not be available in a differentiable form at all—for example, the value of a particular set of airport locations may be determined by running some large-scale economic simulation package. In those

EMPIRICAL GRADIENT

cases, a so-called **empirical gradient** can be determined by evaluating the response to small increments and decrements in each coordinate. Empirical gradient search is the same as steepest-ascent hill climbing in a discretized version of the state space.

Hidden beneath the phrase "α is a small constant" lies a huge variety of methods for adjusting α. The basic problem is that, if α is too small, too many steps are needed; if α

LINE SEARCH

is too large, the search could overshoot the maximum. The technique of **line search** tries to overcome this dilemma by extending the current gradient direction—usually by repeatedly doubling α—until f starts to decrease again. The point at which this occurs becomes the new

current state. There are several schools of thought about how the new direction should be chosen at this point.

NEWTON–RAPHSON

For many problems, the most effective algorithm is the venerable **Newton–Raphson** method (Newton, 1671; Raphson, 1690). This is a general technique for finding roots of functions—that is, solving equations of the form $g(x) = 0$. It works by computing a new estimate for the root x according to Newton's formula

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

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

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

HESSIAN

where $\mathbf{H}_f(\mathbf{x})$ is the **Hessian** matrix of second derivatives, whose elements H_{ij} are given by $\partial^2 f/\partial x_i \partial x_j$. Since the Hessian has n^2 entries, Newton–Raphson becomes expensive in high-dimensional spaces, and many approximations have been developed.

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

CONSTRAINED OPTIMIZATION

A final topic with which a passing acquaintance is useful is **constrained optimization**. An optimization problem is constrained if solutions must satisfy some hard constraints on the values of each variable. For example, in our airport-siting problem, we might constrain sites to be inside Romania and on dry land (rather than in the middle of lakes). The difficulty of constrained optimization problems depends on the nature of the constraints and the objective function. The best-known category is that of **linear programming** problems, in which constraints must be linear inequalities forming a *convex* region and the objective function is also linear. Linear programming problems can be solved in time polynomial in the number of variables. Problems with different types of constraints and objective functions have also been studied—quadratic programming, second-order conic programming, and so on.

LINEAR PROGRAMMING

4.5 Online Search Agents and Unknown Environments

OFFLINE SEARCH

So far we have concentrated on agents that use **offline search** algorithms. They compute a complete solution before setting foot in the real world (see Figure 3.1), and then execute the solution without recourse to their percepts. In contrast, an **online search**[14] agent operates by **interleaving** computation and action: first it takes an action, then it observes the environment and computes the next action. Online search is a good idea in dynamic or semidynamic domains—domains where there is a penalty for sitting around and computing too long. Online search is an even better idea for stochastic domains. In general, an offline search would

ONLINE SEARCH

[14] The term "online" is commonly used in computer science to refer to algorithms that must process input data as they are received, rather than waiting for the entire input data set to become available.

have to come up with an exponentially large contingency plan that considers all possible happenings, while an online search need only consider what actually does happen. For example, a chess playing agent is well-advised to make its first move long before it has figured out the complete course of the game.

EXPLORATION PROBLEM

Online search is a *necessary* idea for an **exploration problem**, where the states and actions are unknown to the agent. An agent in this state of ignorance must use its actions as experiments to determine what to do next, and hence must interleave computation and action.

The canonical example of online search is a robot that is placed in a new building and must explore it to build a map that it can use for getting from A to B. Methods for escaping from labyrinths—required knowledge for aspiring heroes of antiquity—are also examples of online search algorithms. Spatial exploration is not the only form of exploration, however. Consider a newborn baby: it has many possible actions, but knows the outcomes of none of them, and it has experienced only a few of the possible states that it can reach. The baby's gradual discovery of how the world works is, in part, an online search process.

Online search problems

An online search problem can be solved only by an agent executing actions, rather than by a purely computational process. We will assume that the agent knows just the following:

- ACTIONS(s), which returns a list of actions allowed in state s;
- The step-cost function $c(s, a, s')$—note that this cannot be used until the agent knows that s' is the outcome; and
- GOAL-TEST(s).

Note in particular that the agent *cannot* access the successors of a state except by actually trying all the actions in that state. For example, in the maze problem shown in Figure 4.18, the agent does not know that going *Up* from (1,1) leads to (1,2); nor, having done that, does it know that going *Down* will take it back to (1,1). This degree of ignorance can be reduced in some applications—for example, a robot explorer might know how its movement actions work and be ignorant only of the locations of obstacles.

We will assume that the agent can always recognize a state that it has visited before, and we will assume that the actions are deterministic. (These last two assumptions are relaxed in Chapter 17.) Finally, the agent might have access to an admissible heuristic function $h(s)$ that estimates the distance from the current state to a goal state. For example, in Figure 4.18, the agent might know the location of the goal and be able to use the Manhattan distance heuristic.

Typically, the agent's objective is to reach a goal state while minimizing cost. (Another possible objective is simply to explore the entire environment.) The cost is the total path cost of the path that the agent actually travels. It is common to compare this cost with the path cost of the path the agent would follow *if it knew the search space in advance*—that is, the actual shortest path (or shortest complete exploration). In the language of online algorithms,

COMPETITIVE RATIO

this is called the **competitive ratio**; we would like it to be as small as possible.

Although this sounds like a reasonable request, it is easy to see that the best achievable competitive ratio is infinite in some cases. For example, if some actions are irreversible, the online search might accidentally reach a dead-end state from which no goal state is reachable.

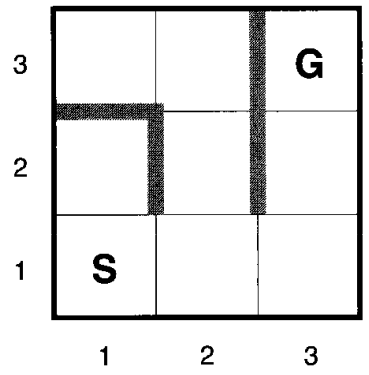


Figure 4.18 A simple maze problem. The agent starts at S and must reach G, but knows nothing of the environment.

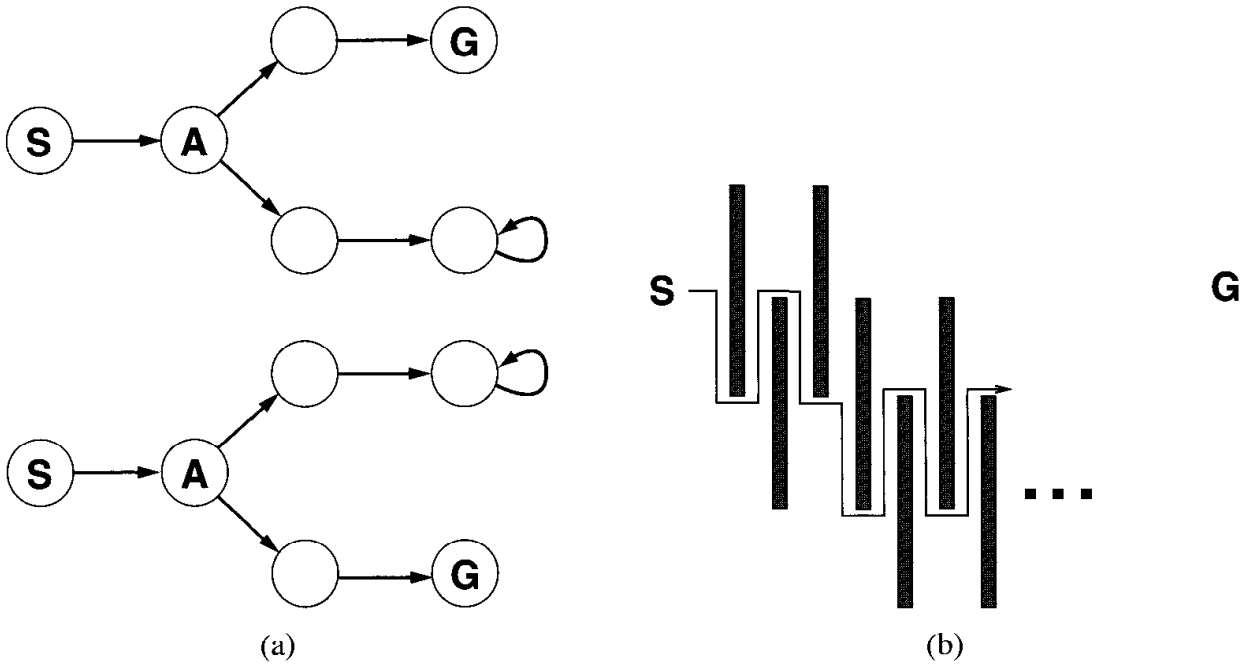


Figure 4.19 (a) Two state spaces that might lead an online search agent into a dead end. Any given agent will fail in at least one of these spaces. (b) A two-dimensional environment that can cause an online search agent to follow an arbitrarily inefficient route to the goal. Whichever choice the agent makes, the adversary blocks that route with another long, thin wall, so that the path followed is much longer than the best possible path.

Perhaps you find the term "accidentally" unconvincing—after all, there might be an algorithm that happens not to take the dead-end path as it explores. Our claim, to be more precise, is that *no algorithm can avoid dead ends in all state spaces.* Consider the two dead-end state spaces in Figure 4.19(a). To an online search algorithm that has visited states S and A, the two state spaces look *identical*, so it must make the same decision in both. Therefore, it will fail in one of them. This is an example of an **adversary argument**—we can imagine an adversary that constructs the state space while the agent explores it and can put the goals and dead ends wherever it likes.

ADVERSARY ARGUMENT

Dead ends are a real difficulty for robot exploration—staircases, ramps, cliffs, and all kinds of natural terrain present opportunities for irreversible actions. To make progress, we will simply assume that the state space is **safely explorable**—that is, some goal state is reachable from every reachable state. State spaces with reversible actions, such as mazes and 8-puzzles, can be viewed as undirected graphs and are clearly safely explorable.

SAFELY EXPLORABLE

Even in safely explorable environments, no bounded competitive ratio can be guaranteed if there are paths of unbounded cost. This is easy to show in environments with irreversible actions, but in fact it remains true for the reversible case as well, as Figure 4.19(b) shows. For this reason, it is common to describe the performance of online search algorithms in terms of the size of the entire state space rather than just the depth of the shallowest goal.

Online search agents

After each action, an online agent receives a percept telling it what state it has reached; from this information, it can augment its map of the environment. The current map is used to decide where to go next. This interleaving of planning and action means that online search algorithms are quite different from the offline search algorithms we have seen previously. For example, offline algorithms such as A* have the ability to expand a node in one part of the space and then immediately expand a node in another part of the space, because node expansion involves simulated rather than real actions. An online algorithm, on the other hand, can expand only a node that it physically occupies. To avoid traveling all the way across the tree to expand the next node, it seems better to expand nodes in a *local* order. Depth-first search has exactly this property, because (except when backtracking) the next node expanded is a child of the previous node expanded.

An online depth-first search agent is shown in Figure 4.20. This agent stores its map in a table, $result[a, s]$, that records the state resulting from executing action a in state s. Whenever an action from the current state has not been explored, the agent tries that action. The difficulty comes when the agent has tried all the actions in a state. In offline depth-first search, the state is simply dropped from the queue; in an online search, the agent has to backtrack physically. In depth-first search, this means going back to the state from which the agent entered the current state most recently. That is achieved by keeping a table that lists, for each state, the predecessor states to which the agent has not yet backtracked. If the agent has run out of states to which it can backtrack, then its search is complete.

We recommend that the reader trace through the progress of ONLINE-DFS-AGENT when applied to the maze given in Figure 4.18. It is fairly easy to see that the agent will, in the worst case, end up traversing every link in the state space exactly twice. For exploration, this is optimal; for finding a goal, on the other hand, the agent's competitive ratio could be arbitrarily bad if it goes off on a long excursion when there is a goal right next to the initial state. An online variant of iterative deepening solves this problem; for an environment that is a uniform tree, the competitive ratio of such an agent is a small constant.

Because of its method of backtracking, ONLINE-DFS-AGENT works only in state spaces where the actions are reversible. There are slightly more complex algorithms that work in general state spaces, but no such algorithm has a bounded competitive ratio.

```
function ONLINE-DFS-AGENT(s') returns an action
  inputs: s', a percept that identifies the current state
  static: result, a table, indexed by action and state, initially empty
          unexplored, a table that lists, for each visited state, the actions not yet tried
          unbacktracked, a table that lists, for each visited state, the backtracks not yet tried
          s, a, the previous state and action, initially null

  if GOAL-TEST(s') then return stop
  if s' is a new state then unexplored[s'] ← ACTIONS(s')
  if s is not null then do
      result[a, s] ← s'
      add s to the front of unbacktracked[s']
  if unexplored[s'] is empty then
      if unbacktracked[s'] is empty then return stop
      else a ← an action b such that result[b, s'] = POP(unbacktracked[s'])
  else a ← POP(unexplored[s'])
  s ← s'
  return a
```

Figure 4.20 An online search agent that uses depth-first exploration. The agent is applicable only in bidirected search spaces.

Online local search

Like depth-first search, **hill-climbing search** has the property of locality in its node expansions. In fact, because it keeps just one current state in memory, hill-climbing search is *already* an online search algorithm! Unfortunately, it is not very useful in its simplest form because it leaves the agent sitting at local maxima with nowhere to go. Moreover, random restarts cannot be used, because the agent cannot transport itself to a new state.

RANDOM WALK

Instead of random restarts, one might consider using a **random walk** to explore the environment. A random walk simply selects at random one of the available actions from the current state; preference can be given to actions that have not yet been tried. It is easy to prove that a random walk will *eventually* find a goal or complete its exploration, provided that the space is finite.[15] On the other hand, the process can be very slow. Figure 4.21 shows an environment in which a random walk will take exponentially many steps to find the goal, because, at each step, backward progress is twice as likely as forward progress. The example is contrived, of course, but there are many real-world state spaces whose topology causes these kinds of "traps" for random walks.

Augmenting hill climbing with *memory* rather than randomness turns out to be a more effective approach. The basic idea is to store a "current best estimate" $H(s)$ of the cost to reach the goal from each state that has been visited. $H(s)$ starts out being just the heuristic

[15] The infinite case is much more tricky. Random walks are complete on infinite one-dimensional and two dimensional grids, but not on three-dimensional grids! In the latter case, the probability that the walk ever returns to the starting point is only about 0.3405. (See Hughes, 1995, for a general introduction.)

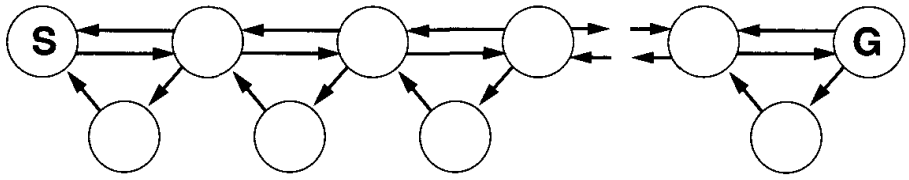


Figure 4.21 An environment in which a random walk will take exponentially many steps to find the goal.

estimate $h(s)$ and is updated as the agent gains experience in the state space. Figure 4.22 shows a simple example in a one-dimensional state space. In (a), the agent seems to be stuck in a flat local minimum at the shaded state. Rather than staying where it is, the agent should follow what seems to be the best path to the goal based on the current cost estimates for its neighbors. The estimated cost to reach the goal through a neighbor s' is the cost to get to s' plus the estimated cost to get to a goal from there—that is, $c(s, a, s') + H(s')$. In the example, there are two actions with estimated costs $1 + 9$ and $1 + 2$, so it seems best to move right. Now, it is clear that the cost estimate of 2 for the shaded state was overly optimistic. Since the best move cost 1 and led to a state that is at least 2 steps from a goal, the shaded state must be at least 3 steps from a goal, so its H should be updated accordingly, as shown in Figure 4.22(b). Continuing this process, the agent will move back and forth twice more, updating H each time and "flattening out" the local minimum until it escapes to the right.

LRTA*

An agent implementing this scheme, which is called learning real-time A* (**LRTA***), is shown in Figure 4.23. Like ONLINE-DFS-AGENT, it builds a map of the environment using the *result* table. It updates the cost estimate for the state it has just left and then chooses the "apparently best" move according to its current cost estimates. One important detail is that actions that have not yet been tried in a state s are always assumed to lead immediately to the goal with the least possible cost, namely $h(s)$. This **optimism under uncertainty** encourages the agent to explore new, possibly promising paths.

OPTIMISM UNDER UNCERTAINTY

An LRTA* agent is guaranteed to find a goal in any finite, safely explorable environment. Unlike A*, however, it is not complete for infinite state spaces—there are cases where it can be led infinitely astray. It can explore an environment of n states in $O(n^2)$ steps in the worst case, but often does much better. The LRTA* agent is just one of a large family of online agents that can be defined by specifying the action selection rule and the update rule in different ways. We will discuss this family, which was developed originally for stochastic environments, in Chapter 21.

Learning in online search

The initial ignorance of online search agents provides several opportunities for learning. First, the agents learn a "map" of the environment—more precisely, the outcome of each action in each state—simply by recording each of their experiences. (Notice that the assumption of deterministic environments means that one experience is enough for each action.) Second, the local search agents acquire more accurate estimates of the value of each state by using local updating rules, as in LRTA*. In Chapter 21 we will see that these updates eventually

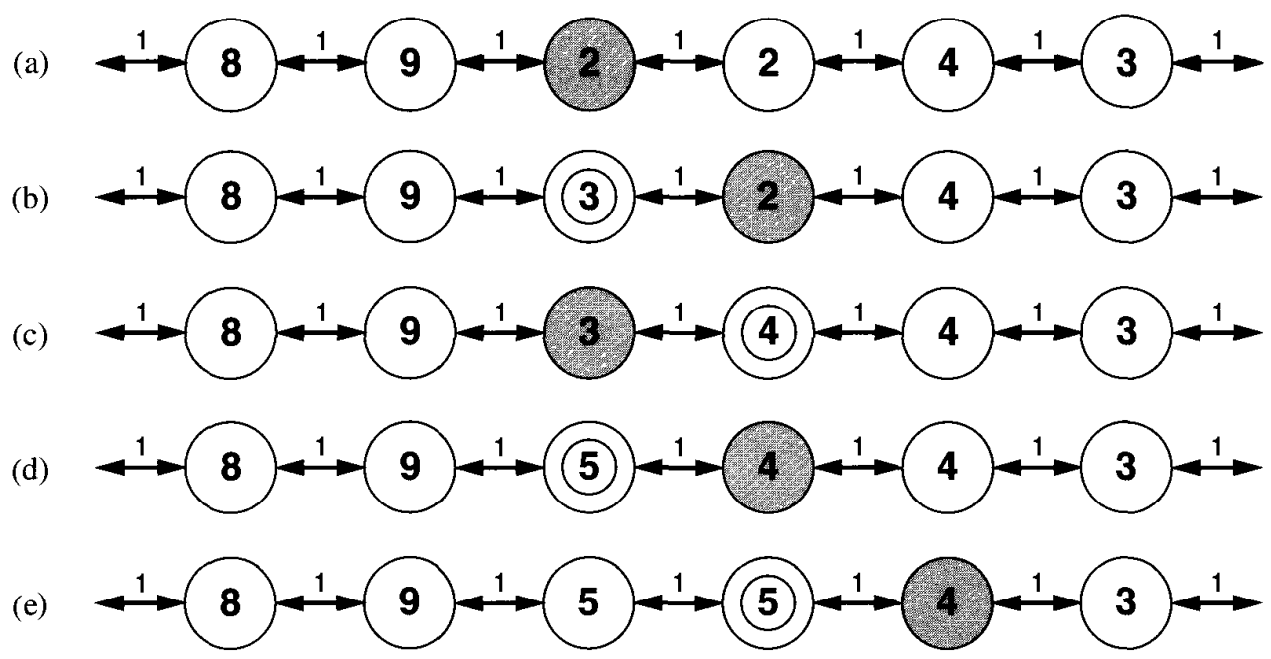


Figure 4.22 Five iterations of LRTA* on a one-dimensional state space. Each state is labeled with $H(s)$, the current cost estimate to reach a goal, and each arc is labeled with its step cost. The shaded state marks the location of the agent, and the updated values at each iteration are circled.

```
function LRTA*-AGENT(s') returns an action
  inputs: s', a percept that identifies the current state
  static: result, a table, indexed by action and state, initially empty
          H, a table of cost estimates indexed by state, initially empty
          s, a, the previous state and action, initially null

  if GOAL-TEST(s') then return stop
  if s' is a new state (not in H) then H[s'] ← h(s')
  unless s is null
      result[a, s] ← s'
      H[s] ← min_{b ∈ ACTIONS(s)} LRTA*-COST(s, b, result[b, s], H)
  a ← an action b in ACTIONS(s') that minimizes LRTA*-COST(s', b, result[b, s'], H)
  s ← s'
  return a

function LRTA*-COST(s, a, s', H) returns a cost estimate
  if s' is undefined then return h(s)
  else return c(s, a, s') + H[s']
```

Figure 4.23 LRTA*-AGENT selects an action according to the values of neighboring states, which are updated as the agent moves about the state space.

converge to *exact* values for every state, provided that the agent explores the state space in the right way. Once exact values are known, optimal decisions can be taken simply by moving to the highest-valued successor—that is, pure hill climbing is then an optimal strategy.

If you followed our suggestion to trace the behavior of ONLINE-DFS-AGENT in the environment of Figure 4.18, you will have noticed that the agent is not very bright. For example, after it has seen that the *Up* action goes from (1,1) to (1,2), the agent still has no idea that the *Down* action goes back to (1,1), or that the *Up* action also goes from (2,1) to (2,2), from (2,2) to (2,3), and so on. In general, we would like the agent to learn that *Up* increases the y-coordinate unless there is a wall in the way, that *Down* reduces it, and so on. For this to happen, we need two things. First, we need a formal and explicitly manipulable representation for these kinds of general rules; so far, we have hidden the information inside the black box called the successor function. Part III is devoted to this issue. Second, we need algorithms that can construct suitable general rules from the specific observations made by the agent. These are covered in Chapter 18.

4.6 SUMMARY

This chapter has examined the application of **heuristics** to reduce search costs. We have looked at a number of algorithms that use heuristics and found that optimality comes at a stiff price in terms of search cost, even with good heuristics.

- **Best-first search** is just GRAPH-SEARCH where the minimum-cost unexpanded nodes (according to some measure) are selected for expansion. Best-first algorithms typically use a **heuristic** function $h(n)$ that estimates the cost of a solution from n.
- **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 we guarantee that $h(n)$ is admissible (for TREE-SEARCH) or consistent (for GRAPH-SEARCH). The space complexity of A* is still prohibitive.
- The performance of heuristic search algorithms depends on the quality of the heuristic function. Good heuristics can sometimes be constructed by relaxing the problem definition, by precomputing solution costs for subproblems in a pattern database, or by learning from experience with the problem class.
- **RBFS** and **SMA*** are robust, optimal search algorithms that use limited amounts of memory; given enough time, they can solve problems that A* cannot solve because it runs out of memory.
- *Local search* methods such as **hill climbing** operate on complete-state formulations, keeping only a small number of nodes in memory. Several stochastic algorithms have been developed, including **simulated annealing**, which returns optimal solutions when given an appropriate cooling schedule. Many local search methods can also be used to solve problems in continuous spaces.

- A **genetic algorithm** is a stochastic hill-climbing search in which a large population of states is maintained. New states are generated by **mutation** and by **crossover**, which combines pairs of states from the population.
- **Exploration problems** arise when the agent has no idea about the states and actions of its environment. For safely explorable environments, **online search** agents can build a map and find a goal if one exists. Updating heuristic estimates from experience provides an effective method to escape from local minima.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

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 as applied to a number of problems, especially the 8-puzzle and the 15-puzzle. Although Doran and Michie carried out theoretical analyses of path length and "penetrance" (the ratio of path length to the total number of nodes examined so far) in heuristic search, they appear to have ignored the information provided by current path length. The A* algorithm, incorporating the current path length into heuristic search, was developed by Hart, Nilsson, and Raphael (1968), with some later corrections (Hart *et al.*, 1972). Dechter and Pearl (1985) demonstrated the optimal efficiency of A*.

The original A* paper 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. A number of algorithms predating A* used the equivalent of open and closed lists; these include breadth-first, depth-first, and uniform-cost search (Bellman, 1957; Dijkstra, 1959). Bellman's work in particular showed the importance of additive path costs in simplifying optimization algorithms.

Pohl (1970, 1977) pioneered the study of the relationship between the error in heuristic functions and the time complexity of A*. The proof that A* runs in linear time if the error in the heuristic function is bounded by a constant can be found in Pohl (1977) and in Gaschnig (1979). Pearl (1984) strengthened this result to allow a logarithmic growth in the error. The "effective branching factor" measure of the efficiency of heuristic search was proposed by Nilsson (1971).

There are many variations on the A* algorithm. Pohl (1973) proposed the use of *dynamic weighting*, which uses a weighted sum $f_w(n) = w_g g(n) + w_h h(n)$ of the current path length and the heuristic function as an evaluation function, rather than the simple sum $f(n) = g(n) + h(n)$ used in A*. The weights w_g and w_h are adjusted dynamically as the search progresses. Pohl's algorithm can be shown to be ϵ-admissible—that is, guaranteed to find solutions within a factor $1 + \epsilon$ of the optimal solution—where ϵ is a parameter supplied to the algorithm. The same property is exhibited by the A^*_ϵ algorithm (Pearl, 1984), which can select any node from the fringe provided its f-cost is within a factor $1 + \epsilon$ of the lowest-f-cost fringe node. The selection can be done so as to minimize search cost.

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). The relationships between state-space search and branch-and-bound have been investigated in depth (Kumar and Kanal, 1983; Nau *et al.*, 1984; Kumar *et al.*, 1988). Martelli and Montanari (1978) demonstrate a connection between dynamic programming (see Chapter 17) and certain types of state-space search. Kumar and Kanal (1988) attempt a "grand unification" of heuristic search, dynamic programming, and branch-and-bound techniques under the name of CDP—the "composite decision process."

Because computers in the late 1950s and early 1960s had at most 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 operator after searching best first up to the memory limit. IDA* (Korf, 1985a, 1985b) was the first widely used 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).

RBFS (Korf, 1991, 1993) is actually somewhat more complicated than the algorithm shown in Figure 4.5, which is 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 (1986) 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.

ITERATIVE EXPANSION

The MA* algorithm appeared in Chakrabarti *et al.* (1989). SMA*, or Simplified MA*, emerged from an attempt to implement MA* as a comparison algorithm for IE (Russell, 1992). Kaindl and Khorsand (1994) have applied SMA* to produce a bidirectional search algorithm that is 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. Korf (1995) surveys memory-bounded search techniques.

The idea that admissible heuristics can be derived by problem relaxation appears in the seminal paper by Held and Karp (1970), who used the the minimum-spanning-tree heuristic to solve the TSP. (See Exercise 4.8.)

The automation of the relaxation process was implemented successfully by Prieditis (1993), building on earlier work with Mostow (Mostow and Prieditis, 1989). The use of pattern databases to derive admissible heuristics is due to Gasser (1995) and Culberson and Schaeffer (1998); disjoint pattern databases are described by Korf and Felner (2002). The probabilistic interpretation of heuristics was investigated in depth by Pearl (1984) and Hansson and Mayer (1989).

By far the most comprehensive source on heuristics and heuristic search algorithms is Pearl's (1984) *Heuristics* text. This book provides especially good coverage of the wide variety of offshoots and variations of A*, including rigorous proofs of their formal properties. Kanal and Kumar (1988) present an anthology of important articles on heuristic search. New results on search algorithms appear regularly in the journal *Artificial Intelligence*.

Local-search techniques have a long history in mathematics and computer science. Indeed, the Newton–Raphson method (Newton, 1671; Raphson, 1690) can be seen as a very efficient local-search method for continuous spaces in which gradient information is available. Brent (1973) is a classic reference for optimization algorithms that do not require such information. Beam search, which we have presented as a local-search algorithm, originated as a bounded-width variant of dynamic programming for speech recognition in the HARPY system (Lowerre, 1976). A related algorithm is analyzed in depth by Pearl (1984, Ch. 5).

The topic of local search has been reinvigorated in recent years by surprisingly good results for large constraint satisfaction problems such as n-queens (Minton *et al.*, 1992) and logical reasoning (Selman *et al.*, 1992) and by the incorporation of randomness, multiple simultaneous searches, and other improvements. This renaissance of what Christos Papadimitriou has called "New Age" algorithms has also sparked increased interest among theoretical computer scientists (Koutsoupias and Papadimitriou, 1992; Aldous and Vazirani, 1994).

TABU SEARCH

In the field of operations research, a variant of hill climbing called **tabu search** has gained popularity (Glover, 1989; Glover and Laguna, 1997). Drawing on models of limited short-term memory in humans, this algorithm maintains a tabu list of k previously visited states that cannot be revisited; as well as improving efficiency when searching graphs, this can allow the algorithm to escape from some local minima. Another useful improvement on hill climbing is the STAGE algorithm (Boyan and Moore, 1998). The idea is to use the local maxima found by random-restart hill climbing to get an idea of the overall shape of the landscape. The algorithm fits a smooth surface to the set of local maxima and then calculates the global maximum of that surface analytically. This becomes the new restart point. The algorithm has been shown to work in practice on hard problems. (Gomes *et al.*, 1998) showed that the run time distributions of systematic backtracking algorithms often have a **heavy-tailed distribution**, which means that the probability of a very long run time is more than would be predicted if the run times were normally distributed. This provides a theoretical justification for random restarts.

HEAVY-TAILED DISTRIBUTION

Simulated annealing was first described by Kirkpatrick *et al.* (1983), who borrowed directly from the **Metropolis algorithm** (which is used to simulate complex systems in physics (Metropolis *et al.*, 1953) and was supposedly invented at a Los Alamos dinner party). Simulated annealing is now a field in itself, with hundreds of papers published every year.

Finding optimal solutions in continuous spaces is the subject matter of several fields, including **optimization theory**, **optimal control theory**, and the **calculus of variations**. Suitable (and practical) entry points are provided by Press *et al.* (2002) and Bishop (1995). **Linear programming** (LP) was one of the first applications of computers; the **simplex algorithm** (Wood and Dantzig, 1949; Dantzig, 1949) is still used despite worst-case exponential complexity. Karmarkar (1984) developed a practical polynomial-time algorithm for LP.

Work by Sewall Wright (1931) on the concept of a **fitness landscape** was an important precursor to the development of genetic algorithms. In the 1950s, several statisticians, including Box (1957) and Friedman (1959), used evolutionary techniques for optimization problems, but it wasn't until Rechenberg (1965, 1973) introduced **evolution strategies** to solve optimization problems for airfoils that the approach gained popularity. In the 1960s and 1970s, John Holland (1975) championed genetic algorithms, both as a useful tool and

EVOLUTION STRATEGIES

as a method to expand our understanding of adaptation, biological or otherwise (Holland, 1995). The **artificial life** movement (Langton, 1995) takes this idea one step further, viewing the products of genetic algorithms as *organisms* rather than solutions to problems. Work in this field by Hinton and Nowlan (1987) and Ackley and Littman (1991) has done much to clarify the implications of the Baldwin effect. For general background on evolution, we strongly recommend Smith and Szathmáry (1999).

ARTIFICIAL LIFE

Most comparisons of genetic algorithms to other approaches (especially stochastic hill-climbing) have found that the genetic algorithms are slower to converge (O'Reilly and Oppacher, 1994; Mitchell *et al.*, 1996; Juels and Wattenberg, 1996; Baluja, 1997). Such findings are not universally popular within the GA community, but recent attempts within that community to understand population-based search as an approximate form of Bayesian learning (see Chapter 20) might help to close the gap between the field and its critics (Pelikan *et al.*, 1999). The theory of **quadratic dynamical systems** may also explain the performance of GAs (Rabani *et al.*, 1998). See Lohn *et al.* (2001) for an example of GAs applied to antenna design, and Larrañaga *et al.* (1999) for an application to the traveling salesperson problem.

The field of **genetic programming** is closely related to genetic algorithms. The principal difference is that the representations that are mutated and combined are programs rather than bit strings. The programs are represented in the form of expression trees; the expressions can be in a standard language such as Lisp or can be specially designed to represent circuits, robot controllers, and so on. Crossover involves splicing together subtrees rather than substrings. This form of mutation guarantees that the offspring are well-formed expressions, which would not be the case if programs were manipulated as strings.

GENETIC PROGRAMMING

Recent interest in genetic programming was spurred by John Koza's work (Koza, 1992, 1994), but it goes back at least to early experiments with machine code by Friedberg (1958) and with finite-state automata by Fogel *et al.* (1966). As with genetic algorithms, there is debate about the effectiveness of the technique. Koza *et al.* (1999) describe a variety of experiments on the automated design of circuit devices using genetic programming.

The journals *Evolutionary Computation* and *IEEE Transactions on Evolutionary Computation* cover genetic algorithms and genetic programming; articles are also found in *Complex Systems*, *Adaptive Behavior*, and *Artificial Life*. The main conferences are the *International Conference on Genetic Algorithms* and the *Conference on Genetic Programming*, recently merged to form the *Genetic and Evolutionary Computation Conference*. The texts by Melanie Mitchell (1996) and David Fogel (2000) give good overviews of the field.

Algorithms for exploring unknown state spaces have been of interest for many centuries. Depth-first search in a maze can be implemented by keeping one's left hand on the wall; loops can be avoided by marking each junction. Depth-first search fails with irreversible actions; the more general problem of exploring of **Eulerian graphs** (i.e., graphs in which each node has equal numbers of incoming and outgoing edges) was solved by an algorithm due to Hierholzer (1873). The first thorough algorithmic study of the exploration problem for arbitrary graphs was carried out by Deng and Papadimitriou (1990), who developed a completely general algorithm, but showed that no bounded competitive ratio is possible for exploring a general graph. Papadimitriou and Yannakakis (1991) examined the question of finding paths to a goal in geometric path-planning environments (where all actions are reversible). They showed that

EULERIAN GRAPHS

a small competitive ratio is achievable with square obstacles, but with general rectangular obstacles no bounded ratio can be achieved. (See Figure 4.19.)

REAL-TIME SEARCH

The LRTA* algorithm was developed by Korf (1990) as part of an investigation into **real-time search** for environments in which the agent must act after searching for only a fixed amount of time (a much more common situation in two-player games). LRTA* is in fact a special case of reinforcement learning algorithms for stochastic environments (Barto *et al.*, 1995). Its policy of optimism under uncertainty—always head for the closest unvisited state—can result in an exploration pattern that is less efficient in the uninformed case than simple depth-first search (Koenig, 2000). Dasgupta *et al.* (1994) show that online iterative deepening search is optimally efficient for finding a goal in a uniform tree with no heuristic information. Several informed variants on the LRTA* theme have been developed with different methods for searching and updating within the known portion of the graph (Pemberton and Korf, 1992). As yet, there is no good understanding of how to find goals with optimal efficiency when using heuristic information.

PARALLEL SEARCH

The topic of **parallel search** algorithms was not covered in the chapter, partly because it requires a lengthy discussion of parallel computer architectures. Parallel search is becoming an important topic in both AI and theoretical computer science. A brief introduction to the AI literature can be found in Mahanti and Daniels (1993).

EXERCISES

4.1 Trace the operation of A* search applied to the problem of getting to Bucharest from Lugoj using the straight-line distance heuristic. That is, show the sequence of nodes that the algorithm will consider and the f, g, and h score for each node.

4.2 The **heuristic path algorithm** is a best-first search in which the objective function is $f(n) = (2 - w)g(n) + wh(n)$. For what values of w is this algorithm guaranteed to be optimal? (You may assume that h is admissable.) What kind of search does this perform when $w = 0$? When $w = 1$? When $w = 2$?

4.3 Prove each of the following statements:

a. Breadth-first search is a special case of uniform-cost search.

b. Breadth-first search, depth-first search, and uniform-cost search are special cases of best-first search.

c. Uniform-cost search is a special case of A* search.

4.4 Devise a state space in which A* using GRAPH-SEARCH returns a suboptimal solution with an $h(n)$ function that is admissible but inconsistent.

4.5 We saw on page 96 that the straight-line distance heuristic leads greedy best-first search astray on the problem of going from Iasi to Fagaras. However, the heuristic is perfect on the opposite problem: going from Fagaras to Iasi. Are there problems for which the heuristic is misleading in both directions?

4.6 Invent a heuristic function for the 8-puzzle that sometimes overestimates, and show how it can lead to a suboptimal solution on a particular problem. (You can use a computer to help if you want.) Prove that, if h never overestimates by more than c, A* using h returns a solution whose cost exceeds that of the optimal solution by no more than c.

4.7 Prove that if a heuristic is consistent, it must be admissible. Construct an admissible heuristic that is not consistent.

4.8 The traveling salesperson problem (TSP) can be solved via the minimum spanning tree (MST) heuristic, which is used to estimate the cost of completing a tour, given that a partial tour has already been constructed. The MST cost of a set of cities is the smallest sum of the link costs of any tree that connects all the cities.

a. Show how this heuristic can be derived from a relaxed version of the TSP.

b. Show that the MST heuristic dominates straight-line distance.

c. Write a problem generator for instances of the TSP where cities are represented by random points in the unit square.

d. Find an efficient algorithm in the literature for constructing the MST, and use it with an admissible search algorithm to solve instances of the TSP.

4.9 On page 108, we defined the relaxation of the 8-puzzle in which a tile can move from square A to square B if B is blank. The exact solution of this problem defines **Gaschnig's heuristic** (Gaschnig, 1979). Explain why Gaschnig's heuristic is at least as accurate as h_1 (misplaced tiles), and show cases where it is more accurate than both h_1 and h_2 (Manhattan distance). Can you suggest a way to calculate Gaschnig's heuristic efficiently?

4.10 We gave two simple heuristics for the 8-puzzle: Manhattan distance and misplaced tiles. Several heuristics in the literature purport to improve on this—see, for example, Nilsson (1971), Mostow and Prieditis (1989), and Hansson *et al.* (1992). Test these claims by implementing the heuristics and comparing the performance of the resulting algorithms.

4.11 Give the name of the algorithm that results from each of the following special cases:

a. Local beam search with $k = 1$.

b. Local beam search with one initial state and no limit on the number of states retained.

c. Simulated annealing with $T = 0$ at all times (and omitting the termination test).

d. Genetic algorithm with population size $N = 1$.

4.12 Sometimes there is no good evaluation function for a problem, but there is a good comparison method: a way to tell whether one node is better than another, without assigning numerical values to either. Show that this is enough to do a best-first search. Is there an analog of A*?

4.13 Relate the time complexity of LRTA* to its space complexity.

4.14 Suppose that an agent is in a 3×3 maze environment like the one shown in Figure 4.18. The agent knows that its initial location is (1,1), that the goal is at (3,3), and that the

four actions *Up*, *Down*, *Left*, *Right* have their usual effects unless blocked by a wall. The agent does *not* know where the internal walls are. In any given state, the agent perceives the set of legal actions; it can also tell whether the state is one it has visited before or a new state.

a. Explain how this online search problem can be viewed as an offline search in belief state space, where the initial belief state includes all possible environment configurations. How large is the initial belief state? How large is the space of belief states?

b. How many distinct percepts are possible in the initial state?

c. Describe the first few branches of a contingency plan for this problem. How large (roughly) is the complete plan?

Notice that this contingency plan is a solution for *every possible environment* fitting the given description. Therefore, interleaving of search and execution is not strictly necessary even in unknown environments.

4.15 In this exercise, we will explore the use of local search methods to solve TSPs of the type defined in Exercise 4.8.

a. Devise a hill-climbing approach to solve TSPs. Compare the results with optimal solutions obtained via the A* algorithm with the MST heuristic (Exercise 4.8).

b. Devise a genetic algorithm approach to the traveling salesperson problem. Compare results to the other approaches. You may want to consult Larrañaga *et al.* (1999) for some suggestions for representations.

4.16 Generate a large number of 8-puzzle and 8-queens instances and solve them (where possible) by hill climbing (steepest-ascent and first-choice variants), hill climbing with random restart, and simulated annealing. Measure the search cost and percentage of solved problems and graph these against the optimal solution cost. Comment on your results.

4.17 In this exercise, we will examine hill climbing in the context of robot navigation, using the environment in Figure 3.22 as an example.

a. Repeat Exercise 3.16 using hill climbing. Does your agent ever get stuck in a local minimum? Is it *possible* for it to get stuck with convex obstacles?

b. Construct a nonconvex polygonal environment in which the agent gets stuck.

c. Modify the hill-climbing algorithm so that, instead of doing a depth-1 search to decide where to go next, it does a depth-k search. It should find the best k-step path and do one step along it, and then repeat the process.

d. Is there some k for which the new algorithm is guaranteed to escape from local minima?

e. Explain how LRTA* enables the agent to escape from local minima in this case.

4.18 Compare the performance of A* and RBFS on a set of randomly generated problems in the 8-puzzle (with Manhattan distance) and TSP (with MST—see Exercise 4.8) domains. Discuss your results. What happens to the performance of RBFS when a small random number is added to the heuristic values in the 8-puzzle domain?

5 CONSTRAINT SATISFACTION PROBLEMS

In which we see how treating states as more than just little black boxes leads to the invention of a range of powerful new search methods and a deeper understanding of problem structure and complexity.

Chapters 3 and 4 explored the idea that problems can be solved by searching in a space of **states**. These states can be evaluated by domain-specific heuristics and tested to see whether they are goal states. From the point of view of the search algorithm, however, each state is a **black box** with no discernible internal structure. It is represented by an arbitrary data structure that can be accessed only by the *problem-specific* routines—the successor function, heuristic function, and goal test.

BLACK BOX

This chapter examines **constraint satisfaction problems**, whose states and goal test conform to a standard, structured, and very simple **representation** (Section 5.1). Search algorithms can be defined that take advantage of the structure of states and use *general-purpose* rather than *problem-specific* heuristics to enable the solution of large problems (Sections 5.2–5.3). Perhaps most importantly, the standard representation of the goal test reveals the structure of the problem itself (Section 5.4). This leads to methods for problem decomposition and to an understanding of the intimate connection between the structure of a problem and the difficulty of solving it.

REPRESENTATION

5.1 CONSTRAINT SATISFACTION PROBLEMS

CONSTRAINT SATISFACTION PROBLEM

VARIABLES

CONSTRAINTS

DOMAIN

VALUES

ASSIGNMENT

CONSISTENT

OBJECTIVE FUNCTION

Formally speaking, a **constraint satisfaction problem** (or CSP) is defined by a set of **variables**, $X_1, X_2, \ldots, X_n$, and a set of **constraints**, $C_1, C_2, \ldots, C_m$. Each variable X_i has a nonempty **domain** D_i of possible **values**. Each constraint C_i involves some subset of the variables and specifies the allowable combinations of values for that subset. A state of the problem is defined by an **assignment** of values to some or all of the variables, $\{X_i = v_i, X_j = v_j, \ldots\}$. An assignment that does not violate any constraints is called a **consistent** or legal assignment. A complete assignment is one in which every variable is mentioned, and a **solution** to a CSP is a complete assignment that satisfies all the constraints. Some CSPs also require a solution that maximizes an **objective function**.

So what does all this mean? Suppose that, having tired of Romania, we are looking at a map of Australia showing each of its states and territories, as in Figure 5.1(a), and that we are given the task of coloring each region either red, green, or blue in such a way that no neighboring regions have the same color. To formulate this as a CSP, we define the variables to be the regions: WA, NT, Q, NSW, V, SA, and T. The domain of each variable is the set $\{red, green, blue\}$. The constraints require neighboring regions to have distinct colors; for example, the allowable combinations for WA and NT are the pairs

$$\{(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)\} .$$

(The constraint can also be represented more succinctly as the inequality $WA \neq NT$, provided the constraint satisfaction algorithm has some way to evaluate such expressions.) There are many possible solutions, such as

$$\{ WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = red \}.$$

CONSTRAINT GRAPH

It is helpful to visualize a CSP as a **constraint graph**, as shown in Figure 5.1(b). The nodes of the graph correspond to variables of the problem and the arcs correspond to constraints.

Treating a problem as a CSP confers several important benefits. Because the representation of states in a CSP conforms to a *standard* pattern—that is, a set of variables with assigned values—the successor function and goal test can written in a generic way that applies to all CSPs. Furthermore, we can develop effective, generic heuristics that require no additional, domain-specific expertise. Finally, the structure of the constraint graph can be used to simplify the solution process, in some cases giving an exponential reduction in complexity. The CSP representation is the first, and simplest, in a series of representation schemes that will be developed throughout the book.

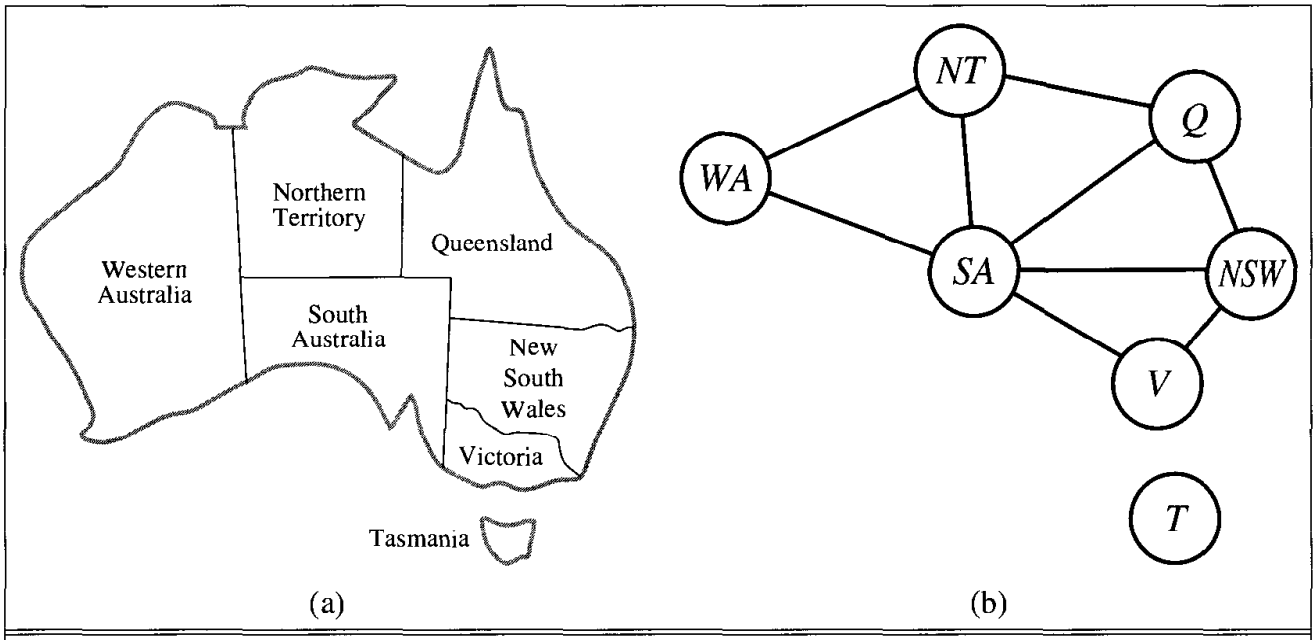


Figure 5.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem. The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

It is fairly easy to see that a CSP can be given an **incremental formulation** as a standard search problem as follows:

- ◇ **Initial state**: the empty assignment {}, in which all variables are unassigned.
- ◇ **Successor function**: a value can be assigned to any unassigned variable, provided that it does not conflict with previously assigned variables.
- ◇ **Goal test**: the current assignment is complete.
- ◇ **Path cost**: a constant cost (e.g., 1) for every step.

Every solution must be a complete assignment and therefore appears at depth n if there are n variables. Furthermore, the search tree extends only to depth n. For these reasons, depth-first search algorithms are popular for CSPs. (See Section 5.2.) It is also the case that *the path by which a solution is reached is irrelevant.* Hence, we can also use a **complete-state formulation**, in which every state is a complete assignment that might or might not satisfy the constraints. Local search methods work well for this formulation. (See Section 5.3.)

FINITE DOMAINS

The simplest kind of CSP involves variables that are **discrete** and have **finite domains**. Map-coloring problems are of this kind. The 8-queens problem described in Chapter 3 can also be viewed as a finite-domain CSP, where the variables $Q_1, \ldots, Q_8$ are the positions of each queen in columns $1, \ldots, 8$ and each variable has the domain $\{1, 2, 3, 4, 5, 6, 7, 8\}$. If the maximum domain size of any variable in a CSP is d, then the number of possible complete assignments is $O(d^n)$—that is, exponential in the number of variables. Finite-domain CSPs include **Boolean CSPs**, whose variables can be either *true* or *false*. Boolean CSPs include as special cases some NP-complete problems, such as 3SAT. (See Chapter 7.) In the worst case, therefore, we cannot expect to solve finite-domain CSPs in less than exponential time. In most practical applications, however, general-purpose CSP algorithms can solve problems *orders of magnitude* larger than those solvable via the general-purpose search algorithms that we saw in Chapter 3.

BOOLEAN CSPS

INFINITE DOMAINS

Discrete variables can also have **infinite domains**—for example, the set of integers or the set of strings. For example, when scheduling construction jobs onto a calendar, each job's start date is a variable and the possible values are integer numbers of days from the current date. With infinite domains, it is no longer possible to describe constraints by enumerating all allowed combinations of values. Instead, a **constraint language** must be used. For example, if Job_1, which takes five days, must precede Job_3, then we would need a constraint language of algebraic inequalities such as $StartJob_1 + 5 \leq StartJob_3$. It is also no longer possible to solve such constraints by enumerating all possible assignments, because there are infinitely many of them. Special solution algorithms (which we will not discuss here) exist for **linear constraints** on integer variables—that is, constraints, such as the one just given, in which each variable appears only in linear form. It can be shown that no algorithm exists for solving general **nonlinear constraints** on integer variables. In some cases, we can reduce integer constraint problems to finite-domain problems simply by bounding the values of all the variables. For example, in a scheduling problem, we can set an upper bound equal to the total length of all the jobs to be scheduled.

CONSTRAINT LANGUAGE

LINEAR CONSTRAINTS

NONLINEAR CONSTRAINTS

CONTINUOUS DOMAINS

Constraint satisfaction problems with **continuous domains** are very common in the real world and are widely studied in the field of operations research. For example, the scheduling

of experiments on the Hubble Space Telescope requires very precise timing of observations; the start and finish of each observation and maneuver are continuous-valued variables that must obey a variety of astronomical, precedence, and power constraints. The best-known category of continuous-domain CSPs is that of **linear programming** problems, where constraints must be linear inequalities forming a *convex* region. Linear programming problems can be solved in time polynomial in the number of variables. Problems with different types of constraints and objective functions have also been studied—quadratic programming, second-order conic programming, and so on.

LINEAR PROGRAMMING

In addition to examining the types of variables that can appear in CSPs, it is useful to look at the types of constraints. The simplest type is the **unary constraint**, which restricts the value of a single variable. For example, it could be the case that South Australians actively dislike the color *green*. Every unary constraint can be eliminated simply by preprocessing the domain of the corresponding variable to remove any value that violates the constraint. A **binary constraint** relates two variables. For example, $SA \neq NSW$ is a binary constraint. A binary CSP is one with only binary constraints; it can be represented as a constraint graph, as in Figure 5.1(b).

UNARY CONSTRAINT

BINARY CONSTRAINT

Higher-order constraints involve three or more variables. A familiar example is provided by **cryptarithmetic** puzzles. (See Figure 5.2(a).) It is usual to insist that each letter in a cryptarithmetic puzzle represent a different digit. For the case in Figure 5.2(a)), this would be represented as the six-variable constraint $Alldiff(F, T, U, W, R, O)$. Alternatively, it can be represented by a collection of binary constraints such as $F \neq T$. The addition constraints on the four columns of the puzzle also involve several variables and can be written as

CRYPTARITHMETIC

$$
\begin{aligned}
O + O &= R + 10 \cdot X_1 \\
X_1 + W + W &= U + 10 \cdot X_2 \\
X_2 + T + T &= O + 10 \cdot X_3 \\
X_3 &= F
\end{aligned}
$$

where X_1, X_2, and X_3 are **auxiliary variables** representing the digit (0 or 1) carried over into the next column. Higher-order constraints can be represented in a **constraint hypergraph**, such as the one shown in Figure 5.2(b). The sharp-eyed reader will have noticed that the *Alldiff* constraint can be broken down into binary constraints—$F \neq T$, $F \neq U$, and so on. In fact, as Exercise 5.11 asks you to prove, every higher-order, finite-domain constraint can be reduced to a set of binary constraints if enough auxiliary variables are introduced. Because of this, we will deal only with binary constraints in this chapter.

AUXILIARY VARIABLES

CONSTRAINT HYPERGRAPH

The constraints we have described so far have all been **absolute** constraints, violation of which rules out a potential solution. Many real-world CSPs include **preference** constraints indicating which solutions are preferred. For example, in a university timetabling problem, Prof. X might prefer teaching in the morning whereas Prof. Y prefers teaching in the afternoon. A timetable that has Prof. X teaching at 2 p.m. would still be a solution (unless Prof. X happens to be the department chair), but would not be an optimal one. Preference constraints can often be encoded as costs on individual variable assignments—for example, assigning an afternoon slot for Prof. X costs 2 points against the overall objective function, whereas a morning slot costs 1. With this formulation, CSPs with preferences can be solved using opti-

PREFERENCE

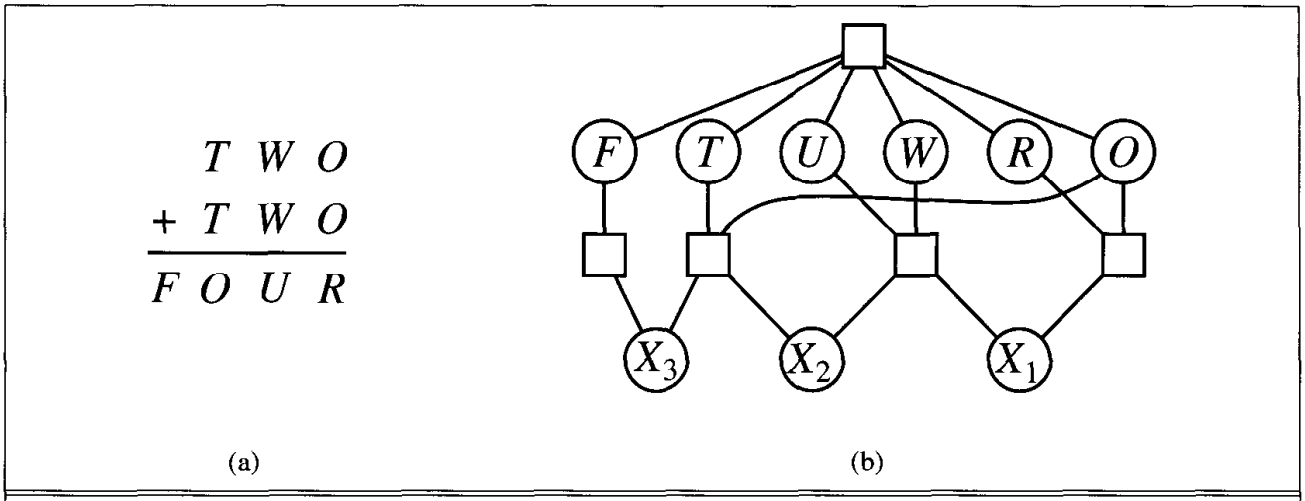


Figure 5.2 (a) A cryptarithmetic problem. Each letter stands for a distinct digit; the aim is to find a substitution of digits for letters such that the resulting sum is arithmetically correct, with the added restriction that no leading zeroes are allowed. (b) The constraint hypergraph for the cryptarithmetic problem, showing the *Alldiff* constraint as well as the column addition constraints. Each constraint is a square box connected to the variables it constrains.

mization search methods, either path-based or local. We do not discuss such CSPs further in this chapter, but we provide some pointers in the bibliographical notes section.

5.2 BACKTRACKING SEARCH FOR CSPS

The preceding section gave a formulation of CSPs as search problems. Using this formulation, any of the search algorithms from Chapters 3 and 4 can solve CSPs. Suppose we apply breadth-first search to the generic CSP problem formulation given in the preceding section. We quickly notice something terrible: the branching factor at the top level is nd, because any of d values can be assigned to any of n variables. At the next level, the branching factor is $(n-1)d$, and so on for n levels. We generate a tree with $n! \cdot d^n$ leaves, even though there are only d^n possible complete assignments!

COMMUTATIVITY

Our seemingly reasonable but naïve problem formulation has ignored a crucial property common to all CSPs: **commutativity**. A problem is commutative if the order of application of any given set of actions has no effect on the outcome. This is the case for CSPs because, when assigning values to variables, we reach the same partial assignment, regardless of order. Therefore, *all CSP search algorithms generate successors by considering possible assignments for only a* single *variable at each node in the search tree.* For example, at the root node of a search tree for coloring the map of Australia, we might have a choice between $SA = red$, $SA = green$, and $SA = blue$, but we would never choose between $SA = red$ and $WA = blue$. With this restriction, the number of leaves is d^n, as we would hope.

BACKTRACKING SEARCH

The term **backtracking search** is used for a depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign. The algorithm is shown in Figure 5.3. Notice that it uses, in effect, the one-at-a-time method of

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment according to CONSTRAINTS[csp] then
      add {var = value} to assignment
      result ← RECURSIVE-BACKTRACKING(assignment, csp)
      if result ≠ failure then return result
      remove {var = value} from assignment
  return failure
```

Figure 5.3 A simple backtracking algorithm for constraint satisfaction problems. The algorithm is modeled on the recursive depth-first search of Chapter 3. The functions SELECT-UNASSIGNED-VARIABLE and ORDER-DOMAIN-VALUES can be used to implement the general-purpose heuristics discussed in the text.

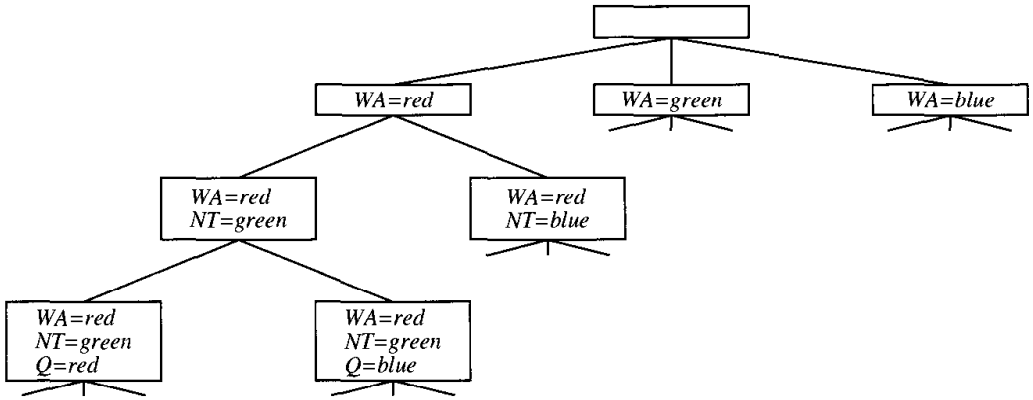


Figure 5.4 Part of the search tree generated by simple backtracking for the map-coloring problem in Figure 5.1.

incremental successor generation described on page 76. Also, it extends the current assignment to generate a successor, rather than copying it. Because the representation of CSPs is standardized, there is no need to supply BACKTRACKING-SEARCH with a domain-specific initial state, successor function, or goal test. Part of the search tree for the Australia problem is shown in Figure 5.4, where we have assigned variables in the order $WA, NT, Q, \ldots$.

Plain backtracking is an uninformed algorithm in the terminology of Chapter 3, so we do not expect it to be very effective for large problems. The results for some sample problems are shown in the first column of Figure 5.5 and confirm our expectations.

In Chapter 4 we remedied the poor performance of uninformed search algorithms by supplying them with domain-specific heuristic functions derived from our knowledge of the problem. It turns out that we can solve CSPs efficiently without such domain-specific knowl-

Problem	Backtracking	BT+MRV	Forward Checking	FC+MRV	Min-Conflicts
USA	(> 1,000K)	(> 1,000K)	2K	60	64
n-Queens	(> 40,000K)	13,500K	(> 40,000K)	817K	4K
Zebra	3,859K	1K	35K	0.5K	2K
Random 1	415K	3K	26K	2K	
Random 2	942K	27K	77K	15K	

Figure 5.5 Comparison of various CSP algorithms on various problems. The algorithms from left to right, are simple backtracking, backtracking with the MRV heuristic, forward checking, forward checking with MRV, and minimum conflicts local search. Listed in each cell is the median number of consistency checks (over five runs) required to solve the problem; note that all entries except the two in the upper right are in thousands (K). Numbers in parentheses mean that no answer was found in the allotted number of checks. The first problem is finding a 4-coloring for the 50 states of the United States of America. The remaining problems are taken from Bacchus and van Run (1995), Table 1. The second problem counts the total number of checks required to solve all n-Queens problems for n from 2 to 50. The third is the "Zebra Puzzle," as described in Exercise 5.13. The last two are artificial random problems. (Min-conflicts was not run on these.) The results suggest that forward checking with the MRV heuristic is better on all these problems than the other backtracking algorithms, but not always better than min-conflicts local search.

edge. Instead, we find general-purpose methods that address the following questions:

1. Which variable should be assigned next, and in what order should its values be tried?
2. What are the implications of the current variable assignments for the other unassigned variables?
3. When a path fails—that is, a state is reached in which a variable has no legal values—can the search avoid repeating this failure in subsequent paths?

The subsections that follow answer each of these questions in turn.

Variable and value ordering

The backtracking algorithm contains the line

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

By default, SELECT-UNASSIGNED-VARIABLE simply selects the next unassigned variable in the order given by the list VARIABLES[csp]. This static variable ordering seldom results in the most efficient search. For example, after the assignments for $WA = red$ and $NT = green$, there is only one possible value for SA, so it makes sense to assign $SA = blue$ next rather than assigning Q. In fact, after SA is assigned, the choices for Q, NSW, and V are all forced. This intuitive idea—choosing the variable with the fewest "legal" values—is called the **minimum remaining values** (MRV) heuristic. It also has been called the "most constrained variable" or "fail-first" heuristic, the latter because it picks a variable that is most likely to cause a failure soon, thereby pruning the search tree. If there is a variable X with zero legal values remaining, the MRV heuristic will select X and failure will be detected immediately—avoiding pointless searches through other variables which always will fail when X is finally selected.

MINIMUM REMAINING VALUES

The second column of Figure 5.5, labeled BT+MRV, shows the performance of this heuristic. The performance is 3 to 3,000 times better than simple backtracking, depending on the problem. Note that our performance measure ignores the extra cost of computing the heuristic values; the next subsection describes a method that makes this cost manageable.

DEGREE HEURISTIC

The MRV heuristic doesn't help at all in choosing the first region to color in Australia, because initially every region has three legal colors. In this case, the **degree heuristic** comes in handy. It attempts to reduce the branching factor on future choices by selecting the variable that is involved in the largest number of constraints on other unassigned variables. In Figure 5.1, SA is the variable with highest degree, 5; the other variables have degree 2 or 3, except for T, which has 0. In fact, once SA is chosen, applying the degree heuristic solves the problem without any false steps—you can choose any consistent color at each choice point and still arrive at a solution with no backtracking. The minimum remaining values heuristic is usually a more powerful guide, but the degree heuristic can be useful as a tie-breaker.

LEAST-CONSTRAINING-VALUE

Once a variable has been selected, the algorithm must decide on the order in which to examine its values. For this, the **least-constraining-value** heuristic can be effective in some cases. It prefers the value that rules out the fewest choices for the neighboring variables in the constraint graph. For example, suppose that in Figure 5.1 we have generated the partial assignment with $WA = red$ and $NT = green$, and that our next choice is for Q. Blue would be a bad choice, because it eliminates the last legal value left for Q's neighbor, SA. The least-constraining-value heuristic therefore prefers red to blue. In general, the heuristic is trying to leave the maximum flexibility for subsequent variable assignments. Of course, if we are trying to find all the solutions to a problem, not just the first one, then the ordering does not matter because we have to consider every value anyway. The same holds if there are no solutions to the problem.

Propagating information through constraints

So far our search algorithm considers the constraints on a variable only at the time that the variable is chosen by SELECT-UNASSIGNED-VARIABLE. But by looking at some of the constraints earlier in the search, or even before the search has started, we can drastically reduce the search space.

Forward checking

FORWARD CHECKING

One way to make better use of constraints during search is called **forward checking**. Whenever a variable X is assigned, the forward checking process looks at each unassigned variable Y that is connected to X by a constraint and deletes from Y's domain any value that is inconsistent with the value chosen for X. Figure 5.6 shows the progress of a map-coloring search with forward checking. There are two important points to notice about this example. First, notice that after assigning $WA = red$ and $Q = green$, the domains of NT and SA are reduced to a single value; we have eliminated branching on these variables altogether by propagating information from WA and Q. The MRV heuristic, which is an obvious partner for forward checking, would automatically select SA and NT next. (Indeed, we can view forward checking as an efficient way to incrementally compute the information that the

	WA	*NT*	*Q*	*NSW*	*V*	*SA*	*T*
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After *WA*=*red*	(R)	G B	R G B	R G B	R G B	G B	R G B
After *Q*=*green*	(R)	B	(G)	R B	R G B	B	R G B
After *V*=*blue*	(R)	B	(G)	R	(B)		R G B

Figure 5.6 The progress of a map-coloring search with forward checking. $WA = red$ is assigned first; then forward checking deletes red from the domains of the neighboring variables NT and SA. After $Q = green$, $green$ is deleted from the domains of NT, SA, and NSW. After $V = blue$, $blue$ is deleted from the domains of NSW and SA, leaving SA with no legal values.

MRV heuristic needs to do its job.) A second point to notice is that, after $V = blue$, the domain of SA is empty. Hence, forward checking has detected that the partial assignment $\{WA = red, Q = green, V = blue\}$ is inconsistent with the constraints of the problem, and the algorithm will therefore backtrack immediately.

Constraint propagation

Although forward checking detects many inconsistencies, it does not detect all of them. For example, consider the third row of Figure 5.6. It shows that when WA is red and Q is $green$, both NT and SA are forced to be blue. But they are adjacent and so cannot have the same value. Forward checking does not detect this as an inconsistency, because it does not look far enough ahead. **Constraint propagation** is the general term for propagating the implications of a constraint on one variable onto other variables; in this case we need to propagate from WA and Q onto NT and SA, (as was done by forward checking) and then onto the constraint between NT and SA to detect the inconsistency. And we want to do this fast: it is no good reducing the amount of search if we spend more time propagating constraints than we would have spent doing a simple search.

CONSTRAINT PROPAGATION

ARC CONSISTENCY

The idea of **arc consistency** provides a fast method of constraint propagation that is substantially stronger than forward checking. Here, "arc" refers to a *directed* arc in the constraint graph, such as the arc from SA to NSW. Given the current domains of SA and NSW, the arc is consistent if, for *every* value x of SA, there is *some* value y of NSW that is consistent with x. In the third row of Figure 5.6, the current domains of SA and NSW are $\{blue\}$ and $\{red, blue\}$ respectively. For $SA = blue$, there is a consistent assignment for NSW, namely, $NSW = red$; therefore, the arc from SA to NSW is consistent. On the other hand, the reverse arc from NSW to SA is not consistent: for the assignment $NSW = blue$, there is no consistent assignment for SA. The arc can be made consistent by deleting the value $blue$ from the domain of NSW.

We can also apply arc consistency to the arc from SA to NT at the same stage in the search process. The third row of the table in Figure 5.6 shows that both variables have the domain $\{blue\}$. The result is that $blue$ must be deleted from the domain of SA, leaving the domain empty. Thus, applying arc consistency has resulted in early detection of an inconsis-

```
function AC-3(csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables {X_1, X_2, ..., X_n}
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (X_i, X_j) ← REMOVE-FIRST(queue)
    if REMOVE-INCONSISTENT-VALUES(X_i, X_j) then
      for each X_k in NEIGHBORS[X_i] do
        add (X_k, X_i) to queue

function REMOVE-INCONSISTENT-VALUES(X_i, X_j) returns true iff we remove a value
  removed ← false
  for each x in DOMAIN[X_i] do
    if no value y in DOMAIN[X_j] allows (x,y) to satisfy the constraint between X_i and X_j
      then delete x from DOMAIN[X_i]; removed ← true
  return removed
```

Figure 5.7 The arc consistency algorithm AC-3. After applying AC-3, either every arc is arc-consistent, or some variable has an empty domain, indicating that the CSP cannot be made arc-consistent (and thus the CSP cannot be solved). The name "AC-3" was used by the algorithm's inventor (Mackworth, 1977) because it's the third version developed in the paper.

tency that is not detected by pure forward checking.

Arc consistency checking can be applied either as a preprocessing step before the beginning of the search process, or as a propagation step (like forward checking) after every assignment during search. (The latter algorithm is sometimes called MAC, for *Maintaining Arc Consistency.*) In either case, the process must be applied *repeatedly* until no more inconsistencies remain. This is because, whenever a value is deleted from some variable's domain to remove an arc inconsistency, a new arc inconsistency could arise in arcs pointing to that variable. The full algorithm for arc consistency, AC-3, uses a queue to keep track of the arcs that need to be checked for inconsistency. (See Figure 5.7.) Each arc (X_i, X_j) in turn is removed from the agenda and checked; if any values need to be deleted from the domain of X_i, then every arc (X_k, X_i) pointing to X_i must be reinserted on the queue for checking. The complexity of arc consistency checking can be analyzed as follows: a binary CSP has at most $O(n^2)$ arcs; each arc (X_k, X_i) can be inserted on the agenda only d times, because X_i has at most d values to delete; checking consistency of an arc can be done in $O(d^2)$ time; so the total worst-case time is $O(n^2 d^3)$. Although this is substantially more expensive than forward checking, the extra cost is usually worthwhile.[1]

Because CSPs include 3SAT as a special case, we do not expect to find a polynomial-time algorithm that can decide whether a given CSP is consistent. Hence, we deduce that arc consistency does not reveal every possible inconsistency. For example, in Figure 5.1, the partial assignment $\{WA = red, NSW = red\}$ is inconsistent, but AC-3 will not find the incon-

[1] The AC-4 algorithm, due to Mohr and Henderson (1986), runs in $O(n^2 d^2)$. See Exercise 5.10.

K-CONSISTENCY

sistency. Stronger forms of propagation can be defined using the notion called **k-consistency**. A CSP is k-consistent if, for any set of $k-1$ variables and for any consistent assignment to those variables, a consistent value can always be assigned to any kth variable. For example, 1-consistency means that each individual variable by itself is consistent; this is also called

NODE CONSISTENCY

node consistency. 2-consistency is the same as arc consistency. 3-consistency means that any pair of adjacent variables can always be extended to a third neighboring variable; this is

PATH CONSISTENCY

also called **path consistency**.

STRONGLY K-CONSISTENT

A graph is **strongly k-consistent** if it is k-consistent and is also $(k-1)$-consistent, $(k-2)$-consistent, ... all the way down to 1-consistent. Now suppose we have a CSP problem with n nodes and make it strongly n-consistent (i.e., strongly k-consistent for $k=n$). We can then solve the problem with no backtracking. First, we choose a consistent value for X_1. We are then guaranteed to be able to choose a value for X_2 because the graph is 2-consistent, for X_3 because it is 3-consistent, and so on. For each variable X_i, we need only search through the d values in the domain to find a value consistent with $X_1, \ldots, X_{i-1}$. We are guaranteed to find a solution in time $O(nd)$. Of course, there is no free lunch: any algorithm for establishing n-consistency must take time exponential in n in the worst case.

There is a broad middle ground between n-consistency and arc consistency: running stronger consistency checks will take more time, but will have a greater effect in reducing the branching factor and detecting inconsistent partial assignments. It is possible to calculate the smallest value k such that running k-consistency ensures that the problem can be solved without backtracking (see Section 5.4), but this is often impractical. In practice, determining the appropriate level of consistency checking is mostly an empirical science.

Handling special constraints

Certain types of constraints occur frequently in real problems and can be handled using special-purpose algorithms that are more efficient than the general-purpose methods described so far. For example, the *Alldiff* constraint says that all the variables involved must have distinct values (as in the cryptarithmetic problem). One simple form of inconsistency detection for *Alldiff* constraints works as follows: if there are m variables involved in the constraint, and if they have n possible distinct values altogether, and $m > n$, then the constraint cannot be satisfied.

This leads to the following simple algorithm: First, remove any variable in the constraint that has a singleton domain, and delete that variable's value from the domains of the remaining variables. Repeat as long as there are singleton variables. If at any point an empty domain is produced or there are more variables than domain values left, then an inconsistency has been detected.

We can use this method to detect the inconsistency in the partial assignment $\{WA = red, NSW = red\}$ for Figure 5.1. Notice that the variables SA, NT, and Q are effectively connected by an *Alldiff* constraint because each pair must be a different color. After applying AC-3 with the partial assignment, the domain of each variable is reduced to $\{green, blue\}$. That is, we have three variables and only two colors, so the *Alldiff* constraint is violated. Thus, a simple consistency procedure for a higher-order constraint is sometimes more effec-

tive than applying arc consistency to an equivalent set of binary constraints.

RESOURCE CONSTRAINT

Perhaps the most important higher-order constraint is the **resource constraint**, sometimes called the *atmost* constraint. For example, let $PA_1, \ldots, PA_4$ denote the numbers of personnel assigned to each of four tasks. The constraint that no more than 10 personnel are assigned in total is written as $atmost(10, PA_1, PA_2, PA_3, PA_4)$. An inconsistency can be detected simply by checking the sum of the minimum values of the current domains; for example, if each variable has the domain $\{3, 4, 5, 6\}$, the *atmost* constraint cannot be satisfied. We can also enforce consistency by deleting the maximum value of any domain if it is not consistent with the minimum values of the other domains. Thus, if each variable in our example has the domain $\{2, 3, 4, 5, 6\}$, the values 5 and 6 can be deleted from each domain.

For large resource-limited problems with integer values—such as logistical problems involving moving thousands of people in hundreds of vehicles—it is usually not possible to represent the domain of each variable as a large set of integers and gradually reduce that set by consistency checking methods. Instead, domains are represented by upper and lower bounds and are managed by bounds propagation. For example, let's suppose there are two flights, 271 and 272, for which the planes have capacities 165 and 385, respectively. The initial domains for the numbers of passengers on each flight are then

$$Flight271 \in [0, 165] \quad \text{and} \quad Flight272 \in [0, 385] \, .$$

Now suppose we have the additional constraint that the two flights together must carry 420 people: $Flight271 + Flight272 \in [420, 420]$. Propagating bounds constraints, we reduce the domains to

$$Flight271 \in [35, 165] \quad \text{and} \quad Flight272 \in [255, 385] \, .$$

We say that a CSP is bounds-consistent if for every variable X, and for both the lower bound and upper bound values of X, there exists some value of Y that satisfies the constraint between X and Y, for every variable Y. This kind of **bounds propagation** is widely used in practical constraint problems.

BOUNDS PROPAGATION

Intelligent backtracking: looking backward

The BACKTRACKING-SEARCH algorithm in Figure 5.3 has a very simple policy for what to do when a branch of the search fails: back up to the preceding variable and try a different value for it. This is called **chronological backtracking**, because the *most recent* decision point is revisited. In this subsection, we will see that there are much better ways.

CHRONOLOGICAL BACKTRACKING

Consider what happens when we apply simple backtracking in Figure 5.1 with a fixed variable ordering Q, NSW, V, T, SA, WA, NT. Suppose we have generated the partial assignment $\{Q = red, NSW = green, V = blue, T = red\}$. When we try the next variable, SA, we see that every value violates a constraint. We back up to T and try a new color for Tasmania! Obviously this is silly—recoloring Tasmania cannot resolve the problem with South Australia.

A more intelligent approach to backtracking is to go all the way back to one of the set of variables that *caused the failure*. This set is called the **conflict set**; here, the conflict set for SA is $\{Q, NSW, V\}$. In general, the conflict set for variable X is the set of previously assigned variables that are connected to X by constraints. The **backjumping** method

CONFLICT SET

BACKJUMPING

backtracks to the *most recent* variable in the conflict set; in this case, backjumping would jump over Tasmania and try a new value for V. This is easily implemented by modifying BACKTRACKING-SEARCH so that it accumulates the conflict set while checking for a legal value to assign. If no legal value is found, it should return the most recent element of the conflict set along with the failure indicator.

The sharp-eyed reader will have noticed that forward checking can supply the conflict set with no extra work: whenever forward checking based on an assignment to X deletes a value from Y's domain, it should add X to Y's conflict set. Also, every time the last value is deleted from Y's domain, the variables in the conflict set of Y are added to the conflict set of X. Then, when we get to Y, we know immediately where to backtrack if needed.

The eagle-eyed reader will have noticed something odd: backjumping occurs when every value in a domain is in conflict with the current assignment; but forward checking detects this event and prevents the search from ever reaching such a node! In fact, it can be shown that *every branch pruned by backjumping is also pruned by forward checking.* Hence, simple backjumping is redundant in a forward-checking search or, indeed, in a search that uses stronger consistency checking, such as MAC.

Despite the observations of the preceding paragraph, the idea behind backjumping remains a good one: to backtrack based on the reasons for failure. Backjumping notices failure when a variable's domain becomes empty, but in many cases a branch is doomed long before this occurs. Consider again the partial assignment $\{WA = red, NSW = red\}$ (which, from our earlier discussion, is inconsistent). Suppose we try $T = red$ next and then assign NT, Q, V, SA. We know that no assignment can work for these last four variables, so eventually we run out of values to try at NT. Now, the question is, where to backtrack? Backjumping cannot work, because NT *does* have values consistent with the preceding assigned variables—NT doesn't have a complete conflict set of preceding variables that caused it to fail. We know, however, that the four variables NT, Q, V, and SA, *taken together*, failed because of a set of preceding variables, which must be those variables which directly conflict with the four. This leads to a deeper notion of the conflict set for a variable such as NT: it is that set of preceding variables that caused NT, *together with any subsequent variables*, to have no consistent solution. In this case, the set is WA and NSW, so the algorithm should backtrack to NSW and skip over Tasmania. A backjumping algorithm that uses conflict sets defined in this way is called **conflict-directed backjumping**.

CONFLICT-DIRECTED BACKJUMPING

We must now explain how these new conflict sets are computed. The method is in fact very simple. The "terminal" failure of a branch of the search always occurs because a variable's domain becomes empty; that variable has a standard conflict set. In our example, SA fails, and its conflict set is (say) $\{WA, NT, Q\}$. We backjump to Q, and Q *absorbs* the conflict set from SA (minus Q itself, of course) into its own direct conflict set, which is $\{NT, NSW\}$; the new conflict set is $\{WA, NT, NSW\}$. That is, there is no solution from Q onwards, given the preceding assignment to $\{WA, NT, NSW\}$. Therefore, we backtrack to NT, the most recent of these. NT absorbs $\{WA, NT, NSW\} - \{NT\}$ into its own direct conflict set $\{WA\}$, giving $\{WA, NSW\}$ (as stated in the previous paragraph). Now the algorithm backjumps to NSW, as we would hope. To summarize: let X_j be the current variable, and let $conf(X_j)$ be its conflict set. If every possible value for X_j fails, backjump

to the most recent variable X_i in $conf(X_j)$, and set

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

CONSTRAINT LEARNING

Conflict-directed backjumping takes us back to the right point in the search tree, but doesn't prevent us from making the same mistakes in another branch of the tree. **Constraint learning** actually modifies the CSP by adding a new constraint that is induced from these conflicts.

5.3 Local Search for Constraint Satisfaction Problems

Local-search algorithms (see Section 4.3) turn out to be very effective in solving many CSPs. They use a complete-state formulation: the initial state assigns a value to every variable, and the successor function usually works by changing the value of one variable at a time. For example, in the 8-queens problem, the initial state might be a random configuration of 8 queens in 8 columns, and the successor function picks one queen and considers moving it elsewhere in its column. Another possibility would be start with the 8 queens, one per column in a permutation of the 8 rows, and to generate a successor by having two queens swap rows.[2] We have actually already seen an example of local search for CSP solving: the application of hill climbing to the n-queens problem (page 112). The application of WALKSAT (page 223) to solve satisfiability problems, which are a special case of CSPs, is another.

MIN-CONFLICTS

In choosing a new value for a variable, the most obvious heuristic is to select the value that results in the minimum number of conflicts with other variables—the **min-conflicts** heuristic. The algorithm is shown in Figure 5.8 and its application to an 8-queens problem is diagrammed in Figure 5.9 and quantified in Figure 5.5.

Min-conflicts is surprisingly effective for many CSPs, particularly when given a reasonable initial state. Its performance is shown in the last column of Figure 5.5. Amazingly, on the n-queens problem, if you don't count the initial placement of queens, the runtime of min-conflicts is roughly *independent of problem size*. It solves even the *million*-queens problem in an average of 50 steps (after the initial assignment). This remarkable observation was the stimulus leading to a great deal of research in the 1990s on local search and the distinction between easy and hard problems, which we take up in Chapter 7. Roughly speaking, n-queens is easy for local search because solutions are densely distributed throughout the state space. Min-conflicts also works well for hard problems. For example, it has been used to schedule observations for the Hubble Space Telescope, reducing the time taken to schedule a week of observations from three weeks (!) to around 10 minutes.

Another advantage of local search is that it can be used in an online setting when the problem changes. This is particularly important in scheduling problems. A week's airline schedule may involve thousands of flights and tens of thousands of personnel assignments, but bad weather at one airport can render the schedule infeasible. We would like to repair the schedule with a minimum number of changes. This can be easily done with a local search algorithm starting from the current schedule. A backtracking search with the new set of

[2] Local search can easily be extended to CSPs with objective functions. In that case, all the techniques for hill climbing and simulated annealing can be applied to optimize the objective function.

```
function MIN-CONFLICTS(csp, max_steps) returns a solution or failure
  inputs: csp, a constraint satisfaction problem
          max_steps, the number of steps allowed before giving up

  current ← an initial complete assignment for csp
  for i = 1 to max_steps do
      if current is a solution for csp then return current
      var ← a randomly chosen, conflicted variable from VARIABLES[csp]
      value ← the value v for var that minimizes CONFLICTS(var, v, current, csp)
      set var = value in current
  return failure
```

Figure 5.8 The MIN-CONFLICTS algorithm for solving CSPs by local search. The initial state may be chosen randomly or by a greedy assignment process that chooses a minimal-conflict value for each variable in turn. The CONFLICTS function counts the number of constraints violated by a particular value, given the rest of the current assignment.

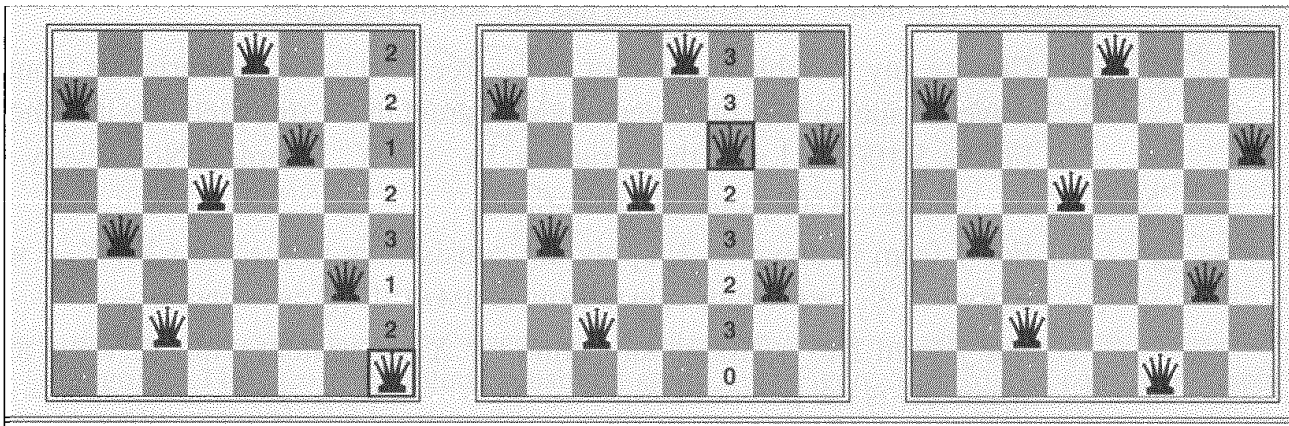


Figure 5.9 A two-step solution for an 8-queens problem using min-conflicts. At each stage, a queen is chosen for reassignment in its column. The number of conflicts (in this case, the number of attacking queens) is shown in each square. The algorithm moves the queen to the min-conflict square, breaking ties randomly.

constraints usually requires much more time and might find a solution with many changes from the current schedule.

5.4 THE STRUCTURE OF PROBLEMS

In this section, we examine ways in which the *structure* of the problem, as represented by the constraint graph, can be used to find solutions quickly. Most of the approaches here are very general and are applicable to other problems besides CSPs, for example probabilistic reasoning. After all, the only way we can possibly hope to deal with the real world is to decompose it into many subproblems. Looking again at Figure 5.1(b) with a view to identifying problem

INDEPENDENT SUBPROBLEMS

CONNECTED COMPONENTS

structure, one fact stands out: Tasmania is not connected to the mainland.[3] Intuitively, it is obvious that coloring Tasmania and coloring the mainland are **independent subproblems**—any solution for the mainland combined with any solution for Tasmania yields a solution for the whole map. Independence can be ascertained simply by looking for **connected components** of the constraint graph. Each component corresponds to a subproblem CSP_i. If assignment S_i is a solution of CSP_i, then $\bigcup_i S_i$ is a solution of $\bigcup_i CSP_i$. Why is this important? Consider the following: suppose each CSP_i has c variables from the total of n variables, where c is a constant. Then there are n/c subproblems, each of which takes at most d^c work to solve. Hence, the total work is $O(d^c n/c)$, which is *linear* in n; without the decomposition, the total work is $O(d^n)$, which is exponential in n. Let's make this more concrete: dividing a Boolean CSP with $n = 80$ into four subproblems with $c = 20$ reduces the worst-case solution time from the lifetime of the universe down to less than a second.

Completely independent subproblems are delicious, then, but rare. In most cases, the subproblems of a CSP are connected. The simplest case is when the constraint graph forms a **tree**: any two variables are connected by at most one path. Figure 5.10(a) shows a schematic example.[4] We will show that *any tree-structured CSP can be solved in time linear in the number of variables.* The algorithm has the following steps:

1. Choose any variable as the root of the tree, and order the variables from the root to the leaves in such a way that every node's parent in the tree precedes it in the ordering. (See Figure 5.10(b).) Label the variables $X_1, \ldots, X_n$ in order. Now, every variable except the root has exactly one parent variable.
2. For j from n down to 2, apply arc consistency to the arc (X_i, X_j), where X_i is the parent of X_j, removing values from DOMAIN$[X_i]$ as necessary.
3. For j from 1 to n, assign any value for X_j consistent with the value assigned for X_i, where X_i is the parent of X_j.

There are two key points to note. First, after step 2 the CSP is directionally arc-consistent, so the assignment of values in step 3 requires no backtracking. (See the discussion of k-consistency on page 147.) Second, by applying the arc-consistency checks in reverse order in step 2, the algorithm ensures that any deleted values cannot endanger the consistency of arcs that have been processed already. The complete algorithm runs in time $O(nd^2)$.

Now that we have an efficient algorithm for trees, we can consider whether more general constraint graphs can be *reduced* to trees somehow. There are two primary ways to do this, one based on removing nodes and one based on collapsing nodes together.

The first approach involves assigning values to some variables so that the remaining variables form a tree. Consider the constraint graph for Australia, shown again in Figure 5.11(a). If we could delete South Australia, the graph would become a tree, as in (b). Fortunately, we can do this (in the graph, not the continent) by fixing a value for SA and deleting from the domains of the other variables any values that are inconsistent with the value chosen for SA.

[3] A careful cartographer or patriotic Tasmanian might object that Tasmania should not be colored the same as its nearest mainland neighbor, to avoid the impression that it *might* be part of that state.

[4] Sadly, very few regions of the world, with the possible exception of Sulawesi, have tree-structured maps.

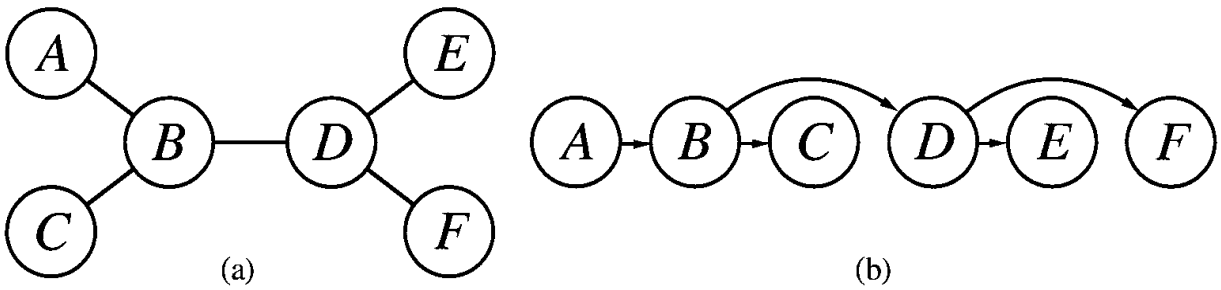


Figure 5.10 (a) The constraint graph of a tree-structured CSP. (b) A linear ordering of the variables consistent with the tree with A as the root.

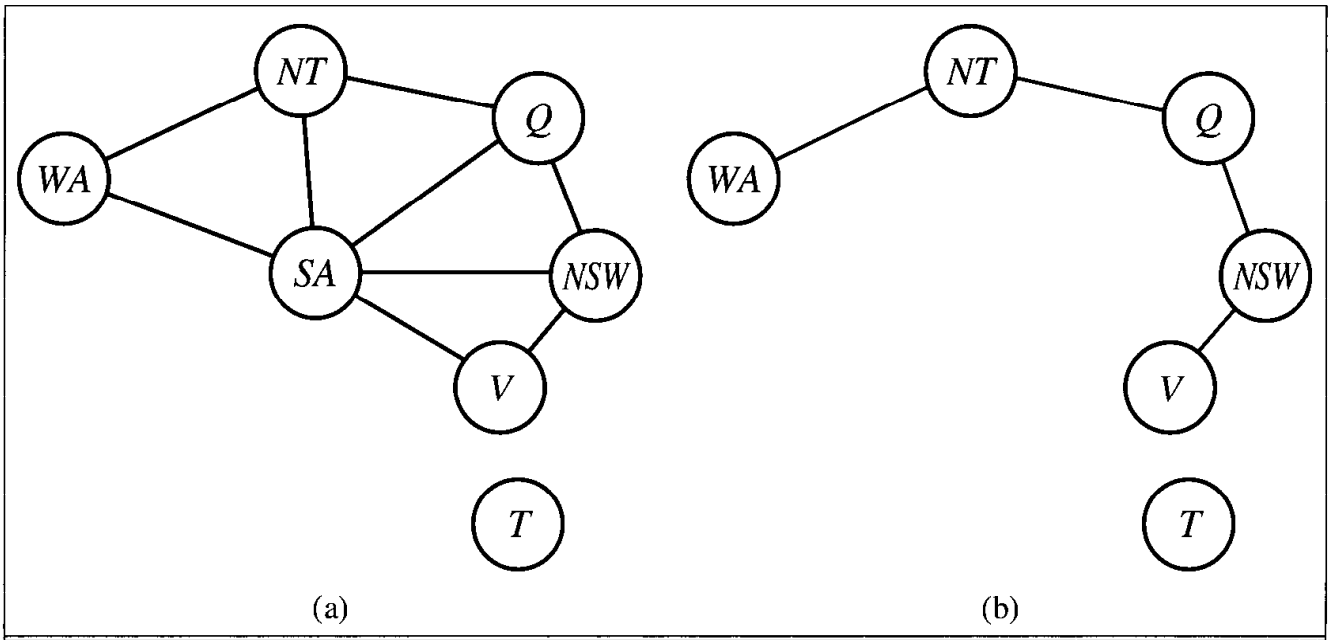


Figure 5.11 (a) The original constraint graph from Figure 5.1. (b) The constraint graph after the removal of SA.

Now, any solution for the CSP after SA and its constraints are removed will be consistent with the value chosen for SA. (This works for binary CSPs; the situation is more complicated with higher-order constraints.) Therefore, we can solve the remaining tree with the algorithm given above and thus solve the whole problem. Of course, in the general case (as opposed to map coloring) the value chosen for SA could be the wrong one, so we would need to try each of them. The general algorithm is as follows:

CYCLE CUTSET

1. Choose a subset S from VARIABLES[csp] such that the constraint graph becomes a tree after removal of S. S is called a **cycle cutset**.
2. For each possible assignment to the variables in S that satisfies all constraints on S,
   (a) remove from the domains of the remaining variables any values that are inconsistent with the assignment for S, and
   (b) If the remaining CSP has a solution, return it together with the assignment for S.

If the cycle cutset has size c, then the total runtime is $O(d^c \cdot (n-c)d^2)$. If the graph is "nearly a tree" then c will be small and the savings over straight backtracking will be huge.

In the worst case, however, c can be as large as $(n-2)$. Finding the *smallest* cycle cutset is NP-hard, but several efficient approximation algorithms are known for this task. The overall algorithmic approach is called **cutset conditioning**; we will see it again in Chapter 14, where it is used for reasoning about probabilities.

CUTSET CONDITIONING

TREE DECOMPOSITION

The second approach is based on constructing a **tree decomposition** of the constraint graph into a set of connected subproblems. Each subproblem is solved independently, and the resulting solutions are then combined. Like most divide-and-conquer algorithms, this works well if no subproblem is too large. Figure 5.12 shows a tree decomposition of the map-coloring problem into five subproblems. A tree decomposition must satisfy the following three requirements:

- Every variable in the original problem appears in at least one of the subproblems.
- If two variables are connected by a constraint in the original problem, they must appear together (along with the constraint) in at least one of the subproblems.
- If a variable appears in two subproblems in the tree, it must appear in every subproblem along the path connecting those subproblems.

The first two conditions ensure that all the variables and constraints are represented in the decomposition. The third condition seems rather technical, but simply reflects the constraint that any given variable must have the same value in every subproblem in which it appears; the links joining subproblems in the tree enforce this constraint. For example, SA appears in all four of the connected subproblems in Figure 5.12. You can verify from Figure 5.11 that this decomposition makes sense.

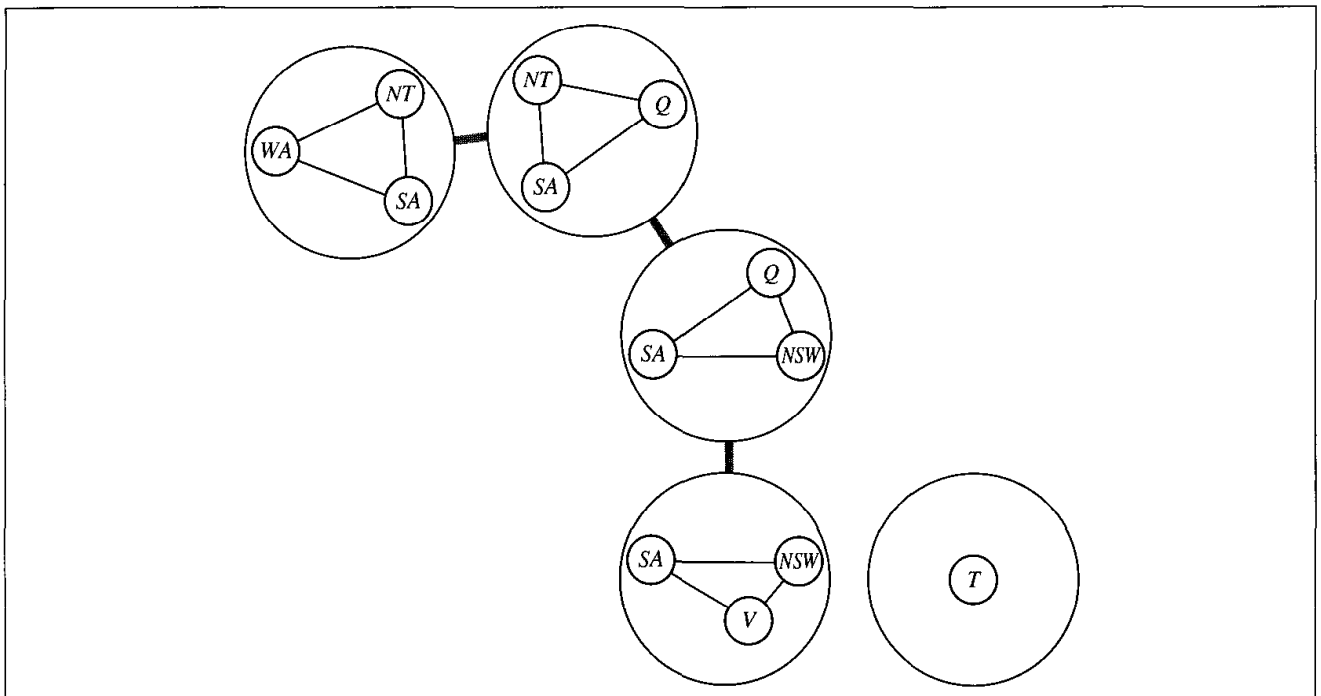


Figure 5.12 A tree decomposition of the constraint graph in Figure 5.11(a).

We solve each subproblem independently; if any one has no solution, we know the entire problem has no solution. If we can solve all the subproblems, then we attempt to construct

a global solution as follows. First, we view each subproblem as a "mega-variable" whose domain is the set of all solutions for the subproblem. For example, the leftmost subproblems in Figure 5.12 is a map-coloring problem with three variables and hence has six solutions—one is $\{WA = red, SA = blue, NT = green\}$. Then, we solve the constraints connecting the subproblems using the efficient algorithm for trees given earlier. The constraints between subproblems simply insist that the subproblem solutions agree on their shared variables. For example, given the solution $\{WA = red, SA = blue, NT = green\}$ for the first subproblem, the only consistent solution for the next subproblem is $\{SA = blue, NT = green, Q = red\}$.

TREE WIDTH

A given constraint graph admits many tree decompositions; in choosing a decomposition, the aim is to make the subproblems as small as possible. The **tree width** of a tree decomposition of a graph is one less than the size of the largest subproblem; the tree width of the graph itself is defined to be the minimum tree width among all its tree decompositions. If a graph has tree width w, and we are given the corresponding tree decomposition, then the problem can be solved in $O(nd^{w+1})$ time. Hence, *CSPs with constraint graphs of bounded tree width are solvable in polynomial time.* Unfortunately, finding the decomposition with minimal tree width is NP-hard, but there are heuristic methods that work well in practice.

5.5 SUMMARY

- **Constraint satisfaction problems** (or CSPs) consist of variables with constraints on them. Many important real-world problems can be described as CSPs. The structure of a CSP can be represented by its **constraint graph**.
- **Backtracking search**, a form of depth-first search, is commonly used for solving CSPs.
- The **minimum remaining values** and **degree** heuristics are domain-independent methods for deciding which variable to choose next in a backtracking search. The **least-constraining-value** heuristic helps in ordering the variable values.
- By propagating the consequences of the partial assignments that it constructs, the backtracking algorithm can reduce greatly the branching factor of the problem. **Forward checking** is the simplest method for doing this. **Arc consistency** enforcement is a more powerful technique, but can be more expensive to run.
- Backtracking occurs when no legal assignment can be found for a variable. **Conflict-directed backjumping** backtracks directly to the source of the problem.
- Local search using the **min-conflicts** heuristic has been applied to constraint satisfaction problems with great success.
- The complexity of solving a CSP is strongly related to the structure of its constraint graph. Tree-structured problems can be solved in linear time. **Cutset conditioning** can reduce a general CSP to a tree-structured one and is very efficient if a small cutset can be found. **Tree decomposition** techniques transform the CSP into a tree of subproblems and are efficient if the **tree width** of the constraint graph is small.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

DIOPHANTINE EQUATIONS

The earliest work related to constraint satisfaction dealt largely with numerical constraints. Equational constraints with integer domains were studied by the Indian mathematician Brahmagupta in the seventh century; they are often called **Diophantine equations**, after the Greek mathematician Diophantus (c. 200–284), who actually considered the domain of positive rationals. Systematic methods for solving linear equations by variable elimination were studied by Gauss (1829); the solution of linear inequality constraints goes back to Fourier (1827).

GRAPH COLORING

Finite-domain constraint satisfaction problems also have a long history. For example, **graph coloring** (of which map coloring is a special case) is an old problem in mathematics. According to Biggs *et al.* (1986), the four-color conjecture (that every planar graph can be colored with four or fewer colors) was first made by Francis Guthrie, a student of De Morgan, in 1852. It resisted solution—despite several published claims to the contrary—until a proof was devised, with the aid of a computer, by Appel and Haken (1977).

Specific classes of constraint satisfaction problems occur throughout the history of computer science. One of the most influential early examples was the SKETCHPAD system (Sutherland, 1963), which solved geometric constraints in diagrams and was the forerunner of modern drawing programs and CAD tools. The identification of CSPs as a *general* class is due to Ugo Montanari (1974). The reduction of higher-order CSPs to purely binary CSPs with auxiliary variables (see Exercise 5.11) is due originally to the 19th-century logician Charles Sanders Peirce. It was introduced into the CSP literature by Dechter (1990b) and was elaborated by Bacchus and van Beek (1998). CSPs with preferences among solutions are studied widely in the optimization literature; see Bistarelli *et al.* (1997) for a generalization of the CSP framework to allow for preferences. The bucket-elimination algorithm (Dechter, 1999) can also be applied to optimization problems.

Backtracking search for constraint satisfaction is due to Bitner and Reingold (1975), although they trace the basic algorithm back to the 19th century. Bitner and Reingold also introduced the MRV heuristic, which they called the *most-constrained-variable* heuristic. Brelaz (1979) used the degree heuristic as a tie-breaker after applying the MRV heuristic. The resulting algorithm, despite its simplicity, is still the best method for k-coloring arbitrary graphs. Haralick and Elliot (1980) proposed the *least-constraining-value heuristic*.

Constraint propagation methods were popularized by Waltz's (1975) success on polyhedral line-labeling problems for computer vision. Waltz showed that, in many problems, propagation completely eliminates the need for backtracking. Montanari (1974) introduced the notion of constraint networks and propagation by path consistency. Alan Mackworth (1977) proposed the AC-3 algorithm for enforcing arc consistency as well as the general idea of combining backtracking with some degree of consistency enforcement. AC-4, a more efficient arc consistency algorithm, was developed by Mohr and Henderson (1986). Soon after Mackworth's paper appeared, researchers began experimenting with the tradeoff between the cost of consistency enforcement and the benefits in terms of search reduction. Haralick and Elliot (1980) favored the minimal forward checking algorithm described by McGregor (1979), whereas Gaschnig (1979) suggested full arc consistency checking after each variable

assignment—an algorithm later called MAC by Sabin and Freuder (1994). The latter paper provides somewhat convincing evidence that, on harder CSPs, full arc consistency checking pays off. Freuder (1978, 1982) investigated the notion of k-consistency and its relationship to the complexity of solving CSPs. Apt (1999) describes a generic algorithmic framework within which consistency propagation algorithms can be analyzed.

Special methods for handling higher-order constraints have been developed primarily within the context of **constraint logic programming**. Marriott and Stuckey (1998) provide excellent coverage of research in this area. The *Alldiff* constraint was studied by Regin (1994). Bounds constraints were incorporated into constraint logic programming by Van Hentenryck *et al.* (1998).

The basic backjumping method is due to John Gaschnig (1977, 1979). Kondrak and van Beek (1997) showed that this algorithm is essentially subsumed by forward checking. Conflict-directed backjumping was devised by Prosser (1993). The most general and powerful form of intelligent backtracking was actually developed very early on by Stallman and Sussman (1977). Their technique of **dependency-directed backtracking** led to the development of **truth maintenance systems** (Doyle, 1979), which we will discuss in Section 10.8. The connection between the two areas is analyzed by de Kleer (1989).

DEPENDENCY-DIRECTED BACKTRACKING

The work of Stallman and Sussman also introduced the idea of **constraint recording**, in which partial results obtained by search can be saved and reused later in the search. The idea was introduced formally into backtracking search by Dechter (1990a). **Backmarking** (Gaschnig, 1979) is a particularly simple method in which consistent and inconsistent pairwise assignments are saved and used to avoid rechecking constraints. Backmarking can be combined with conflict-directed backjumping; Kondrak and van Beek (1997) present a hybrid algorithm that provably subsumes either method taken separately. The method of **dynamic backtracking** (Ginsberg, 1993) retains successful partial assignments from later subsets of variables when backtracking over an earlier choice that does not invalidate the later success.

CONSTRAINT RECORDING

BACKMARKING

DYNAMIC BACKTRACKING

Local search in constraint satisfaction problems was popularized by the work of Kirkpatrick *et al.* (1983) on **simulated annealing** (see Chapter 4), which is widely used for scheduling problems. The *min-conflicts* heuristic was first proposed by Gu (1989) and was developed independently by Minton *et al.* (1992). Sosic and Gu (1994) showed how it could be applied to solve the 3,000,000 queens problem in less than a minute. The astounding success of local search using min-conflicts on the n-queens problem led to a reappraisal of the nature and prevalence of "easy" and "hard" problems. Peter Cheeseman *et al.* (1991) explored the difficulty of randomly generated CSPs and discovered that almost all such problems either are trivially easy or have no solutions. Only if the parameters of the problem generator are set in a certain narrow range, within which roughly half of the problems are solvable, do we find "hard" problem instances. We discuss this phenomenon further in Chapter 7.

Work relating the structure and complexity of CSPs originates with Freuder (1985), who showed that search on arc-consistent trees works without any backtracking. A similar result, with extensions to acyclic hypergraphs, was developed in the database community (Beeri *et al.*, 1983). Since those papers were published, there has been a great deal of progress in developing more general results relating the complexity of solving a CSP to the structure of

its constraint graph. The notion of tree width was introduced by the graph theorists Robertson and Seymour (1986). Dechter and Pearl (1987, 1989), building on the work of Freuder, applied the same notion (which they called **induced width**) to constraint satisfaction problems and developed the tree decomposition approach sketched in Section 5.4. Drawing on this work and on results from database theory, Gottlob *et al.* (1999a, 1999b) developed a notion, **hypertree width**, that is based on the characterization of the CSP as a hypergraph. In addition to showing that any CSP with hypertree width w can be solved in time $O(n^{w+1} \log n)$, they also showed that hypertree width subsumes all previously defined measures of "width" in the sense that there are cases where the hypertree width is bounded and the other measures are unbounded.

There are several good surveys of CSP techniques, including those by Kumar (1992), Dechter and Frost (1999), and Bartak (2001); and the encyclopedia articles by Dechter (1992) and Mackworth (1992). Pearson and Jeavons (1997) survey tractable classes of CSPs, covering both structural decomposition methods and methods that rely on properties of the domains or constraints themselves. Kondrak and van Beek (1997) give an analytical survey of backtracking search algorithms, and Bacchus and van Run (1995) give a more empirical survey. The texts by Tsang (1993) and by Marriott and Stuckey (1998) go into much more depth than has been possible in this chapter. Several interesting applications are described in the collection edited by Freuder and Mackworth (1994). Papers on constraint satisfaction appear regularly in *Artificial Intelligence* and in the specialist journal, *Constraints*. The primary conference venue is the International Conference on Principles and Practice of Constraint Programming, often called *CP*.

EXERCISES

5.1 Define in your own words the terms constraint satisfaction problem, constraint, backtracking search, arc consistency, backjumping and min-conflicts.

5.2 How many solutions are there for the map-coloring problem in Figure 5.1?

5.3 Explain why it is a good heuristic to choose the variable that is *most* constrained, but the value that is *least* constraining in a CSP search.

5.4 Consider the problem of constructing (not solving) crossword puzzles:[5] fitting words into a rectangular grid. The grid, which is given as part of the problem, specifies which squares are blank and which are shaded. Assume that a list of words (i.e., a dictionary) is provided and that the task is to fill in the blank squares using any subset of the list. Formulate this problem precisely in two ways:

a. As a general search problem. Choose an appropriate search algorithm, and specify a heuristic function, if you think one is needed. Is it better to fill in blanks one letter at a time or one word at a time?

[5] Ginsberg *et al.* (1990) discuss several methods for constructing crossword puzzles. Littman *et al.* (1999) tackle the harder problem of solving them.

b. As a constraint satisfaction problem. Should the variables be words or letters?

Which formulation do you think will be better? Why?

5.5 Give precise formulations for each of the following as constraint satisfaction problems:

FLOOR-PLANNING

a. Rectilinear **floor-planning**: find nonoverlapping places in a large rectangle for a number of smaller rectangles.

CLASS SCHEDULING

b. **Class scheduling**: There is a fixed number of professors and classrooms, a list of classes to be offered, and a list of possible time slots for classes. Each professor has a set of classes that he or she can teach.

5.6 Solve the cryptarithmetic problem in Figure 5.2 by hand, using backtracking, forward checking, and the MRV and least-constraining-value heuristics.

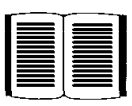
5.7 Figure 5.5 tests out various algorithms on the n-queens problem. Try these same algorithms on map-coloring problems generated randomly as follows: scatter n points on the unit square; selecting a point X at random, connect X by a straight line to the nearest point Y such that X is not already connected to Y and the line crosses no other line; repeat the previous step until no more connections are possible. Construct the performance table for the largest n you can manage, using both $d = 3$ and $d = 4$ colors. Comment on your results.

5.8 Use the AC-3 algorithm to show that arc consistency is able to detect the inconsistency of the partial assignment $\{WA = red, V = blue\}$ for the problem shown in Figure 5.1.

5.9 What is the worst-case complexity of running AC-3 on a tree-structured CSP?

5.10 AC-3 puts back on the queue *every* arc (X_k, X_i) whenever *any* value is deleted from the domain of X_i, even if each value of X_k is consistent with several remaining values of X_i. Suppose that, for every arc (X_k, X_i), we keep track of the number of remaining values of X_i that are consistent with each value of X_k. Explain how to update these numbers efficiently and hence show that arc consistency can be enforced in total time $O(n^2 d^2)$.

5.11 Show how a single ternary constraint such as "$A + B = C$" can be turned into three binary constraints by using an auxiliary variable. You may assume finite domains. (*Hint:* consider a new variable that takes on values which are pairs of other values, and consider constraints such as "X is the first element of the pair Y.") Next, show how constraints with more than three variables can be treated similarly. Finally, show how unary constraints can be eliminated by altering the domains of variables. This completes the demonstration that any CSP can be transformed into a CSP with only binary constraints.



5.12 Suppose that a graph is known to have a cycle cutset of no more than k nodes. Describe a simple algorithm for finding a minimal cycle cutset whose runtime is not much more than $O(n^k)$ for a CSP with n variables. Search the literature for methods for finding approximately minimal cycle cutsets in time that is polynomial in the size of the cutset. Does the existence of such algorithms make the cycle cutset method practical?

5.13 Consider the following logic puzzle: In five houses, each with a different color, live 5 persons of different nationalities, each of whom prefer a different brand of cigarette, a

different drink, and a different pet. Given the following facts, the question to answer is "Where does the zebra live, and in which house do they drink water?"

The Englishman lives in the red house.
The Spaniard owns the dog.
The Norwegian lives in the first house on the left.
Kools are smoked in the yellow house.
The man who smokes Chesterfields lives in the house next to the man with the fox.
The Norwegian lives next to the blue house.
The Winston smoker owns snails.
The Lucky Strike smoker drinks orange juice.
The Ukrainian drinks tea.
The Japanese smokes Parliaments.
Kools are smoked in the house next to the house where the horse is kept.
Coffee is drunk in the green house.
The Green house is immediately to the right (your right) of the ivory house.
Milk is drunk in the middle house.

Discuss different representations of this problem as a CSP. Why would one prefer one representation over another?

6 ADVERSARIAL SEARCH

In which we examine the problems that arise when we try to plan ahead in a world where other agents are planning against us.

6.1 GAMES

Chapter 2 introduced **multiagent environments**, in which any given agent will need to consider the actions of other agents and how they affect its own welfare. The unpredictability of these other agents can introduce many possible **contingencies** into the agent's problem-solving process, as discussed in Chapter 3. The distinction between **cooperative** and **competitive** multiagent environments was also introduced in Chapter 2. Competitive environments, in which the agents' goals are in conflict, give rise to **adversarial search** problems—often known as **games**.

GAMES

Mathematical **game theory**, a branch of economics, views any multiagent environment as a game provided that the impact of each agent on the others is "significant," regardless of whether the agents are cooperative or competitive.[1] In AI, "games" are usually of a rather specialized kind—what game theorists call deterministic, turn-taking, two-player, **zero-sum games** of **perfect information**. In our terminology, this means deterministic, fully observable environments in which there are two agents whose actions must alternate and in which the utility values at the end of the game are always equal and opposite. For example, if one player wins a game of chess (+1), the other player necessarily loses (−1). It is this opposition between the agents' utility functions that makes the situation adversarial. We will consider multiplayer games, non-zero-sum games, and stochastic games briefly in this chapter, but will delay discussion of game theory proper until Chapter 17.

ZERO-SUM GAMES

PERFECT INFORMATION

Games have engaged the intellectual faculties of humans—sometimes to an alarming degree—for as long as civilization has existed. For AI researchers, the abstract nature of games makes them an appealing subject for study. The state of a game is easy to represent, and agents are usually restricted to a small number of actions whose outcomes are defined by

[1] Environments with very many agents are best viewed as **economies** rather than games.

precise rules. Physical games, such as croquet and ice hockey, have much more complicated descriptions, a much larger range of possible actions, and rather imprecise rules defining the legality of actions. With the exception of robot soccer, these physical games have not attracted much interest in the AI community.

Game playing was one of the first tasks undertaken in AI. By 1950, almost as soon as computers became programmable, chess had been tackled by Konrad Zuse (the inventor of the first programmable computer and the first programming language), by Claude Shannon (the inventor of information theory), by Norbert Wiener (the creator of modern control theory), and by Alan Turing. Since then, there has been steady progress in the standard of play, to the point that machines have surpassed humans in checkers and Othello, have defeated human champions (although not every time) in chess and backgammon, and are competitive in many other games. The main exception is Go, in which computers perform at the amateur level.

Games, unlike most of the toy problems studied in Chapter 3, are interesting *because* they are too hard to solve. For example, chess has an average branching factor of about 35, and games often go to 50 moves by each player, so the search tree has about 35^{100} or 10^{154} nodes (although the search graph has "only" about 10^{40} distinct nodes). Games, like the real world, therefore require the ability to make *some* decision even when calculating the *optimal* decision is infeasible. Games also penalize inefficiency severely. Whereas an implementation of A* search that is half as efficient will simply cost twice as much to run to completion, a chess program that is half as efficient in using its available time probably will be beaten into the ground, other things being equal. Game-playing research has therefore spawned a number of interesting ideas on how to make the best possible use of time.

We begin with a definition of the optimal move and an algorithm for finding it. We then look at techniques for choosing a good move when time is limited. **Pruning** allows us to ignore portions of the search tree that make no difference to the final choice, and heuristic **evaluation functions** allow us to approximate the true utility of a state without doing a complete search. Section 6.5 discusses games such as backgammon that include an element of chance; we also discuss bridge, which includes elements of **imperfect information** because not all cards are visible to each player. Finally, we look at how state-of-the-art game-playing programs fare against human opposition and at directions for future developments.

IMPERFECT INFORMATION

6.2 Optimal Decisions in Games

We will consider games with two players, whom we will call MAX and MIN for reasons that will soon become obvious. MAX moves first, and then they take turns moving until the game is over. At the end of the game, points are awarded to the winning player and penalties are given to the loser. A game can be formally defined as a kind of search problem with the following components:

- The **initial state**, which includes the board position and identifies the player to move.
- A **successor function**, which returns a list of (*move, state*) pairs, each indicating a legal move and the resulting state.

TERMINAL TEST

- A **terminal test**, which determines when the game is over. States where the game has ended are called **terminal states**.
- A **utility function** (also called an objective function or payoff function), which gives a numeric value for the terminal states. In chess, the outcome is a win, loss, or draw, with values $+1$, -1, or 0. Some games have a wider variety of possible outcomes; the payoffs in backgammon range from $+192$ to -192. This chapter deals mainly with zero-sum games, although we will briefly mention non-zero-sum games.

GAME TREE

The initial state and the legal moves for each side define the **game tree** for the game. Figure 6.1 shows part of the game tree for tic-tac-toe (noughts and crosses). From the initial state, MAX has nine possible moves. Play alternates between MAX's placing an X and MIN's placing an O until we reach leaf nodes corresponding to terminal states such that one player has three in a row or all the squares are filled. The number on each leaf node indicates the utility value of the terminal state from the point of view of MAX; high values are assumed to be good for MAX and bad for MIN (which is how the players get their names). It is MAX's job to use the search tree (particularly the utility of terminal states) to determine the best move.

Optimal strategies

In a normal search problem, the optimal solution would be a sequence of moves leading to a goal state—a terminal state that is a win. In a game, on the other hand, MIN has something

STRATEGY

to say about it. MAX therefore must find a contingent **strategy**, which specifies MAX's move in the initial state, then MAX's moves in the states resulting from every possible response by MIN, then MAX's moves in the states resulting from every possible response by MIN to *those* moves, and so on. Roughly speaking, an optimal strategy leads to outcomes at least as good as any other strategy when one is playing an infallible opponent. We will begin by showing how to find this optimal strategy, even though it should be infeasible for MAX to compute it for games more complex than tic-tac-toe.

Even a simple game like tic-tac-toe is too complex for us to draw the entire game tree, so we will switch to the trivial game in Figure 6.2. The possible moves for MAX at the root node are labeled a_1, a_2, and a_3. The possible replies to a_1 for MIN are b_1, b_2, b_3, and so on. This particular game ends after one move each by MAX and MIN. (In game parlance, we say

PLY

that this tree is one move deep, consisting of two half-moves, each of which is called a **ply**.) The utilities of the terminal states in this game range from 2 to 14.

Given a game tree, the optimal strategy can be determined by examining the **minimax**

MINIMAX VALUE

value of each node, which we write as MINIMAX-VALUE(n). The minimax value of a node is the utility (for MAX) of being in the corresponding state, *assuming that both players play optimally* from there to the end of the game. Obviously, the minimax value of a terminal state is just its utility. Furthermore, given a choice, MAX will prefer to move to a state of maximum value, whereas MIN prefers a state of minimum value. So we have the following:

$$\text{MINIMAX-VALUE}(n) = \begin{cases} \text{UTILITY}(n) & \text{if } n \text{ is a terminal state} \\ \max_{s \in Successors(n)} \text{MINIMAX-VALUE}(s) & \text{if } n \text{ is a MAX node} \\ \min_{s \in Successors(n)} \text{MINIMAX-VALUE}(s) & \text{if } n \text{ is a MIN node.} \end{cases}$$

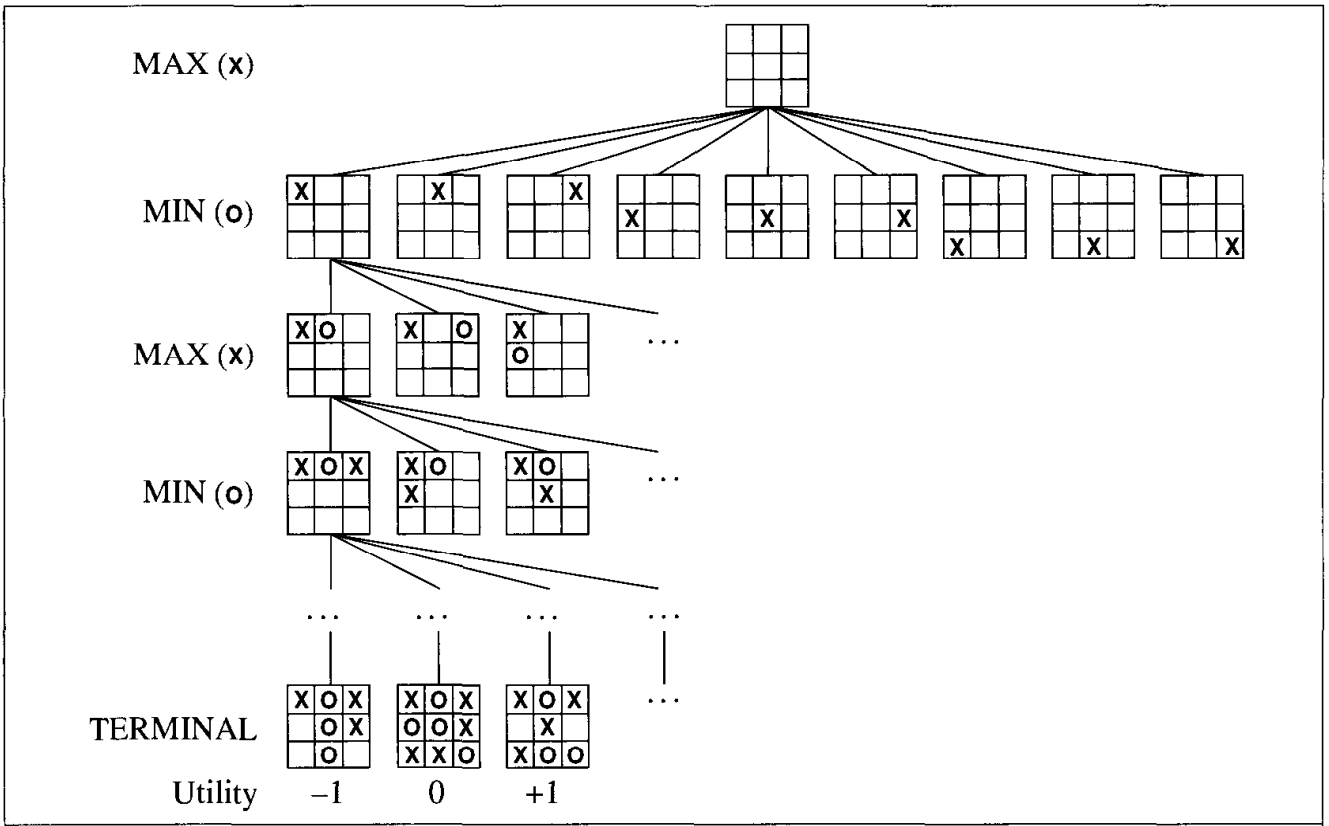


Figure 6.1 A (partial) search tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the search tree, giving alternating moves by MIN (O) and MAX, until we eventually reach terminal states, which can be assigned utilities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

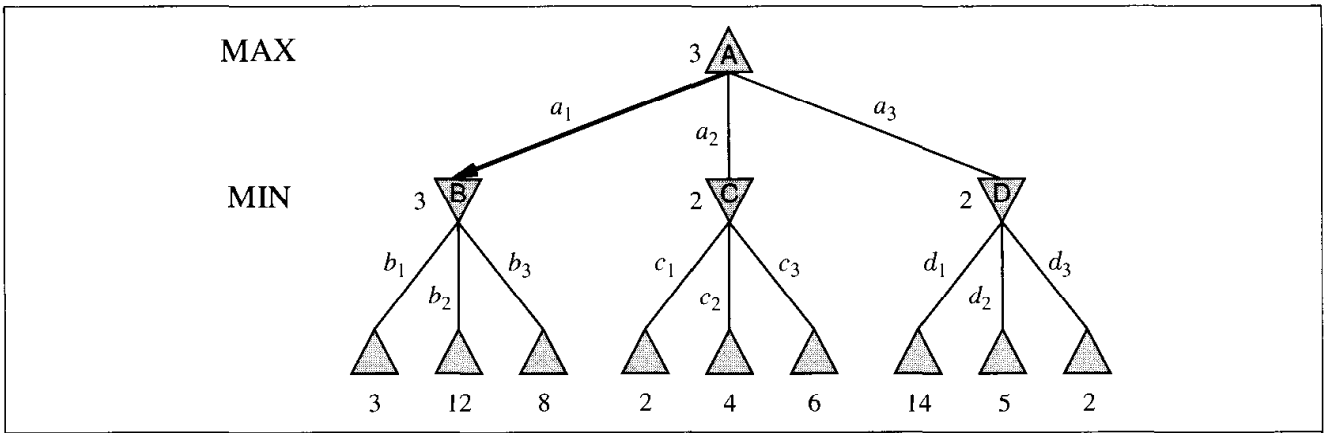


Figure 6.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the successor with the highest minimax value, and MIN's best reply is b_1, because it leads to the successor with the lowest minimax value.

Let us apply these definitions to the game tree in Figure 6.2. The terminal nodes on the bottom level are already labeled with their utility values. The first MIN node, labeled B, has three successors with values 3, 12, and 8, so its minimax value is 3. Similarly, the other two MIN nodes have minimax value 2. The root node is a MAX node; its successors have minimax

MINIMAX DECISION

values 3, 2, and 2; so it has a minimax value of 3. We can also identify the **minimax decision** at the root: action a_1 is the optimal choice for MAX because it leads to the successor with the highest minimax value.

This definition of optimal play for MAX assumes that MIN also plays optimally—it maximizes the *worst-case* outcome for MAX. What if MIN does not play optimally? Then it is easy to show (Exercise 6.2) that MAX will do even better. There may be other strategies against suboptimal opponents that do better than the minimax strategy; but these strategies necessarily do worse against optimal opponents.

The minimax algorithm

MINIMAX ALGORITHM

The **minimax algorithm** (Figure 6.3) computes the minimax decision from the current state. It uses a simple recursive computation of the minimax values of each successor state, directly implementing the defining equations. The recursion proceeds all the way down to the leaves

BACKED UP

of the tree, and then the minimax values are **backed up** through the tree as the recursion unwinds. For example, in Figure 6.2, the algorithm first recurses down to the three bottom-left nodes, and uses the UTILITY function on them to discover that their values are 3, 12, and 8 respectively. Then it takes the minimum of these values, 3, and returns it as the backed-up value of node B. A similar process gives the backed up values of 2 for C and 2 for D. Finally, we take the maximum of 3, 2, and 2 to get the backed-up value of 3 for the root node.

The minimax algorithm performs a complete depth-first exploration of the game tree. If the maximum depth of the tree is m, and there are b legal moves at each point, then the time complexity of the minimax algorithm is $O(b^m)$. The space complexity is $O(bm)$ for an algorithm that generates all successors at once, or $O(m)$ for an algorithm that generates successors one at a time (see page 76). For real games, of course, the time cost is totally impractical, but this algorithm serves as the basis for the mathematical analysis of games and for more practical algorithms.

Optimal decisions in multiplayer games

Many popular games allow more than two players. Let us examine how to extend the minimax idea to multiplayer games. This is straightforward from the technical viewpoint, but raises some interesting new conceptual issues.

First, we need to replace the single value for each node with a *vector* of values. For example, in a three-player game with players A, B, and C, a vector $\langle v_A, v_B, v_C \rangle$ is associated with each node. For terminal states, this vector gives the utility of the state from each player's viewpoint. (In two-player, zero-sum games, the two-element vector can be reduced to a single value because the values are always opposite.) The simplest way to implement this is to have the UTILITY function return a vector of utilities.

Now we have to consider nonterminal states. Consider the node marked X in the game tree shown in Figure 6.4. In that state, player C chooses what to do. The two choices lead to terminal states with utility vectors $\langle v_A = 1, v_B = 2, v_C = 6 \rangle$ and $\langle v_A = 4, v_B = 2, v_C = 3 \rangle$. Since 6 is bigger than 3, C should choose the first move. This means that if state X is reached, subsequent play will lead to a terminal state with utilities $\langle v_A = 1, v_B = 2, v_C = 6 \rangle$. Hence,

```
function MINIMAX-DECISION(state) returns an action
  inputs: state, current state in game

  v ← MAX-VALUE(state)
  return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for a, s in SUCCESSORS(state) do
    v ← MAX(v, MIN-VALUE(s))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a, s in SUCCESSORS(state) do
    v ← MIN(v, MAX-VALUE(s))
  return v
```

Figure 6.3 An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state.

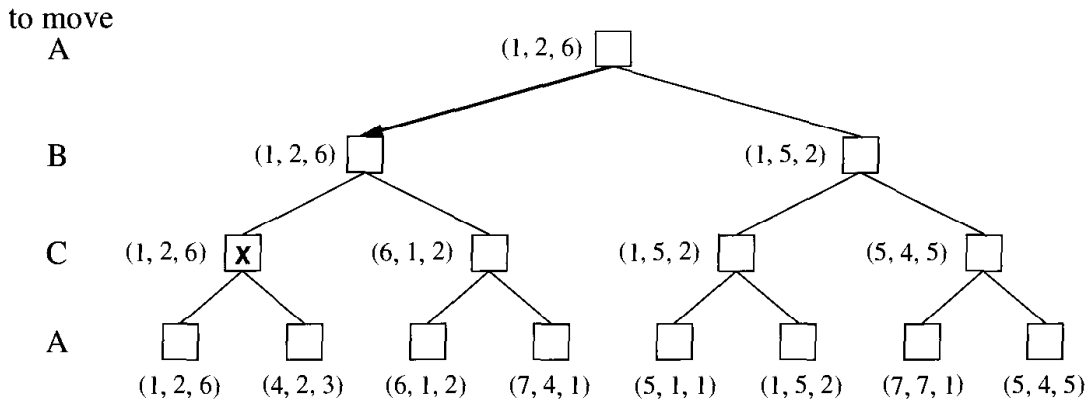


Figure 6.4 The first three ply of a game tree with three players (A, B, C). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

the backed-up value of X is this vector. In general, the backed-up value of a node n is the utility vector of whichever successor has the highest value for the player choosing at n.

Anyone who plays multiplayer games, such as Diplomacy^{TM}, quickly becomes aware that there is a lot more going on than in two-player games. Multiplayer games usually involve

ALLIANCES **alliances**, whether formal or informal, among the players. Alliances are made and broken

as the game proceeds. How are we to understand such behavior? Are alliances a natural consequence of optimal strategies for each player in a multiplayer game? It turns out that they can be. For example suppose A and B are in weak positions and C is in a stronger position. Then it is often optimal for both A and B to attack C rather than each other, lest C destroy each of them individually. In this way, collaboration emerges from purely selfish behavior. Of course, as soon as C weakens under the joint onslaught, the alliance loses its value, and either A or B could violate the agreement. In some cases, explicit alliances merely make concrete what would have happened anyway. In other cases there is a social stigma to breaking an alliance, so players must balance the immediate advantage of breaking an alliance against the long-term disadvantage of being perceived as untrustworthy. See Section 17.6 for more on these complications.

If the game is not zero-sum, then collaboration can also occur with just two players. Suppose, for example, that there is a terminal state with utilities $\langle v_A = 1000, v_B = 1000 \rangle$, and that 1000 is the highest possible utility for each player. Then the optimal strategy is for both players to do everything possible to reach this state—that is, the players will automatically cooperate to achieve a mutually desirable goal.

6.3 ALPHA–BETA PRUNING

The problem with minimax search is that the number of game states it has to examine is exponential in the number of moves. Unfortunately we can't eliminate the exponent, but we can effectively cut it in half. The trick is that it is possible to compute the correct minimax decision without looking at every node in the game tree. That is, we can borrow the idea of **pruning** from Chapter 4 in order to eliminate large parts of the tree from consideration.

ALPHA–BETA PRUNING

The particular technique we will examine is called **alpha–beta pruning**. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

Consider again the two-ply game tree from Figure 6.2. Let's go through the calculation of the optimal decision once more, this time paying careful attention to what we know at each point in the process. The steps are explained in Figure 6.5. The outcome is that we can identify the minimax decision without ever evaluating two of the leaf nodes.

Another way to look at this is as a simplification of the formula for MINIMAX-VALUE. Let the two unevaluated successors of node C in Figure 6.5 have values x and y and let z be the minimum of x and y. The value of the root node is given by

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

In other words, the value of the root and hence the minimax decision are *independent* of the values of the pruned leaves x and y.

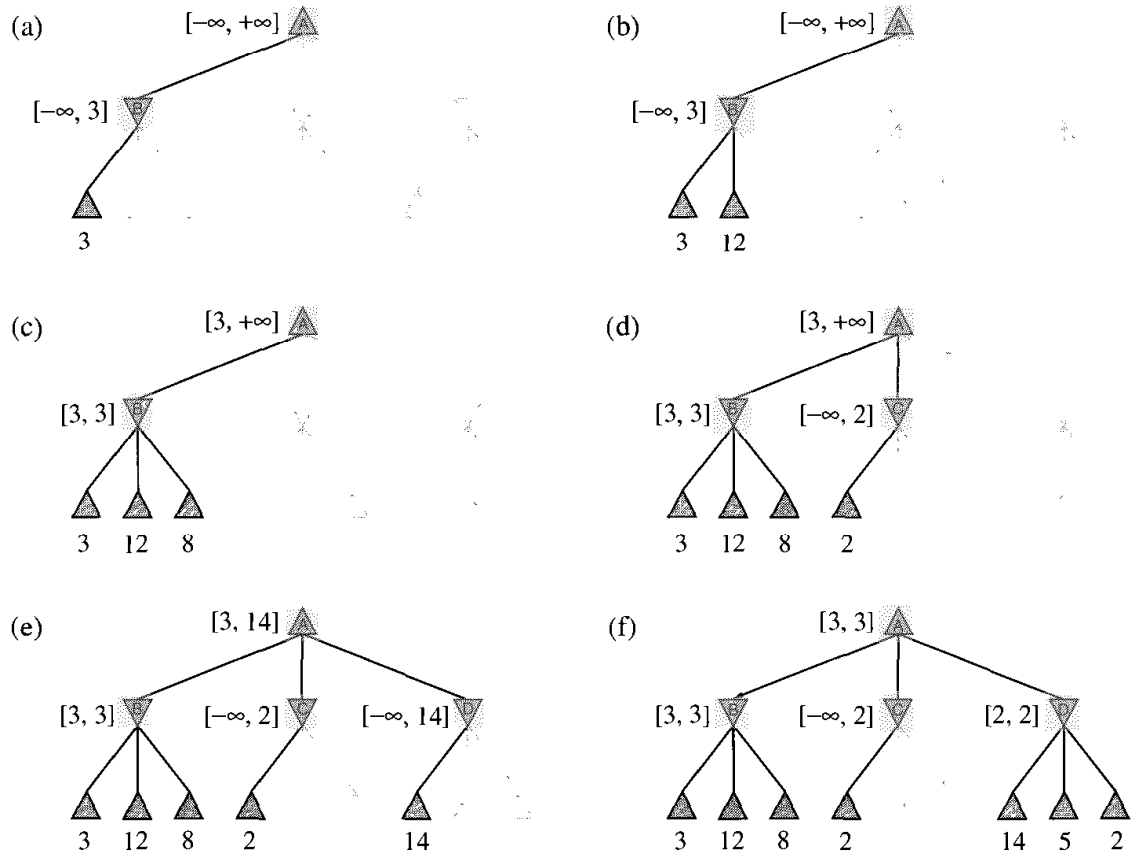


Figure 6.5 Stages in the calculation of the optimal decision for the game tree in Figure 6.2. At each point, we show the range of possible values for each node. (a) The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of *at most* 3. (b) The second leaf below B has a value of 12; MIN would avoid this move, so the value of B is still at most 3. (c) The third leaf below B has a value of 8; we have seen all B's successors, so the value of B is exactly 3. Now, we can infer that the value of the root is *at least* 3, because MAX has a choice worth 3 at the root. (d) The first leaf below C has the value 2. Hence, C, which is a MIN node, has a value of *at most* 2. But we know that B is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other successors of C. This is an example of alpha–beta pruning. (e) The first leaf below D has the value 14, so D is worth *at most* 14. This is still higher than MAX's best alternative (i.e., 3), so we need to keep exploring D's successors. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14. (f) The second successor of D is worth 5, so again we need to keep exploring. The third successor is worth 2, so now D is worth exactly 2. MAX's decision at the root is to move to B, giving a value of 3.

Alpha–beta pruning can be applied to trees of any depth, and it is often possible to prune entire subtrees rather than just leaves. The general principle is this: consider a node n somewhere in the tree (see Figure 6.6), such that Player has a choice of moving to that node. If Player has a better choice m either at the parent node of n or at any choice point further up, then n *will never be reached in actual play.* So once we have found out enough about n (by examining some of its descendants) to reach this conclusion, we can prune it.

Remember that minimax search is depth-first, so at any one time we just have to consider the nodes along a single path in the tree. Alpha–beta pruning gets its name from the

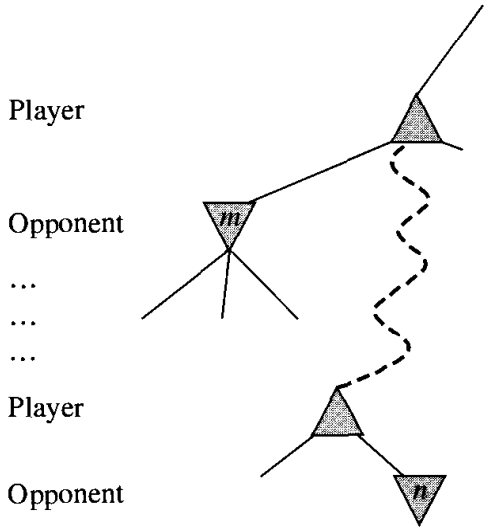


Figure 6.6 Alpha–beta pruning: the general case. If m is better than n for Player, we will never get to n in play.

following two parameters that describe bounds on the backed-up values that appear anywhere along the path:

$\alpha =$ the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.

$\beta =$ the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN.

Alpha–beta search updates the values of α and β as it goes along and prunes the remaining branches at a node (i.e., terminates the recursive call) as soon as the value of the current node is known to be worse than the current α or β value for MAX or MIN, respectively. The complete algorithm is given in Figure 6.7. We encourage the reader to trace its behavior when applied to the tree in Figure 6.5.

The effectiveness of alpha–beta pruning is highly dependent on the order in which the successors are examined. For example, in Figure 6.5(e) and (f), we could not prune any successors of D at all because the worst successors (from the point of view of MIN) were generated first. If the third successor had been generated first, we would have been able to prune the other two. This suggests that it might be worthwhile to try to examine first the successors that are likely to be best.

If we assume that this can be done,[2] then it turns out that alpha–beta needs to examine only $O(b^{m/2})$ nodes to pick the best move, instead of $O(b^m)$ for minimax. This means that the effective branching factor becomes $\sqrt{b}$ instead of b—for chess, 6 instead of 35. Put another way, alpha–beta can look ahead roughly twice as far as minimax in the same amount of time. If successors are examined in random order rather than best-first, the total number of nodes examined will be roughly $O(b^{3m/4})$ for moderate b. For chess, a fairly simple ordering function (such as trying captures first, then threats, then forward moves, and then backward moves) gets you to within about a factor of 2 of the best-case $O(b^{m/2})$ result. Adding dynamic

[2] Obviously, it cannot be done perfectly; otherwise the ordering function could be used to play a perfect game!

```
function ALPHA-BETA-SEARCH(state) returns an action
  inputs: state, current state in game

  v ← MAX-VALUE(state, −∞, +∞)
  return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state

  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for a, s in SUCCESSORS(state) do
    v ← MAX(v, MIN-VALUE(s, α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state

  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for a, s in SUCCESSORS(state) do
    v ← MIN(v, MAX-VALUE(s, α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Figure 6.7 The alpha–beta search algorithm. Notice that these routines are the same as the MINIMAX routines in Figure 6.3, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β (and the bookkeeping to pass these parameters along).

move-ordering schemes, such as trying first the moves that were found to be best last time, brings us quite close to the theoretical limit.

In Chapter 3, we noted that repeated states in the search tree can cause an exponential increase in search cost. In games, repeated states occur frequently because of **transpositions**—different permutations of the move sequence that end up in the same position. For example, if White has one move a_1 that can be answered by Black with b_1 and an unrelated move a_2 on the other side of the board that can be answered by b_2, then the sequences $[a_1, b_1, a_2, b_2]$ and $[a_1, b_2, a_2, b_1]$ both end up in the same position (as do the permutations beginning with a_2). It is worthwhile to store the evaluation of this position in a hash table the first time it is encountered, so that we don't have to recompute it on subsequent occurrences.

TRANSPOSITIONS

TRANSPOSITION TABLE

The hash table of previously seen positions is traditionally called a **transposition table**; it is essentially identical to the *closed* list in GRAPH-SEARCH (page 83). Using a transposition table can have a dramatic effect, sometimes as much as doubling the reachable search depth in chess. On the other hand, if we are evaluating a million nodes per second, it is not practical to keep *all* of them in the transposition table. Various strategies have been used to choose the most valuable ones.

6.4 IMPERFECT, REAL-TIME DECISIONS

The minimax algorithm generates the entire game search space, whereas the alpha–beta algorithm allows us to prune large parts of it. However, alpha–beta still has to search all the way to terminal states for at least a portion of the search space. This depth is usually not practical, because moves must be made in a reasonable amount of time—typically a few minutes at most. Shannon's 1950 paper, *Programming a computer for playing chess*, proposed instead that programs should cut off the search earlier and apply a heuristic **evaluation function** to states in the search, effectively turning nonterminal nodes into terminal leaves. In other words, the suggestion is to alter minimax or alpha–beta in two ways: the utility function is replaced by a heuristic evaluation function EVAL, which gives an estimate of the position's utility, and the terminal test is replaced by a **cutoff test** that decides when to apply EVAL.

CUTOFF TEST

Evaluation functions

An evaluation function returns an *estimate* of the expected utility of the game from a given position, just as the heuristic functions of Chapter 4 return an estimate of the distance to the goal. The idea of an estimator was not new when Shannon proposed it. For centuries, chess players (and aficionados of other games) have developed ways of judging the value of a position, because humans are even more limited in the amount of search they can do than are computer programs. It should be clear that the performance of a game-playing program is dependent on the quality of its evaluation function. An inaccurate evaluation function will guide an agent toward positions that turn out to be lost. How exactly do we design good evaluation functions?

First, the evaluation function should order the *terminal* states in the same way as the true utility function; otherwise, an agent using it might select suboptimal moves even if it can see ahead all the way to the end of the game. Second, the computation must not take too long! (The evaluation function could call MINIMAX-DECISION as a subroutine and calculate the exact value of the position, but that would defeat the whole purpose: to save time.) Third, for nonterminal states, the evaluation function should be strongly correlated with the actual chances of winning.

One might well wonder about the phrase "chances of winning." After all, chess is not a game of chance: we know the current state with certainty, and there are no dice involved. But if the search must be cut off at nonterminal states, then the algorithm will necessarily be *uncertain* about the final outcomes of those states. This type of uncertainty is induced by

computational, rather than informational, limitations. Given the limited amount of computation that the evaluation function is allowed to do for a given state, the best it can do is make a guess about the final outcome.

FEATURES

Let us make this idea more concrete. Most evaluation functions work by calculating various **features** of the state—for example, the number of pawns possessed by each side in a game of chess. The features, taken together, define various *categories* or *equivalence classes* of states: the states in each category have the same values for all the features. Any given category, generally speaking, will contain some states that lead to wins, some that lead to draws, and some that lead to losses. The evaluation function cannot know which states are which, but it can return a single value that reflects the *proportion* of states with each outcome. For example, suppose our experience suggests that 72% of the states encountered in the category lead to a win (utility +1); 20% to a loss (−1), and 8% to a draw (0). Then a reasonable evaluation for states in the category is the weighted average or **expected value**: $(0.72 \times +1) + (0.20 \times -1) + (0.08 \times 0) = 0.52$. In principle, the expected value can be determined for each category, resulting in an evaluation function that works for any state. As with terminal states, the evaluation function need not return actual expected values, as long as the *ordering* of the states is the same.

EXPECTED VALUE

In practice, this kind of analysis requires too many categories and hence too much experience to estimate all the probabilities of winning. Instead, most evaluation functions compute separate numerical contributions from each feature and then *combine* them to find the total value. For example, introductory chess books give an approximate **material value** for each piece: each pawn is worth 1, a knight or bishop is worth 3, a rook 5, and the queen 9. Other features such as "good pawn structure" and "king safety" might be worth half a pawn, say. These feature values are then simply added up to obtain the evaluation of the position. A secure advantage equivalent to a pawn gives a substantial likelihood of winning, and a secure advantage equivalent to three pawns should give almost certain victory, as illustrated in Figure 6.8(a). Mathematically, this kind of evaluation function is called a **weighted linear function**, because it can be expressed as

MATERIAL VALUE

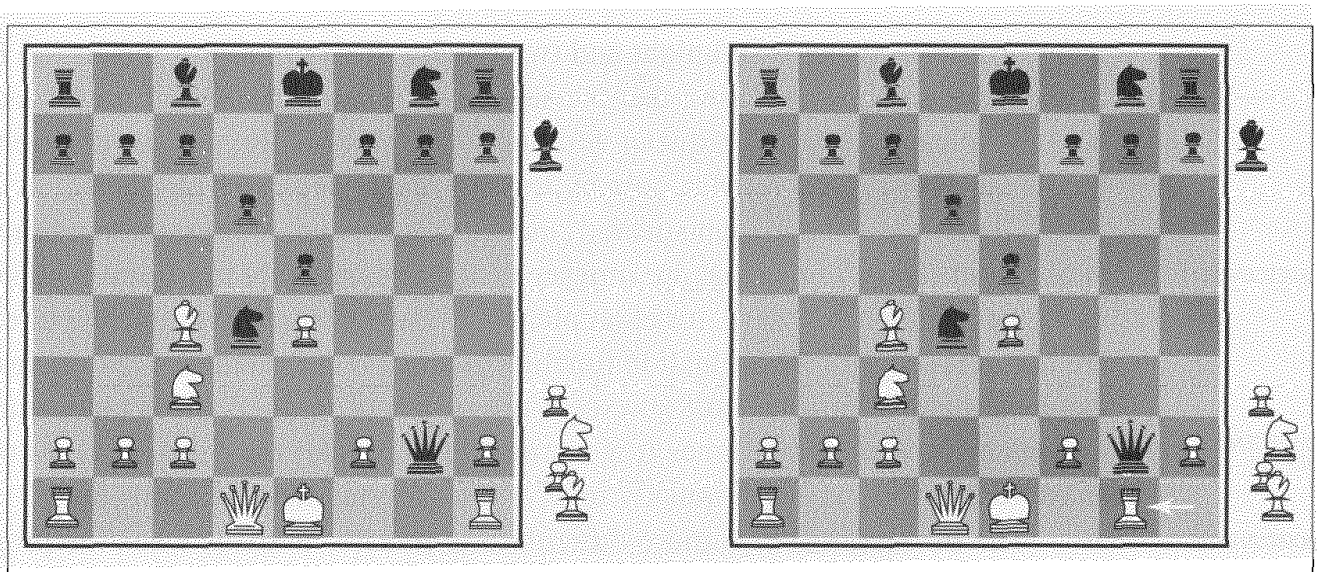
WEIGHTED LINEAR FUNCTION

$$\textsc{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s) \,,$$

where each w_i is a weight and each f_i is a feature of the position. For chess, the f_i could be the numbers of each kind of piece on the board, and the w_i could be the values of the pieces (1 for pawn, 3 for bishop, etc.).

Adding up the values of features seems like a reasonable thing to do, but in fact it involves a very strong assumption: that the contribution of each feature is *independent* of the values of the other features. For example, assigning the value 3 to a bishop ignores the fact that bishops are more powerful in the endgame, when they have a lot of space to maneuver. For this reason, current programs for chess and other games also use *nonlinear* combinations of features. For example, a pair of bishops might be worth slightly more than twice the value of a single bishop, and a bishop is worth more in the endgame than at the beginning.

The astute reader will have noticed that the features and weights are *not* part of the rules of chess! They come from centuries of human chess-playing experience. Given the



(a) White to move (b) White to move

Figure 6.8 Two slightly different chess positions. In (a), black has an advantage of a knight and two pawns and will win the game. In (b), black will lose after white captures the queen.

linear form of the evaluation, the features and weights result in the best approximation to the true ordering of states by value. In particular, experience suggests that a secure material advantage of more than one point will probably win the game, all other things being equal; a three-point advantage is sufficient for near-certain victory. In games where this kind of experience is not available, the weights of the evaluation function can be estimated by the machine learning techniques of Chapter 18. Reassuringly, applying these techniques to chess has confirmed that a bishop is indeed worth about three pawns.

Cutting off search

The next step is to modify ALPHA-BETA-SEARCH so that it will call the heuristic EVAL function when it is appropriate to cut off the search. In terms of implementation, we replace the two lines in Figure 6.7 that mention TERMINAL-TEST with the following line:

if CUTOFF-TEST(*state*, *depth*) **then return** EVAL(*state*)

We also must arrange for some bookkeeping so that the current *depth* is incremented on each recursive call. The most straightforward approach to controlling the amount of search is to set a fixed depth limit, so that CUTOFF-TEST(*state*, *depth*) returns *true* for all *depth* greater than some fixed depth d. (It must also return *true* for all terminal states, just as TERMINAL-TEST did.) The depth d is chosen so that the amount of time used will not exceed what the rules of the game allow.

A more robust approach is to apply iterative deepening, as defined in Chapter 3. When time runs out, the program returns the move selected by the deepest completed search. However, these approaches can lead to errors due to the approximate nature of the evaluation function. Consider again the simple evaluation function for chess based on material advantage. Suppose the program searches to the depth limit, reaching the position in Figure 6.8(b),

where Black is ahead by a knight and two pawns. It would report this as the heuristic value of the state, thereby declaring that the state will likely lead to a win by Black. But White's next move captures Black's queen with no compensation. Hence, the position is really won for White, but this can be seen only by looking ahead one more ply.

QUIESCENCE

Obviously, a more sophisticated cutoff test is needed. The evaluation function should be applied only to positions that are **quiescent**—that is, unlikely to exhibit wild swings in value in the near future. In chess, for example, positions in which favorable captures can be made are not quiescent for an evaluation function that just counts material. Nonquiescent positions can be expanded further until quiescent positions are reached. This extra search is called a **quiescence search**; sometimes it is restricted to consider only certain types of moves, such as capture moves, that will quickly resolve the uncertainties in the position.

QUIESCENCE SEARCH

HORIZON EFFECT

The **horizon effect** is more difficult to eliminate. It arises when the program is facing a move by the opponent that causes serious damage and is ultimately unavoidable. Consider the chess game in Figure 6.9. Black is ahead in material, but if White can advance its pawn from the seventh row to the eighth, the pawn will become a queen and create an easy win for White. Black can forestall this outcome for 14 ply by checking White with the rook, but inevitably the pawn will become a queen. The problem with fixed-depth search is that it believes that these stalling moves have avoided the queening move—we say that the stalling moves push the inevitable queening move "over the search horizon" to a place where it cannot be detected.

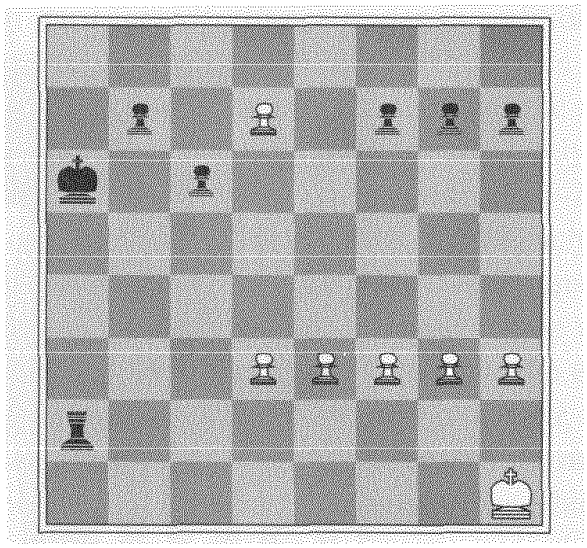
As hardware improvements lead to deeper searches, one expects that the horizon effect will occur less frequently—very long delaying sequences are quite rare. The use of **singular extensions** has also been quite effective in avoiding the horizon effect without adding too much search cost. A singular extension is a move that is "clearly better" than all other moves in a given position. A singular-extension search can go beyond the normal depth limit without incurring much cost because its branching factor is 1. (Quiescence search can be thought of as a variant of singular extensions.) In Figure 6.9, a singular extension search will find the eventual queening move, provided that black's checking moves and white's king moves can be identified as "clearly better" than the alternatives.

SINGULAR EXTENSIONS

So far we have talked about cutting off search at a certain level and about doing alpha–beta pruning that provably has no effect on the result. It is also possible to do **forward pruning**, meaning that some moves at a given node are pruned immediately without further consideration. Clearly, most humans playing chess only consider a few moves from each position (at least consciously). Unfortunately, the approach is rather dangerous because there is no guarantee that the best move will not be pruned away. This can be disastrous if applied near the root, because every so often the program will miss some "obvious" moves. Forward pruning can be used safely in special situations—for example, when two moves are symmetric or otherwise equivalent, only one of them need be considered—or for nodes that are deep in the search tree.

FORWARD PRUNING

Combining all the techniques described here results in a program that can play creditable chess (or other games). Let us assume we have implemented an evaluation function for chess, a reasonable cutoff test with a quiescence search, and a large transposition table. Let us also assume that, after months of tedious bit-bashing, we can generate and evaluate



Black to move

Figure 6.9 The horizon effect. A series of checks by the black rook forces the inevitable queening move by white "over the horizon" and makes this position look like a win for black, when it is really a win for white.

around a million nodes per second on the latest PC, allowing us to search roughly 200 million nodes per move under standard time controls (three minutes per move). The branching factor for chess is about 35, on average, and 35^5 is about 50 million, so if we used minimax search we could look ahead only about five plies. Though not incompetent, such a program can be fooled easily by an average human chess player, who can occasionally plan six or eight plies ahead. With alpha–beta search we get to about 10 ply, which results in an expert level of play. Section 6.7 describes additional pruning techniques that can extend the effective search depth to roughly 14 plies. To reach grandmaster status we would need an extensively tuned evaluation function and a large database of optimal opening and endgame moves. It wouldn't hurt to have a supercomputer to run the program on.

6.5 Games That Include an Element of Chance

In real life, there are many unpredictable external events that put us into unforeseen situations. Many games mirror this unpredictability by including a random element, such as the throwing of dice. In this way, they take us a step nearer reality, and it is worthwhile to see how this affects the decision-making process.

Backgammon is a typical game that combines luck and skill. Dice are rolled at the beginning of a player's turn to determine the legal moves. In the backgammon position of Figure 6.10, for example, white has rolled a 6–5, and has four possible moves.

Although White knows what his or her own legal moves are, White does not know what Black is going to roll and thus does not know what Black's legal moves will be. That means White cannot construct a standard game tree of the sort we saw in chess and tic-tac-toe. A

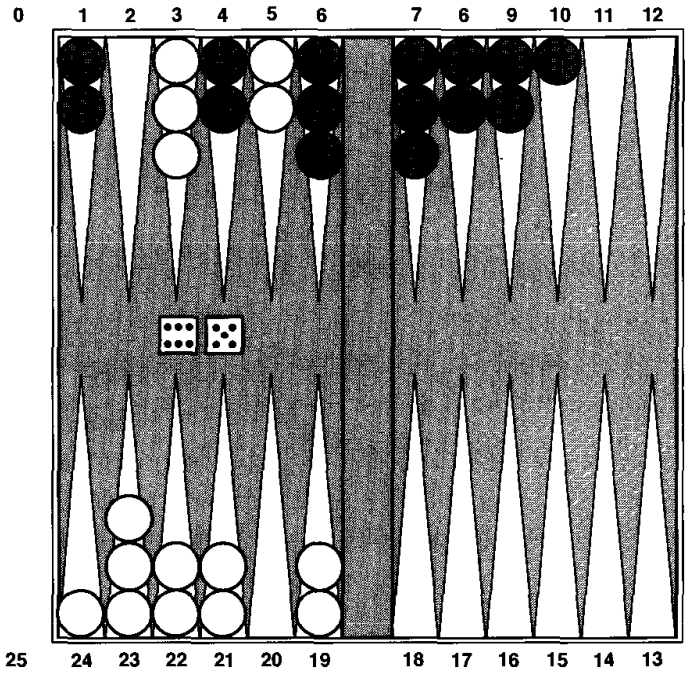


Figure 6.10 A typical backgammon position. The goal of the game is to move all one's pieces off the board. White moves clockwise toward 25, and black moves counterclockwise toward 0. A piece can move to any position unless there are multiple opponent pieces there; if there is one opponent, it is captured and must start over. In the position shown, White has rolled 6–5 and must choose among four legal moves: (5–10,5–11), (5–11,19–24), (5–10,10–16), and (5–11,11–16).

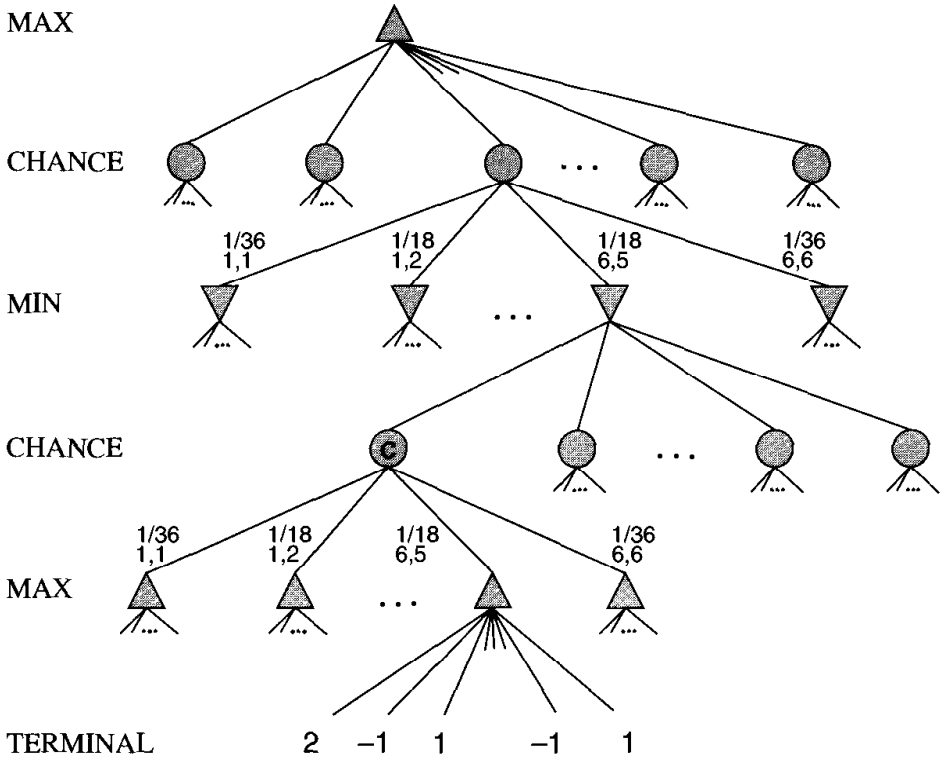


Figure 6.11 Schematic game tree for a backgammon position.

CHANCE NODES

game tree in backgammon must include **chance nodes** in addition to MAX and MIN nodes. Chance nodes are shown as circles in Figure 6.11. The branches leading from each chance node denote the possible dice rolls, and each is labeled with the roll and the chance that it will occur. There are 36 ways to roll two dice, each equally likely; but because a 6–5 is the same as a 5–6, there are only 21 distinct rolls. The six doubles (1–1 through 6–6) have a 1/36 chance of coming up, the other 15 distinct rolls a 1/18 chance each.

The next step is to understand how to make correct decisions. Obviously, we still want to pick the move that leads to the best position. However, the resulting positions do not have definite minimax values. Instead, we can only calculate the **expected value**, where the expectation is taken over all the possible dice rolls that could occur. This leads us to generalize

EXPECTIMINIMAX VALUE

the **minimax value** for deterministic games to an **expectiminimax value** for games with chance nodes. Terminal nodes and MAX and MIN nodes (for which the dice roll is known) work exactly the same way as before; chance nodes are evaluated by taking the weighted average of the values resulting from all possible dice rolls, that is,

$$\text{EXPECTIMINIMAX}(n) = \begin{cases} \text{UTILITY}(n) & \text{if } n \text{ is a terminal state} \\ \max_{s \in Successors(n)} \text{EXPECTIMINIMAX}(s) & \text{if } n \text{ is a MAX node} \\ \min_{s \in Successors(n)} \text{EXPECTIMINIMAX}(s) & \text{if } n \text{ is a MIN node} \\ \sum_{s \in Successors(n)} P(s) \cdot \text{EXPECTIMINIMAX}(s) & \text{if } n \text{ is a chance node} \end{cases}$$

where the successor function for a chance node n simply augments the state of n with each possible dice roll to produce each successor s and $P(s)$ is the probability that that dice roll occurs. These equations can be backed up recursively all the way to the root of the tree, just as in minimax. We leave the details of the algorithm as an exercise.

Position evaluation in games with chance nodes

As with minimax, the obvious approximation to make with expectiminimax is to cut the search off at some point and apply an evaluation function to each leaf. One might think that evaluation functions for games such as backgammon should be just like evaluation functions for chess—they just need to give higher scores to better positions. But in fact, the presence of chance nodes means that one has to be more careful about what the evaluation values mean. Figure 6.12 shows what happens: with an evaluation function that assigns values [1, 2, 3, 4] to the leaves, move A_1 is best; with values [1, 20, 30, 400], move A_2 is best. Hence, the program behaves totally differently if we make a change in the scale of some evaluation values! It turns out that, to avoid this sensitivity, the evaluation function must be a *positive linear* transformation of the probability of winning from a position (or, more generally, of the expected utility of the position). This is an important and general property of situations in which uncertainty is involved, and we discuss it further in Chapter 16.

Complexity of expectiminimax

If the program knew in advance all the dice rolls that would occur for the rest of the game, solving a game with dice would be just like solving a game without dice, which minimax

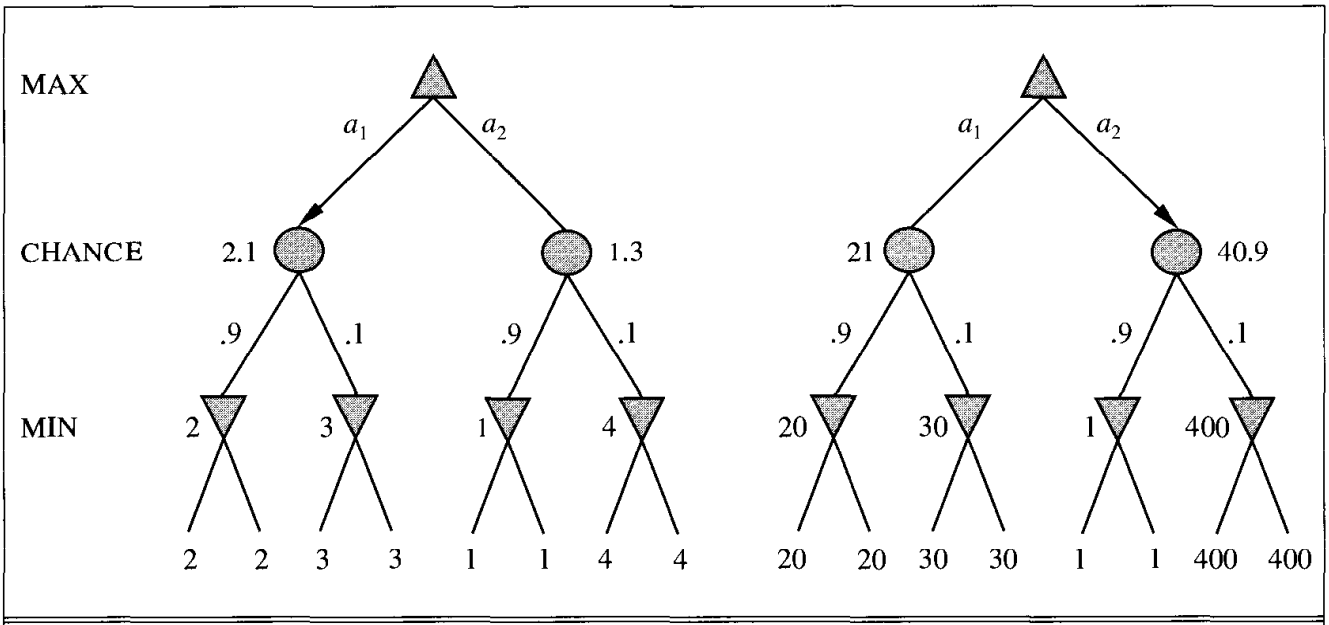


Figure 6.12 An order-preserving transformation on leaf values changes the best move.

does in $O(b^m)$ time. Because expectiminimax is also considering all the possible dice-roll sequences, it will take $O(b^m n^m)$, where n is the number of distinct rolls.

Even if the search depth is limited to some small depth d, the extra cost compared with that of minimax makes it unrealistic to consider looking ahead very far in most games of chance. In backgammon n is 21 and b is usually around 20, but in some situations can be as high as 4000 for dice rolls that are doubles. Three plies is probably all we could manage.

Another way to think about the problem is this: the advantage of alpha–beta is that it ignores future developments that just are not going to happen, given best play. Thus, it concentrates on likely occurrences. In games with dice, there are *no* likely sequences of moves, because for those moves to take place, the dice would first have to come out the right way to make them legal. This is a general problem whenever uncertainty enters the picture: the possibilities are multiplied enormously, and forming detailed plans of action becomes pointless, because the world probably will not play along.

No doubt it will have occurred to the reader that perhaps something like alpha–beta pruning could be applied to game trees with chance nodes. It turns out that it can. The analysis for MIN and MAX nodes is unchanged, but we can also prune chance nodes, using a bit of ingenuity. Consider the chance node C in Figure 6.11 and what happens to its value as we examine and evaluate its children. Is it possible to find an upper bound on the value of C before we have looked at all its children? (Recall that this is what alpha–beta needs to prune a node and its subtree.) At first sight, it might seem impossible, because the value of C is the *average* of its children's values. Until we have looked at all the dice rolls, this average could be anything, because the unexamined children might have any value at all. But if we put bounds on the possible values of the utility function, then we can arrive at bounds for the average. For example, if we say that all utility values are between $+3$ and -3, then the value of leaf nodes is bounded, and in turn we *can* place an upper bound on the value of a chance node without looking at all its children.

Card games

Card games are interesting for many reasons besides their connection with gambling. Among the huge variety of games, we will focus on those in which cards are dealt randomly at the beginning of the game, with each player receiving a hand of cards that is not visible to the other players. Such games include bridge, whist, hearts, and some forms of poker.

At first sight, it might seem that card games are just like dice games: the cards are dealt randomly and determine the moves available to each player, but all the dice are rolled at the beginning! We will pursue this observation further. It will turn out to be quite useful in practice. It is also quite wrong, for interesting reasons.

Imagine two players, MAX and MIN, playing some practice hands of four-card two handed bridge with all the cards showing. The hands are as follows, with MAX to play first:

MAX : ♡6 ◇6 ♣9 8 MIN : ♡4 ♠2 ♣10 5 .

Suppose that MAX leads the ♣9. MIN must now follow suit, playing either the ♣10 or the ♣5. MIN plays the ♣10 and wins the trick. MIN goes next and leads the ♠2. MAX has no spades (and so cannot win the trick) and therefore must throw away some card. The obvious choice is the ◇6 because the other two remaining cards are winners. Now, whichever card MIN leads for the next trick, MAX will win both remaining tricks and the game will be tied at two tricks each. It is easy to show, using a suitable variant of minimax (Exercise 6.12), that MAX's lead of the ♣9 is in fact an optimal choice.

Now let's modify MIN's hand, replacing the ♡4 with the ◇4:

MAX : ♡6 ◇6 ♣9 8 MIN : ◇4 ♠2 ♣10 5 .

The two cases are entirely symmetric: play will be identical, except that on the second trick MAX will throw away the ♡6. Again, the game will be tied at two tricks each and the lead of the ♣9 is an optimal choice.

So far, so good. Now let's hide one of MIN's cards: MAX knows that MIN has either the first hand (with the ♡4) or the second hand (with the ◇4), but has no idea which. MAX reasons as follows:

> The ♣9 is an optimal choice against MIN's first hand and against MIN's second hand, so it must be optimal now because I know that MIN has one of the two hands.

More generally, MAX is using what we might call "averaging over clairvoyancy." The idea is to evaluate a given course of action when there are unseen cards by first computing the minimax value of that action for each possible deal of the cards, and then computing the expected value over all deals using the probability of each deal.

If you think this is reasonable (or if you have no idea because you don't understand bridge), consider the following story:

> Day 1: Road *A* leads to a heap of gold pieces; Road *B* leads to a fork. Take the left fork and you'll find a mound of jewels, but take the right fork and you'll be run over by a bus.
> Day 2: Road *A* leads to a heap of gold pieces; Road *B* leads to a fork. Take the right fork and you'll find a mound of jewels, but take the left fork and you'll be run over by a bus.
> Day 3: Road *A* leads to a heap of gold pieces; Road *B* leads to a fork. Guess correctly and you'll find a mound of jewels, but guess incorrectly and you'll be run over by a bus.

Obviously, it's not unreasonable to take Road *B* on the first two days. No sane person, though, would take Road *B* on Day 3. Yet this is exactly what averaging over clairvoyancy suggests: Road *B* is optimal in the situations of Day 1 and Day 2; therefore it is optimal on Day 3, because one of the two previous situations must hold. Let us return to the card game: after MAX leads the ♣9, MIN wins with the ♣10. As before, MIN leads the ♠2, and now MAX is at the fork in the road without any instructions. If MAX throws away the ♡6 and MIN still has the ♡4, the ♡4 becomes a winner and MAX loses the game. Similarly, If MAX throws away the ♢6 and MIN still has the ♢4, MAX also loses. Therefore, playing the ♣9 first leads to a situation where MAX has a 50% chance of losing. (It would be much better to play the ♡6 and the ♢6 first, guaranteeing a tied game.)

The lesson to be drawn from all this is that when information is missing, one must consider *what information one will have* at each point in the game. The problem with MAX's algorithm is that it assumes that in each possible deal, play will proceed *as if all the cards are visible*. As our example shows, this leads MAX to act as if all *future* uncertainty will be resolved when the time comes. MAX's algorithm will also never decide to *gather* information (or *provide* information to a partner), because within each deal there's no need to do so; yet in games such as bridge, it is often a good idea to play a card that will help one discover things about one's opponent's cards or that will tell one's partner about one's own cards. These kinds of behaviors are generated automatically by an optimal algorithm for games of imperfect information. Such an algorithm searches not in the space of world states (hands of cards), but in the space of **belief states** (beliefs about who has which cards, with what probabilities). We will be able to explain the algorithm properly in Chapter 17, once we have developed the necessary probabilistic machinery. In that chapter, we will also expand on one final and very important point: in games of imperfect information, it's best to give away as little information to the opponent as possible, and often the best way to do this is to act *unpredictably*. This is why restaurant hygiene inspectors do random inspection visits.

6.6 State-of-the-Art Game Programs

One might say that game playing is to AI as Grand Prix motor racing is to the car industry: state-of-the-art game programs are blindingly fast, incredibly well-tuned machines that incorporate very advanced engineering techniques, but they aren't much use for doing the shopping. Although some researchers believe that game playing is somewhat irrelevant to mainstream AI, it continues to generate both excitement and a steady stream of innovations that have been adopted by the wider community.

CHESS

Chess: In 1957, Herbert Simon predicted that within 10 years computers would beat the human world champion. Forty years later, the Deep Blue program defeated Garry Kasparov in a six-game exhibition match. Simon was wrong, but only by a factor of 4. Kasparov wrote:

> The decisive game of the match was Game 2, which left a scar in my memory . . . we saw something that went well beyond our wildest expectations of how well a computer would be able to foresee the long-term positional consequences of its decisions. The machine

> refused to move to a position that had a decisive short-term advantage—showing a very human sense of danger. (Kasparov, 1997)

Deep Blue was developed by Murray Campbell, Feng-Hsiung Hsu, and Joseph Hoane at IBM (see Campbell *et al.*, 2002), building on the Deep Thought design developed earlier by Campbell and Hsu at Carnegie Mellon. The winning machine was a parallel computer with 30 IBM RS/6000 processors running the "software search" and 480 custom VLSI chess processors that performed move generation (including move ordering), the "hardware search" for the last few levels of the tree, and the evaluation of leaf nodes. Deep Blue searched 126 million nodes per second on average, with a peak speed of 330 million nodes per second. It generated up to 30 billion positions per move, reaching depth 14 routinely. The heart of the machine is a standard iterative-deepening alpha–beta search with a transposition table, but the key to its success seems to have been its ability to generate extensions beyond the depth limit for sufficiently interesting lines of forcing/forced moves. In some cases the search reached a depth of 40 plies. The evaluation function had over 8000 features, many of them describing highly specific patterns of pieces. An "opening book" of about 4000 positions was used, as well as a database of 700,000 grandmaster games from which consensus recommendations could be extracted. The system also used a large endgame database of solved positions, containing all positions with five pieces and many with six pieces. This database has the effect of substantially extending the effective search depth, allowing Deep Blue to play perfectly in some cases even when it is many moves away from checkmate.

The success of Deep Blue reinforced the widely held belief that progress in computer game-playing has come primarily from ever-more-powerful hardware—a view encouraged by IBM. Deep Blue's creators, on the other hand, state that the search extensions and evaluation function were also critical (Campbell *et al.*, 2002). Moreover, we know that several recent algorithmic improvements have allowed programs running on standard PCs to win every World Computer-Chess Championship since 1992, often defeating massively parallel opponents that could search 1000 times more nodes. A variety of pruning heuristics are used to reduce the effective branching factor to less than 3 (compared with the actual branching factor of about 35). The most important of these is the **null move** heuristic, which generates a good lower bound on the value of a position, using a shallow search in which the opponent gets to move twice at the beginning. This lower bound often allows alpha–beta pruning without the expense of a full-depth search. Also important is **futility pruning**, which helps decide in advance which moves will cause a beta cutoff in the successor nodes.

NULL MOVE

FUTILITY PRUNING

The Deep Blue team declined a chance for a rematch with Kasparov. Instead, the most recent major competition in 2002 featured the program FRITZ against world champion Vladimir Kramnik. The eight game match ended in a draw. The conditions of the match were much more favorable to the human, and the hardware was an ordinary PC, not a supercomputer. Still, Kramnik commented that "It is now clear that the top program and the world champion are approximately equal."

CHECKERS

Checkers: Beginning in 1952, Arthur Samuel of IBM, working in his spare time, developed a checkers program that learned its own evaluation function by playing itself thousands of times. We describe this idea in more detail in Chapter 21. Samuel's program began as a

novice, but after only a few days' self-play had improved itself beyond Samuel's own level (although he was not a strong player). In 1962 it defeated Robert Nealy, a champion at "blind checkers," through an error on his part. Many people felt that this meant computers were superior to people at checkers, but this was not the case. Still, when one considers that Samuel's computing equipment (an IBM 704) had 10,000 words of main memory, magnetic tape for long-term storage, and a .000001–GHz processor, the win remains a great accomplishment.

Few other people attempted to do better until Jonathan Schaeffer and colleagues developed Chinook, which runs on regular PCs and uses alpha–beta search. Chinook uses a precomputed database of all 444 billion positions with eight or fewer pieces on the board to make its endgame play flawless. Chinook came in second in the 1990 U.S. Open and earned the right to challenge for the world championship. It then ran up against a problem, in the form of Marion Tinsley. Dr. Tinsley had been world champion for over 40 years, losing only three games in all that time. In the first match against Chinook, Tinsley suffered his fourth and fifth losses, but won the match 20.5–18.5. The world championship match in August 1994 between Tinsley and Chinook ended prematurely when Tinsley had to withdraw for health reasons. Chinook became the official world champion.

Schaeffer believes that, with enough computing power, the database of endgames could be enlarged to the point where a forward search from the initial position would always reach solved positions, i.e., checkers would be completely solved. (Chinook has announced a win as early as move 5.) This kind of exhaustive analysis can be done by hand for 3×3 tic-tac-toe and has been done by computer for Qubic ($4 \times 4 \times 4$ tic-tac-toe), Go-Moku (five in a row), and Nine-Men's Morris (Gasser, 1998). Remarkable work by Ken Thompson and Lewis Stiller (1992) solved all five-piece and some six-piece chess endgames, making them available on the Internet. Stiller discovered one case where a forced mate existed but required 262 moves; this caused some consternation because the rules of chess require some "progress" to occur within 50 moves.

OTHELLO

Othello, also called Reversi, is probably more popular as a computer game than as a board game. It has a smaller search space than chess, usually 5 to 15 legal moves, but evaluation expertise had to be developed from scratch. In 1997, the Logistello program (Buro, 2002) defeated the human world champion, Takeshi Murakami , by six games to none. It is generally acknowledged that humans are no match for computers at Othello.

BACKGAMMON

Backgammon: Section 6.5 explained why the inclusion of uncertainty from dice rolls makes deep search an expensive luxury. Most work on backgammon has gone into improving the evaluation function. Gerry Tesauro (1992) combined Samuel's reinforcement learning method with neural network techniques (Chapter 20) to develop a remarkably accurate evaluator that is used with a search to depth 2 or 3. After playing more than a million training games against itself, Tesauro's program, TD-GAMMON, is reliably ranked among the top three players in the world. The program's opinions on the opening moves of the game have in some cases radically altered the received wisdom.

GO

Go is the most popular board game in Asia, requiring at least as much discipline from its professionals as chess. Because the board is 19×19, the branching factor starts at 361, which is too daunting for regular search methods. Up to 1997 there were no competent

programs at all, but now programs often play respectable moves. Most of the best programs combine pattern recognition techniques (when the following pattern of pieces appears, this move should be considered) with limited search (decide whether these pieces can be captured, staying within the local area). The strongest programs at the time of writing are probably Chen Zhixing's Goemate and Michael Reiss' Go4++, each rated somewhere around 10 kyu (weak amateur). Go is an area that is likely to benefit from intensive investigation using more sophisticated reasoning methods. Success may come from finding ways to integrate several lines of local reasoning about each of the many, loosely connected "subgames" into which Go can be decomposed. Such techniques would be of enormous value for intelligent systems in general.

BRIDGE

Bridge is a game of imperfect information: a player's cards are hidden from the other players. Bridge is also a *multiplayer* game with four players instead of two, although the players are paired into two teams. As we saw in Section 6.5, optimal play in bridge can include elements of information-gathering, communication, bluffing, and careful weighing of probabilities. Many of these techniques are used in the Bridge Baron^{TM} program (Smith *et al.*, 1998), which won the 1997 computer bridge championship. While it does not play optimally, Bridge Baron is one of the few successful game-playing systems to use complex, hierarchical plans (see Chapter 12) involving high-level ideas such as **finessing** and **squeezing** that are familiar to bridge players.

The GIB program (Ginsberg, 1999) won the 2000 championship quite decisively. GIB uses the "averaging over clairvoyancy" method, with two crucial modifications. First, rather than examining how well each choice works for every possible arrangement of the hidden cards—of which there can be up to 10 million—it examines a random sample of 100 arrangements. Second, GIB uses **explanation-based generalization** to compute and cache general rules for optimal play in various standard classes of situations. This enables it to solve each deal *exactly*. GIB's tactical accuracy makes up for its inability to reason about information. It finished 12th in a field of 35 in the par contest (involving just play of the hand) at the 1998 human world championship, far exceeding the expectations of many human experts.

6.7 DISCUSSION

Because calculating optimal decisions in games is intractable in most cases, all algorithms must make some assumptions and approximations. The standard approach, based on minimax, evaluation functions, and alpha–beta, is just one way to do this. Probably because it was proposed so early on, the standard approach had been developed intensively and dominates other methods in tournament play. Some in the field believe that this has caused game playing to become divorced from the mainstream of AI research, because the standard approach no longer provides much room for new insight into general questions of decision making. In this section, we look at the alternatives.

First, let us consider minimax. Minimax selects an optimal move in a given search tree *provided that the leaf node evaluations are exactly correct.* In reality, evaluations are

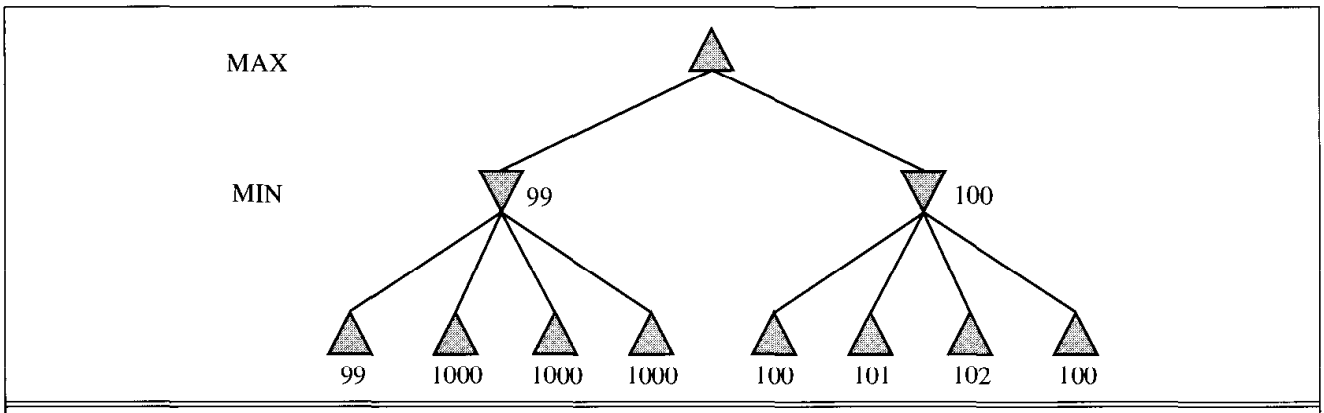


Figure 6.13 A two-ply game tree for which minimax may be inappropriate.

usually crude estimates of the value of a position and can be considered to have large errors associated with them. Figure 6.13 shows a two-ply game tree for which minimax seems inappropriate. Minimax suggests taking the right-hand branch, whereas it is quite likely that the true value of the left-hand branch is higher. The minimax choice relies on the assumption that *all* of the nodes labeled with values 100, 101, 102, and 100 are *actually* better than the node labeled with value 99. However, the fact that the node labeled 99 has siblings labeled 1000 suggests that in fact it might have a higher true value. One way to deal with this problem is to have an evaluation that returns a *probability distribution* over possible values. Then one can calculate the probability distribution for the parent's value using standard statistical techniques. Unfortunately, the values of sibling nodes are usually highly correlated, so this can be an expensive calculation, requiring hard to obtain information.

Next, we consider the search algorithm that generates the tree. The aim of an algorithm designer is to specify a computation that runs quickly and yields a good move. The most obvious problem with the alpha–beta algorithm is that it is designed not just to select a good move, but also to calculate bounds on the values of all the legal moves. To see why this extra information is unnecessary, consider a position in which there is only one legal move. Alpha–beta search still will generate and evaluate a large, and totally useless, search tree. Of course, we can insert a test into the algorithm, but this merely hides the underlying problem: many of the calculations done by alpha–beta are largely irrelevant. Having only one legal move is not much different from having several legal moves, one of which is fine and the rest of which are obviously disastrous. In a "clear favorite" situation like this, it would be better to reach a quick decision after a small amount of search than to waste time that could be more productively used later on a more problematic position. This leads to the idea of the *utility of a node expansion*. A good search algorithm should select node expansions of high utility—that is, ones that are likely to lead to the discovery of a significantly better move. If there are no node expansions whose utility is higher than their cost (in terms of time), then the algorithm should stop searching and make a move. Notice that this works not only for clear-favorite situations, but also for the case of *symmetrical* moves, for which no amount of search will show that one move is better than another.

METAREASONING

This kind of reasoning about what computations to do is called **metareasoning** (reasoning about reasoning). It applies not just to game playing, but to any kind of reasoning

at all. All computations are done in the service of trying to reach better decisions, all have costs, and all have some likelihood of resulting in a certain improvement in decision quality. Alpha–beta incorporates the simplest kind of metareasoning, namely, a theorem to the effect that certain branches of the tree can be ignored without loss. It is possible to do much better. In Chapter 16, we will see how these ideas can be made precise and implementable.

Finally, let us reexamine the nature of search itself. Algorithms for heuristic search and for game playing work by generating sequences of concrete states, starting from the initial state and then applying an evaluation function. Clearly, this is not how humans play games. In chess, one often has a particular goal in mind—for example, trapping the opponent's queen—and can use this goal to *selectively* generate plausible plans for achieving it. This kind of **goal-directed reasoning** or **planning** sometimes eliminates combinatorial search altogether. (See Part IV.) David Wilkins' (1980) PARADISE is the only program to have used goal-directed reasoning successfully in chess: it was capable of solving some chess problems requiring an 18-move combination. As yet there is no good understanding of how to *combine* the two kinds of algorithm into a robust and efficient system, although Bridge Baron might be a step in the right direction. A fully integrated system would be a significant achievement not just for game-playing research, but also for AI research in general, because it would be a good basis for a general intelligent agent.

6.8 SUMMARY

We have looked at a variety of games to understand what optimal play means and to understand how to play well in practice. The most important ideas are as follows:

- A game can be defined by the **initial state** (how the board is set up), the legal **actions** in each state, a **terminal test** (which says when the game is over), and a **utility function** that applies to terminal states.
- In two-player zero-sum games with **perfect information**, the **minimax** algorithm can select optimal moves using a depth-first enumeration of the game tree.
- The **alpha–beta** search algorithm computes the same optimal move as minimax, but achieves much greater efficiency by eliminating subtrees that are provably irrelevant.
- Usually, it is not feasible to consider the whole game tree (even with alpha–beta), so we need to cut the search off at some point and apply an **evaluation function** that gives an estimate of the utility of a state.
- Games of chance can be handled by an extension to the minimax algorithm that evaluates a **chance node** by taking the average utility of all its children nodes, weighted by the probability of each child.
- Optimal play in games of **imperfect information**, such as bridge, requires reasoning about the current and future **belief states** of each player. A simple approximation can be obtained by averaging the value of an action over each possible configuration of missing information.

- Programs can match or beat the best human players in checkers, Othello, and backgammon and are close behind in bridge. A program has beaten the world chess champion in one exhibition match. Programs remain at the amateur level in Go.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

The early history of mechanical game playing was marred by numerous frauds. The most notorious of these was Baron Wolfgang von Kempelen's (1734-1804) "The Turk," a supposed chess-playing automaton that defeated Napoleon before being exposed as a magician's trick cabinet housing a human chess expert (see Levitt, 2000). It played from 1769 to 1854. In 1846, Charles Babbage (who had been fascinated by the Turk) appears to have contributed the first serious discussion of the feasibility of computer chess and checkers (Morrison and Morrison, 1961). He also designed, but did not build, a special-purpose machine for playing tic-tac-toe. The first true game-playing machine was built around 1890 by the Spanish engineer Leonardo Torres y Quevedo. It specialized in the "KRK" (king and rook vs. king) chess endgame, guaranteeing a win with king and rook from any position.

The minimax algorithm is often traced to a paper published in 1912 by Ernst Zermelo, the developer of modern set theory. The paper unfortunately contained several errors and did not describe minimax correctly. A solid foundation for game theory was developed in the seminal work *Theory of Games and Economic Behavior* (von Neumann and Morgenstern, 1944), which included an analysis showing that some games *require* strategies that are randomized (or otherwise unpredictable). See Chapter 17 for more information.

Many influential figures of the early computer era were intrigued by the possibility of computer chess. Konrad Zuse (1945), the first person to design a programmable computer, developed fairly detailed ideas about how it might be done. Norbert Wiener's (1948) influential book *Cybernetics* discussed one possible design for a chess program, including the ideas of minimax search, depth cutoffs, and evaluation functions. Claude Shannon (1950) laid out the basic principles of modern game-playing programs in much more detail than Wiener. He introduced the idea of quiescence search and described some ideas for selective (nonexhaustive) game-tree search. Slater (1950) and the commentators on his article also explored the possibilities for computer chess play. In particular, I. J. Good (1950) developed the notion of quiescence independently of Shannon.

In 1951, Alan Turing wrote the first computer program capable of playing a full game of chess (see Turing *et al.*, 1953). But Turing's program never actually ran on a computer; it was tested by hand simulation against a very weak human player, who defeated it. Meanwhile D. G. Prinz (1952) had written, and actually run, a program that solved chess problems, although it did not play a full game. Alex Bernstein wrote the first program to play a full game of standard chess (Bernstein and Roberts, 1958; Bernstein *et al.*, 1958).[3]

John McCarthy conceived the idea of alpha–beta search in 1956, although he did not publish it. The NSS chess program (Newell *et al.*, 1958) used a simplified version of alpha–

[3] Newell *et al.* (1958) mention a Russian program, BESM, that may have predated Bernstein's program.

beta; it was the first chess program to do so. According to Nilsson (1971), Arthur Samuel's checkers program (Samuel, 1959, 1967) also used alpha–beta, although Samuel did not mention it in the published reports on the system. Papers describing alpha–beta were published in the early 1960s (Hart and Edwards, 1961; Brudno, 1963; Slagle, 1963b). An implementation of full alpha–beta is described by Slagle and Dixon (1969) in a program for playing the game of Kalah. Alpha–beta was also used by the "Kotok–McCarthy" chess program written by a student of John McCarthy (Kotok, 1962). Knuth and Moore (1975) provide a history of alpha–beta, along with a proof of its correctness and a time complexity analysis. Their analysis of alpha–beta with random successor ordering showed an asymptotic complexity of $O((b/\log b)^d)$, which seemed rather dismal because the effective branching factor $b/\log b$ is not much less than b itself. They then realized that the asymptotic formula is accurate only for $b > 1000$ or so, whereas the often-quoted $O(b^{3d/4})$ applies to the range of branching factors encountered in actual games. Pearl (1982b) shows alpha–beta to be asymptotically optimal among all fixed-depth game-tree search algorithms.

The first computer chess match featured the Kotok-McCarthy program and the "ITEP" program written in the mid-1960s at Moscow's Institute of Theoretical and Experimental Physics (Adelson-Velsky *et al.*, 1970). This intercontinental match was played by telegraph. It ended with a 3–1 victory for the ITEP program in 1967. The first chess program to compete successfully with humans was MacHack 6 (Greenblatt *et al.*, 1967). Its rating of approximately 1400 was well above the novice level of 1000, but it fell far short of the rating of 2800 or more that would have been needed to fulfill Herb Simon's 1957 prediction that a computer program would be world chess champion within 10 years (Simon and Newell, 1958).

Beginning with the first ACM North American Computer-Chess Championship in 1970, competition among chess programs became serious. Programs in the early 1970s became extremely complicated, with various kinds of tricks for eliminating some branches of search, for generating plausible moves, and so on. In 1974, the first World Computer-Chess Championship was held in Stockholm and won by Kaissa (Adelson-Velsky *et al.*, 1975), another program from ITEP. Kaissa used the much more straightforward approach of exhaustive alpha–beta search combined with quiescence search. The dominance of this approach was confirmed by the convincing victory of CHESS 4.6 in the 1977 World Computer-Chess Championship. CHESS 4.6 examined up to 400,000 positions per move and had a rating of 1900.

A later version of Greenblatt's MacHack 6 was the first chess program to run on custom hardware designed specifically for chess (Moussouris *et al.*, 1979), but the first program to achieve notable success through the use of custom hardware was Belle (Condon and Thompson, 1982). Belle's move generation and position evaluation hardware enabled it to explore several million positions per move. Belle achieved a rating of 2250, becoming the first master-level program. The HITECH system, also a special-purpose computer, was designed by former World Correspondence Chess Champion Hans Berliner and his student Carl Ebeling at CMU to allow rapid calculation of evaluation functions (Ebeling, 1987; Berliner and Ebeling, 1989). Generating about 10 million positions per move, HITECH became North American computer champion in 1985 and was the first program to defeat a human grandmaster, in 1987. Deep Thought, which was also developed at CMU, went further in the direction of pure search speed (Hsu *et al.*, 1990). It achieved a rating of 2551 and was the

forcrunner of Deep Blue. The Fredkin Prize, cstablished in 1980, offercd $5000 to the first program to achieve a master rating, $10,000 to the first program to achieve a USCF (United States Chess Federation) rating of 2500 (near the grandmaster level), and $100,000 for the first program to defeat the human world champion. The $5000 prize was claimed by Belle in 1983, the $10,000 prize by Deep Thought in 1989, and the $100,000 prize by Deep Blue for its victory over Garry Kasparov in 1997. It is important to remember that Deep Blue's success was due to algorithmic improvements as well as hardware (Hsu, 1999; Campbell *et al.*, 2002). Techniques such as the null-move heuristic (Beal, 1990) have led to programs that are quite selective in their searches. The last three World Computer-Chess Championships in 1992, 1995, and 1999 were won by programs running on standard PCs. Probably the most complete description of a modern chess program is provided by Ernst Heinz (2000), whose DARKTHOUGHT program was the highest-ranked noncommercial PC program at the 1999 world championships.

Several attempts have been made to overcome the problems with the "standard approach" that were outlined in Section 6.7. The first selective search algorithm with some theoretical grounding was probably B* (Berliner, 1979), which attempts to maintain interval bounds on the possible value of a node in the game tree, rather than giving it a single point-valued estimate. Leaf nodes are selected for expansion in an attempt to refine the top-level bounds until one move is "clearly best." Palay (1985) extends the B* idea using probability distributions on values in place of intervals. David McAllester's (1988) conspiracy number search expands leaf nodes that, by changing their values, could cause the program to prefer a new move at the root. MGSS* (Russell and Wefald, 1989) uses the decision-theoretic techniques of Chapter 16 to estimate the value of expanding each leaf in terms of the expected improvement in decision quality at the root. It outplayed an alpha–beta algorithm at Othello despite searching an order of magnitude fewer nodes. The MGSS* approach is, in principle, applicable to the control of any form of deliberation.

Alpha–beta search is in many ways the two-player analog of depth-first branch-and-bound, which is dominated by A* in the single-agent case. The SSS* algorithm (Stockman, 1979) can be viewed as a two-player A* and never expands more nodes than alpha–beta to reach the same decision. The memory requirements and computational overhead of the queue make SSS* in its original form impractical, but a linear-space version has been developed from the RBFS algorithm (Korf and Chickering, 1996). Plaat *et al.* (1996) developed a new view of SSS* as a combination of alpha–beta and transposition tables, showing how to overcome the drawbacks of the original algorithm and developing a new variant called MTD(*f*) that has been adopted by a number of top programs.

D. F. Beal (1980) and Dana Nau (1980, 1983) studied the weaknesses of minimax applied to approximate evaluations. They showed that under certain independence assumptions about the distribution of leaf values in the tree, minimaxing can yield values at the root that are actually *less* reliable than the direct use of the evaluation function itself. Pearl's book *Heuristics* (1984) partially explains this apparent paradox and analyzes many game-playing algorithms. Baum and Smith (1997) propose a probability-based replacement for minimax, showing that it results in better choices in certain games. There is still little theory of the effects of cutting off search at different levels and applying evaluation functions.

The expectiminimax algorithm was proposed by Donald Michie (1966), although of course it follows directly from the principles of game-tree evaluation due to von Neumann and Morgenstern. Bruce Ballard (1983) extended alpha–beta pruning to cover trees with chance nodes. The first successful backgammon program was BKG (Berliner, 1977, 1980b); it used a complex, manually constructed evaluation function and searched only to depth 1. It was the first program to defeat a human world champion at a major classic game (Berliner, 1980a). Berliner readily acknowledged that this was a very short exhibition match (not a world championship match) and that BKG was very lucky with the dice. Work by Gerry Tesauro, first on NEUROGAMMON (Tesauro, 1989) and later on TD-GAMMON (Tesauro, 1995), showed that much better results could be obtained via reinforcement learning, which we will cover in Chapter 21.

Checkers, rather than chess, was the first of the classic games fully played by a computer. Christopher Strachey (1952) wrote the first working program for checkers. Schaeffer (1997) gives a highly readable, "warts and all" account of the development of his Chinook world champion checkers program.

The first Go-playing programs were developed somewhat later than those for checkers and chess (Lefkovitz, 1960; Remus, 1962) and have progressed more slowly. Ryder (1971) used a pure search-based approach with a variety of selective pruning methods to overcome the enormous branching factor. Zobrist (1970) used condition–action rules to suggest plausible moves when known patterns appeared. Reitman and Wilcox (1979) combined rules and search to good effect, and most modern programs have followed this hybrid approach. Müller (2002) summarizes the state of the art of computerized Go and provides a wealth of references. Anshelevich (2000) uses related techniques for the game of Hex. The *Computer Go Newsletter*, published by the Computer Go Association, describes current developments.

Papers on computer game playing appear in a variety of venues. The rather misleadingly named conference proceedings *Heuristic Programming in Artificial Intelligence* report on the Computer Olympiads, which include a wide variety of games. There are also several edited collections of important papers on game-playing research (Levy, 1988a, 1988b; Marsland and Schaeffer, 1990). The International Computer Chess Association (ICCA), founded in 1977, publishes the quarterly *ICGA Journal* (formerly the *ICCA Journal*). Important papers have been published in the serial anthology *Advances in Computer Chess*, starting with Clarke (1977). Volume 134 of the journal *Artificial Intelligence* (2002) contains descriptions of state-of-the-art programs for chess, Othello, Hex, shogi, Go, backgammon, poker, Scrabble.™ and other games.

EXERCISES

6.1 This problem exercises the basic concepts of game playing, using tic-tac-toe (noughts and crosses) as an example. We define X_n as the number of rows, columns, or diagonals with exactly n X's and no O's. Similarly, O_n is the number of rows, columns, or diagonals with just n O's. The utility function assigns $+1$ to any position with $X_3 = 1$ and -1 to any

position with $O_3 = 1$. All other terminal positions have utility 0. For nonterminal positions, we use a linear evaluation function defined as $Eval(s) = 3X_2(s)+X_1(s)-(3O_2(s)+O_1(s))$.

a. Approximately how many possible games of tic-tac-toe are there?

b. Show the whole game tree starting from an empty board down to depth 2 (i.e., one X and one O on the board), taking symmetry into account.

c. Mark on your tree the evaluations of all the positions at depth 2.

d. Using the minimax algorithm, mark on your tree the backed-up values for the positions at depths 1 and 0, and use those values to choose the best starting move.

e. Circle the nodes at depth 2 that would *not* be evaluated if alpha–beta pruning were applied, assuming the nodes are generated *in the optimal order for alpha–beta pruning.*

6.2 Prove the following assertion: for every game tree, the utility obtained by MAX using minimax decisions against a suboptimal MIN will be never be lower than the utility obtained playing against an optimal MIN. Can you come up with a game tree in which MAX can do still better using a *suboptimal* strategy against a suboptimal MIN?

6.3 Consider the two-player game described in Figure 6.14.

a. Draw the complete game tree, using the following conventions:

- Write each state as (s_A, s_B) where s_A and s_B denote the token locations.
- Put each terminal state in a square boxes and write its game value in a circle.
- Put *loop states* (states that already appear on the path to the root) in double square boxes. Since it is not clear how to assign values to loop states, annotate each with a "?" in a circle.

b. Now mark each node with its backed-up minimax value (also in a circle). Explain how you handled the "?" values and why.

c. Explain why the standard minimax algorithm would fail on this game tree and briefly sketch how you might fix it, drawing on your answer to (b). Does your modified algorithm give optimal decisions for all games with loops?

d. This 4-square game can be generalized to n squares for any $n > 2$. Prove that A wins if n is even and loses if n is odd.

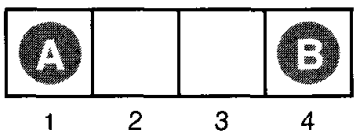


Figure 6.14 The starting position of a simple game. Player A moves first. The two players take turns moving, and each player must move his token to an open adjacent space *in either direction.* If the opponent occupies an adjacent space, then a player may jump over the opponent to the next open space if any. (For example, if A is on 3 and B is on 2, then A may move back to 1.) The game ends when one player reaches the opposite end of the board. If player A reaches space 4 first, then the value of the game to A is $+1$; if player B reaches space 1 first, then the value of the game to A is -1.

6.4 Implement move generators and evaluation functions for one or more of the following games: Kalah, Othello, checkers, and chess. Construct a general alpha–beta game-playing agent that uses your implementation. Compare the effect of increasing search depth, improving move ordering, and improving the evaluation function. How close does your effective branching factor come to the ideal case of perfect move ordering?

6.5 Develop a formal proof of correctness for alpha-beta pruning. To do this, consider the situation shown in Figure 6.15. The question is whether to prune node n_j, which is a max-node and a descendant of node n_1. The basic idea is to prune it if and only if the minimax value of n_1 can be shown to be independent of the value of n_j.

a. The value of n_1 is given by

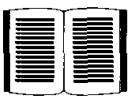
$$n_1 = \min(n_2, n_{21}, \ldots, n_{2b_2}) \ .$$

Find a similar expression for n_2 and hence an expression for n_1 in terms of n_j.

b. Let l_i be the minimum (or maximum) value of the nodes to the *left* of node n_i at depth i, whose minimax value is already known. Similarly, let r_i be the minimum (or maximum) value of the unexplored nodes to the right of n_i at depth i. Rewrite your expression for n_1 in terms of the l_i and r_i values.

c. Now reformulate the expression to show that in order to affect n_1, n_j must not exceed a certain bound derived from the l_i values.

d. Repeat the process for the case where n_j is a min-node.



6.6 Implement the expectiminimax algorithm and the *-alpha–beta algorithm, which is described by Ballard (1983), for pruning game trees with chance nodes. Try them on a game such as backgammon and measure the pruning effectiveness of *-alpha–beta.

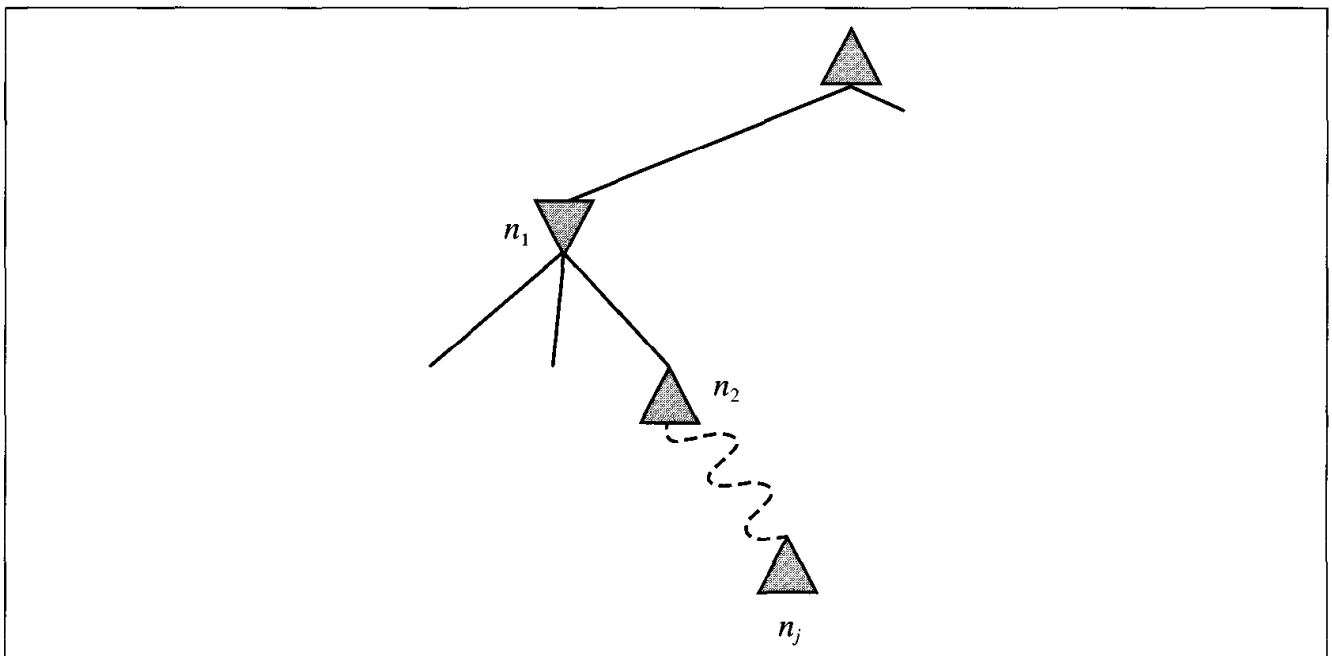


Figure 6.15 Situation when considering whether to prune node n_j.

6.7 Prove that with a positive linear transformation of leaf values (i.e., transforming a value x to $ax + b$ where $a > 0$), the choice of move remains unchanged in a game tree, even when there are chance nodes.

6.8 Consider the following procedure for choosing moves in games with chance nodes:

- Generate some die-roll sequences (say, 50) down to a suitable depth (say, 8).
- With known die rolls, the game tree becomes deterministic. For each die-roll sequence, solve the resulting deterministic game tree using alpha–beta.
- Use the results to estimate the value of each move and to choose the best.

Will this procedure work well? Why (not)?

6.9 Describe and implement a *real-time*, *multiplayer* game-playing environment, where time is part of the environment state and players are given fixed time allocations.

6.10 Describe or implement state descriptions, move generators, terminal tests, utility functions, and evaluation functions for one or more of the following games: Monopoly, Scrabble, bridge (assuming a given contract), and poker (choose your favorite variety).

6.11 Consider carefully the interplay of chance events and partial information in each of the games in Exercise 6.10.

a. For which is the standard expectiminimax model appropriate? Implement the algorithm and run it in your game-playing agent, with appropriate modifications to the game-playing environment.

b. For which would the scheme described in Exercise 6.8 be appropriate?

c. Discuss how you might deal with the fact that in some of the games, the players do not have the same knowledge of the current state.

6.12 The minimax algorithm assumes that players take turns moving, but in card games such as whist and bridge, the winner of the previous trick plays first on the next trick.

a. Modify the algorithm to work properly for these games. You may assume that a function WINNER(s) is available that reports which player won the trick just completed (if any).

b. Draw the game tree for the first pair of hands shown on page 179.

6.13 The Chinook checkers program makes extensive use of endgame databases, which provide exact values for every position with eight or fewer pieces. How might such databases be generated efficiently?

6.14 Discuss how well the standard approach to game playing would apply to games such as tennis, pool, and croquet, which take place in a continuous physical state space.

6.15 Describe how the minimax and alpha–beta algorithms change for two-player, **non-zero-sum games** in which each player has his or her own utility function. You may assume that each player knows the other's utility function. If there are no constraints on the two terminal utilities, is it possible for any node to be pruned by alpha–beta?

6.16 Suppose you have a chess program that can evaluate 1 million nodes per second. Decide on a compact representation of a game state for storage in a transposition table. About how many entries can you fit in a 500MB in-memory table? Will that be enough for the three minutes of search allocated for one move? How many table lookups can you do in the time it would take to do one evaluation? Now suppose the transposition table is larger than can fit in memory. About how many evaluations could you do in the time it takes to do one disk seek with standard disk hardware?

7 LOGICAL AGENTS

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

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

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

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

Understanding natural language also requires inferring hidden state, namely, the intention of the speaker. When we hear, "John saw the diamond through the window and coveted it," we know "it" refers to the diamond and not the window—we reason, perhaps unconsciously, with our knowledge of relative value. Similarly, when we hear, "John threw the brick through the window and broke it," we know "it" refers to the window. Reasoning allows

us to cope with the virtually infinite variety of utterances using a finite store of commonsense knowledge. Problem-solving agents have difficulty with this kind of ambiguity because their representation of contingency problems is inherently exponential.

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

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

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

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

7.1 KNOWLEDGE-BASED AGENTS

KNOWLEDGE BASE
SENTENCE
KNOWLEDGE REPRESENTATION LANGUAGE

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

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

INFERENCE
LOGICAL AGENTS

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

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

Figure 7.1 A generic knowledge-based agent.

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

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

BACKGROUND KNOWLEDGE

The details of the representation language are hidden inside three functions that implement the interface between the sensors and actuators and the core representation and reasoning system. MAKE-PERCEPT-SENTENCE constructs a sentence asserting that the agent perceived the given percept at the given time. MAKE-ACTION-QUERY constructs a sentence that asks what action should be done at the current time. Finally, MAKE-ACTION-SENTENCE constructs a sentence asserting that the chosen action was executed. The details of the inference mechanisms are hidden inside TELL and ASK. Later sections will reveal these details.

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

KNOWLEDGE LEVEL

IMPLEMENTATION LEVEL

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

DECLARATIVE

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

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

7.2 THE WUMPUS WORLD

WUMPUS WORLD

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

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

- ◇ **Performance measure**: +1000 for picking up the gold, −1000 for falling into a pit or being eaten by the wumpus, −1 for each action taken and −10 for using up the arrow.
- ◇ **Environment**: A 4×4 grid of rooms. The agent always starts in the square labeled [1,1], facing to the right. The locations of the gold and the wumpus are chosen randomly, with a uniform distribution, from the squares other than the start square. In addition, each square other than the start can be a pit, with probability 0.2.
- ◇ **Actuators**: The agent can move forward, turn left by 90°, or turn right by 90°. The agent dies a miserable death if it enters a square containing a pit or a live wumpus. (It is safe, albeit smelly, to enter a square with a dead wumpus.) Moving forward has no

effect if there is a wall in front of the agent. The action *Grab* can be used to pick up an object that is in the same square as the agent. The action *Shoot* can be used to fire an arrow in a straight line in the direction the agent is facing. The arrow continues until it either hits (and hence kills) the wumpus or hits a wall. The agent only has one arrow, so only the first *Shoot* action has any effect.

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

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

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

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

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

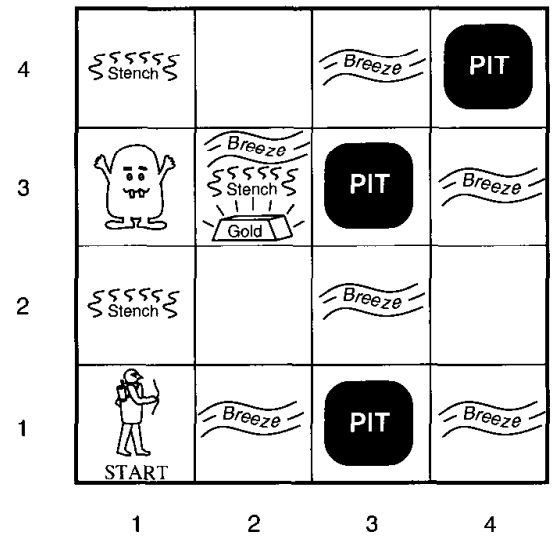


Figure 7.2 A typical wumpus world. The agent is in the bottom left corner.

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

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

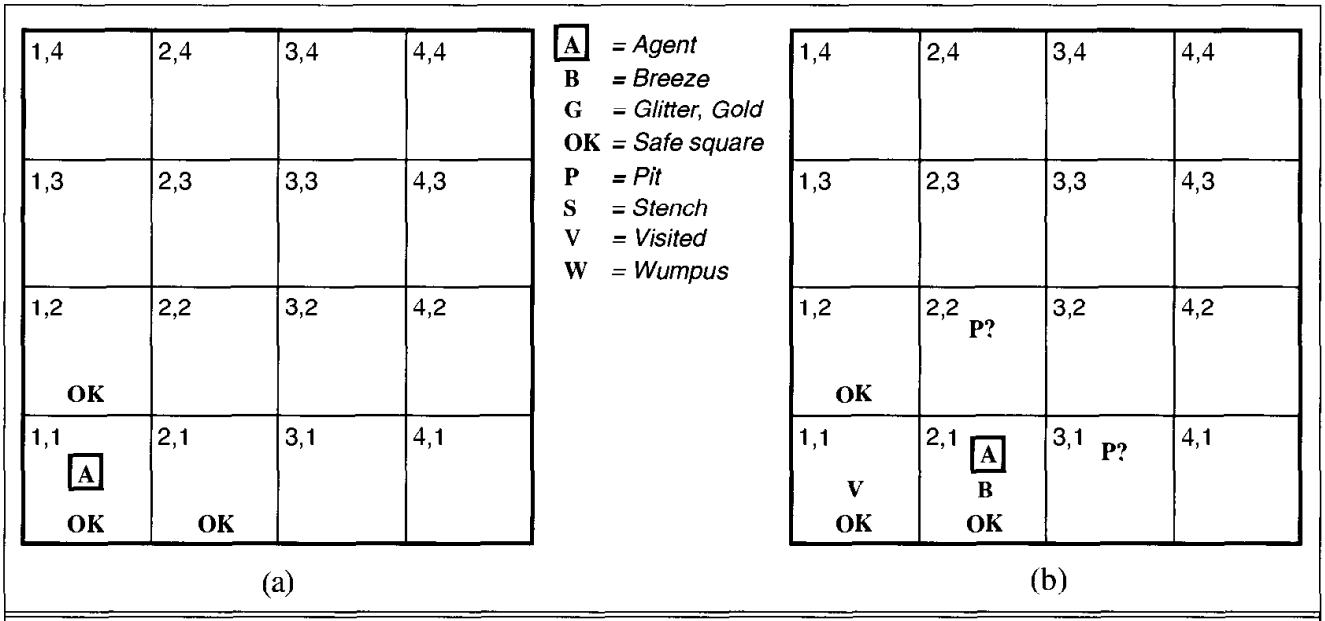


Figure 7.3 The first step taken by the agent in the wumpus world. (a) The initial situation, after percept $[None, None, None, None, None]$. (b) After one move, with percept $[None, Breeze, None, None, None]$.

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

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

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

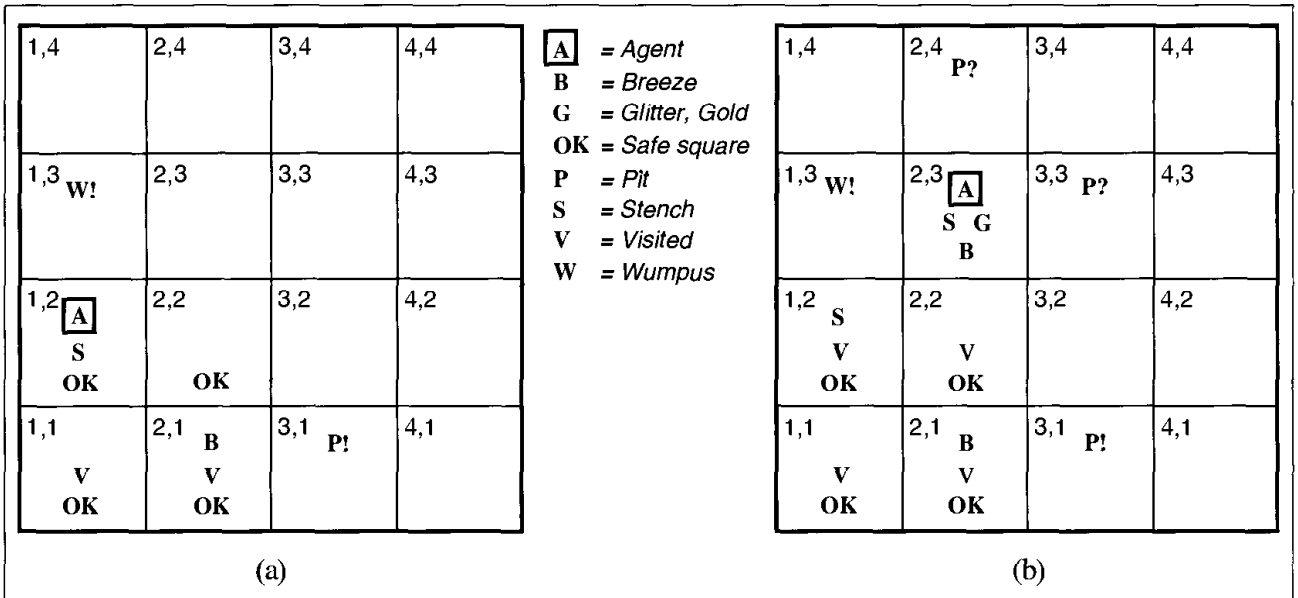


Figure 7.4 Two later stages in the progress of the agent. (a) After the third move, with percept [*Stench*, *None*, *None*, *None*, *None*]. (b) After the fifth move, with percept [*Stench*, *Breeze*, *Glitter*, *None*, *None*].

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

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

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

7.3 Logic

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

SYNTAX

In Section 7.1, we said that knowledge bases consist of sentences. These sentences are expressed according to the **syntax** of the representation language, which specifies all the sentences that are well formed. The notion of syntax is clear enough in ordinary arithmetic: "$x + y = 4$" is a well-formed sentence, whereas "$x2y+ =$" is not. The syntax of logical

languages (and of arithmetic, for that matter) is usually designed for writing papers and books. There are literally dozens of different syntaxes, some with lots of Greek letters and exotic mathematical symbols, and some with rather visually appealing diagrams with arrows and bubbles. In all cases, however, sentences in an agent's knowledge base are real physical configurations of (parts of) the agent. Reasoning will involve generating and manipulating those configurations.

SEMANTICS

A logic must also define the **semantics** of the language. Loosely speaking, semantics has to do with the "meaning" of sentences. In logic, the definition is more precise. The

TRUTH

semantics of the language defines the **truth** of each sentence with respect to each **possible**

POSSIBLE WORLD

world. For example, the usual semantics adopted for arithmetic specifies that the sentence "$x + y = 4$" is true in a world where x is 2 and y is 2, but false in a world where x is 1 and y is 1.[1] In standard logics, every sentence must be either true or false in each possible world—there is no "in between."[2]

MODEL

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

Now that we have a notion of truth, we are ready to talk about logical reasoning. This

ENTAILMENT

involves the relation of logical **entailment** between sentences—the idea that a sentence *follows logically* from another sentence. In mathematical notation, we write as

$$\alpha \models \beta$$

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

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

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

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

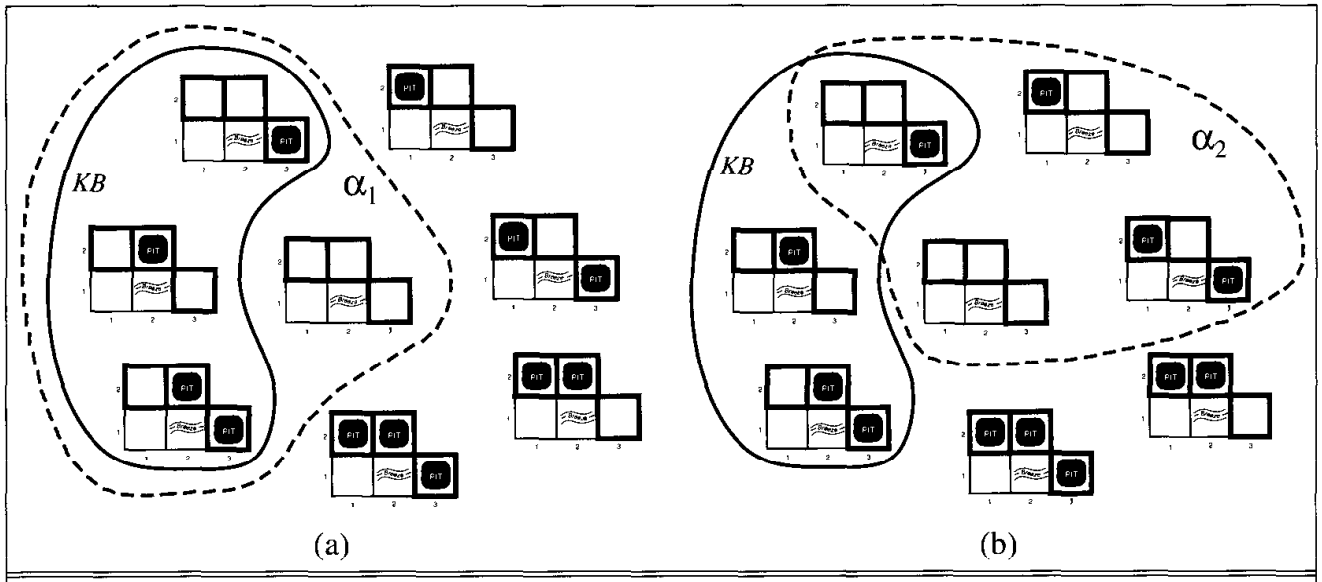


Figure 7.5 Possible models for the presence of pits in squares [1,2], [2,2], and [3,1], given observations of nothing in [1,1] and a breeze in [2,1]. (a) Models of the knowledge base and α_1 (no pit in [1,2]). (b) Models of the knowledge base and α_2 (no pit in [2,2]).

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

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

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

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

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

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

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

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

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

LOGICAL INFERENCE

MODEL CHECKING

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

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

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

$$KB \vdash_i \alpha ,$$

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

SOUND
TRUTH-PRESERVING

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

COMPLETENESS

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

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

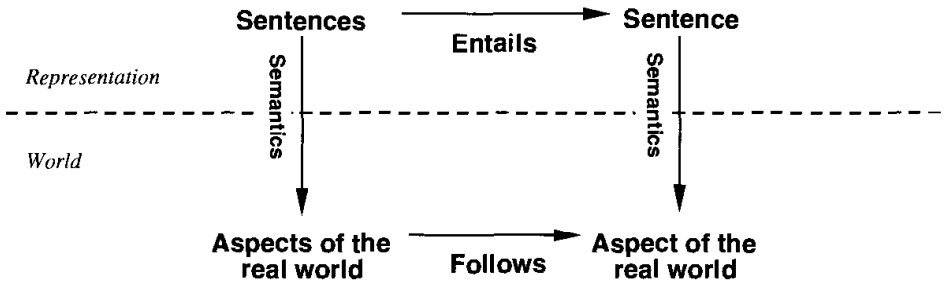


Figure 7.6 Sentences are physical configurations of the agent, and reasoning is a process of constructing new physical configurations from old ones. Logical reasoning should ensure that the new configurations represent aspects of the world that actually follow from the aspects that the old configurations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

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

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

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

GROUNDING

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

7.4 PROPOSITIONAL LOGIC: A VERY SIMPLE LOGIC

PROPOSITIONAL LOGIC

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

Syntax

ATOMIC SENTENCES
PROPOSITION SYMBOL

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

COMPLEX SENTENCES
LOGICAL CONNECTIVES

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

NEGATION
LITERAL

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

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

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

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

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

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

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

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

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

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

$$\neg P \vee Q \wedge R \Rightarrow S$$

is equivalent to the sentence

$$((\neg P) \vee (Q \wedge R)) \Rightarrow S\,.$$

Precedence does not resolve ambiguity in sentences such as $A \wedge B \wedge C$, which could be read as $((A \wedge B) \wedge C)$ or as $(A \wedge (B \wedge C))$. Because these two readings mean the same thing according to the semantics given in the next section, sentences such as $A \wedge B \wedge C$ are allowed. We also allow $A \vee B \vee C$ and $A \Leftrightarrow B \Leftrightarrow C$. Sentences such as $A \Rightarrow B \Rightarrow C$ are not

allowed because the two readings have different meanings; we insist on parentheses in this case. Finally, we will sometimes use square brackets instead of parentheses when it makes the sentence clearer.

Semantics

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

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

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

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

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

For complex sentences, we have rules such as

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

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

TRUTH TABLE

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

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

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

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

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

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

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

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

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

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

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

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

A simple knowledge base

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

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

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

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

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

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

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

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

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

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

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

Inference

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

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

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

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

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

  symbols ← a list of the proposition symbols in KB and α
  return TT-CHECK-ALL(KB, α, *symbols*, [])

---

function TT-CHECK-ALL(KB, α, *symbols*, *model*) **returns** *true* or *false*
  if EMPTY?(*symbols*) **then**
    if PL-TRUE?(KB, *model*) **then return** PL-TRUE?(α, *model*)
    else return *true*
  else do
    P ← FIRST(*symbols*); *rest* ← REST(*symbols*)
    return TT-CHECK-ALL(KB, α, *rest*, EXTEND(P, *true*, *model*)) **and**
      TT-CHECK-ALL(KB, α, *rest*, EXTEND(P, *false*, *model*))

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

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

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

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

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

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

Equivalence, validity, and satisfiability

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

LOGICAL EQUIVALENCE

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

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

(Recall that $\models$ means entailment.)

VALIDITY

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

TAUTOLOGY

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

What good are valid sentences? From our definition of entailment, we can derive the

DEDUCTION THEOREM

deduction theorem, which was known to the ancient Greeks:

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

(Exercise 7.4 asks for a proof.) We can think of the inference algorithm in Figure 7.10 as

checking the validity of $(KB \Rightarrow \alpha)$. Conversely, every valid implication sentence describes a legitimate inference.

SATISFIABILITY

The final concept we will need is **satisfiability**. A sentence is satisfiable if it is true in *some* model. For example, the knowledge base given earlier, $(R_1 \wedge R_2 \wedge R_3 \wedge R_4 \wedge R_5)$, is satisfiable because there are three models in which it is true, as shown in Figure 7.9. If

SATISFIES

a sentence α is true in a model m, then we say that m **satisfies** α, or that m **is a model of** α. Satisfiability can be checked by enumerating the possible models until one is found that satisfies the sentence. Determining the satisfiability of sentences in propositional logic was the first problem proved to be NP-complete.

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

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

Proving β from α by checking the unsatisfiability of $(\alpha \wedge \neg\beta)$ corresponds exactly to the

REDUCTIO AD ABSURDUM

standard mathematical proof technique of *reductio ad absurdum* (literally, "reduction to an

REFUTATION

absurd thing"). It is also called proof by **refutation** or proof by **contradiction**. One assumes a sentence β to be false and shows that this leads to a contradiction with known axioms α. This contradiction is exactly what is meant by saying that the sentence $(\alpha \wedge \neg\beta)$ is unsatisfiable.

7.5 Reasoning Patterns in Propositional Logic

This section covers standard patterns of inference that can be applied to derive chains of conclusions that lead to the desired goal. These patterns of inference are called **inference**

INFERENCE RULES

rules. The best-known rule is called **Modus Ponens** and is written as follows:

MODUS PONENS

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

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

AND-ELIMINATION

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

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

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

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

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

$$\frac{\alpha \Leftrightarrow \beta}{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)} \quad \text{and} \quad \frac{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)}{\alpha \Leftrightarrow \beta} .$$

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

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

$$R_6 : \quad (B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}) .$$

Then we apply And-Elimination to R_6 to obtain

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

Logical equivalence for contrapositives gives

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

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

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

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

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

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

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

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

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

MONOTONICITY

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

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

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

Resolution

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

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

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

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

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

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

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

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

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

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

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

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

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

UNIT RESOLUTION

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

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

COMPLEMENTARY LITERALS

CLAUSE

UNIT CLAUSE

RESOLUTION

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

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

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

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

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

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

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

FACTORING

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

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

REFUTATION COMPLETENESS

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

Conjunctive normal form

CONJUNCTIVE NORMAL FORM

K-CNF

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

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

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

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

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

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

A resolution algorithm

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

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

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

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

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

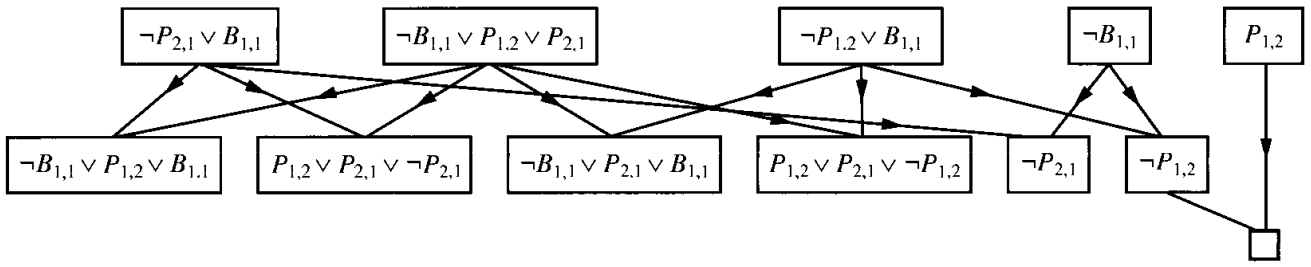


Figure 7.13 Partial application of PL-RESOLUTION to a simple inference in the wumpus world. $\neg P_{1,2}$ is shown to follow from the first four clauses in the top row.

- there are no new clauses that can be added, in which case KB does not entail α; or,
- two clauses resolve to yield the *empty* clause, in which case KB entails α.

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

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

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

and we wish to prove α which is, say, $\neg P_{1,2}$. When we convert $(KB \wedge \neg\alpha)$ into CNF, we obtain the clauses shown at the top of Figure 7.13. The second row of the figure shows all the clauses obtained by resolving pairs in the first row. Then, when $P_{1,2}$ is resolved with $\neg P_{1,2}$, we obtain the empty clause, shown as a small square. Inspection of Figure 7.13 reveals that

many resolution steps are pointless. For example, the clause $B_{1,1} \vee \neg B_{1,1} \vee P_{1,2}$ is equivalent to $True \vee P_{1,2}$ which is equivalent to $True$. Deducing that $True$ is true is not very helpful. Therefore, any clause in which two complementary literals appear can be discarded.

Completeness of resolution

RESOLUTION CLOSURE

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

GROUND RESOLUTION THEOREM

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

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

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

For i from 1 to k,

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

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

Forward and backward chaining

HORN CLAUSES

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

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

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

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

DEFINITE CLAUSES
HEAD
BODY
FACT

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

INTEGRITY CONSTRAINTS

FORWARD CHAINING
BACKWARD CHAINING

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

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

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

The best way to understand the algorithm is through an example and a picture. Figure 7.15(a) shows a simple knowledge base of Horn clauses with A and B as known facts.

AND–OR GRAPH

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

It is easy to see that forward chaining is **sound**: every inference is essentially an application of Modus Ponens. Forward chaining is also **complete**: every entailed atomic sentence will be derived. The easiest way to see this is to consider the final state of the *inferred* table

FIXED POINT

(after the algorithm reaches a **fixed point** where no new inferences are possible). The table

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

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

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

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

DATA-DRIVEN

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

$P \Rightarrow Q$

$L \wedge M \Rightarrow P$

$B \wedge L \Rightarrow M$

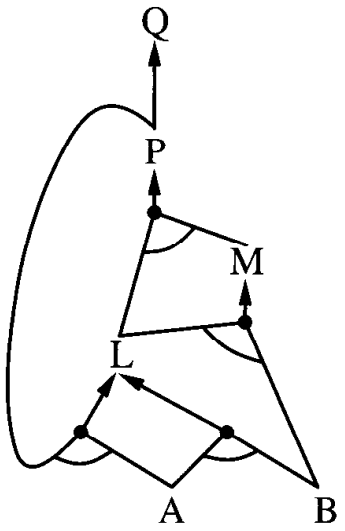
$A \wedge P \Rightarrow L$

$A \wedge B \Rightarrow L$

A

B

(a)



(b)

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

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

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

GOAL-DIRECTED REASONING

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

7.6 EFFECTIVE PROPOSITIONAL INFERENCE

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

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

A complete backtracking algorithm

DAVIS–PUTNAM ALGORITHM

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

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

PURE SYMBOL

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

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

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

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

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

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

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

UNIT PROPAGATION

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

Local-search algorithms

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

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

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

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

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

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

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

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

Hard satisfiability problems

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

UNDERCONSTRAINED

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

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

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

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

CRITICAL POINT

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

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

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

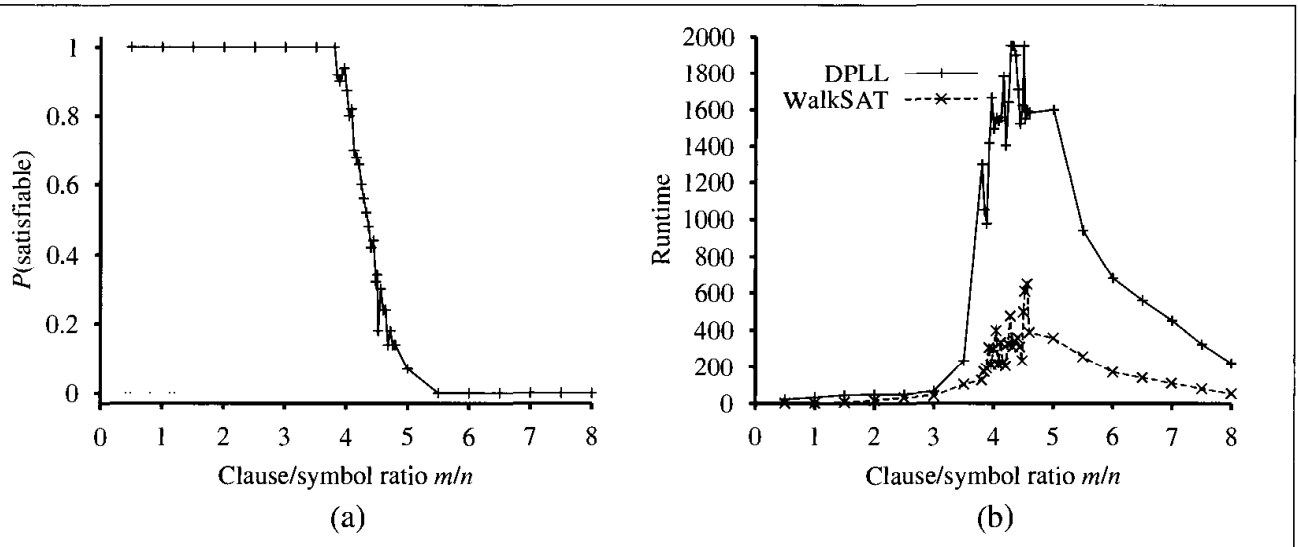


Figure 7.18 (a) Graph showing the probability that a random 3-CNF sentence with $n = 50$ symbols is satisfiable, as a function of the clause/symbol ratio m/n. (b) Graph of the median runtime of DPLL and WALKSAT on 100 *satisfiable* random 3-CNF sentences with $n = 50$, for a narrow range of m/n around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

7.7 AGENTS BASED ON PROPOSITIONAL LOGIC

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

Finding pits and wumpuses using logical inference

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

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

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

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

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

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

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

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

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

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

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

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

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

PL-WUMPUS-AGENT works quite well in a small wumpus world. There is, however, something deeply unsatisfying about the agent's knowledge base. *KB* contains "physics" sentences of the form given in Equations (7.1) and (7.2) *for every single square*. The larger

the environment, the larger the initial knowledge base needs to be. We would much prefer to have just two sentences that say how breezes and stenches arise in *all* squares. These are beyond the powers of propositional logic to express. In the next chapter, we will see a more expressive logical language in which such sentences are easy to express.

Keeping track of location and orientation

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

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

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

$$L^1_{1,1} \wedge \mathit{FacingRight}^1 \wedge \mathit{Forward}^1 \Rightarrow L^2_{2,1}$$
$$\mathit{FacingRight}^1 \wedge \mathit{TurnLeft}^1 \Rightarrow \mathit{FacingUp}^2 ,$$

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

$$L^t_{x,y} \wedge \mathit{FacingRight}^t \wedge \mathit{Forward}^t \Rightarrow L^{t+1}_{x+1,y} . \tag{7.3}$$

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

Circuit-based agents

CIRCUIT-BASED AGENT
SEQUENTIAL CIRCUIT
GATES
REGISTERS

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

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

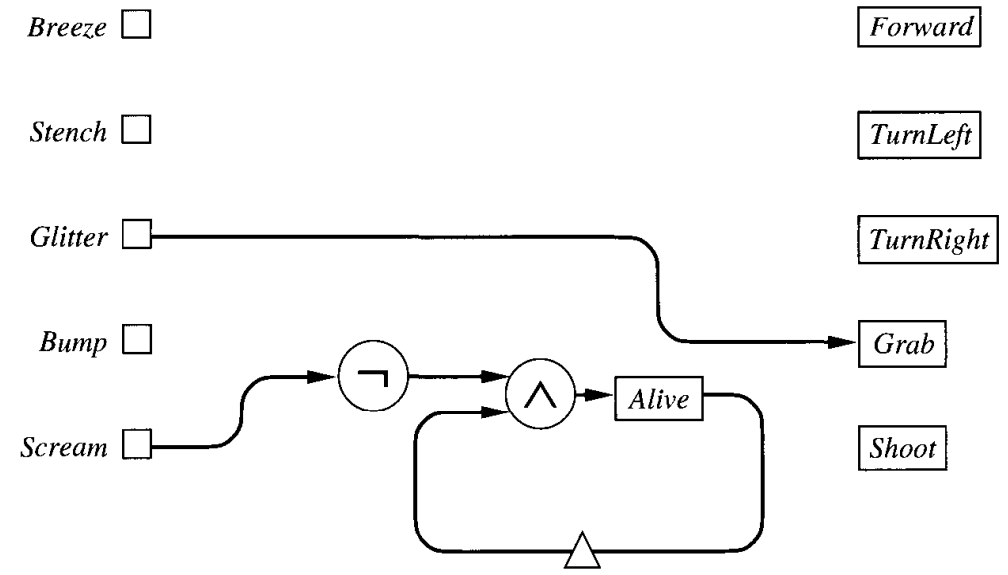


Figure 7.20 Part of a circuit-based agent for the wumpus world, showing inputs, outputs, the circuit for grabbing the gold, and the circuit for determining whether the wumpus is alive. Registers are shown as rectangles and one-step delays are shown as small triangles.

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

DATAFLOW

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

We said in the preceding section that circuit-based agents handle time more satisfactorily than propositional inference-based agents. This is because the value stored in each register gives the truth value of the corresponding proposition symbol *at the current time* t, rather than having a different copy for each different time step. For example, we might have an *Alive* register that should contain *true* when the wumpus is alive and *false* when it is dead. This register corresponds to the proposition symbol $Alive^t$, so on each time step it refers to a different proposition. The internal state of the agent—i.e., its memory—is maintained by

DELAY LINE

connecting the output of a register back into the circuit through a **delay line**. This delivers the value of the register at the *previous* time step. Figure 7.20 shows an example. The value for *Alive* is given by the conjunction of the negation of *Scream* and the delayed value of *Alive* itself. In terms of propositions, the circuit for *Alive* implements the biconditional

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

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

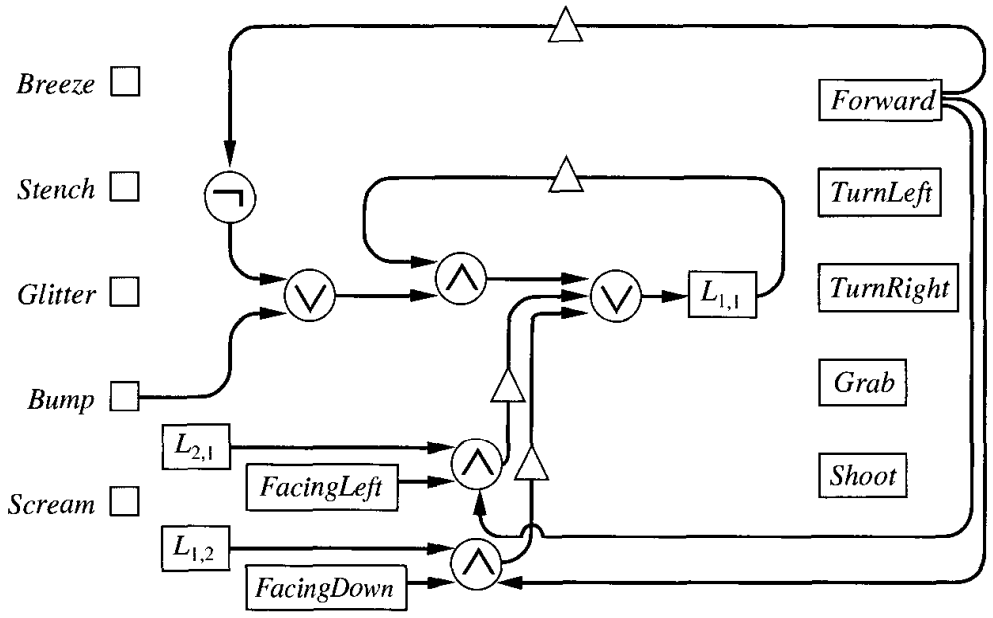


Figure 7.21 The circuit for determining whether the agent is at [1,1]. Every location and orientation register has a similar circuit attached.

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

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

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

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

The solution to this problem is to use two bits instead of one. $B_{4,4}$ is represented by two registers that we will call $K(B_{4,4})$ and $K(\neg B_{4,4})$, where K stands for "known.". (Remember that these are still just symbols with complicated names, even though they look like structured

expressions!) When both $K(B_{4,4})$ and $K(\neg B_{4,4})$ are false, it means the truth value of $B_{4,4}$ is unknown. (If both are true, there's a bug in the knowledge base!) Now whenever we would use $B_{4,4}$ in some part of the circuit, we use $K(B_{4,4})$ instead ; and whenever we would use $\neg B_{4,4}$, we use $K(\neg B_{4,4})$. In general, we represent each potentially indeterminate proposition with two **knowledge propositions** that state whether the underlying proposition is known to be true and known to be false.

KNOWLEDGE PROPOSITION

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

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

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

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

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

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

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

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

ACYCLICITY

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

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

The extra disjuncts, $K(P_{3,4})^t$ and $K(P_{4,3})^t$, allow the agent to determine breeziness from the known presence of adjacent pits, which seems entirely reasonable. Now, unfortunately,

breeziness depends on adjacent pits, and pits depend on adjacent breeziness through equations such as Equation (7.8). Therefore, the complete circuit would contain cycles.

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

A comparison

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

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

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

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

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

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

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

COMPILATION

7.8 SUMMARY

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

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

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

BIBLIOGRAPHICAL AND HISTORICAL NOTES

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

Logic itself had its origins in ancient Greek philosophy and mathematics. Various logical principles—principles connecting the syntactic structure of sentences with their truth and falsity, with their meaning, or with the validity of arguments in which they figure—are scattered in the works of Plato. The first known systematic study of logic was carried out by Aristotle, whose work was assembled by his students after his death in 322 B.C. as a treatise called the *Organon*. Aristotle's **syllogisms** were what we would now call inference rules. Although the syllogisms included elements of both propositional and first-order logic, the system as a whole was very weak by modern standards. It did not allow for patterns of inference that apply to sentences of arbitrary complexity, as in modern propositional logic.

SYLLOGISMS

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

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

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

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

Truth tables as a method of testing the validity or unsatisfiability of sentences in the language of propositional logic were introduced independently by Ludwig Wittgenstein (1922) and Emil Post (1921). In the 1930s, a great deal of progress was made on inference methods for first-order logic. In particular, Gödel (1930) showed that a complete procedure for inference in first-order logic could be obtained via a reduction to propositional logic, using Herbrand's theorem (Herbrand, 1930). We will take up this history again in Chapter 9; the important point here is that the development of efficient propositional algorithms in the 1960s was motivated largely by the interest of mathematicians in an effective theorem prover

for first-order logic. The Davis–Putnam algorithm (Davis and Putnam, 1960) was the first effective algorithm for propositional resolution but was in most cases much less efficient than the DPLL backtracking algorithm introduced two years later (1962). The full resolution rule and a proof of its completeness appeared in a seminal paper by J. A. Robinson (1965), which also showed how to do first-order reasoning without resort to propositional techniques.

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

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

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

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

are still a very active area of research; the collection of articles in Du *et al.* (1999) provides a good starting point.

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

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

EXERCISES

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

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

α_2 = "There is no pit in [2,2]."
α_3 = "There is a wumpus in [1,3]."

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

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

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

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

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

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

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

7.4 Prove each of the following assertions:

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

b. For any α, $False \models \alpha$.

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

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

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

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

a. $(A \wedge B) \vee (B \wedge C)$

b. $A \vee B$

c. $A \Leftrightarrow B \Leftrightarrow C$

7.6 We have defined four different binary logical connectives.

a. Are there any others that might be useful?

b. How many binary connectives can there be?

c. Why are some of them not very useful?

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

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

a. $Smoke \Rightarrow Smoke$

b. $Smoke \Rightarrow Fire$

c. $(Smoke \Rightarrow Fire) \Rightarrow (\neg Smoke \Rightarrow \neg Fire)$

d. $Smoke \vee Fire \vee \neg Fire$

e. $((Smoke \wedge Heat) \Rightarrow Fire) \Leftrightarrow ((Smoke \Rightarrow Fire) \vee (Heat \Rightarrow Fire))$

f. $(Smoke \Rightarrow Fire) \Rightarrow ((Smoke \wedge Heat) \Rightarrow Fire)$

g. $Big \vee Dumb \vee (Big \Rightarrow Dumb)$

h. $(Big \wedge Dumb) \vee \neg Dumb$

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

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

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

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

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

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

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

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

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

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

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

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

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

IMPLICATIVE NORMAL FORM

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

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

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

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

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

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

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

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

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

8 FIRST-ORDER LOGIC

In which we notice that the world is blessed with many objects, some of which are related to other objects, and in which we endeavor to reason about them.

In Chapter 7, we showed how a knowledge-based agent could represent the world in which it operates and deduce what actions to take. We used propositional logic as our representation language because it sufficed to illustrate the basic concepts of logic and knowledge-based agents. Unfortunately, propositional logic is too puny a language to represent knowledge of complex environments in a concise way. In this chapter, we examine **first-order logic**,[1] which is sufficiently expressive to represent a good deal of our commonsense knowledge. It also either subsumes or forms the foundation of many other representation languages and has been studied intensively for many decades. We begin in Section 8.1 with a discussion of representation languages in general; Section 8.2 covers the syntax and semantics of first-order logic; Sections 8.3 and 8.4 illustrate the use of first-order logic for simple representations.

FIRST-ORDER LOGIC

8.1 REPRESENTATION REVISITED

In this section, we will discuss the nature of representation languages. Our discussion will motivate the development of first-order logic, a much more expressive language than the propositional logic introduced in Chapter 7. We will look at propositional logic and at other kinds of languages to understand what works and what fails. Our discussion will be cursory, compressing centuries of thought, trial, and error into a few paragraphs.

Programming languages (such as C++ or Java or Lisp) are by far the largest class of formal languages in common use. Programs themselves represent, in a direct sense, only computational processes. Data structures within programs can represent facts; for example, a program could use a 4×4 array to represent the contents of the wumpus world. Thus, the programming language statement $World[2,2] \leftarrow Pit$ is a fairly natural way to assert that there is a pit in square [2,2]. (Such representations might be considered *ad hoc*; database systems were developed precisely to provide a more general, domain-independent way to

[1] Also called **first-order predicate calculus**, sometimes abbreviated as **FOL** or **FOPC**.

store and retrieve facts.) What programming languages lack is any general mechanism for deriving facts from other facts; each update to a data structure is done by a domain-specific procedure whose details are derived by the programmer from his or her own knowledge of the domain. This **procedural** approach can be contrasted with the **declarative** nature of propositional logic, in which knowledge and inference are separate, and inference is entirely domain-independent.

A second drawback of data structures in programs (and of databases, for that matter) is the lack of any easy way to say, for example, "There is a pit in [2,2] or [3,1]" or "If the wumpus is in [1,1] then he is not in [2,2]." Programs can store a single value for each variable, and some systems allow the value to be "unknown," but they lack the expressiveness required to handle partial information.

Propositional logic is a declarative language because its semantics is based on a truth relation between sentences and possible worlds. It also has sufficient expressive power to deal with partial information, using disjunction and negation. Propositional logic has a third property that is desirable in representation languages, namely **compositionality**. In a compositional language, the meaning of a sentence is a function of the meaning of its parts. For example, "$S_{1,4} \wedge S_{1,2}$" is related to the meanings of "$S_{1,4}$" and "$S_{1,2}$." It would be very strange if "$S_{1,4}$" meant that there is a stench in square [1,4] and "$S_{1,2}$" meant that there is a stench in square [1,2], but "$S_{1,4} \wedge S_{1,2}$" meant that France and Poland drew 1–1 in last week's ice hockey qualifying match. Clearly, noncompositionality makes life much more difficult for the reasoning system.

COMPOSITIONALITY

As we saw in Chapter 7, propositional logic lacks the expressive power to describe an environment with many objects *concisely*. For example, we were forced to write a separate rule about breezes and pits for each square, such as

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

In English, on the other hand, it seems easy enough to say, once and for all, "Squares adjacent to pits are breezy." The syntax and semantics of English somehow make it possible to describe the environment concisely.

A moment's thought suggests that natural languages (such as English or Spanish) are very expressive indeed. We managed to write almost this whole book in natural language, with only occasional lapses into other languages (including logic, mathematics, and the language of diagrams). There is a long tradition in linguistics and the philosophy of language that views natural language as essentially a declarative knowledge representation language and attempts to pin down its formal semantics. Such a research program, if successful, would be of great value to artificial intelligence because it would allow a natural language (or some derivative) to be used within representation and reasoning systems.

The modern view of natural language is that it serves a somewhat different purpose, namely as a medium for **communication** rather than pure representation. When a speaker points and says, "Look!" the listener comes to know that, say, Superman has finally appeared over the rooftops. Yet we would not want to say that the sentence "Look!" encoded that fact. Rather, the meaning of the sentence depends both on the sentence itself and on the **context** in which the sentence was spoken. Clearly, one could not store a sentence such as "Look!" in

a knowledge base and expect to recover its meaning without also storing a representation of the context—which raises the question of how the context itself can be represented. Natural languages are also noncompositional—the meaning of a sentence such as "Then she saw it" can depend on a context constructed by many preceding and succeeding sentences. Finally, natural languages suffer from **ambiguity**, which would cause difficulties for thinking. As Pinker (1995) puts it: "When people think about *spring*, surely they are not confused as to whether they are thinking about a season or something that goes *boing*—and if one word can correspond to two thoughts, thoughts can't be words."

Our approach will be to adopt the foundation of propositional logic—a declarative, compositional semantics that is context-independent and unambiguous—and build a more expressive logic on that foundation, borrowing representational ideas from natural language while avoiding its drawbacks. When we look at the syntax of natural language, the most obvious elements are nouns and noun phrases that refer to **objects** (squares, pits, wumpuses) and verbs and verb phrases that refer to **relations** among objects (is breezy, is adjacent to, shoots). Some of these relations are **functions**—relations in which there is only one "value" for a given "input." It is easy to start listing examples of objects, relations, and functions:

OBJECTS
RELATIONS
FUNCTIONS

- Objects: people, houses, numbers, theories, Ronald McDonald, colors, baseball games, wars, centuries ...
- Relations: these can be unary relations or **properties** such as red, round, bogus, prime, multistoried ..., or more general n-ary relations such as brother of, bigger than, inside, part of, has color, occurred after, owns, comes between, ...
- Functions: father of, best friend, third inning of, one more than, beginning of ...

PROPERTIES

Indeed, almost any assertion can be thought of as referring to objects and properties or relations. Some examples follow:

- "One plus two equals three"
  Objects: one, two, three, one plus two; Relation: equals; Function: plus. ("One plus two" is a name for the object that is obtained by applying the function "plus" to the objects "one" and "two." Three is another name for this object.)
- "Squares neighboring the wumpus are smelly."
  Objects: wumpus, squares; Property: smelly; Relation: neighboring.
- "Evil King John ruled England in 1200."
  Objects: John, England, 1200; Relation: ruled; Properties: evil, king.

The language of **first-order logic**, whose syntax and semantics we will define in the next section, is built around objects and relations. It has been so important to mathematics, philosophy, and artificial intelligence precisely because those fields—and indeed, much of everyday human existence—can be usefully thought of as dealing with objects and the relations among them. First-order logic can also express facts about *some* or *all* of the objects in the universe. This enables one to represent general laws or rules, such as the statement "Squares neighboring the wumpus are smelly."

The primary difference between propositional and first-order logic lies in the **ontological commitment** made by each language—that is, what it assumes about the nature of *reality*.

ONTOLOGICAL COMMITMENT

THE LANGUAGE OF THOUGHT

Philosophers and psychologists have long pondered how it is that humans and other animals represent knowledge. It is clear that the evolution of natural language has played an important role in developing this ability in humans. On the other hand, much psychological evidence suggests that humans do not employ language directly in their internal representations. For example, which of the following two phrases formed the opening of Section 8.1?

"In this section, we will discuss the nature of representation languages . . ."

"This section covers the topic of knowledge representation languages . . ."

Wanner (1974) found that subjects made the right choice in such tests at chance level—about 50% of the time—but remembered the content of what they read with better than 90% accuracy. This suggests that people process the words to form some kind of nonverbal representation, which we call **memory**.

The exact mechanism by which language enables and shapes the representation of ideas in humans remains a fascinating question. The famous **Sapir–Whorf hypothesis** claims that the language we speak profoundly influences the way in which we think and make decisions, in particular by setting up the category structure by which we divide up the world into different sorts of objects. Whorf (1956) claimed that Eskimos have many words for snow and thus experience snow in a different way from speakers of other languages. Some linguists dispute the factual basis for this claim—Pullum (1991) argues that Inuit, Yupik, and other related languages seem to have about the same number of words for snow-related concepts as English—while others support it (Fortescue, 1984). It seems unarguably true that populations having greater familiarity with some aspects of the world develop much more detailed vocabularies—for example, field entomologists divide what most of us call *beetles* into hundreds of thousands of species and are personally familiar with many of these. (The evolutionary biologist J. B. S. Haldane once complained of "An inordinate fondness for beetles" on the part of the Creator.) Moreover, expert skiers have many terms for snow—powder, chowder, mashed potatoes, crud, corn, cement, crust, sugar, asphalt, corduroy, fluff, glop, and so on—that represent distinctions unfamiliar to the lay person. What is unclear is the direction of causality—do skiers become aware of the distinctions only by learning the words, or do the distinctions emerge from individual experience and become matched with the labels current in the community? This question is especially important in the study of child development. As yet, we have little understanding of the extent to which learning language and learning to think are intertwined. For example, does the knowledge of a name for a concept, such as *bachelor*, make it easier to construct and reason with more complex concepts that include that name, such as *eligible bachelor*?

For example, propositional logic assumes that there are facts that either hold or do not hold in the world. Each fact can be in one of two states: true or false.[2] First-order logic assumes more; namely, that the world consists of objects with certain relations among them that do or do not hold. Special-purpose logics make still further ontological commitments; for example,

TEMPORAL LOGIC

temporal logic assumes that facts hold at particular *times* and that those times (which may be points or intervals) are ordered. Thus, special-purpose logics give certain kinds of objects (and the axioms about them) "first-class" status within the logic, rather than simply defining them within the knowledge base.

HIGHER-ORDER LOGIC

Higher-order logic views the relations and functions referred to by first-order logic as objects in themselves. This allows one to make assertions about *all* relations—for example, one could wish to define what it means for a relation to be transitive. Unlike most special-purpose logics, higher-order logic is strictly more expressive than first-order logic, in the sense that some sentences of higher-order logic cannot be expressed by any finite number of first-order logic sentences.

EPISTEMOLOGICAL COMMITMENTS

A logic can also be characterized by its **epistemological commitments**—the possible states of knowledge that it allows with respect to each fact. In both propositional and first-order logic, a sentence represents a fact and the agent either believes the sentence to be true, believes it to be false, or has no opinion. These logics therefore have three possible states of knowledge regarding any sentence. Systems using **probability theory**, on the other hand, can have any *degree of belief*, ranging from 0 (total disbelief) to 1 (total belief).[3] For example, a probabilistic wumpus-world agent might believe that the wumpus is in [1,3] with probability 0.75. The ontological and epistemological commitments of five different logics are summarized in Figure 8.1.

Language	Ontological Commitment (What exists in the world)	Epistemological Commitment (What an agent believes about facts)
Propositional logic	facts	true/false/unknown
First-order logic	facts, objects, relations	true/false/unknown
Temporal logic	facts, objects, relations, times	true/false/unknown
Probability theory	facts	degree of belief $\in [0, 1]$
Fuzzy logic	facts with degree of truth $\in [0, 1]$	known interval value

Figure 8.1 Formal languages and their ontological and epistemological commitments.

In the next section, we will launch into the details of first-order logic. Just as a student of physics requires some familiarity with mathematics, a student of AI must develop a talent for working with logical notation. On the other hand, it is also important *not* to get too concerned with the *specifics* of logical notation—after all, there are dozens of different versions. The main things to keep hold of are how the language facilitates concise representations and how its semantics leads to sound reasoning procedures.

[2] In contrast, facts in **fuzzy logic** have a **degree of truth** between 0 and 1. For example, the sentence "Vienna is a large city" might be true in our world only to degree 0.6.

[3] It is important not to confuse the degree of belief in probability theory with the degree of truth in fuzzy logic. Indeed, some fuzzy systems allow uncertainty (degree of belief) about degrees of truth.

8.2 SYNTAX AND SEMANTICS OF FIRST-ORDER LOGIC

We begin this section by specifying more precisely the way in which the possible worlds of first-order logic reflect the ontological commitment to objects and relations. Then we introduce the various elements of the language, explaining their semantics as we go along.

Models for first-order logic

Recall from Chapter 7 that the models of a logical language are the formal structures that constitute the possible worlds under consideration. Models for propositional logic are just sets of truth values for the proposition symbols. Models for first-order logic are more interesting.

DOMAIN
DOMAIN ELEMENTS

First, they have objects in them! The **domain** of a model is the set of objects it contains; these objects are sometimes called **domain elements**. Figure 8.2 shows a model with five objects: Richard the Lionheart, King of England from 1189 to 1199; his younger brother, the evil King John, who ruled from 1199 to 1215; the left legs of Richard and John; and a crown.

TUPLES

The objects in the model may be related in various ways. In the figure, Richard and John are brothers. Formally speaking, a relation is just the set of **tuples** of objects that are related. (A tuple is a collection of objects arranged in a fixed order and is written with angle brackets surrounding the objects.) Thus, the brotherhood relation in this model is the set

$$\{\langle \text{Richard the Lionheart, King John}\rangle, \langle \text{King John, Richard the Lionheart}\rangle\}. \quad (8.1)$$

(Here we have named the objects in English, but you may, if you wish, mentally substitute the pictures for the names.) The crown is on King John's head, so the "on head" relation contains

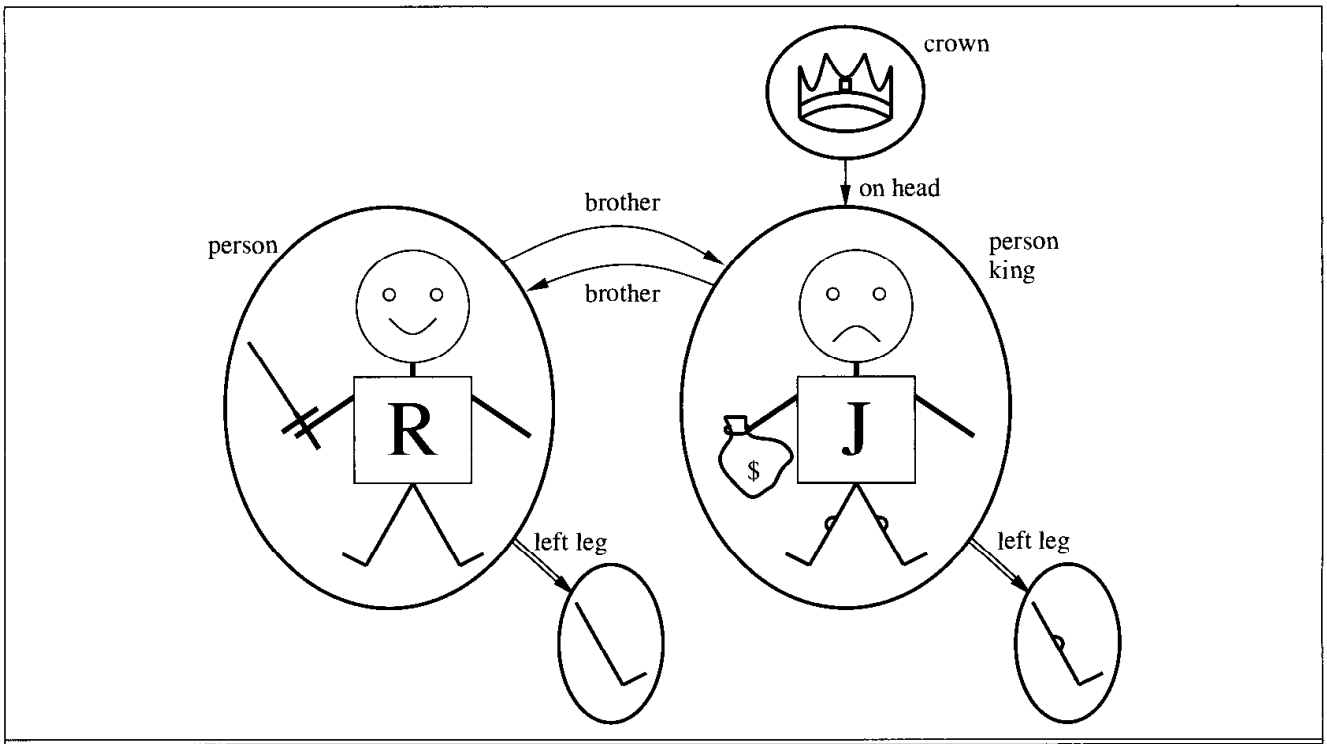


Figure 8.2 A model containing five objects, two binary relations, three unary relations (indicated by labels on the objects), and one unary function, left-leg.

just one tuple, ⟨the crown, King John⟩. The "brother" and "on head" relations are binary relations—that is, they relate pairs of objects. The model also contains unary relations, or properties: the "person" property is true of both Richard and John; the "king" property is true only of John (presumably because Richard is dead at this point); and the "crown" property is true only of the crown.

Certain kinds of relationships are best considered as functions, in that a given object must be related to exactly one object in this way. For example, each person has one left leg, so the model has a unary "left leg" function that includes the following mappings:

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

TOTAL FUNCTIONS

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

Symbols and interpretations

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

CONSTANT SYMBOLS
PREDICATE SYMBOLS
FUNCTION SYMBOLS

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

ARITY

INTERPRETATION

The semantics must relate sentences to models in order to determine truth. For this to happen, we need an **interpretation** that specifies exactly which objects, relations and functions are referred to by the constant, predicate, and function symbols. One possible interpretation for our example—which we will call the **intended interpretation**—is as follows:

INTENDED INTERPRETATION

- *Richard* refers to Richard the Lionheart and *John* refers to the evil King John.
- *Brother* refers to the brotherhood relation, that is, the set of tuples of objects given in Equation (8.1); *OnHead* refers to the "on head" relation that holds between the crown and King John; *Person*, *King*, and *Crown* refer to the sets of objects that are persons, kings, and crowns.
- *LeftLeg* refers to the "left leg" function, that is, the mapping given in Equation (8.2).

There are many other possible interpretations relating these symbols to this particular model. For example, one interpretation maps *Richard* to the crown and *John* to King John's left leg. There are five objects in the model, so there are 25 possible interpretations just for the

$$
\begin{aligned}
\mathit{Sentence} &\rightarrow \mathit{AtomicSentence} \\
&\mid (\ \mathit{Sentence}\ \mathit{Connective}\ \mathit{Sentence}\ ) \\
&\mid \mathit{Quantifier}\ \mathit{Variable}, \ldots\ \mathit{Sentence} \\
&\mid \neg\ \mathit{Sentence} \\
\\
\mathit{AtomicSentence} &\rightarrow \mathit{Predicate}(\mathit{Term}, \ldots) \mid \mathit{Term} = \mathit{Term} \\
\\
\mathit{Term} &\rightarrow \mathit{Function}(\mathit{Term}, \ldots) \\
&\mid \mathit{Constant} \\
&\mid \mathit{Variable} \\
\\
\mathit{Connective} &\rightarrow \Rightarrow \mid \wedge \mid \vee \mid \Leftrightarrow \\
\mathit{Quantifier} &\rightarrow \forall \mid \exists \\
\mathit{Constant} &\rightarrow A \mid X_1 \mid \mathit{John} \mid \cdots \\
\mathit{Variable} &\rightarrow a \mid x \mid s \mid \cdots \\
\mathit{Predicate} &\rightarrow \mathit{Before} \mid \mathit{HasColor} \mid \mathit{Raining} \mid \cdots \\
\mathit{Function} &\rightarrow \mathit{Mother} \mid \mathit{LeftLeg} \mid \cdots
\end{aligned}
$$

Figure 8.3 The syntax of first-order logic with equality, specified in Backus–Naur form. (See page 984 if you are not familiar with this notation.) The syntax is strict about parentheses; the comments about parentheses and operator precedence on page 205 apply equally to first-order logic.

constant symbols *Richard* and *John*. Notice that not all the objects need have a name—for example, the intended interpretation does not name the crown or the legs. It is also possible for an object to have several names; there is an interpretation under which both *Richard* and *John* refer to the crown. If you find this possibility confusing, remember that, in propositional logic, it is perfectly possible to have a model in which *Cloudy* and *Sunny* are both true; it is the job of the knowledge base to rule out models that are inconsistent with our knowledge.

The truth of any sentence is determined by a model and an interpretation for the sentence's symbols. Therefore, entailment, validity, and so on are defined in terms of *all possible models* and *all possible interpretations*. It is important to note that the number of domain elements in each model may be unbounded—for example, the domain elements may be integers or real numbers. Hence, the number of possible models is unbounded, as is the number of interpretations. Checking entailment by the enumeration of all possible models, which works for propositional logic, is not an option for first-order logic. Even if the number of objects is restricted, the number of combinations can be very large. With the symbols in our example, there are roughly 10^{25} combinations for a domain with five objects. (See Exercise 8.5.)

Terms

TERM

A **term** is a logical expression that refers to an object. Constant symbols are therefore terms, but it is not always convenient to have a distinct symbol to name every object. For example, in English we might use the expression "King John's left leg" rather than giving a name to his leg. This is what function symbols are for: instead of using a constant symbol, we use $LeftLeg(John)$. In the general case, a complex term is formed by a function symbol followed by a parenthesized list of terms as arguments to the function symbol. It is important to remember that a complex term is just a complicated kind of name. It is not a "subroutine call" that "returns a value." There is no *LeftLeg* subroutine that takes a person as input and returns a leg. We can reason about left legs (e.g., stating the general rule that everyone has one and then deducing that John must have one) without ever providing a definition of *LeftLeg*. This is something that cannot be done with subroutines in programming languages.[4]

The formal semantics of terms is straightforward. Consider a term $f(t_1, \ldots, t_n)$. The function symbol f refers to some function in the model (call it F); the argument terms refer to objects in the domain (call them $d_1, \ldots, d_n$); and the term as a whole refers to the object that is the value of the function F applied to $d_1, \ldots, d_n$. For example, suppose the *LeftLeg* function symbol refers to the function shown in Equation (8.2) and *John* refers to King John, then $LeftLeg(John)$ refers to King John's left leg. In this way, the interpretation fixes the referent of every term.

Atomic sentences

Now that we have both terms for referring to objects and predicate symbols for referring to relations, we can put them together to make **atomic sentences** that state facts. An atomic sentence is formed from a predicate symbol followed by a parenthesized list of terms:

$$Brother(Richard, John).$$

This states, under the intended interpretation given earlier, that Richard the Lionheart is the brother of King John.[5] Atomic sentences can have complex terms as arguments. Thus,

$$Married(Father(Richard), Mother(John))$$

states that Richard the Lionheart's father is married to King John's mother (again, under a suitable interpretation).

*An atomic sentence is **true** in a given model, under a given interpretation, if the relation referred to by the predicate symbol holds among the objects referred to by the arguments.*

---

[4] **λ-expressions** provide a useful notation in which new function symbols are constructed "on the fly." For example, the function that squares its argument can be written as $(\lambda x \; x \times x)$ and can be applied to arguments just like any other function symbol. A λ-expression can also be defined and used as a predicate symbol. (See Chapter 22.) The `lambda` operator in Lisp plays exactly the same role. Notice that the use of λ in this way does *not* increase the formal expressive power of first-order logic, because any sentence that includes a λ-expression can be rewritten by "plugging in" its arguments to yield an equivalent sentence.

[5] We will usually follow the argument ordering convention that $P(x, y)$ is interpreted as "x is a P of y."

Complex sentences

We can use **logical connectives** to construct more complex sentences, just as in propositional calculus. The semantics of sentences formed with logical connectives is identical to that in the propositional case. Here are four sentences that are true in the model of Figure 8.2 under our intended interpretation:

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

Quantifiers

QUANTIFIERS

Once we have a logic that allows objects, it is only natural to want to express properties of entire collections of objects, instead of enumerating the objects by name. **Quantifiers** let us do this. First-order logic contains two standard quantifiers, called *universal* and *existential*.

Universal quantification ($\forall$)

Recall the difficulty we had in Chapter 7 with the expression of general rules in propositional logic. Rules such as "Squares neighboring the wumpus are smelly" and "All kings are persons" are the bread and butter of first-order logic. We will deal with the first of these in Section 8.3. The second rule, "All kings are persons," is written in first-order logic as

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

$\forall$ is usually pronounced "For all ...". (Remember that the upside-down A stands for "all.") Thus, the sentence says, "For all x, if x is a king, then x is a person." The symbol x is called a **variable**. By convention, variables are lowercase letters. A variable is a term all by itself, and as such can also serve as the argument of a function—for example, $LeftLeg(x)$. A term with no variables is called a **ground term**.

VARIABLE

GROUND TERM

Intuitively, the sentence $\forall x \; P$, where P is any logical expression, says that P is true for every object x. More precisely, $\forall x \; P$ is true in a given model under a given interpretation if P is true in all possible **extended interpretations** constructed from the given interpretation, where each extended interpretation specifies a domain element to which x refers.

EXTENDED INTERPRETATION

This sounds complicated, but it is really just a careful way of stating the intuitive meaning of universal quantification. Consider the model shown in Figure 8.2 and the intended interpretation that goes with it. We can extend the interpretation in five ways:

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

The universally quantified sentence $\forall x \;\; King(x) \Rightarrow Person(x)$ is true under the original interpretation if the sentence $King(x) \Rightarrow Person(x)$ is true in each of the five extended inter-

pretations. That is, the universally quantified sentence is equivalent to asserting the following five sentences:

Richard the Lionheart is a king $\Rightarrow$ Richard the Lionheart is a person.
King John is a king $\Rightarrow$ King John is a person.
Richard's left leg is a king $\Rightarrow$ Richard's left leg is a person.
John's left leg is a king $\Rightarrow$ John's left leg is a person.
The crown is a king $\Rightarrow$ the crown is a person.

Let us look carefully at this set of assertions. Since, in our model, King John is the only king, the second sentence asserts that he is a person, as we would hope. But what about the other four sentences, which appear to make claims about legs and crowns? Is that part of the meaning of "All kings are persons"? In fact, the other four assertions are true in the model, but make no claim whatsoever about the personhood qualifications of legs, crowns, or indeed Richard. This is because none of these objects is a king. Looking at the truth table for $\Rightarrow$ (Figure 7.8), we see that the implication is true whenever its premise is false—*regardless* of the truth of the conclusion. Thus, by asserting the universally quantified sentence, which is equivalent to asserting a whole list of individual implications, we end up asserting the conclusion of the rule just for those objects for whom the premise is true and saying nothing at all about those individuals for whom the premise is false. Thus, the truth-table entries for $\Rightarrow$ turn out to be perfect for writing general rules with universal quantifiers.

A common mistake, made frequently even by diligent readers who have read this paragraph several times, is to use conjunction instead of implication. The sentence

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

would be equivalent to asserting

Richard the Lionheart is a king $\wedge$ Richard the Lionheart is a person,
King John is a king $\wedge$ King John is a person,
Richard's left leg is a king $\wedge$ Richard's left leg is a person,

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

Existential quantification ($\exists$)

Universal quantification makes statements about every object. Similarly, we can make a statement about *some* object in the universe without naming it, by using an existential quantifier. To say, for example, that King John has a crown on his head, we write

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

$\exists x$ is pronounced "There exists an x such that . . ." or "For some x . . .".

Intuitively, the sentence $\exists x \;\; P$ says that P is true for at least one object x. More precisely, $\exists x \;\; P$ is true in a given model under a given interpretation if P is true in *at least one* extended interpretation that assigns x to a domain element. For our example, this means

that at least one of the following must be true:

Richard the Lionheart is a crown $\wedge$ Richard the Lionheart is on John's head;
King John is a crown $\wedge$ King John is on John's head;
Richard's left leg is a crown $\wedge$ Richard's left leg is on John's head;
John's left leg is a crown $\wedge$ John's left leg is on John's head;
The crown is a crown $\wedge$ the crown is on John's head.

The fifth assertion is true in the model, so the original existentially quantified sentence is true in the model. Notice that, by our definition, the sentence would also be true in a model in which King John was wearing two crowns. This is entirely consistent with the original sentence "King John has a crown on his head." [6]

Just as $\Rightarrow$ appears to be the natural connective to use with $\forall$, $\wedge$ is the natural connective to use with $\exists$. Using $\wedge$ as the main connective with $\forall$ led to an overly strong statement in the example in the previous section; using $\Rightarrow$ with $\exists$ usually leads to a very weak statement, indeed. Consider the following sentence:

$$\exists x \;\; Crown(x) \;\Rightarrow\; OnHead(x, John) \,.$$

On the surface, this might look like a reasonable rendition of our sentence. Applying the semantics, we see that the sentence says that at least one of the following assertions is true:

Richard the Lionheart is a crown $\Rightarrow$ Richard the Lionheart is on John's head;
King John is a crown $\Rightarrow$ King John is on John's head;
Richard's left leg is a crown $\Rightarrow$ Richard's left leg is on John's head;

and so on. Now an implication is true if both premise and conclusion are true, *or if its premise is false*. So if Richard the Lionheart is not a crown, then the first assertion is true and the existential is satisfied. So, an existentially quantified implication sentence is true in any model containing an object for which the premise of the implication is false; hence such sentences really do not say much at all.

Nested quantifiers

We will often want to express more complex sentences using multiple quantifiers. The simplest case is where the quantifiers are of the same type. For example, "Brothers are siblings" can be written as

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

Consecutive quantifiers of the same type can be written as one quantifier with several variables. For example, to say that siblinghood is a symmetric relationship, we can write

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

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

$$\forall x \;\exists y \;\; Loves(x, y) \,.$$

[6] There is a variant of the existential quantifier, usually written $\exists^1$ or $\exists!$, that means "There exists exactly one." The same meaning can be expressed using equality statements, as we show in Section 8.2.

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

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

The order of quantification is therefore very important. It becomes clearer if we insert parentheses. $\forall x \ (\exists y \ Loves(x, y))$ says that *everyone* has a particular property, namely, the property that somebody loves them. On the other hand, $\exists x \ (\forall y \ Loves(x, y))$ says that *someone* in the world has a particular property, namely the property of being loved by everybody.

Some confusion can arise when two quantifiers are used with the same variable name. Consider the sentence

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

Here the x in $Brother(Richard, x)$ is *existentially* quantified. The rule is that the variable belongs to the innermost quantifier that mentions it; then it will not be subject to any other quantification.[7] Another way to think of it is this: $\exists x \ Brother(Richard, x)$ is a sentence about Richard (that he has a brother), not about x; so putting a $\forall x$ outside it has no effect. It could equally well have been written $\exists z \ Brother(Richard, z)$. Because this can be a source of confusion, we will always use different variables.

Connections between $\forall$ and $\exists$

The two quantifiers are actually intimately connected with each other, through negation. Asserting that everyone dislikes parsnips is the same as asserting there does not exist someone who likes them, and vice versa:

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

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

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

Because $\forall$ is really a conjunction over the universe of objects and $\exists$ is a disjunction, it should not be surprising that they obey De Morgan's rules. The De Morgan rules for quantified and unquantified sentences are as follows:

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

Thus, we do not really need both $\forall$ and $\exists$, just as we do not really need both $\wedge$ and $\vee$. Still, readability is more important than parsimony, so we will keep both of the quantifiers.

[7] It is the potential for interference between quantifiers using the same variable name that motivates the slightly baroque mechanism of extended interpretations in the semantics of quantified sentences. The more intuitively obvious approach of substituting objects for every occurrence of x fails in our example because the x in $Brother(Richard, x)$ would be "captured" by the substitution. Extended interpretations handle this correctly because the inner quantifier's assignment for x overrides the outer quantifier's.

Equality

EQUALITY SYMBOL

First-order logic includes one more way to make atomic sentences, other than using a predicate and terms as described earlier. We can use the **equality symbol** to make statements to the effect that two terms refer to the same object. For example,

$$Father(John) = Henry$$

says that the object referred to by $Father(John)$ and the object referred to by $Henry$ are the same. Because an interpretation fixes the referent of any term, determining the truth of an equality sentence is simply a matter of seeing that the referents of the two terms are the same object.

The equality symbol can be used to state facts about a given function, as we just did for the *Father* symbol. It can also be used with negation to insist that two terms are not the same object. To say that Richard has at least two brothers, we would write

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

The sentence

$$\exists x, y \;\; Brother(x, Richard) \wedge Brother(y, Richard) \; ,$$

does not have the intended meaning. In particular, it is true in the model of Figure 8.2, where Richard has only one brother. To see this, consider the extended interpretation in which both x and y are assigned to King John. The addition of $\neg(x = y)$ rules out such models. The notation $x \neq y$ is sometimes used as an abbreviation for $\neg(x = y)$.

8.3 Using First-Order Logic

Now that we have defined an expressive logical language, it is time to learn how to use it. The best way to do this is through examples. We have seen some simple sentences illustrating the various aspects of logical syntax; in this section, we will provide more systematic representations of some simple **domains**. In knowledge representation, a domain is just some part of the world about which we wish to express some knowledge.

DOMAINS

We will begin with a brief description of the TELL/ASK interface for first-order knowledge bases. Then we will look at the domains of family relationships, numbers, sets, and lists, and at the wumpus world. The next section contains a more substantial example (electronic circuits) and Chapter 10 covers everything in the universe.

Assertions and queries in first-order logic

ASSERTIONS

Sentences are added to a knowledge base using TELL, exactly as in propositional logic. Such sentences are called **assertions**. For example, we can assert that John is a king and that kings are persons:

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

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

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

QUERIES
GOALS

returns *true*. Questions asked using ASK are called **queries** or **goals** (not to be confused with goals as used to describe an agent's desired states). Generally speaking, any query that is logically entailed by the knowledge base should be answered affirmatively. For example, given the two assertions in the preceding paragraph, the query

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

should also return *true*. We can also ask quantified queries, such as

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

The answer to this query could be *true*, but this is neither helpful nor amusing. (It is rather like answering "Can you tell me the time?" with "Yes.") A query with existential variables is asking "Is there an x such that . . .," and we solve it by providing such an x. The standard

SUBSTITUTION
BINDING LIST

form for an answer of this sort is a **substitution** or **binding list**, which is a set of variable/term pairs. In this particular case, given just the two assertions, the answer would be $\{x/John\}$. If there is more than one possible answer, a list of substitutions can be returned.

The kinship domain

The first example we consider is the domain of family relationships, or kinship. This domain includes facts such as "Elizabeth is the mother of Charles" and "Charles is the father of William" and rules such as "One's grandmother is the mother of one's parent."

Clearly, the objects in our domain are people. We will have two unary predicates, *Male* and *Female*. Kinship relations—parenthood, brotherhood, marriage, and so on—will be represented by binary predicates: *Parent*, *Sibling*, *Brother*, *Sister*, *Child*, *Daughter*, *Son*, *Spouse*, *Wife*, *Husband*, *Grandparent*, *Grandchild*, *Cousin*, *Aunt*, and *Uncle*. We will use functions for *Mother* and *Father*, because every person has exactly one of each of these (at least according to nature's design).

We can go through each function and predicate, writing down what we know in terms of the other symbols. For example, one's mother is one's female parent:

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

One's husband is one's male spouse:

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

Male and female are disjoint categories:

$$\forall x\ \ Male(x)\ \Leftrightarrow\ \neg Female(x)\ .$$

Parent and child are inverse relations:

$$\forall p, c\ \ Parent(p, c)\ \Leftrightarrow\ Child(c, p)\ .$$

A grandparent is a parent of one's parent:

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

A sibling is another child of one's parents:

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

We could go on for several more pages like this, and Exercise 8.11 asks you to do just that.

AXIOM

Each of these sentences can be viewed as an **axiom** of the kinship domain. Axioms are commonly associated with purely mathematical domains—we will see some axioms for numbers shortly—but they are needed in all domains. They provide the basic factual information from which useful conclusions can be derived. Our kinship axioms are also **definitions**; they have the form $\forall x, y \;\; P(x, y) \Leftrightarrow \ldots$. The axioms define the *Mother* function and the *Husband*, *Male*, *Parent*, *Grandparent*, and *Sibling* predicates in terms of other predicates. Our definitions "bottom out" at a basic set of predicates (*Child*, *Spouse*, and *Female*) in terms of which the others are ultimately defined. This is a very natural way in which to build up the representation of a domain, and it is analogous to the way in which software packages are built up by successive definitions of subroutines from primitive library functions. Notice that there is not necessarily a unique set of primitive predicates; we could equally well have used *Parent*, *Spouse*, and *Male*. In some domains, as we will see, there is no clearly identifiable basic set.

DEFINITION

THEOREM

Not all logical sentences about a domain are axioms. Some are **theorems**—that is, they are entailed by the axioms. For example, consider the assertion that siblinghood is symmetric:

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

Is this an axiom or a theorem? In fact, it is a theorem that follows logically from the axiom that defines siblinghood. If we ASK the knowledge base this sentence, it should return *true*.

From a purely logical point of view, a knowledge base need contain only axioms and no theorems, because the theorems do not increase the set of conclusions that follow from the knowledge base. From a practical point of view, theorems are essential to reduce the computational cost of deriving new sentences. Without them, a reasoning system has to start from first principles every time, rather like a physicist having to rederive the rules of calculus for every new problem.

Not all axioms are definitions. Some provide more general information about certain predicates without constituting a definition. Indeed, some predicates have no complete definition because we do not know enough to characterize them fully. For example, there is no obvious way to complete the sentence:

$$\forall x \;\; Person(x) \Leftrightarrow \ldots$$

Fortunately, first-order logic allows us to make use of the *Person* predicate without completely defining it. Instead, we can write partial specifications of properties that every person has and properties that make something a person:

$$\forall x \;\; Person(x) \Rightarrow \ldots$$
$$\forall x \;\; \ldots \Rightarrow Person(x) \; .$$

Axioms can also be "just plain facts," such as $Male(Jim)$ and $Spouse(Jim, Laura)$. Such facts form the descriptions of specific problem instances, enabling specific questions to be answered. The answers to these questions will then be theorems that follow from the

axioms. Often, one finds that the expected answers are not forthcoming—for example, from $Male(George)$ and $Spouse(George, Laura)$, one expects to be able to infer $Female(Laura)$; but this does not follow from the axioms given earlier. This is a sign that an axiom is missing. Exercise 8.8 asks you to supply it.

Numbers, sets, and lists

NATURAL NUMBERS

Numbers are perhaps the most vivid example of how a large theory can be built up from a tiny kernel of axioms. We will describe here the theory of **natural numbers** or nonnegative integers. We need a predicate $NatNum$ that will be true of natural numbers; we need one constant symbol, 0; and we need one function symbol, S (successor). The **Peano axioms** define natural numbers and addition.[8] Natural numbers are defined recursively:

PEANO AXIOMS

$$NatNum(0) .$$
$$\forall n \;\; NatNum(n) \Rightarrow NatNum(S(n)) .$$

That is, 0 is a natural number, and for every object n, if n is a natural number then $S(n)$ is a natural number. So the natural numbers are 0, $S(0)$, $S(S(0))$, and so on. We also need axioms to constrain the successor function:

$$\forall n \;\; 0 \neq S(n) .$$
$$\forall m, n \;\; m \neq n \Rightarrow S(m) \neq S(n) .$$

Now we can define addition in terms of the successor function:

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

The first of these axioms says that adding 0 to any natural number m gives m itself. Notice the use of the binary function symbol "+" in the term $+(m, 0)$; in ordinary mathematics, the term would be written $m + 0$ using **infix** notation. (The notation we have used for first-order logic is called **prefix**.) To make our sentences about numbers easier to read, we will allow the use of infix notation. We can also write $S(n)$ as $n + 1$, so that the second axiom becomes

INFIX

PREFIX

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

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

SYNTACTIC SUGAR

The use of infix notation is an example of **syntactic sugar**, that is, an extension to or abbreviation of the standard syntax that does not change the semantics. Any sentence that uses sugar can be "de-sugared" to produce an equivalent sentence in ordinary first-order logic.

Once we have addition, it is straightforward to define multiplication as repeated addition, exponentiation as repeated multiplication, integer division and remainders, prime numbers, and so on. Thus, the whole of number theory (including cryptography) can be built up from one constant, one function, one predicate and four axioms.

SETS

The domain of **sets** is also fundamental to mathematics as well as to commonsense reasoning. (In fact, it is possible to build number theory on top of set theory.) We want to be able to represent individual sets, including the empty set. We need a way to build up sets by

[8] The Peano axioms also include the principle of induction, which is a sentence of second-order logic rather than of first-order logic. The importance of this distinction is explained in Chapter 9.

adding an element to a set or taking the union or intersection of two sets. We will want to know whether an element is a member of a set and to be able to distinguish sets from objects that are not sets.

We will use the normal vocabulary of set theory as syntactic sugar. The empty set is a constant written as $\{\}$. There is one unary predicate, Set, which is true of sets. The binary predicates are $x \in s$ (x is a member of set s) and $s_1 \subseteq s_2$ (set s_1 is a subset, not necessarily proper, of set s_2). The binary functions are $s_1 \cap s_2$ (the intersection of two sets), $s_1 \cup s_2$ (the union of two sets), and $\{x|s\}$ (the set resulting from adjoining element x to set s). One possible set of axioms is as follows:

1. The only sets are the empty set and those made by adjoining something to a set:
$$\forall s \;\; Set(s) \Leftrightarrow (s = \{\}) \vee (\exists x, s_2 \;\; Set(s_2) \wedge s = \{x|s_2\}) .$$
2. The empty set has no elements adjoined into it, in other words, there is no way to decompose $EmptySet$ into a smaller set and an element:
$$\neg \exists x, s \;\; \{x|s\} = \{\} .$$
3. Adjoining an element already in the set has no effect:
$$\forall x, s \;\; x \in s \Leftrightarrow s = \{x|s\} .$$
4. The only members of a set are the elements that were adjoined into it. We express this recursively, saying that x is a member of s if and only if s is equal to some set s_2 adjoined with some element y, where either y is the same as x or x is a member of s_2:
$$\forall x, s \;\; x \in s \Leftrightarrow [\exists y, s_2 \;\; (s = \{y|s_2\} \wedge (x = y \vee x \in s_2))] .$$
5. A set is a subset of another set if and only if all of the first set's members are members of the second set:
$$\forall s_1, s_2 \;\; s_1 \subseteq s_2 \Leftrightarrow (\forall x \;\; x \in s_1 \Rightarrow x \in s_2) .$$
6. Two sets are equal if and only if each is a subset of the other:
$$\forall s_1, s_2 \;\; (s_1 = s_2) \Leftrightarrow (s_1 \subseteq s_2 \wedge s_2 \subseteq s_1) .$$
7. An object is in the intersection of two sets if and only if it is a member of both sets:
$$\forall x, s_1, s_2 \;\; x \in (s_1 \cap s_2) \Leftrightarrow (x \in s_1 \wedge x \in s_2) .$$
8. An object is in the union of two sets if and only if it is a member of either set:
$$\forall x, s_1, s_2 \;\; x \in (s_1 \cup s_2) \Leftrightarrow (x \in s_1 \vee x \in s_2) .$$

LISTS

Lists are similar to sets. The differences are that lists are ordered and the same element can appear more than once in a list. We can use the vocabulary of Lisp for lists: Nil is the constant list with no elements; $Cons$, $Append$, $First$, and $Rest$ are functions; and $Find$ is the predicate that does for lists what $Member$ does for sets. $List?$ is a predicate that is true only of lists. As with sets, it is common to use syntactic sugar in logical sentences involving lists. The empty list is $[]$. The term $Cons(x, y)$, where y is a nonempty list, is written $[x|y]$. The term $Cons(x, Nil)$, (i.e., the list containing the element x), is written as $[x]$. A list of several elements, such as $[A, B, C]$, corresponds to the nested term $Cons(A, Cons(B, Cons(C, Nil)))$. Exercise 8.14 asks you to write out the axioms for lists.

The wumpus world

Some propositional logic axioms for the wumpus world were given in Chapter 7. The first-order axioms in this section are much more concise, capturing in a very natural way exactly what we want to say.

Recall that the wumpus agent receives a percept vector with five elements. The corresponding first-order sentence stored in the knowledge base must include both the percept and the time at which it occurred; otherwise the agent will get confused about when it saw what. We will use integers for time steps. A typical percept sentence would be

$$Percept([Stench, Breeze, Glitter, None, None], 5) .$$

Here, *Percept* is a binary predicate and *Stench* and so on are constants placed in a list. The actions in the wumpus world can be represented by logical terms:

$$Turn(Right),\ Turn(Left),\ Forward,\ Shoot,\ Grab,\ Release,\ Climb .$$

To determine which is best, the agent program constructs a query such as

$$\exists a\ \ BestAction(a, 5) .$$

ASK should solve this query and return a binding list such as $\{a/Grab\}$. The agent program can then return *Grab* as the action to take, but first it must TELL its own knowledge base that it is performing a *Grab*.

The raw percept data implies certain facts about the current state. For example:

$$\forall t, s, g, m, c\ \ Percept([s, Breeze, g, m, c], t) \Rightarrow Breeze(t) ,$$
$$\forall t, s, b, m, c\ \ Percept([s, b, Glitter, m, c], t) \Rightarrow Glitter(t) ,$$

and so on. These rules exhibit a trivial form of the reasoning process called **perception**, which we study in depth in Chapter 24. Notice the quantification over time t. In propositional logic, we would need copies of each sentence for each time step.

Simple "reflex" behavior can also be implemented by quantified implication sentences. For example, we have

$$\forall t\ \ Glitter(t) \Rightarrow BestAction(Grab, t) .$$

Given the percept and rules from the preceding paragraphs, this would yield the desired conclusion $BestAction(Grab, 5)$—that is, *Grab* is the right thing to do. Notice the correspondence between this rule and the direct percept–action connection in the circuit-based agent in Figure 7.20; the circuit connection *implicitly* quantifies over time.

SYNCHRONIC

So far in this section, the sentences dealing with time have been **synchronic** ("same time") sentences, that is, they relate properties of a world state to other properties of the same world state. Sentences that allow reasoning "across time" are called **diachronic**; for example, the agent needs to know how to combine information about its previous location with information about the action just taken in order to determine its current location. We will defer discussion of diachronic sentences until Chapter 10; for now, just assume that the required inferences have been made for location and other time-dependent predicates.

DIACHRONIC

We have represented the percepts and actions; now it is time to represent the environment itself. Let us begin with objects. Obvious candidates are squares, pits, and the wumpus.

We could name each square—$Square_{1,2}$ and so on—but then the fact that $Square_{1,2}$ and $Square_{1,3}$ are adjacent would have to be an "extra" fact, and we would need one such fact for each pair of squares. It is better to use a complex term in which the row and column appear as integers; for example, we can simply use the list term $[1, 2]$. Adjacency of any two squares can be defined as

$$\forall x, y, a, b \;\; Adjacent([x, y], [a, b]) \Leftrightarrow [a, b] \in \{[x + 1, y], [x - 1, y], [x, y + 1], [x, y - 1]\} .$$

We could also name each pit, but this would be inappropriate for a different reason: there is no reason to distinguish among the pits.[9] It is much simpler to use a unary predicate Pit that is true of squares containing pits. Finally, since there is exactly one wumpus, a constant $Wumpus$ is just as good as a unary predicate (and perhaps more dignified from the wumpus's viewpoint). The wumpus lives in exactly one square, so it is a good idea to use a function such as $Home(Wumpus)$ to name that square. This completely avoids the cumbersome set of sentences required in propositional logic to say that exactly one square contains a wumpus. (It would be even worse for propositional logic with two wumpuses.)

The agent's location changes over time, so we will write $At(Agent, s, t)$ to mean that the agent is at square s at time t. Given its current location, the agent can infer properties of the square from properties of its current percept. For example, if the agent is at a square and perceives a breeze, then that square is breezy:

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

It is useful to know that a *square* is breezy because we know that the pits cannot move about. Notice that $Breezy$ has no time argument.

Having discovered which places are breezy (or smelly) and, very importantly, *not* breezy (or *not* smelly), the agent can deduce where the pits are (and where the wumpus is). There are two kinds of synchronic rules that could allow such deductions:

DIAGNOSTIC RULES

◇ **Diagnostic rules**:

Diagnostic rules lead from observed effects to hidden causes. For finding pits, the obvious diagnostic rules say that if a square is breezy, some adjacent square must contain a pit, or

$$\forall s \;\; Breezy(s) \Rightarrow \exists r \;\; Adjacent(r, s) \wedge Pit(r) ,$$

and that if a square is not breezy, no adjacent square contains a pit: [10]

$$\forall s \;\; \neg Breezy(s) \Rightarrow \neg\exists r \;\; Adjacent(r, s) \wedge Pit(r) .$$

Combining these two, we obtain the biconditional sentence

$$\forall s \;\; Breezy(s) \Leftrightarrow \exists r \;\; Adjacent(r, s) \wedge Pit(r) . \tag{8.3}$$

[9] Similarly, most of us do not name each bird that flies overhead as it migrates to warmer regions in winter. An ornithologist wishing to study migration patterns, survival rates, and so on *does* name each bird, by means of a ring on its leg, because individual birds must be tracked.

[10] There is a natural human tendency to forget to write down negative information such as this. In conversation, this tendency is entirely normal—it would be strange to say "There are two cups on the table *and there are not three or more*," even though "There are two cups on the table" is, strictly speaking, still true when there are three. We will return to this topic in Chapter 10.

CAUSAL RULES

◇ **Causal rules**:
Causal rules reflect the assumed direction of causality in the world: some hidden property of the world causes certain percepts to be generated. For example, a pit causes all adjacent squares to be breezy:

$$\forall r \;\; Pit(r) \Rightarrow [\forall s \;\; Adjacent(r, s) \Rightarrow Breezy(s)]$$

and if all squares adjacent to a given square are pitless, the square will not be breezy:

$$\forall s \;\; [\forall r \;\; Adjacent(r, s) \Rightarrow \neg Pit(r)] \Rightarrow \neg Breezy(s) \; .$$

With some work, it is possible to show that these two sentences together are logically equivalent to the biconditional sentence in Equation (8.3). The biconditional itself can also be thought of as causal, because it states how the truth value of *Breezy* is generated from the world state.

MODEL-BASED REASONING

Systems that reason with causal rules are called **model-based reasoning** systems, because the causal rules form a model of how the environment operates. The distinction between model-based and diagnostic reasoning is important in many areas of AI. Medical diagnosis in particular has been an active area of research, in which approaches based on direct associations between symptoms and diseases (a diagnostic approach) have gradually been replaced by approaches using an explicit model of the disease process and how it manifests itself in symptoms. The issues come up again in Chapter 13.

Whichever kind of representation the agent uses, *if the axioms correctly and completely describe the way the world works and the way that percepts are produced, then any complete logical inference procedure will infer the strongest possible description of the world state, given the available percepts.* Thus, the agent designer can concentrate on getting the knowledge right, without worrying too much about the processes of deduction. Furthermore, we have seen that first-order logic can represent the wumpus world no less concisely than the original English-language description given in Chapter 7.

8.4 KNOWLEDGE ENGINEERING IN FIRST-ORDER LOGIC

The preceding section illustrated the use of first-order logic to represent knowledge in three simple domains. This section describes the general process of knowledge base construction—

KNOWLEDGE ENGINEERING

a process called **knowledge engineering**. A knowledge engineer is someone who investigates a particular domain, learns what concepts are important in that domain, and creates a formal representation of the objects and relations in the domain. We will illustrate the knowledge engineering process in an electronic circuit domain that should already be fairly familiar, so that we can concentrate on the representational issues involved. The approach we will take is suitable for developing *special-purpose* knowledge bases whose domain is carefully circumscribed and whose range of queries is known in advance. *General-purpose* knowledge bases, which are intended to support queries across the full range of human knowledge, are discussed in Chapter 10.

The knowledge engineering process

Knowledge engineering projects vary widely in content, scope, and difficulty, but all such projects include the following steps:

1. *Identify the task.* The knowledge engineer must delineate the range of questions that the knowledge base will support and the kinds of facts that will be available for each specific problem instance. For example, does the wumpus knowledge base need to be able to choose actions or is it required to answer questions only about the contents of the environment? Will the sensor facts include the current location? The task will determine what knowledge must be represented in order to connect problem instances to answers. This step is analogous to the PEAS process for designing agents in Chapter 2.
2. *Assemble the relevant knowledge.* The knowledge engineer might already be an expert in the domain, or might need to work with real experts to extract what they know—a process called **knowledge acquisition**. At this stage, the knowledge is not represented formally. The idea is to understand the scope of the knowledge base, as determined by the task, and to understand how the domain actually works.

   KNOWLEDGE ACQUISITION

   For the wumpus world, which is defined by an artificial set of rules, the relevant knowledge is easy to identify. (Notice, however, that the definition of adjacency was not supplied explicitly in the wumpus-world rules.) For real domains, the issue of relevance can be quite difficult—for example, a system for simulating VLSI designs might or might not need to take into account stray capacitances and skin effects.
3. *Decide on a vocabulary of predicates, functions, and constants.* That is, translate the important domain-level concepts into logic-level names. This involves many questions of knowledge engineering *style*. Like programming style, this can have a significant impact on the eventual success of the project. For example, should pits be represented by objects or by a unary predicate on squares? Should the agent's orientation be a function or a predicate? Should the wumpus's location depend on time? Once the choices have been made, the result is a vocabulary that is known as the **ontology** of the domain. The word *ontology* means a particular theory of the nature of being or existence. The ontology determines what kinds of things exist, but does not determine their specific properties and interrelationships.

   ONTOLOGY
4. *Encode general knowledge about the domain.* The knowledge engineer writes down the axioms for all the vocabulary terms. This pins down (to the extent possible) the meaning of the terms, enabling the expert to check the content. Often, this step reveals misconceptions or gaps in the vocabulary that must be fixed by returning to step 3 and iterating through the process.
5. *Encode a description of the specific problem instance.* If the ontology is well thought out, this step will be easy. It will involve writing simple atomic sentences about instances of concepts that are already part of the ontology. For a logical agent, problem instances are supplied by the sensors, whereas a "disembodied" knowledge base is supplied with additional sentences in the same way that traditional programs are supplied with input data.

6. *Pose queries to the inference procedure and get answers.* This is where the reward is: we can let the inference procedure operate on the axioms and problem-specific facts to derive the facts we are interested in knowing.
7. *Debug the knowledge base.* Alas, the answers to queries will seldom be correct on the first try. More precisely, the answers will be correct *for the knowledge base as written*, assuming that the inference procedure is sound, but they will not be the ones that the user is expecting. For example, if an axiom is missing, some queries will not be answerable from the knowledge base. A considerable debugging process could ensue. Missing axioms or axioms that are too weak can be identified easily by noticing places where the chain of reasoning stops unexpectedly. For example, if the knowledge base includes one of the diagnostic axioms for pits,

   $$\forall s \;\; Breezy(s) \Rightarrow \exists r \;\; Adjacent(r, s) \wedge Pit(r) \,,$$

   but not the other, then the agent will never be able to prove the *absence* of pits. Incorrect axioms can be identified because they are false statements about the world. For example, the sentence

   $$\forall x \;\; NumOfLegs(x, 4) \Rightarrow Mammal(x)$$

   is false for reptiles, amphibians, and, more important, tables. *The falsehood of this sentence can be determined independently of the rest of the knowledge base.* In contrast, a typical error in a program looks like this:

   ```
   offset = position + 1.
   ```

   It is impossible to tell whether this statement is correct without looking at the rest of the program to see whether, for example, `offset` is used to refer to the current position, or to one beyond the current position, or whether the value of `position` is changed by another statement and so `offset` should also be changed again.

To understand this seven-step process better, we now apply it to an extended example—the domain of electronic circuits.

The electronic circuits domain

We will develop an ontology and knowledge base that allow us to reason about digital circuits of the kind shown in Figure 8.4. We follow the seven-step process for knowledge engineering.

Identify the task

There are many reasoning tasks associated with digital circuits. At the highest level, one analyzes the circuit's functionality. For example, does the circuit in Figure 8.4 actually add properly? If all the inputs are high, what is the output of gate A2? Questions about the circuit's structure are also interesting. For example, what are all the gates connected to the first input terminal? Does the circuit contain feedback loops? These will be our tasks in this section. There are more detailed levels of analysis, including those related to timing delays, circuit area, power consumption, production cost, and so on. Each of these levels would require additional knowledge.

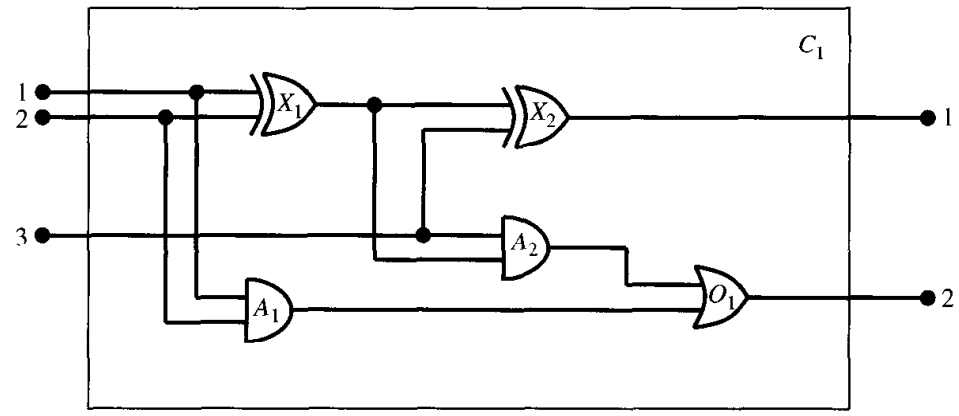


Figure 8.4 A digital circuit C1, purporting to be a one-bit full adder. The first two inputs are the two bits to be added and the third input is a carry bit. The first output is the sum, and the second output is a carry bit for the next adder. The circuit contains two XOR gates, two AND gates and one OR gate.

Assemble the relevant knowledge

What do we know about digital circuits? For our purposes, they are composed of wires and gates. Signals flow along wires to the input terminals of gates, and each gate produces a signal on the output terminal that flows along another wire. To determine what these signals will be, we need to know how the gates transform their input signals. There are four types of gates: AND, OR, and XOR gates have two input terminals, and NOT gates have one. All gates have one output terminal. Circuits, like gates, have input and output terminals.

To reason about functionality and connectivity, we do not need to talk about the wires themselves, the paths the wires take, or the junctions where two wires come together. All that matters is the connections between terminals—we can say that one output terminal is connected to another input terminal without having to mention the wire that actually connects them. There are many other factors of the domain that are irrelevant to our analysis, such as the size, shape, color, or cost of the various components.

If our purpose were something other than verifying designs at the gate level, the ontology would be different. For example, if we were interested in debugging faulty circuits, then it would probably be a good idea to include the wires in the ontology, because a faulty wire can corrupt the signal flowing along it. For resolving timing faults, we would need to include gate delays. If we were interested in designing a product that would be profitable, then the cost of the circuit and its speed relative to other products on the market would be important.

Decide on a vocabulary

We now know that we want to talk about circuits, terminals, signals, and gates. The next step is to choose functions, predicates, and constants to represent them. We will start from individual gates and move up to circuits.

First, we need to be able to distinguish a gate from other gates. This is handled by naming gates with constants: X_1, X_2, and so on. Although each gate is connected into the circuit in its own individual way, its *behavior*—the way it transforms input signals into output