The Intelligent Agent Book

Artificial Intelligence
A Modern Approach

Stuart Russell • Peter Norvig

Prentice Hall Series in Artificial Intelligence

Artificial Intelligence
A Modern Approach

Stuart J. Russell and Peter Norvig

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

Prentice Hall, Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging-in-Publication Data

Russell, Stuart J. (Stuart Jonathan)
Artificial intelligence : a modern approach / Stuart Russell, Peter Norvig.
p. cm.
Includes bibliographical references and index.
ISBN 0-13-103805-2
1. Artificial intelligence I. Norvig, Peter. II. Title.
Q335.R86 1995
006.3--dc20 94-36444
CIP

Publisher: Alan Apt
Production Editor: Mona Pompili
Developmental Editor: Sondra Chavez
Cover Designers: Stuart Russell and Peter Norvig
Production Coordinator: Lori Bulwin
Editorial Assistant: Shirley McGuire

A Simon & Schuster Company
Englewood Cliffs, New Jersey 07632

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-103805-2

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada, Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

Preface

There are many textbooks that offer an introduction to artificial intelligence (AI). This text has five principal features that together distinguish it from other texts.

1. *Unified presentation of the field.*
   Some texts are organized from a historical perspective, describing each of the major problems and solutions that have been uncovered in 40 years of AI research. Although there is value to this perspective, the result is to give the impression of a dozen or so barely related subfields, each with its own techniques and problems. We have chosen to present AI as a unified field, working on a common problem in various guises. This has entailed some reinterpretation of past research, showing how it fits within a common framework and how it relates to other work that was historically separate. It has also led us to include material not normally covered in AI texts.
2. *Intelligent agent design.*
   The unifying theme of the book is the concept of an *intelligent agent.* In this view, the problem of AI is to describe and build agents that receive percepts from the environment and perform actions. Each such agent is implemented by a function that maps percepts to actions, and we cover different ways to represent these functions, such as production systems, reactive agents, logical planners, neural networks, and decision-theoretic systems. We explain the role of learning as extending the reach of the designer into unknown environments, and show how it 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 goal achievement. We stress the importance of the task environment characteristics in determining the appropriate agent design.
3. *Comprehensive and up-to-date coverage.*
   We cover areas that are sometimes underemphasized, including reasoning under uncertainty, learning, neural networks, natural language, vision, robotics, and philosophical foundations. We cover many of the more recent ideas in the field, including simulated annealing, memory-bounded search, global ontologies, dynamic and adaptive probabilistic (Bayesian) networks, computational learning theory, and reinforcement learning. We also provide extensive notes and references on the historical sources and current literature for the main ideas in each chapter.
4. *Equal emphasis on theory and practice.*
   Theory and practice are given equal emphasis. All material is grounded in first principles with rigorous theoretical analysis where appropriate, but the point of the theory is to get the concepts across and explain how they are used in actual, fielded systems. The reader of this book will come away with an appreciation for the basic concepts and mathematical methods of AI, and also with an idea of what can and cannot be done with today's technology, at what cost, and using what techniques.
5. *Understanding through implementation.*
   The principles of intelligent agent design are clarified by using them to actually build agents. Chapter 2 provides an overview of agent design, including a basic agent and environment

project. Subsequent chapters include programming exercises that ask the student to add capabilities to the agent, making it behave more and more interestingly and (we hope) intelligently. Algorithms are presented at three levels of detail: prose descriptions and pseudo-code in the text, and complete Common Lisp programs available on the Internet or on floppy disk. All the agent programs are interoperable and work in a uniform framework for simulated environments.

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 the 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. We also hope that AI researchers could benefit from thinking about the unifying approach we advocate.

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 adaptive probabilistic networks in detail. Some experience with nonnumeric programming is desirable, but can be picked up in a few weeks study. We provide implementations of all algorithms in Common Lisp (see Appendix B), but other languages such as Scheme, Prolog, Smalltalk, C++, or ML could be used instead.

Overview of the book

The book is divided into eight parts. Part I, "Artificial Intelligence," sets the stage for all the others, and offers a view of the AI enterprise based around the idea of intelligent agents—systems that can decide what to do and 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 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, what one's actions might do—and how to reason logically with that knowledge. Part IV, "Acting Logically," 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 to 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; it also introduces a new kind of component, the *neural network*, and its associated learning procedures. 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," analyses the past and future of AI, and provides some light amusement by discussing what AI really is and why it has already succeeded to some degree, and airing the views of those philosophers who believe that AI can never succeed at all.

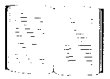
Using this book

This is a big book; covering *all* the chapters and the projects would take two semesters. You will notice that the book is divided into 27 chapters, which makes it easy to select the appropriate material for any chosen course of study. Each chapter can be covered in approximately one week. Some reasonable choices for a variety of quarter and semester courses are as follows:

- *One-quarter general introductory course:*
  Chapters 1, 2, 3, 6, 7, 9, 11, 14, 15, 18, 22.
- *One-semester general introductory course:*
  Chapters 1, 2, 3, 4, 6, 7, 9, 11, 13, 14, 15, 18, 19, 22, 24, 26, 27.
- *One-quarter course with concentration on search and planning:*
  Chapters 1, 2, 3, 4, 5, 6, 7, 9, 11, 12, 13.
- *One-quarter course with concentration on reasoning and expert systems:*
  Chapters 1, 2, 3, 6, 7, 8, 9, 10, 11, 14, 15, 16.
- *One-quarter course with concentration on natural language:*
  Chapters 1, 2, 3, 6, 7, 8, 9, 14, 15, 22, 23, 26, 27.
- *One-semester course with concentration on learning and neural networks:*
  Chapters 1, 2, 3, 4, 6, 7, 9, 14, 15, 16, 17, 18, 19, 20, 21.
- *One-semester course with concentration on vision and robotics:*
  Chapters 1, 2, 3, 4, 6, 7, 11, 13, 14, 15, 16, 17, 24, 25, 20.

These sequences could be used for both undergraduate and graduate courses. The relevant parts of the book could also be used to provide the first phase of graduate specialty courses. For example, Part VI could be used in conjunction with readings from the literature in a course on machine learning.

We have decided *not* to designate certain sections as "optional" or certain exercises as "difficult," as individual tastes and backgrounds vary widely. Exercises requiring significant programming are marked with a keyboard icon, and those requiring some investigation of the literature are marked with a book icon. Altogether, over 300 exercises are included. Some of them are large enough to be considered term projects. Many of the exercises can best be solved by taking advantage of the code repository, which is described in Appendix B. Throughout the book, important points are marked with a *pointing icon.*



If you have any comments on the book, we'd like to hear from you. Appendix B includes information on how to contact us.

Acknowledgements

Jitendra Malik wrote most of Chapter 24 (Vision) and John Canny wrote most of Chapter 25 (Robotics). Doug Edwards researched the Historical Notes sections for all chapters and wrote much of them. Tim Huang helped with formatting of the diagrams and algorithms. Maryann Simmons prepared the 3-D model from which the cover illustration was produced, and Lisa Marie Sardegna did the postprocessing for the final image. Alan Apt, Mona Pompili, and Sondra Chavez at Prentice Hall tried their best to keep us on schedule and made many helpful suggestions on design and content.

Stuart would like to thank his parents, brother, and sister for their encouragement and their patience at his extended absence. He hopes to be home for Christmas. He would also like to thank Loy Sheflott for her patience and support. He hopes to be home some time tomorrow afternoon. His intellectual debt to his Ph.D. advisor, Michael Genesereth, is evident throughout the book. 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, his advisor (Bob Wilensky), supervisors (Bill Woods and Bob Sproull) and employer (Sun Microsystems) for supporting his work in AI, and his wife (Kris) and friends for encouraging and tolerating him through the long hours of writing.

Before publication, drafts of this book were used in 26 courses by about 1000 students. Both of us deeply appreciate the many comments of these students and instructors (and other reviewers). We can't thank them all individually, but we would like to acknowledge the especially helpful comments of these people:

Tony Barrett, Howard Beck, John Binder, Larry Bookman, Chris Brown, Lauren Burka, Murray Campbell, Anil Chakravarthy, Roberto Cipolla, Doug Edwards, Kutluhan Erol, Jeffrey Forbes, John Fosler, Bob Futrelle, Sabine Glesner, Barbara Grosz, Steve Hanks, Othar Hansson, Jim Hendler, Tim Huang, Seth Hutchinson, Dan Jurafsky, Leslie Pack Kaelbling, Keiji Kanazawa, Surekha Kasibhatla, Simon Kasif, Daphne Koller, Rich Korf, James Kurien, John Lazzaro, Jason Leatherman, Jon LeBlanc, Jim Martin, Andy Mayer, Steve Minton, Leora Morgenstern, Ron Musick, Stuart Nelson, Steve Omohundro, Ron Parr, Tony Passera, Michael Pazzani, Ira Pohl, Martha Pollack, Bruce Porter, Malcolm Pradhan, Lorraine Prior, Greg Provan, Philip Resnik, Richard Scherl, Daniel Sleator, Robert Sproull, Lynn Stein, Devika Subramanian, Rich Sutton, Jonathan Tash, Austin Tate, Mark Torrance, Randall Upham, Jim Waldo, Bonnie Webber, Michael Wellman, Dan Weld, Richard Yen, Shlomo Zilberstein.

Summary of Contents

I Artificial Intelligence 1
- 1 **Introduction** ... 3
- 2 Intelligent **Agents** ... 31

II Problem-solving 53
- 3 Solving Problems by Searching ... 55
- 4 Informed Search Methods ... 92
- 5 Game **Playing** ... 122

III Knowledge and reasoning 149
- 6 Agents that Reason **Logically** ... 151
- 7 First-Order **Logic** ... 185
- 8 **Building** a Knowledge Base ... 217
- 9 Inference in First-Order **Logic** ... 265
- 10 Logical Reasoning **Systems** ... 297

IV Acting logically 335
- 11 **Planning** ... 337
- 12 Practical Planning ... 367
- 13 Planning and **Acting** ... 392

V Uncertain knowledge and reasoning 413
- 14 **Uncertainty** ... 415
- 15 Probabilistic Reasoning **Systems** ... 436
- 16 Making Simple Decisions ... 471
- 17 Making Complex Decisions ... 498

VI Learning 523
- 18 Learning from **Observations** ... 525
- 19 Learning in Neural and Belief **Networks** ... 563
- 20 Reinforcement **Learning** ... 598
- 21 Knowledge in **Learning** ... 625

VII Communicating, perceiving, and acting 649
- 22 Agents that Communicate ... 651
- 23 Practical Natural Language Processing ... 691
- 24 Perception ... 724
- 25 **Robotics** ... 773

VIII Conclusions 815
- 26 Philosophical Foundations ... 817
- 27 **AI:** Present and Future ... 842
- A Complexity analysis and O() **notation** ... 851
- B Notes on Languages and **Algorithms** ... 854

Bibliography 859

Index 905

Contents

I Artificial Intelligence **1**

1 Introduction **3**
1.1 What is AI? . . . 4
Acting humanly: The Turing Test approach . . . 5
Thinking humanly: The cognitive modelling approach . . . 6
Thinking rationally: The laws of thought approach . . . 6
Acting rationally: The rational agent approach . . . 7
1.2 The Foundations of Artificial Intelligence . . . 8
Philosophy (428 B.C.-present) . . . 8
Mathematics (c. 800–present) . . . 11
Psychology (1879–present) . . . 12
Computer engineering (1940–present) . . . 14
Linguistics (1957–present) . . . 15
1.3 The History of Artificial Intelligence . . . 16
The gestation of artificial intelligence (1943–1956) . . . 16
Early enthusiasm, great expectations (1952-1969) . . . 17
A dose of reality (1966-1974) . . . 20
Knowledge-based systems: The key to power? (1969–1979) . . . 22
AI becomes an industry (1980-1988) . . . 24
The return of neural networks (1986–present) . . . 24
Recent events (1987–present) . . . 25
1.4 The State of the Art . . . 26
1.5 Summary . . . 27
Bibliographical and Historical Notes . . . 28
Exercises . . . 28

2 Intelligent Agents **31**
2.1 Introduction . . . 31
2.2 How Agents Should Act . . . 31
The ideal mapping from percept sequences to actions . . . 34
Autonomy . . . 35
2.3 Structure of Intelligent Agents . . . 35
Agent programs . . . 37
Why not just look up the answers? . . . 38
An example . . . 39
Simple reflex agents . . . 40
Agents that keep track of the world . . . 41
Goal-based agents . . . 42
Utility-based agents . . . 44
2.4 Environments . . . 45

Properties of environments . . . 46
Environment programs . . . 47
2.5 Summary . . . 49
Bibliographical and Historical Notes . . . 50
Exercises . . . 50

II Problem-solving **53**

3 Solving Problems by Searching 55
3.1 Problem-Solving Agents . . . 55
3.2 Formulating Problems . . . 57
Knowledge and problem types . . . 58
Well-defined problems and solutions . . . 60
Measuring problem-solving performance . . . 61
Choosing states and actions . . . 61
3.3 Example Problems . . . 63
Toy problems . . . 63
Real-world problems . . . 68
3.4 Searching for Solutions . . . 70
Generating action sequences . . . 70
Data structures for search trees . . . 72
3.5 Search Strategies . . . 73
Breadth-first search . . . 74
Uniform cost search . . . 75
Depth-first search . . . 77
Depth-limited search . . . 78
Iterative deepening search . . . 78
Bidirectional search . . . 80
Comparing search strategies . . . 81
3.6 Avoiding Repeated States . . . 82
3.7 Constraint Satisfaction Search . . . 83
3.8 Summary . . . 85
Bibliographical and Historical Notes . . . 86
Exercises . . . 87

4 Informed Search Methods **92**
4.1 Best-First Search . . . 92
Minimize estimated cost to reach a goal: Greedy search . . . 93
Minimizing the total path cost: A* search . . . 96
4.2 Heuristic Functions . . . 101
The effect of heuristic accuracy on performance . . . 102
Inventing heuristic functions . . . 103
Heuristics for constraint satisfaction problems . . . 104
4.3 Memory Bounded Search . . . 106

Iterative deepening A* search (IDA*) . . . 106
SMA* search . . . 107
4.4 Iterative Improvement Algorithms . . . 111
Hill-climbing search . . . 111
Simulated annealing . . . 113
Applications in constraint satisfaction problems . . . 114
4.5 Summary . . . 115
Bibliographical and Historical Notes . . . 115
Exercises . . . 118

5 Game Playing **122**
5.1 Introduction: Games as Search Problems . . . 122
5.2 Perfect Decisions in Two-Person Games . . . 123
5.3 Imperfect Decisions . . . 126
Evaluation functions . . . 127
Cutting off search . . . 129
5.4 Alpha-Beta Pruning . . . 129
Effectiveness of alpha-beta pruning . . . 131
5.5 Games That Include an Element of Chance . . . 133
Position evaluation in games with chance nodes . . . 135
Complexity of expectiminimax . . . 135
5.6 State-of-the-Art Game Programs . . . 136
Chess . . . 137
Checkers or Draughts . . . 138
Othello . . . 138
Backgammon . . . 139
Go . . . 139
5.7 Discussion . . . 139
5.8 Summary . . . 141
Bibliographical and Historical Notes . . . 141
Exercises . . . 145

III Knowledge and reasoning **149**

6 Agents that Reason Logically **151**
6.1 A Knowledge-Based Agent . . . 151
6.2 The Wumpus World Environment . . . 153
Specifying the environment . . . 154
Acting and reasoning in the wumpus world . . . 155
6.3 Representation, Reasoning, and Logic . . . 157
Representation . . . 160
Inference . . . 163
Logics . . . 165
6.4 Propositional Logic: A Very Simple Logic . . . 166

Syntax . . . 166
Semantics . . . 168
Validity and inference . . . 169
Models . . . 170
Rules of inference for propositional logic . . . 171
Complexity of propositional inference . . . 173
6.5 An Agent for the Wumpus World . . . 174
The knowledge base . . . 174
Finding the wumpus . . . 175
Translating knowledge into action . . . 176
Problems with the propositional agent . . . 176
6.6 Summary . . . 178
Bibliographical and Historical Notes . . . 178
Exercises . . . 180

7 First-Order Logic **185**
7.1 Syntax and Semantics . . . 186
Terms . . . 188
Atomic sentences . . . 189
Complex sentences . . . 189
Quantifiers . . . 189
Equality . . . 193
7.2 Extensions and Notational Variations . . . 194
Higher-order logic . . . 195
Functional and predicate expressions using the A operator . . . 195
The uniqueness quantifier ∃! . . . 196
The uniqueness operator ι . . . 196
Notational variations . . . 196
7.3 Using First-Order Logic . . . 197
The kinship domain . . . 197
Axioms, definitions, and theorems . . . 198
The domain of sets . . . 199
Special notations for sets, lists and arithmetic . . . 200
Asking questions and getting answers . . . 200
7.4 Logical Agents for the Wumpus World . . . 201
7.5 A Simple Reflex Agent . . . 202
Limitations of simple reflex agents . . . 203
7.6 Representing Change in the World . . . 203
Situation calculus . . . 204
Keeping track of location . . . 206
7.7 Deducing Hidden Properties of the World . . . 208
7.8 Preferences Among Actions . . . 210
7.9 Toward a Goal-Based Agent . . . 211
7.10 **Summary** . . . 211

Bibliographical and Historical Notes . . . 212
Exercises . . . 213

8 Building a Knowledge Base 217
8.1 Properties of Good and Bad Knowledge Bases . . . 218
8.2 Knowledge Engineering . . . 221
8.3 The Electronic Circuits Domain . . . 223
Decide what to talk about . . . 223
Decide on a vocabulary . . . 224
Encode general rules . . . 225
Encode the specific instance . . . 225
Pose queries to the inference procedure . . . 226
8.4 General Ontology . . . 226
Representing Categories . . . 229
Measures . . . 231
Composite objects . . . 233
Representing change with events . . . 234
Times, intervals, and actions . . . 238
Objects revisited . . . 240
Substances and objects . . . 241
Mental events and mental objects . . . 243
Knowledge and action . . . 247
8.5 The Grocery Shopping World . . . 247
Complete description of the shopping simulation . . . 248
Organizing knowledge . . . 249
Menu-planning . . . 249
Navigating . . . 252
Gathering . . . 253
Communicating . . . 254
Paying . . . 255
8.6 Summary . . . 256
Bibliographical and Historical Notes . . . 256
Exercises . . . 261

9 Inference in First-Order Logic **265**
9.1 Inference Rules Involving Quantifiers . . . 265
9.2 An Example Proof . . . 266
9.3 Generalized Modus Ponens . . . 269
Canonical form . . . 270
Unification . . . 270
Sample proof revisited . . . 271
9.4 Forward and Backward Chaining . . . 272
Forward-chaining algorithm . . . 273
Backward-chaining algorithm . . . 275

9.5 Completeness . . . 276
9.6 Resolution: A Complete Inference Procedure . . . 277
The resolution inference rule . . . 278
Canonical forms for resolution . . . 278
Resolution proofs . . . 279
Conversion to Normal Form . . . 281
Example proof . . . 282
Dealing with equality . . . 284
Resolution strategies . . . 284
9.7 Completeness of resolution . . . 286
9.8 Summary . . . 290
Bibliographical and Historical Notes . . . 291
Exercises . . . 294

10 Logical Reasoning Systems **297**
10.1 Introduction . . . 297
10.2 Indexing, Retrieval, and Unification . . . 299
Implementing sentences and terms . . . 299
Store and fetch . . . 299
Table-based indexing . . . 300
Tree-based indexing . . . 301
The unification algorithm . . . 302
10.3 Logic Programming Systems . . . 304
The Prolog language . . . 304
Implementation . . . 305
Compilation of logic programs . . . 306
Other logic programming languages . . . 308
Advanced control facilities . . . 308
10.4 Theorem Provers . . . 310
Design of a theorem prover . . . 310
Extending Prolog . . . 311
Theorem provers as assistants . . . 312
Practical uses of theorem provers . . . 313
10.5 Forward-Chaining Production Systems . . . 313
Match phase . . . 314
Conflict resolution phase . . . 315
Practical uses of production systems . . . 316
10.6 Frame Systems and Semantic Networks . . . 316
Syntax and semantics of semantic networks . . . 317
Inheritance with exceptions . . . 319
Multiple inheritance . . . 320
Inheritance and change . . . 320
Implementation of semantic networks . . . 321
Expressiveness of semantic networks . . . 323

10.7 Description Logics . . . 323
Practical uses of description logics . . . 325
10.8 Managing Retractions, Assumptions, and Explanations . . . 325
10.9 Summary . . . 327
Bibliographical and Historical Notes . . . 328
Exercises . . . 332

IV Acting logically 335

11 Planning **337**
11.1 A Simple Planning Agent . . . 337
11.2 From Problem Solving to Planning . . . 338
11.3 Planning in Situation Calculus . . . 341
11.4 Basic Representations for Planning . . . 343
Representations for states and goals . . . 343
Representations for actions . . . 344
Situation space and plan space . . . 345
Representations for plans . . . 346
Solutions . . . 349
11.5 A Partial-Order Planning Example . . . 349
11.6 A Partial-Order Planning Algorithm . . . 355
11.7 Planning with Partially Instantiated Operators . . . 357
11.8 Knowledge Engineering for Planning . . . 359
The blocks world . . . 359
Shakey's world . . . 360
11.9 Summary . . . 362
Bibliographical and Historical Notes . . . 363
Exercises . . . 364

12 Practical Planning **367**
12.1 Practical Planners . . . 367
Spacecraft assembly, integration, and verification . . . 367
Job shop scheduling . . . 369
Scheduling for space missions . . . 369
Buildings, aircraft carriers, and beer factories . . . 371
12.2 Hierarchical Decomposition . . . 371
Extending the language . . . 372
Modifying the planner . . . 374
12.3 Analysis of Hierarchical Decomposition . . . 375
Decomposition and sharing . . . 379
Decomposition versus approximation . . . 380
12.4 More Expressive Operator Descriptions . . . 381
Conditional effects . . . 381
Negated and disjunctive goals . . . 382

Universal quantification . . . 383
A planner for expressive operator descriptions . . . 384
12.5 Resource Constraints . . . 386
Using measures in planning . . . 386
Temporal constraints . . . 388
12.6 Summary . . . 388
Bibliographical and Historical Notes . . . 389
Exercises . . . 390

13 Planning and Acting **392**
13.1 Conditional Planning . . . 393
The nature of conditional plans . . . 393
An algorithm for generating conditional plans . . . 395
Extending the plan language . . . 398
13.2 A Simple Replanning Agent . . . 401
Simple replanning with execution monitoring . . . 402
13.3 Fully Integrated Planning and Execution . . . 403
13.4 Discussion and Extensions . . . 407
Comparing conditional planning and replanning . . . 407
Coercion and abstraction . . . 409
13.5 Summary . . . 410
Bibliographical and Historical Notes . . . 411
Exercises . . . 412

V Uncertain knowledge and reasoning **413**

14 Uncertainty **415**
14.1 Acting under Uncertainty . . . 415
Handling uncertain knowledge . . . 416
Uncertainty and rational decisions . . . 418
Design for a decision-theoretic agent . . . 419
14.2 Basic Probability Notation . . . 420
Prior probability . . . 420
Conditional probability . . . 421
14.3 The Axioms of Probability . . . 422
Why the axioms of probability are reasonable . . . 423
The joint probability distribution . . . 425
14.4 Bayes' Rule and Its Use . . . 426
Applying Bayes' rule: The simple case . . . 426
Normalization . . . 427
Using Bayes' rule: Combining evidence . . . 428
14.5 Where Do Probabilities Come From? . . . 430
14.6 Summary . . . 431
Bibliographical and Historical Notes . . . 431

Exercises . . . 433

15 Probabilistic Reasoning Systems 436
15.1 Representing Knowledge in an Uncertain Domain . . . 436
15.2 The Semantics of Belief Networks . . . 438
Representing the joint probability distribution . . . 439
Conditional independence relations in belief networks . . . 444
15.3 Inference in Belief Networks . . . 445
The nature of probabilistic inferences . . . 446
An algorithm for answering queries . . . 447
15.4 Inference in Multiply Connected Belief Networks . . . 453
Clustering methods . . . 453
Cutset conditioning methods . . . 454
Stochastic simulation methods . . . 455
15.5 Knowledge Engineering for Uncertain Reasoning . . . 456
Case study: The Pathfinder system . . . 457
15.6 Other Approaches to Uncertain Reasoning . . . 458
Default reasoning . . . 459
Rule-based methods for uncertain reasoning . . . 460
Representing ignorance: Dempster-Shafer theory . . . 462
Representing vagueness: Fuzzy sets and fuzzy logic . . . 463
15.7 Summary . . . 464
Bibliographical and Historical Notes . . . 464
Exercises . . . 467

16 Making Simple Decisions 471
16.1 Combining Beliefs and Desires Under Uncertainty . . . 471
16.2 The Basis of Utility Theory . . . 473
Constraints on rational preferences . . . 473
. . . and then there was Utility . . . 474
16.3 Utility Functions . . . 475
The utility of money . . . 476
Utility scales and utility assessment . . . 478
16.4 Multiattribute utility functions . . . 480
Dominance . . . 481
Preference structure and multiattribute utility . . . 483
16.5 Decision Networks . . . 484
Representing a decision problem using decision networks . . . 484
Evaluating decision networks . . . 486
16.6 The Value of Information . . . 487
A simple example . . . 487
A general formula . . . 488
Properties of the value of information . . . 489
Implementing an information-gathering agent . . . 490

16.7 Decision-Theoretic Expert Systems . . . 491
16.8 Summary . . . 493
Bibliographical and Historical Notes . . . 493
Exercises . . . 495

17 Making Complex Decisions **498**
17.1 Sequential Decision Problems . . . 498
17.2 Value Iteration . . . 502
17.3 Policy Iteration . . . 505
17.4 Decision-Theoretic Agent Design . . . 508
The decision cycle of a rational agent . . . 508
Sensing in uncertain worlds . . . 510
17.5 Dynamic Belief Networks . . . 514
17.6 Dynamic Decision Networks . . . 516
Discussion . . . 518
17.7 Summary . . . 519
Bibliographical and Historical Notes . . . 520
Exercises . . . 521

VI Learning **523**

18 Learning from Observations 525
18.1 A General Model of Learning Agents . . . 525
Components of the performance element . . . 527
Representation of the components . . . 528
Available feedback . . . 528
Prior knowledge . . . 528
Bringing it all together . . . 529
18.2 Inductive Learning . . . 529
18.3 Learning Decision Trees . . . 531
Decision trees as performance elements . . . 531
Expressiveness of decision trees . . . 532
Inducing decision trees from examples . . . 534
Assessing the performance of the learning algorithm . . . 538
Practical uses of decision tree learning . . . 538
18.4 Using Information Theory . . . 540
Noise and overfitting . . . 542
Broadening the applicability of decision trees . . . 543
18.5 Learning General Logical Descriptions . . . 544
Hypotheses . . . 544
Examples . . . 545
Current-best-hypothesis search . . . 546
Least-commitment search . . . 549
Discussion . . . 552

18.6 Why Learning Works: Computational Learning Theory . . . 552
How many examples are needed? . . . 553
Learning decision lists . . . 555
Discussion . . . 557
18.7 Summary . . . 558
Bibliographical and Historical Notes . . . 559
Exercises . . . 560

19 Learning in Neural and Belief Networks **563**
19.1 How the Brain Works . . . 564
Comparing brains with digital computers . . . 565
19.2 Neural Networks . . . 567
Notation . . . 567
Simple computing elements . . . 567
Network structures . . . 570
Optimal network structure . . . 572
19.3 Perceptrons . . . 573
What perceptrons can represent . . . 573
Learning linearly separable functions . . . 575
19.4 Multilayer Feed-Forward Networks . . . 578
Back-propagation learning . . . 578
Back-propagation as gradient descent search . . . 580
Discussion . . . 583
19.5 Applications of Neural Networks . . . 584
Pronunciation . . . 585
Handwritten character recognition . . . 586
Driving . . . 586
19.6 Bayesian Methods for Learning Belief Networks . . . 588
Bayesian learning . . . 588
Belief network learning problems . . . 589
Learning networks with fixed structure . . . 589
A comparison of belief networks and neural networks . . . 592
19.7 Summary . . . 593
Bibliographical and Historical Notes . . . 594
Exercises . . . 596

20 Reinforcement Learning **598**
20.1 Introduction . . . 598
20.2 Passive Learning in a Known Environment . . . 600
Naïve updating . . . 601
Adaptive dynamic programming . . . 603
Temporal difference learning . . . 604
20.3 Passive Learning in an Unknown Environment . . . 605
20.4 Active Learning in an Unknown Environment . . . 607

20.5 Exploration . . . 609
20.6 Learning an Action-Value Function . . . 612
20.7 Generalization in Reinforcement Learning . . . 615
Applications to game-playing . . . 617
Application to robot control . . . 617
20.8 Genetic Algorithms and Evolutionary Programming . . . 619
20.9 Summary . . . 621
Bibliographical and Historical Notes . . . 622
Exercises . . . 623

21 Knowledge in Learning **625**
21.1 Knowledge in Learning . . . 625
Some simple examples . . . 626
Some general schemes . . . 627
21.2 Explanation-Based Learning . . . 629
Extracting general rules from examples . . . 630
Improving efficiency . . . 631
21.3 Learning Using Relevance Information . . . 633
Determining the hypothesis space . . . 633
Learning and using relevance information . . . 634
21.4 Inductive Logic Programming . . . 636
An example . . . 637
Inverse resolution . . . 639
Top-down learning methods . . . 641
21.5 Summary . . . 644
Bibliographical and Historical Notes . . . 645
Exercises . . . 647

VII Communicating, perceiving, and acting **649**

22 **Agents that Communicate** **651**
22.1 Communication as Action . . . 652
Fundamentals of language . . . 654
The component steps of communication . . . 655
Two models of communication . . . 659
22.2 Types of Communicating Agents . . . 659
Communicating using Tell and Ask . . . 660
Communicating using formal language . . . 661
An agent that communicates . . . 662
22.3 A Formal Grammar for a Subset of English . . . 662
The Lexicon of $\mathcal{E}_0$ . . . 664
The Grammar of $\mathcal{E}_0$ . . . 664
22.4 Syntactic Analysis (Parsing) . . . 664
22.5 Definite Clause Grammar (DCG) . . . 667

22.6 Augmenting a Grammar . . . 668
Verb Subcategorization . . . 669
Generative Capacity of Augmented Grammars . . . 671
22.7 Semantic Interpretation . . . 672
Semantics as DCG Augmentations . . . 673
The semantics of "John loves Mary" . . . 673
The semantics of $\mathcal{E}_1$ . . . 675
Converting quasi-logical form to logical form . . . 677
Pragmatic Interpretation . . . 678
22.8 Ambiguity and Disambiguation . . . 680
Disambiguation . . . 682
22.9 A Communicating Agent . . . 683
22.10 Summary . . . 684
Bibliographical and Historical Notes . . . 685
Exercises . . . 688

23 Practical Natural Language Processing 691
23.1 Practical Applications . . . 691
Machine translation . . . 691
Database access . . . 693
Information retrieval . . . 694
Text categorization . . . 695
Extracting data from text . . . 696
23.2 Efficient Parsing . . . 696
Extracting parses from the chart: Packing . . . 701
23.3 Scaling Up the Lexicon . . . 703
23.4 Scaling Up the Grammar . . . 705
Nominal compounds and apposition . . . 706
Adjective phrases . . . 707
Determiners . . . 708
Noun phrases revisited . . . 709
Clausal complements . . . 710
Relative clauses . . . 710
Questions . . . 711
Handling agrammatical strings . . . 712
23.5 Ambiguity . . . 712
Syntactic evidence . . . 713
Lexical evidence . . . 713
Semantic evidence . . . 713
Metonymy . . . 714
Metaphor . . . 715
23.6 Discourse Understanding . . . 715
The structure of coherent discourse . . . 717
23.7 Summary . . . 719

Bibliographical and Historical Notes . . . 720
Exercises . . . 721

24 Perception 724
24.1 Introduction . . . 724
24.2 Image Formation . . . 725
Pinhole camera . . . 725
Lens systems . . . 727
Photometry of image formation . . . 729
Spectrophotometry of image formation . . . 730
24.3 Image-Processing Operations for Early Vision . . . 730
Convolution with linear filters . . . 732
Edge detection . . . 733
24.4 Extracting 3-D Information Using Vision . . . 734
Motion . . . 735
Binocular stereopsis . . . 737
Texture gradients . . . 742
Shading . . . 743
Contour . . . 745
24.5 Using Vision for Manipulation and Navigation . . . 749
24.6 Object Representation and Recognition . . . 751
The alignment method . . . 752
Using projective invariants . . . 754
24.7 Speech Recognition . . . 757
Signal processing . . . 758
Defining the overall speech recognition model . . . 760
The language model: P(words) . . . 760
The acoustic model: P(signallwords) . . . 762
Putting the models together . . . 764
The search algorithm . . . 765
Training the model . . . 766
24.8 Summary . . . 767
Bibliographical and Historical Notes . . . 767
Exercises . . . 771

25 **Robotics** 773
25.1 Introduction . . . 773
25.2 Tasks: What Are Robots Good For? . . . 774
Manufacturing and materials handling . . . 774
Gofer robots . . . 775
Hazardous environments . . . 775
Telepresence and virtual reality . . . 776
Augmentation of human abilities . . . 776
25.3 Parts: What Are Robots Made Of? . . . 777

Effectors: Tools for action . . . 777
Sensors: Tools for perception . . . 782
25.4 Architectures . . . 786
Classical architecture . . . 787
Situated automata . . . 788
25.5 Configuration Spaces: A Framework for Analysis . . . 790
Generalized configuration space . . . 792
Recognizable Sets . . . 795
25.6 Navigation and Motion Planning . . . 796
Cell decomposition . . . 796
Skeletonization methods . . . 798
Fine-motion planning . . . 802
Landmark-based navigation . . . 805
Online algorithms . . . 806
25.7 Summary . . . 809
Bibliographical and Historical Notes . . . 809
Exercises . . . 811

VIII Conclusions **815**

26 Philosophical Foundations 817
26.1 The Big Questions . . . 817
26.2 Foundations of Reasoning and Perception . . . 819
26.3 On the Possibility of Achieving Intelligent Behavior . . . 822
The mathematical objection . . . 824
The argument from informality . . . 826
26.4 Intentionality and Consciousness . . . 830
The Chinese Room . . . 831
The Brain Prosthesis Experiment . . . 835
Discussion . . . 836
26.5 Summary . . . 837
Bibliographical and Historical Notes . . . 838
Exercises . . . 840

27 **AI: Present and Future** **842**
27.1 Have We Succeeded Yet? . . . 842
27.2 What Exactly Are We Trying to Do? . . . 845
27.3 What If We Do Succeed? . . . 848

A Complexity analysis and O() notation **851**
A.1 Asymptotic Analysis . . . 851
A.2 Inherently Hard Problems . . . 852
Bibliographical and Historical Notes . . . 853

B Notes on Languages and Algorithms **854**
B.1 Defining Languages with Backus-Naur Form (BNF) 854
B.2 Describing Algorithms with Pseudo-Code 855
Nondeterminism 855
Static variables 856
Functions as values 856
B.3 The Code Repository 857
B.4 Comments 857

Bibliography 859

Index **905**

Part I

ARTIFICIAL INTELLIGENCE

The two chapters in this part introduce the subject of Artificial Intelligence or AI and our approach to the subject: that AI is the study of *agents* that exist in an environment and perceive and act.

and subtracting machine called the Pascaline. Leibniz improved on this in 1694, building a mechanical device that multiplied by doing repeated addition. Progress stalled for over a century until Charles Babbage (1792–1871) dreamed that logarithm tables could be computed by machine. He designed a machine for this task, but never completed the project. Instead, he turned to the design of the Analytical Engine, for which Babbage invented the ideas of addressable memory, stored programs, and conditional jumps. Although the idea of programmable machines was not new—in 1805, Joseph Marie Jacquard invented a loom that could be programmed using punched cards—Babbage's machine was the first artifact possessing the characteristics necessary for universal computation. Babbage's colleague Ada Lovelace, daughter of the poet Lord Byron, wrote programs for the Analytical Engine and even speculated that the machine could play chess or compose music. Lovelace was the world's first programmer, and the first of many to endure massive cost overruns and to have an ambitious project ultimately abandoned.[11] Babbage's basic design was proven viable by Doron Swade and his colleagues, who built a working model using only the mechanical techniques available at Babbage's time (Swade, 1993). Babbage had the right idea, but lacked the organizational skills to get his machine built.

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, the linked list data type, automatic storage management, and some of the key concepts of object-oriented programming and integrated program development environments with graphical user interfaces.

Linguistics (1957–present)

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.

Later developments in linguistics showed the problem to be considerably more complex than it seemed in 1957. Language is ambiguous and leaves much unsaid. This means that understanding language requires an understanding of the subject matter and context, not just an understanding of the structure of sentences. This may seem obvious, but it was not appreciated until the early 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.

[11] She also gave her name to Ada, the U.S. Department of Defense's all-purpose programming language.

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.

ARTIFICIAL INTELLIGENCE

Humankind has given itself the scientific name **homo sapiens**—man the wise—because our mental capacities are so important to our everyday lives and our sense of self. The field of **artificial intelligence,** or AI, attempts to understand intelligent entities. Thus, one reason to study it is to learn more about ourselves. But unlike philosophy and psychology, which are also concerned with intelligence, AI strives to *build* intelligent entities as well as understand them. Another reason to study AI is that these constructed intelligent entities are interesting and useful in their own right. AI has produced many significant and impressive products even at this early stage in its development. Although no one can predict the future in detail, it is clear that computers with human-level intelligence (or better) would have a huge impact on our everyday lives and on the future course of civilization.

AI addresses one of the ultimate puzzles. How is it possible for a slow, tiny brain, whether biological or electronic, to perceive, understand, predict, and manipulate a world far larger and more complicated than itself? How do we go about making something with those properties? These are hard questions, but unlike the search for faster-than-light travel or an antigravity device, the researcher in AI has solid evidence that the quest is possible. All the researcher has to do is look in the mirror to see an example of an intelligent system.

AI is one of the newest disciplines. It was formally initiated in 1956, when the name was coined, although at that point work had been under way for about five years. Along with modern genetics, it 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, and that it takes many years of study before one can contribute new ideas. AI, on the other hand, still has openings for a full-time Einstein.

The study of intelligence is also one of the oldest disciplines. For over 2000 years, philosophers have tried to understand how seeing, learning, remembering, and reasoning could, or should,

be done.[1] The advent of usable computers in the early 1950s turned the learned but armchair speculation concerning these mental faculties into a real experimental and theoretical discipline. Many felt that the new "Electronic Super-Brains" had unlimited potential for intelligence. "Faster Than Einstein" was a typical headline. But as well as providing a vehicle for creating artificially intelligent entities, the computer provides a tool for testing theories of intelligence, and many theories failed to withstand the test—a case of "out of the armchair, into the fire." AI has turned out to be more difficult than many at first imagined, and modem ideas are much richer, more subtle, and more interesting as a result.

AI currently encompasses a huge variety of subfields, from general-purpose areas such as perception and logical reasoning, to specific tasks such as playing chess, proving mathematical theorems, writing poetry, and diagnosing diseases. Often, scientists in other fields move gradually into artificial intelligence, where they find the tools and vocabulary to systematize and automate the intellectual tasks on which they have been working all their lives. Similarly, workers in AI can choose to apply their methods to any area of human intellectual endeavor. In this sense, it is truly a universal field.

1.1 WHAT IS AI?

We have now explained why AI is exciting, but we have not said what it *is*. We could just say, "Well, it has to do with smart programs, so let's get on and write some." But the history of science shows that it is helpful to aim at the right goals. Early alchemists, looking for a potion for eternal life and a method to turn lead into gold, were probably off on the wrong foot. Only when the aim changed, to that of finding explicit theories that gave accurate predictions of the terrestrial world, in the same way that early astronomy predicted the apparent motions of the stars and planets, could the scientific method emerge and productive science take place.

Definitions of artificial intelligence according to eight recent textbooks are shown in Figure 1.1. These definitions vary along two main dimensions. The ones on top are concerned with *thought processes* and *reasoning*, whereas the ones on the bottom address *behavior*. Also, the definitions on the left measure success in terms of *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. This gives us four possible goals to pursue in artificial intelligence, as seen in the caption of Figure 1.1.

RATIONALITY

Historically, all four approaches have been followed. As one might expect, a tension exists between approaches centered around humans and approaches centered around rationality.[2] A human-centered approach must be an empirical science, involving hypothesis and experimental

1 A more recent branch of philosophy is concerned with proving that AI is impossible. We will return to this interesting viewpoint in Chapter 26.

2 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 often make mistakes; we are not all chess grandmasters even though we may 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).

"The exciting new effort to make computers think . . . *machines with minds,* in the full and literal sense" (Haugeland, 1985) "[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning . . ." (Bellman, 1978)	"The study of mental faculties through the use of computational models" (Charniak and McDermott, 1985) "The study of the computations that make it possible to perceive, reason, and act" (Winston, 1992)
"The art of creating machines that perform functions that require intelligence when performed by people" (Kurzweil, 1990) "The study of how to make computers do things at which, at the moment, people are better" (Rich and Knight, 1991)	"A field of study that seeks to explain and emulate intelligent behavior in terms of computational processes" (Schalkoff, 1990) "The branch of computer science that is concerned with the automation of intelligent behavior" (Luger and Stubblefield, 1993)

Figure 1.1 Some definitions of AI. They are organized into four categories:

Systems that think like humans.	Systems that think rationally.
Systems that act like humans.	Systems that act rationally.

confirmation. A rationalist approach involves a combination of mathematics and engineering. People in each group sometimes cast aspersions on work done in the other groups, but the truth is that each direction has yielded valuable insights. Let us look at each 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. Turing defined intelligent behavior as the ability to achieve human-level performance in all cognitive tasks, sufficient to fool an interrogator. Roughly speaking, the test he proposed is that the computer should be interrogated by a human via a teletype, and passes the test if the interrogator cannot tell if there is a computer or a human at the other end. Chapter 26 discusses the details of the test, and whether or not a computer is really intelligent if it passes. For now, 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 (or some other human language);

KNOWLEDGE REPRESENTATION

- **knowledge representation** to store information provided before or during the interrogation;

AUTOMATED REASONING

- **automated reasoning** to use the stored information to answer questions and to draw new conclusions;

MACHINE LEARNING

- **machine learning** to adapt to new circumstances and to detect and extrapolate patterns.

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,

TOTAL TURING TEST

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

◇ **computer vision** to perceive objects, and

ROBOTICS

◇ **robotics** to move them about.

Within AI, there has not been a big effort to try to pass the Turing test. The issue of acting like a human comes up primarily when AI programs have to interact with people, as when an expert system explains how it came to its diagnosis, or a natural language processing system has a dialogue with a user. These programs must behave according to certain normal conventions of human interaction in order to make themselves understood. The underlying representation and reasoning in such a system may or may not be based on a human model.

Thinking humanly: The cognitive modelling 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—or 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 behavior matches human behavior, that is evidence that some of the program's mechanisms may also be operating in humans. For example, Newell and Simon, who developed GPS, the "General Problem Solver" (Newell and Simon, 1961), were not content to have their program correctly solve problems. They were more concerned with comparing the trace of its reasoning steps to traces of human subjects solving the same problems. This is in contrast to other researchers of the same time (such as Wang (1960)), who were concerned with getting the right answers regardless of how humans might do it. The interdisciplinary field of

COGNITIVE SCIENCE

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.

Although cognitive science is a fascinating field in itself, we are not going to be discussing it all that much 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 only has access to a computer for experimentation. We will simply note that AI and cognitive science continue to fertilize each other, especially in the areas of vision, natural language, and learning. The history of psychological theories of cognition is briefly covered on page 12.

Thinking rationally: The laws of thought approach

The Greek philosopher Aristotle was one of the first to attempt to codify "right thinking," that is,

SYLLOGISMS

irrefutable reasoning processes. His famous **syllogisms** provided patterns for argument structures that always gave correct conclusions 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, and initiated the field of **logic.**

LOGIC

The development of formal logic in the late nineteenth and early twentieth centuries, which we describe in more detail in Chapter 6, provided a precise notation for statements about all kinds of things in the world and the relations between them. (Contrast this with ordinary arithmetic notation, which provides mainly for equality and inequality statements about numbers.) By 1965, programs existed that could, given enough time and memory, take a description of a problem in logical notation and find the solution to the problem, if one exists. (If there is no solution, the program might never stop looking for it.) 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 because the power of the representation and reasoning systems are well-defined and fairly well understood.

Acting rationally: The rational agent approach

AGENT

Acting rationally means acting so as to achieve one's goals, given one's beliefs. An **agent** is just something that perceives and acts. (This may be an unusual use of the word, but you will get used to it.) In this approach, AI is viewed as the study and construction of rational agents.

In the "laws of thought" approach to AI, the whole 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 reasonably said to involve inference. For example, pulling one's hand off of a hot stove is a reflex action that is more successful than a slower action taken after careful deliberation.

All the "cognitive 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 in order 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.

The study of AI as rational agent design therefore has two advantages. First, it is more general than the "laws of thought" approach, because correct inference is only a useful mechanism for achieving rationality, and not a necessary one. Second, it is more amenable to scientific

development than 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 may be far from achieving 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 possible in complicated environments. The computational demands are just too high. However, for most of the book, we will adopt the working hypothesis that understanding perfect decision making is a good place to start. It simplifies the problem and provides the appropriate setting for most of the foundational material in the field. Chapters 5 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 and the next, we provide a brief history of AI. Although AI itself is a young field, it has inherited many ideas, viewpoints, and techniques from other disciplines. From over 2000 years of tradition in philosophy, theories of reasoning and learning have emerged, along with the viewpoint that the mind is constituted by the operation of a physical system. From over 400 years of mathematics, we have formal theories of logic, probability, decision making, and computation. From psychology, we have the tools with which to investigate the human mind, and a scientific language within which to express the resulting theories. From linguistics, we have theories of the structure and meaning of language. Finally, from computer science, we have the tools with which to make AI a reality.

Like any history, this one is forced to concentrate on a small number of people and events, and ignore others that were also important. We choose to arrange events to tell the story of how the various intellectual components of modern AI came into being. We certainly would not wish to give the impression, however, that the disciplines from which the components came have all been working toward AI as their ultimate fruition.

Philosophy (428 B.C.-present)

> The safest characterization of the European philosophical tradition is that it consists of a series of footnotes to Plato.
> —Alfred North Whitehead

We begin with the birth of Plato in 428 B.C. His writings range across politics, mathematics, physics, astronomy, and several branches of philosophy. Together, Plato, his teacher Socrates,

and his student Aristotle laid the foundation for much of western thought and culture. The philosopher Hubert Dreyfus (1979, p. 67) says that "The story of artificial intelligence might well begin around 450 B.C." when Plato reported a dialogue in which Socrates asks Euthyphro,[3] "I want to know what is characteristic of piety which makes all actions pious ... that I may have it to turn to, and to use as a standard whereby to judge your actions and those of other men."[4] In other words, Socrates was asking for an *algorithm* to distinguish piety from non-piety. Aristotle went on to try to formulate more precisely the laws governing the rational part of the mind. He developed an informal system of syllogisms for proper reasoning, which in principle allowed one to mechanically generate conclusions, given initial premises. Aristotle did not believe all parts of the mind were governed by logical processes; he also had a notion of intuitive reason.

Now that we have the idea of a set of rules that can describe the working of (at least part of) the mind, the next step is to consider the mind as a physical system. We have to wait for René Descartes (1596–1650) for a clear discussion of the distinction between mind and matter, and 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 mind (or soul or spirit) that is outside of nature, exempt from physical laws. On the other hand, he felt that animals did not possess this dualist quality; they could be considered as if they were machines.

DUALISM

MATERIALISM

An alternative to dualism is **materialism,** which holds that all the world (including the brain and mind) operate according to physical law.[5] Wilhelm Leibniz (1646–1716) was probably the first to take the materialist position to its logical conclusion and build a mechanical device intended to carry out mental operations. Unfortunately, his formulation of logic was so weak that his mechanical concept generator could not produce interesting results.

It is also possible to adopt an intermediate position, in which one accepts that the mind has a physical basis, but denies that it can be *explained* by a reduction to ordinary physical processes. Mental processes and consciousness are therefore part of the physical world, but inherently unknowable; they are beyond rational understanding. Some philosophers critical of AI have adopted exactly this position, as we discuss in Chapter 26.

Barring these possible objections to the aims of AI, philosophy had thus established a tradition in which the mind was conceived of as a physical device operating principally by reasoning with the knowledge that it contained. The next problem is then to establish the source of knowledge. The **empiricist** movement, starting with Francis Bacon's (1561-1626) *Novum Organum*,[6] is characterized by the 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, 1978) proposed what is now known as the principle of **induction:**

EMPIRICIST

INDUCTION

---

[3] The *Euthyphro* describes the events just before the trial of Socrates in 399 B.C. Dreyfus has clearly erred in placing it 51 years earlier.

[4] Note that other translations have "goodness/good" instead of "piety/pious."

[5] In this view, the perception of "free will" arises because the deterministic generation of behavior is constituted by the operation of the mind selecting among what appear to be the possible courses of action. They remain "possible" because the brain does not have access to its own future states.

[6] An update of Aristotle's *organon*, or instrument of thought.

that general rules are acquired by exposure to repeated associations between their elements. The theory was given more formal shape by Bertrand Russell (1872-1970) who introduced **logical positivism.** This doctrine holds that all knowledge can be characterized by logical theories connected, ultimately, to **observation sentences** that correspond to sensory inputs.[7] The **confirmation theory** of Rudolf Carnap and Carl Hempel attempted to establish the nature of the connection between the observation sentences and the more general theories—in other words, to understand how knowledge can be acquired from experience.

LOGICAL POSITIVISM
OBSERVATION SENTENCES
CONFIRMATION THEORY

The final element in the philosophical picture of the mind is the connection between knowledge and action. What form should this connection take, and how can particular actions be justified? These questions are vital to AI, because only by understanding how actions are justified can we understand how to build an agent whose actions are justifiable, or rational. Aristotle provides an elegant answer in the *Nicomachean Ethics* (Book III. 3, 1112b):

> 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, nor a statesman whether he shall produce law and order, nor does any one else deliberate about his end. 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, which in the order of discovery is last . . . 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 approach (with a few minor refinements) was implemented 2300 years later by Newell and Simon in their GPS program, about which they write (Newell and Simon, 1972):

MEANS–ENDS ANALYSIS

> The main methods of GPS jointly embody the heuristic **of means-ends analysis.** Means–ends analysis is typified by the following kind of common-sense argument:
>
> > I want to take my son to nursery school. What's the difference between what I have and what I want? One of distance. What changes distance? My automobile. My automobile won't work. What is needed to make it work? A new battery. What has new batteries? An auto repair shop. I want the repair shop to put in a new battery; but the shop doesn't know I need one. What is the difficulty? One of communication. What allows communication? A telephone . . . and so on.
>
> This kind of analysis—classifying things in terms of the functions they serve and oscillating among ends, functions required, and means that perform them—forms the basic system of heuristic of GPS.

Means-ends analysis is useful, but does not say what to do when several actions will achieve the goal, or when no action will completely achieve it. Arnauld, a follower of Descartes, 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) amplifies on this idea. The more formal theory of decisions is discussed in the following section.

---

[7] 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.

Mathematics (c. 800-present)

Philosophers staked out most of the important ideas of AI, but to make the leap to a formal science required a level of mathematical formalization in three main areas: computation, logic, and probability. The notion of expressing a computation as a formal **algorithm** goes back to al-Khowarazmi, an Arab mathematician of the ninth century, whose writings also introduced Europe to Arabic numerals and algebra.

ALGORITHM

Logic goes back at least to Aristotle, but it was a philosophical rather than mathematical subject until George Boole (1815-1864) introduced his formal language for making logical inference in 1847. Boole's approach was incomplete, but good enough that others filled in the gaps. In 1879, Gottlob Frege (1848-1925) produced a logic that, except for some notational changes, forms the first-order logic that is used today as the most basic knowledge representation system.[8] 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.

David Hilbert (1862-1943), a great mathematician in his own right, is most remembered for the problems he did not solve. In 1900, he presented a list of 23 problems that he correctly predicted would occupy mathematicians for the bulk of the century. The final problem asks if 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 if there were fundamental limits to the power of effective proof procedures. In 1930, Kurt Godel (1906-1978) showed that there exists an effective procedure to prove any true statement in the first-order logic of Frege and Russell; but 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. 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: their truth cannot be established by any algorithm.

INCOMPLETENESS THEOREM

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.

Although undecidability and noncomputability are important to an understanding of computation, the notion of **intractability** has had a much greater impact. Roughly speaking, a class of problems is called intractable if the time required to solve instances of the class grows at least 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 moderate-sized in-

INTRACTABILITY

[8] To understand why Frege's notation was not universally adopted, see the cover of this book.

stances cannot be solved in any reasonable time. Therefore, one should strive to divide the overall problem of generating intelligent behavior into tractable subproblems rather than intractable ones.

REDUCTION

The second important concept in the theory of complexity is **reduction,** which also emerged in the 1960s (Dantzig, 1960; Edmonds, 1962). A reduction is a general transformation from one class of problems to another, such that solutions to the first class can be found by reducing them to problems of the second class and solving the latter problems.

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 an NP-complete problem class can be reduced is likely to be intractable. (Although it has not yet been proved that NP-complete problems are necessarily intractable, few theoreticians believe otherwise.) These results contrast sharply with the "Electronic Super-Brain" enthusiasm accompanying the advent of computers. Despite the ever-increasing speed of computers, subtlety and careful use of resources will characterize intelligent systems. Put crudely, the world is an *extremely* large problem instance!

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. Before his time, the outcomes of gambling games were seen as the will of the gods rather than the whim of chance. 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. Bernoulli also framed an alternative view of probability, as a subjective "degree of belief" rather than an objective ratio of outcomes. Subjective probabilities therefore can be updated as new evidence is obtained. Thomas Bayes (1702-1761) proposed a rule for updating subjective probabilities in the light of new evidence (published posthumously in 1763). Bayes' rule, and the subsequent field of Bayesian analysis, form the basis of the modern approach to uncertain reasoning in AI systems. Debate still rages between supporters of the objective and subjective views of probability, but it is not clear if the difference has great significance for AI. Both versions obey the same set of axioms. Savage's (1954) *Foundations of Statistics* gives a good introduction to the field.

DECISION THEORY

As with logic, a connection must be made between probabilistic reasoning and action. **Decision theory,** pioneered by John Von Neumann and Oskar Morgenstern (1944), combines probability theory with utility theory (which provides a formal and complete framework for specifying the preferences of an agent) to give the first general theory that can distinguish good actions from bad ones. Decision theory is the mathematical successor to utilitarianism, and provides the theoretical basis for many of the agent designs in this book.

Psychology (1879–present)

Scientific psychology can be said to have begun with 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 to this day" (Nalwa, 1993, p.15). In 1879, the same year that Frege launched first-order logic, Wundt opened the first laboratory of experimental psychology at the University of Leipzig. Wundt insisted on carefully controlled experiments in which his workers would perform a perceptual or associative task while introspecting on their thought processes. The careful controls went a long way to make psychology a science, but as the methodology spread, a curious phenomenon arose: each laboratory would report introspective data that just happened to match the theories that were popular in that laboratory. The **behaviorism** movement of John Watson (1878–1958) and Edward Lee Thorndike (1874–1949) rebelled against this subjectivism, rejecting *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 understanding humans. Nevertheless, it had a stronghold on psychology (especially in the United States) from about 1920 to 1960.

BEHAVIORISM

The view that the brain possesses and processes information, which is the principal characteristic of **cognitive psychology,** can be traced back at least to the works of William James[9] (1842–1910). Helmholtz also insisted that perception involved a form of unconscious logical inference. The cognitive viewpoint was largely eclipsed by behaviorism until 1943, when Kenneth Craik published *The Nature of Explanation.* Craik put back the missing mental step between stimulus and response. He claimed that beliefs, goals, and reasoning steps could be useful valid components of a theory of human behavior, and 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)

An agent designed this way can, for example, plan a long trip by considering various possible routes, comparing them, and choosing the best one, all before starting the journey. Since the 1960s, the information-processing view has dominated psychology. It it now almost taken for granted among many psychologists that "a cognitive theory should be like a computer program" (Anderson, 1980). By this it is meant that the theory should describe cognition as consisting of well-defined transformation processes operating at the level of the information carried by the input signals.

For most of the early history of AI and cognitive science, no significant distinction was drawn between the two fields, and it was common to see AI programs described as psychological

[9] 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.

results without any claim as to the exact human behavior they were modelling. In the last decade or so, however, the methodological distinctions have become clearer, and most work now falls into one field or the other.

Computer engineering (1940–present)

For artificial intelligence to succeed, we need two things: intelligence and an artifact. The computer has been unanimously acclaimed as the artifact with the best chance of demonstrating intelligence. The modern digital electronic computer was invented independently and almost simultaneously by scientists in three countries embattled in World War II. The first operational modern computer was the Heath Robinson,[10] built in 1940 by Alan Turing's team for the single purpose of deciphering German messages. When the Germans switched to a more sophisticated code, the electromechanical relays in the Robinson proved to be too slow, and a new machine called the Colossus was built from vacuum tubes. It was completed in 1943, and by the end of the war, ten Colossus machines were in everyday use.

The first operational *programmable* computer was the Z-3, the invention of Konrad Zuse in Germany in 1941. Zuse invented floating-point numbers for the Z-3, and went on in 1945 to develop Plankalkul, the first high-level programming language. Although Zuse received some support from the Third Reich to apply his machine to aircraft design, the military hierarchy did not attach as much importance to computing as did its counterpart in Britain.

In the United States, the first *electronic* computer, the ABC, was assembled by John Atanasoff and his graduate student Clifford Berry between 1940 and 1942 at Iowa State University. The project received little support and was abandoned after Atanasoff became involved in military research in Washington. Two other computer projects were started as secret military research: the Mark I, II, and III computers were developed at Harvard by a team under Howard Aiken; and the ENIAC was developed at the University of Pennsylvania by a team including John Mauchly and John Eckert. ENIAC was the first general-purpose, electronic, digital computer. One of its first applications was computing artillery firing tables. A successor, the EDVAC, followed John Von Neumann's suggestion to use a stored program, so that technicians would not have to scurry about changing patch cords to run a new program.

But perhaps the most critical breakthrough was the IBM 701, built in 1952 by Nathaniel Rochester and his group. This was the first computer to yield a profit for its manufacturer. IBM went on to become one of the world's largest corporations, and sales of computers have grown to j $150 billion/year. In the United States, the computer industry (including software and services) j now accounts for about 10% of the gross national product.

Each generation of computer hardware has brought an increase in speed and capacity, and I a decrease in price. Computer engineering has been remarkably successful, regularly doubling j performance every two years, with no immediate end in sight for this rate of increase. Massively j parallel machines promise to add several more zeros to the overall throughput achievable.

Of course, there were calculating devices before the electronic computer. The abacus is roughly 7000 years old. In the mid-17th century, Blaise Pascal built a mechanical adding 1

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

and subtracting machine called the Pascaline. Leibniz improved on this in 1694, building a mechanical device that multiplied by doing repeated addition. Progress stalled for over a century until Charles Babbage (1792–1871) dreamed that logarithm tables could be computed by machine. He designed a machine for this task, but never completed the project. Instead, he turned to the design of the Analytical Engine, for which Babbage invented the ideas of addressable memory, stored programs, and conditional jumps. Although the idea of programmable machines was not new—in 1805, Joseph Marie Jacquard invented a loom that could be programmed using punched cards—Babbage's machine was the first artifact possessing the characteristics necessary for universal computation. Babbage's colleague Ada Lovelace, daughter of the poet Lord Byron, wrote programs for the Analytical Engine and even speculated that the machine could play chess or compose music. Lovelace was the world's first programmer, and the first of many to endure massive cost overruns and to have an ambitious project ultimately abandoned.[11] Babbage's basic design was proven viable by Doron Swade and his colleagues, who built a working model using only the mechanical techniques available at Babbage's time (Swade, 1993). Babbage had the right idea, but lacked the organizational skills to get his machine built.

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, the linked list data type, automatic storage management, and some of the key concepts of object-oriented programming and integrated program development environments with graphical user interfaces.

Linguistics (1957–present)

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.

Later developments in linguistics showed the problem to be considerably more complex than it seemed in 1957. Language is ambiguous and leaves much unsaid. This means that understanding language requires an understanding of the subject matter and context, not just an understanding of the structure of sentences. This may seem obvious, but it was not appreciated until the early 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.

[11] She also gave her name to Ada, the U.S. Department of Defense's all-purpose programming language.

Modern linguistics and AI were "born" at about the same time, so linguistics does not play a large foundational role in the growth of AI. Instead, the two grew up together, intersecting in a hybrid field called **computational linguistics or natural language processing,** which concentrates on the problem of language use.

1.3 THE HISTORY OF ARTIFICIAL INTELLIGENCE

With the background material behind us, we are now ready to outline the development of AI proper. We could do this by identifying loosely defined and overlapping phases in its development, or by chronicling the various different and intertwined conceptual threads that make up the field. In this section, we will take the former approach, at the risk of doing some degree of violence to the real relationships among subfields. The history of each subfield is covered in individual chapters later in the book.

The gestation of artificial intelligence (1943-1956)

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; the 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 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, such that learning could take place.

The work of McCulloch and Pitts was arguably the forerunner of both the logicist tradition in AI and the connectionist tradition. In the early 1950s, Claude Shannon (1950) and Alan Turing (1953) were writing chess programs for von Neumann-style conventional computers.[12] At the same time, 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 whether this kind of work should be considered mathematics, but von Neumann was on the committee and reportedly said, "If it isn't now it will be someday." Ironically, Minsky was later to prove theorems that contributed to the demise of much of neural network research during the 1970s.

[12] Shannon actually had no real computer to work with, and Turing was eventually denied access to his own team's computers by the British government, on the grounds that research into artificial intelligence was surely frivolous.

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. All together there were ten attendees, including Trenchard More from Princeton, Arthur Samuel from IBM, and Ray Solomonoff and Oliver Selfridge from MIT.

Two researchers from Carnegie Tech,[13] Alien 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 most lasting thing to come out of the workshop was an agreement to adopt McCarthy's new name for the field: **artificial intelligence.**

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. Some modern AI researchers refer 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 the way humans approached the same problems. Thus, GPS was probably the first program to embody the "thinking humanly" approach. The combination of AI and cognitive science has continued at CMU up to the present day.

---

[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.

At IBM, Nathaniel Rochester and his colleagues produced some of the first AI programs. Herbert Gelernter (1959) constructed the Geometry Theorem Prover. Like the Logic Theorist, it proved theorems using explicitly represented axioms. Gelernter soon found that there were too many possible reasoning paths to follow, most of which turned out to be dead ends. To help focus the search, he added the capability to create a numerical representation of a diagram—a particular case of the general theorem to be proved. Before the program tried to prove something, it could first check the diagram to see if it was true in the particular case.

Starting in 1952, Arthur Samuel wrote a series of programs for checkers (draughts) that eventually learned to play tournament-level checkers. Along the way, he disproved the idea that computers can only do what they are told to, as 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 5 covers game playing, and Chapter 20 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 **Lisp,** which was to become the dominant AI programming language. Lisp is the second-oldest language in current use.[15] With Lisp, McCarthy had the tool he needed, but access to scarce and expensive computing resources was also a serious problem. Thus, he and others at MIT invented time sharing. After getting an experimental time-sharing system up at MIT, McCarthy eventually attracted the interest of a group of MIT grads who formed Digital Equipment Corporation, which was to become the world's second largest computer manufacturer, thanks to their time-sharing minicomputers. 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 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 after more than 35 years.

1958 also marked the year that Marvin Minsky moved to MIT. For years he and McCarthy were inseparable as they defined the field together. But they grew apart as 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 took the opportunity to go to Stanford and start the AI lab there. His research agenda of using 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 Section 9.6). Work at Stanford emphasized general-purpose methods for logical reasoning. Applications of

[15] FORTRAN is one year older than Lisp.

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 integration problems typical of first-year college calculus courses. Tom Evans's ANALOGY program (1968) solved geometric analogy problems that appear in IQ tests, such as the one in Figure 1.2. Bertram Raphael's (1968) SIR (Semantic Information Retrieval) was able to accept input statements in a very restricted subset of English and answer questions thereon. Daniel Bobrow's STUDENT program (1967) solved algebra story problems such as

> 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?

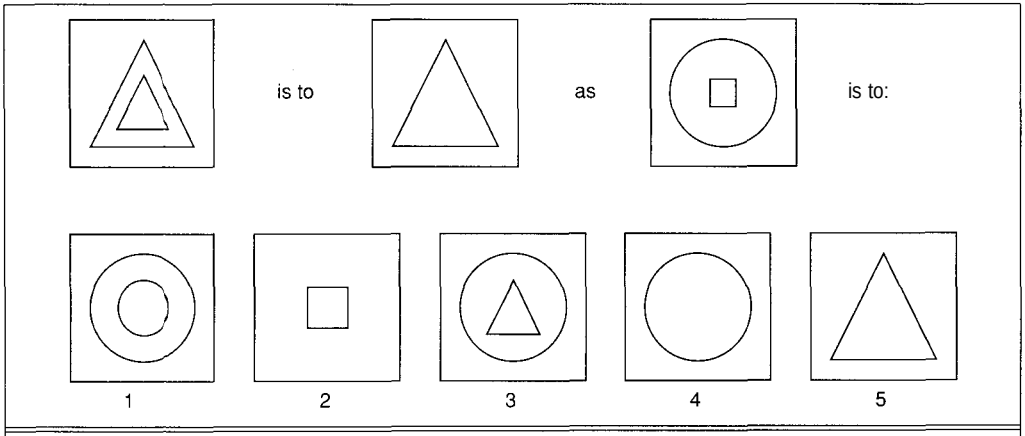


Figure 1.2 An example problem solved by Evans's ANALOGY program.

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 1.3. A task in this world is to rearrange the blocks in a certain way, using a robot hand that can pick up one block at a time. The blocks world was home to the vision project of David Huffman (1971), the vision and constraint-propagation work of David Waltz (1975), the learning theory of Patrick Winston (1970), the natural language understanding program of Terry Winograd (1972), and the planner of Scott Fahlman (1974).

Early work building on the neural networks of McCulloch and Pitts also flourished. The work of 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**.

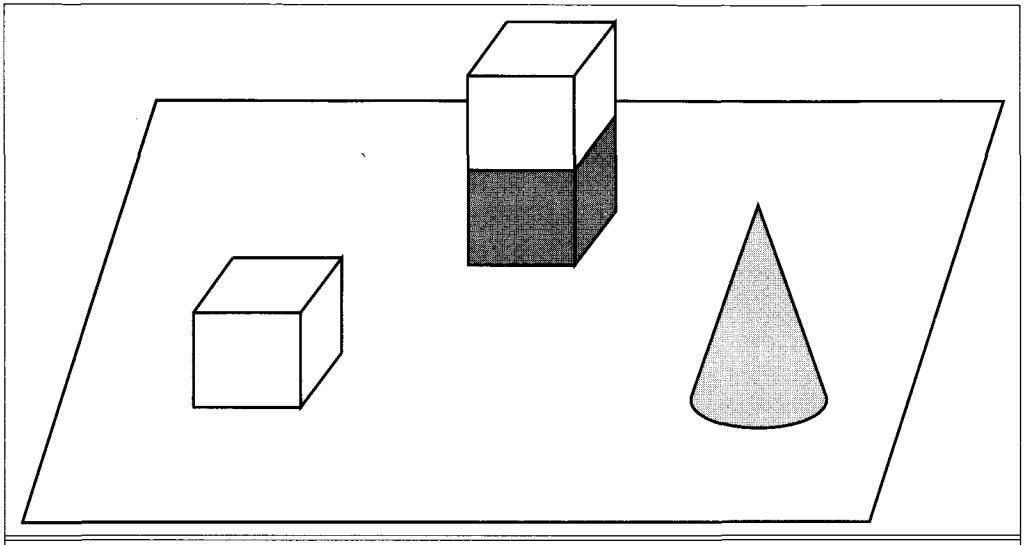


Figure 1.3 A scene from the blocks world. A task for the robot might be "Pick up a big red block," expressed either in natural language or in a formal notation.

Rosenblatt proved the famous **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 Section 19.3.

A dose of reality (1966-1974)

From the beginning, AI researchers were not shy in 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 human mind has been applied.

Although one might argue that terms such as "visible future" can be interpreted in various ways, some of Simon's predictions were more concrete. In 1958, he predicted that within 10 years a computer would be chess champion, and an important new mathematical theorem would be proved by machine. Claims such as these turned out to be wildly optimistic. The barrier that faced almost all AI research projects was that methods that sufficed for demonstrations on one or two simple examples 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 early programs often contained little or no knowledge of their subject matter, and succeeded by means of simple syntactic manipulations. Weizenbaum's ELIZA program (1965), which could apparently engage in serious conversation

on any topic, actually just borrowed and manipulated the sentences typed into it by a human. A typical story occurred in early machine translation efforts, which were generously funded by the 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. In fact, translation requires general knowledge of the subject matter in order to resolve ambiguity and establish the content of the sentence. The famous retranslation 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 cancelled.

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 worked by representing the basic facts about a problem and trying out a series of steps to solve it, combining different combinations of steps until the right one was found. The early programs were feasible only because microworlds contained very few objects. Before the theory of NP-completeness 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 and then apply a selection process to preserve mutations that seemed to improve behavior. Despite thousands of hours of CPU time, almost no progress was demonstrated.

Failure to come to grips with the "combinatorial explosion" was one of the main criticisms of AI contained in the Lighthill report (Lighthill, 1973), which formed the basis for the decision by the British government to end support for AI research in all but two universities. (Oral tradition paints a somewhat different and more colorful picture, with political ambitions and personal animosities that cannot be put in print.)

A third difficulty arose because of some fundamental limitations on the basic structures being used to generate intelligent behavior. For example, in 1969, Minsky and Papert's book *Perceptrons* (1969) proved that although perceptrons 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)

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 they use weak information about the domain. For many complex domains, it turns out that their performance is also weak. The only way around this is to use knowledge more suited to making larger reasoning steps and to solving typically occurring cases in narrow areas of expertise. One might say that to solve a hard problem, you almost have to know the answer already.

WEAK METHODS

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 rapidly became 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:

> 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

Having recognized that the molecule contains a particular substructure, the number of possible candidates is enormously reduced. The DENDRAL team concluded that the new system 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 arguably 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) and 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

EXPERT SYSTEMS

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 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 14), which seemed (at the time) to fit well with how doctors assessed the impact of evidence on the diagnosis.

Other approaches to medical diagnosis were also followed. At Rutgers University, Saul Amarel's *Computers in Biomedicine* project began an ambitious attempt to diagnose diseases based on explicit knowledge of the causal mechanisms of the disease process. Meanwhile, large groups at MIT and the New England Medical Center were pursuing an approach to diagnosis and treatment based on the theories of probability and utility. Their aim was to build systems that gave provably optimal medical recommendations. In medicine, the Stanford approach using rules provided by doctors proved more popular at first. But another probabilistic reasoning system, PROSPECTOR (Duda *et al.*, 1979), generated enormous publicity by recommending exploratory drilling at a geological site that proved to contain a large molybdenum deposit.

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 by 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; 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). William Woods (1973) built the LUNAR system, which allowed geologists to ask questions in English about the rock samples brought back by the Apollo moon mission. LUNAR was the first natural language program that was used by people other than the system's author to get real work done. Since then, many natural language programs have been used as interfaces to databases.

The widespread growth of applications to real-world problems caused a concomitant increase in the demands for workable knowledge representation schemes. A large number of different representation 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 rather more structured approach, collecting together 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 (1980-1988)

The first successful commercial expert system, R1, began operation at Digital Equipment Corporation (McDermott, 1982). The program helped configure orders for new computer systems, and by 1986, it was saving the company an estimated $40 million a year. By 1988, DEC's AI group had 40 deployed expert systems, 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 system technology.

In 1981, the Japanese announced the "Fifth Generation" project, a 10-year plan to build intelligent computers running Prolog in much the same way that ordinary computers run machine code. The idea was that with the ability to make millions of inferences per second, computers would be able to take advantage of vast stores of rules. The project proposed to achieve full-scale natural language understanding, among other ambitious goals.

The Fifth Generation project fueled interest in AI, and by taking advantage of fears of Japanese domination, researchers and corporations were able to generate support for a similar investment in the United States. The Microelectronics and Computer Technology Corporation (MCC) was formed as a research consortium to counter the Japanese project. In Britain, the Alvey report reinstated the funding that was cut by the Lighthill report.[16] In both cases, AI was part of a broad effort, including chip design and human-interface research.

The booming AI industry also included companies such as Carnegie Group, Inference, Intellicorp, and Teknowledge that offered the software tools to build expert systems, and hardware companies such as Lisp Machines Inc., Texas Instruments, Symbolics, and Xerox that were building workstations optimized for the development of Lisp programs. Over a hundred companies built industrial robotic vision systems. Overall, the industry went from a few million in sales in 1980 to $2 billion in 1988.

The return of neural networks (1986–present)

Although computer science had neglected the field of neural networks after Minsky and Papert's *Perceptrons* book, work had continued in other fields, particularly physics. Large collections of simple neurons could be understood in much the same way as large collections of atoms in solids. Physicists such as Hopfield (1982) used techniques from statistical mechanics to analyze the storage and optimization properties of networks, leading to significant cross-fertilization of ideas. Psychologists including David Rumelhart and Geoff Hinton continued the study of neural net models of memory. As we discuss in Chapter 19, 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.

At about the same time, some disillusionment was occurring concerning the applicability of the expert system technology derived from MYCIN-type systems. Many corporations and

[16] To save embarrassment, a new field called IKBS (Intelligent Knowledge-BasedSystems) was defined because Artificial Intelligence had been officially cancelled.

research groups found that building a successful expert system involved much more than simply buying a reasoning system and filling it with rules. Some predicted an "AI Winter" in which AI funding would be squeezed severely. It was perhaps this fear, and the historical factors on the neural network side, that led to a period in which neural networks and traditional AI were seen as rival fields, rather than as mutually supporting approaches to the same problem.

Recent events (1987–present)

Recent years have seen a sea change in both the content and the methodology of research in artificial intelligence.[17] 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.

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 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 to the present discussion. 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 steadily improving their scores. Speech technology and the related field of handwritten character recognition are already making the transition to widespread industrial and consumer applications.

Another area that seems to have benefitted from formalization is planning. Early work by Austin Tate (1977), followed up by David Chapman (1987), has resulted in an elegant synthesis of existing planning programs into a simple framework. There have been a number of advances that built upon each other rather than starting from scratch each time. The result is that planning systems that were only good for microworlds in the 1970s are now used for scheduling of factory work and space missions, among other things. See Chapters 11 and 12 for more details.

Judea Pearl's (1988) *Probabilistic Reasoning in Intelligent Systems* marked 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 **belief network** formalism was invented to allow efficient reasoning about the combination of uncertain evidence. This approach largely overcomes the problems with probabilistic reasoning systems of the 1960s and 1970s, and has come to dominate AI research on uncertain reasoning and expert systems. 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 human experts. Chapters 14 to 16 cover this area.

[17] 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 increased 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.)

Similar gentle revolutions have occurred in robotics, computer vision, machine learning (including neural networks), 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. 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 Alien Newell, John Laird, and Paul Rosenbloom on SOAR (Newell, 1990; Laird *et al.*, 1987) is the best-known example of a complete agent architecture in AI. The so-called "situated" movement aims to understand the workings of agents embedded in real environments with continuous sensory inputs. Many interesting results are coming out of such work, including the realization that the previously isolated subfields of AI may need to be reorganized somewhat when their results are to be tied together into a single agent design.

1.4 THE STATE OF THE ART

International grandmaster Arnold Denker studies the pieces on the board in front of him. He realizes there is no hope; he must resign the game. His opponent, HITECH, becomes the first computer program to defeat a grandmaster in a game of chess (Berliner, 1989).

"I want to go from Boston to San Francisco," the traveller says into the microphone. "What date will you be travelling on?" is the reply. The traveller explains she wants to go October 20th, nonstop, on the cheapest available fare, returning on Sunday. A speech understanding program named PEGASUS handles the whole transaction, which results in a confirmed reservation that saves the traveller $894 over the regular coach fare. Even though the speech recognizer gets one out of ten words wrong,[18] it is able to recover from these errors because of its understanding of how dialogs are put together (Zue *et al.*, 1994).

An analyst in the Mission Operations room of the Jet Propulsion Laboratory suddenly starts paying attention. A red message has flashed onto the screen indicating an "anomaly" with the Voyager spacecraft, which is somewhere in the vicinity of Neptune. Fortunately, the analyst is able to correct the problem from the ground. Operations personnel believe the problem might have been overlooked had it not been for MARVEL, a real-time expert system that monitors the massive stream of data transmitted by the spacecraft, handling routine tasks and alerting the analysts to more serious problems (Schwuttke, 1992).

Cruising the highway outside of Pittsburgh at a comfortable 55 mph, the man in the driver's seat seems relaxed. He should be—for the past 90 miles, he has not had to touch the steering wheel, brake, or accelerator. The real driver is a robotic system that gathers input from video cameras, sonar, and laser range finders attached to the van. It combines these inputs with experience learned from training runs and succesfully computes how to steer the vehicle (Pomerleau, 1993).

A leading expert on lymph-node pathology describes a fiendishly difficult case to the expert system, and examines the system's diagnosis. He scoffs at the system's response. Only slightly worried, the creators of the system suggest he ask the computer for an explanation of

[18] Some other existing systems err only half as often on this task.

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. The expert admits his error, eventually (Heckerman, 1991).

From a camera perched on a street light above the crossroads, the traffic monitor watches the scene. If any humans were awake to read the main screen, they would see "Citroen 2CV turning from Place de la Concorde into Champs Elysees," "Large truck of unknown make stopped on Place de la Concorde," and so on into the night. And occasionally, "Major incident on Place de la Concorde, speeding van collided with motorcyclist," and an automatic call to the emergency services (King *et al.*, 1993; Koller *et al.*, 1994).

These are just a few examples of artificial intelligence systems that exist today. Not magic or science fiction—butrather 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 help arrive at the right 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 reasoning about algorithms.
- Psychologists strengthened the idea that humans and other animals can be considered information processing machines. Linguists showed that language use fits into this model.
- Computer engineering provided the artifact that makes 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.
- 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.
- Recent progress in understanding the theoretical basis for intelligence has gone hand in hand with improvements in the capabilities of real systems.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Daniel Crevier's (1993) *Artificial Intelligence* gives a complete history of the field, and Raymond Kurzweil's (1990) *Age of Intelligent Machines* situates AI in the broader context of computer science and intellectual history in general. Dianne Martin (1993) documents the degree to which early computers were endowed by the media with mythical powers of intelligence.

The methodological status of artificial intelligence is discussed 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.

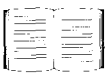
Artificial Intelligence: The Very Idea, by John Haugeland (1985) gives a readable account of the philosophical and practical problems of AI. Cognitive science is well-described by Johnson-Laird's *The Computer and the Mind: An Introduction to Cognitive Science.* Baker (1989) covers the syntactic part of modern linguistics, and Chierchia and McConnell-Ginet (1990) cover semantics. Alien (1995) covers linguistics from the AI point of view.

Early AI work is covered in Feigenbaum and Feldman's *Computers and Thought,* Minsky's *Semantic Information Processing,* and the *Machine Intelligence* series edited by Donald Michie. A large number of influential papers are collected in *Readings in Artificial Intelligence* (Webber and Nilsson, 1981). 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 four-volume *Handbook of Artificial Intelligence* (Barr and Feigenbaum, 1981) contains descriptions of almost every major AI system published before 1981.

The most recent work appears in the proceedings of the major AI conferences: the biennial International Joint Conference on AI (IJCAI), and the annual 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,* and the electronic *Journal of Artificial Intelligence Research.* There are also many journals devoted to specific areas, which we cover in the appropriate chapters. Commercial products are covered in the magazines *AI Expert* and *PC AI.* 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* and the *SIGART Bulletin* contain many topical and tutorial articles as well as announcements of conferences and workshops.

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 completing the book.



1.1 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. Do you think this is reasonable?

1.2 We characterized the definitions of AI along two dimensions, human vs. ideal and thought vs. action. But there are other dimensions that are worth considering. One dimension is whether we are interested in theoretical results or in practical applications. Another is whether we intend our intelligent computers to be conscious or not. Philosophers have had a lot to say about this issue, and although most AI researchers are happy to leave the questions to the philosophers, there has been heated debate. The claim that machines can be conscious is called the strong AI claim; the **weak AI** position makes no such claim. Characterize the eight definitions on page 5 and the seven following definitions according to the four dimensions we have mentioned and whatever other ones you feel are helpful.

STRONG AI

WEAK AI

Artificial intelligence is …

a. "a collection of algorithms that are computationally tractable, adequate approximations of intractably specified problems" (Partridge, 1991)

b. "the enterprise of constructing a physical symbol system that can reliably pass the Turing Test" (Ginsberg, 1993)

c. "the field of computer science that studies how machines can be made to act intelligently" (Jackson, 1986)

d. "a field of study that encompasses computational techniques for performing tasks that apparently require intelligence when performed by humans" (Tanimoto, 1990)

e. "a very general investigation of the nature of intelligence and the principles and mechanisms required for understanding or replicating it" (Sharpies *et al.*, 1989)

f. "the getting of computers to do things that seem to be intelligent" (Rowe, 1988).

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

1.4 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.

1.5 Examine the AI literature to discover whether or not 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. Playing a decent game of bridge at a competitive level.

d. Discovering and proving new mathematical theorems.

e. Writing an intentionally funny story.

f. Giving competent legal advice in a specialized area of law.

g. Translating spoken English into spoken Swedish in real time.

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

1.6 Find an article written by a lay person in a reputable newspaper or magazine claiming the achievement of some intelligent capacity by a machine, where the claim is either wildly exaggerated or false.

1.7 Fact, fiction, and forecast:

a. Find a claim in print by a reputable philosopher or scientist to the effect that a certain capacity will never be exhibited by computers, where that capacity has now been exhibited.

b. Find a claim by a reputable computer scientist to the effect that a certain capacity would be exhibited by a date that has since passed, without the appearance of that capacity.

c. Compare the accuracy of these predictions to predictions in other fields such as biomedicine, fusion power, nanotechnology, transportation, or home electronics.

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

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

2 INTELLIGENT AGENTS

In which we discuss what an intelligent agent does, how it is related to its environment, how it is evaluated, and how we might go about building one.

2.1 INTRODUCTION

An agent is anything that can be viewed as **perceiving** its environment through **sensors and acting** upon that environment through **effectors.** A human agent has eyes, ears, and other organs for sensors, and hands, legs, mouth, and other body parts for effectors. A robotic agent substitutes cameras and infrared range finders for the sensors and various motors for the effectors. A software agent has encoded bit strings as its percepts and actions. A generic agent is diagrammed in Figure 2.1.

Our aim in this book is to design agents that do a good job of acting on their environment. First, we will be a little more precise about what we mean by a good job. Then we will talk about different designs for successful agents—filling in the question mark in Figure 2.1. We discuss some of the general principles used in the design of agents throughout the book, chief among which is the principle that agents should *know* things. Finally, we show how to couple an agent to an environment and describe several kinds of environments.

2.2 How AGENTS SHOULD ACT

RATIONAL AGENT

A **rational agent** is one that does the right thing. Obviously, this is better than doing the wrong thing, but what does it mean? As a first approximation, we will say that the right action is the one that will cause the agent to be most successful. That leaves us with the problem of deciding *how* and *when* to evaluate the agent's success.

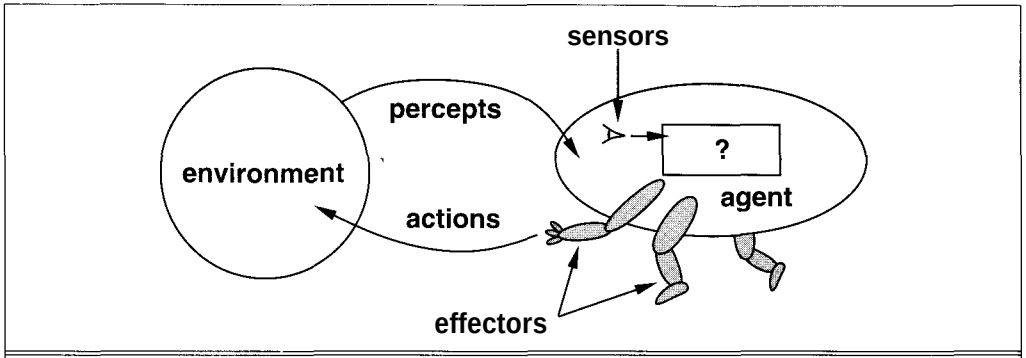


Figure 2.1 Agents interact with environments through sensors and effectors.

PERFORMANCE MEASURE

We use the term **performance measure** for the *how*—the criteria that determine how successful an agent is. 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. (Human agents in particular are notorious for "sour grapes"—saying they did not really want something after they are unsuccessful at getting it.) Therefore, we will insist on an objective performance measure imposed by some authority. In other words, we as outside observers establish a standard of what it means to be successful in an environment and use it to measure the performance of agents.

As an example, consider the case of an agent that is supposed to vacuum a dirty floor. A plausible performance measure would be the amount of dirt cleaned up in a single eight-hour shift. A more sophisticated performance measure would factor in the amount of electricity consumed and the amount of noise generated as well. A third performance measure might give highest marks to an agent that not only cleans the floor quietly and efficiently, but also finds time to go windsurfing at the weekend.[1]

The *when* of evaluating performance is also important. If we measured how much dirt the agent had cleaned up in the first hour of the day, we would be rewarding those agents that start fast (even if they do little or no work later on), and punishing those that work consistently. Thus, we want to measure performance over the long run, be it an eight-hour shift or a lifetime.

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,[2] 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

1 There is a danger here for those who establish performance measures: you often get what you ask for. That is. if you measure success by the amount of dirt cleaned up, then some clever agent is bound to bring in a load of dirt each morning, quickly clean it up, and get a good performance score. What you really want to measure is how clean the floor is, but determining that is more difficult than just weighing the dirt cleaned up.

2 See N. Henderson, "New door latches urged for Boeing 747 jumbo jets." *Washington Post,* 8/24/89.

street." Rather, this points out that rationality is concerned with *expected* success *given what has been perceived.* Crossing the street was rational because most of the time the crossing would be successful, and there was no way I could have foreseen the falling door. Note that another agent that was equipped with radar for detecting falling doors or a steel cage strong enough to repel them would be more successful, but it would not be any more rational.

In other words, we cannot blame an agent for failing to take into account something it could not perceive, or for failing to take an action (such as repelling the cargo door) that it is incapable of taking. But relaxing the requirement of perfection is not just a question of being fair to agents. The point is that if we specify that an intelligent agent should always do what is *actually* the right thing, it will be impossible to design an agent to fulfill this specification—unless we improve the performance of crystal balls.

In summary, what is rational at any given time depends on four things:

- The performance measure that defines degree of success.
- Everything that the agent has perceived so far. We will call this complete perceptual history the **percept sequence.**
- What the agent knows about the environment.
- The actions that the agent can perform.

PERCEPT SEQUENCE

IDEAL RATIONAL AGENT

This leads to a definition of an **ideal rational agent:** *For each possible percept sequence, an ideal rational agent should do whatever action is expected to maximize its performance measure, on the basis of the evidence provided by the percept sequence and whatever built-in knowledge the agent has.*

We need to look carefully at this definition. At first glance, it might appear to allow an agent to indulge in some 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. The definition seems to say that it would be OK for it to cross the road. In fact, this interpretation is wrong on two counts. First, it would not be rational to cross the road: the risk of crossing without looking is too great. Second, an ideal rational agent would have chosen the "looking" action before stepping into the street, because looking helps maximize the expected performance. Doing actions *in order to obtain useful information* is an important part of rationality and is covered in depth in Chapter 16.

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. Consider a clock. It can be thought of as just an inanimate object, or it can be thought of as a simple agent. As an agent, most clocks always do the right action: moving their hands (or displaying digits) in the proper fashion. Clocks are a kind of degenerate agent in that their percept sequence is empty; no matter what happens outside, the clock's action should be unaffected.

Well, this is not quite true. If the clock and its owner take a trip from California to Australia, the right thing for the clock to do would be to turn itself back six hours. We do not get upset at our clocks for failing to do this because we realize that they are acting rationally, given their lack of perceptual equipment.[3]

[3] One of the authors still gets a small thrill when his computer successfully resets itself at daylight savings time.

The ideal mapping from percept sequences to actions

Once we realize that an agent's behavior depends only on its percept sequence to date, then we can describe any particular agent by making a table of the action it takes in response to each possible percept sequence. (For most agents, this would be a very long list—infinite, in fact, unless we place a bound on the length of percept sequences we want to consider.) Such a list is called a **mapping** from percept sequences to actions. We can, in principle, find out which mapping correctly describes an agent by trying out all possible percept sequences and recording which actions the agent does in response. (If the agent uses some randomization in its computations, then we would have to try some percept sequences several times to get a good idea of the agent's average behavior.) And if mappings describe agents, then **ideal mappings** describe ideal agents. *Specifying which action an agent ought to take in response to any given percept sequence provides a design for an ideal agent.*

MAPPING

IDEAL MAPPINGS

This does not mean, of course, that we have to create an explicit table with an entry for every possible percept sequence. It is possible to define a specification of the mapping without exhaustively enumerating it. Consider a very simple agent: the square-root function on a calculator. The percept sequence for this agent is a sequence of keystrokes representing a number, and the action is to display a number on the display screen. The ideal mapping is that when the percept is a positive number x, the right action is to display a positive number z such that $z^2 \approx x$, accurate to, say, 15 decimal places. This specification of the ideal mapping does not require the designer to actually construct a table of square roots. Nor does the square-root function have to use a table to behave correctly: Figure 2.2 shows part of the ideal mapping and a simple program that implements the mapping using Newton's method.

The square-root example illustrates the relationship between the ideal mapping and an ideal agent design, for a very restricted task. Whereas the table is very large, the agent is a nice, compact program. It turns out that it is possible to design nice, compact agents that implement j

Percept x	Action z
1.0	1.000000000000000
1.1	1.048808848170152
1.2	1.095445115010332
1.3	1.140175425099138
1.4	1.183215956619923
1.5	1.224744871391589
1.6	1.264911064067352
1.7	1.303840481040530
1.8	1.341640786499874
1.9	1.378404875209022
⋮	⋮

```
function SQRT(x)
  z ← 1.0                    /* initial guess */
  repeat until |z² - x| < 10⁻¹⁵
        z ← z - (z² - x)/(2z)
  end
  return z
```

Figure 2.2 Part of the ideal mapping for the square-root problem (accurate to 15 digits), and a corresponding program that implements the ideal mapping.

the ideal mapping for much more general situations: agents that can solve a limitless variety of tasks in a limitless variety of environments. Before we discuss how to do this, we need to look at one more requirement that an intelligent agent ought to satisfy.

Autonomy

There is one more thing to deal with in the definition of an ideal rational agent: the "built-in knowledge" part. If the agent's actions are based completely on built-in knowledge, such that it need pay no attention to its percepts, then we say that the agent lacks **autonomy.** For example, if the clock manufacturer was prescient enough to know that the clock's owner would be going to Australia at some particular date, then a mechanism could be built in to adjust the hands automatically by six hours at just the right time. This would certainly be successful behavior, but the intelligence seems to belong to the clock's designer rather than to the clock itself.

AUTONOMY

An agent's behavior can be based on both its own experience and the built-in knowledge used in constructing the agent for the particular environment in which it operates. *A system is autonomous*[4] *to the extent that its behavior is determined by its own experience.* It would be too stringent, though, to require complete autonomy from the word go: 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.

Autonomy not only fits in with our intuition, but it is an example of sound engineering practices. An agent that operates on the basis of built-in assumptions will only operate successfully when those assumptions hold, and thus lacks flexibility. Consider, for example, 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. A truly autonomous intelligent agent should be able to operate successfully in a wide variety of environments, given sufficient time to adapt.

2.3 STRUCTURE OF INTELLIGENT AGENTS

So far we have talked about agents by describing their *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:** a function that implements the agent mapping from percepts to actions. We assume this program will run on some sort of computing device, which we will call the **architecture.** Obviously, the program we choose has

AGENT PROGRAM

ARCHITECTURE

[4] The word "autonomous" has also come to mean something like "not under the immediate control of a human," as in "autonomous land vehicle." We are using it in a stronger sense.

to be one that the architecture will accept and run. The architecture might be a plain computer, or it might include special-purpose hardware for certain tasks, such as processing camera images or filtering audio input. It might also include software that provides a degree of insulation between the raw computer and the agent program, so that we can program at a higher level. 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 effectors as they are generated. The relationship among agents, architectures, and programs can be summed up as follows:

agent = architecture + program

Most of this book is about designing agent programs, although Chapters 24 and 25 deal directly with the architecture.

Before we design an agent program, we must have a pretty good idea of the possible percepts and actions, what goals or performance measure the agent is supposed to achieve, and what sort of environment it will operate in.[5] These come in a wide variety. Figure 2.3 shows the basic elements for a selection of agent types.

It may come as a surprise to some readers that we include in our list of agent types some programs that seem to 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 goals that the agent is supposed to achieve. Some "real" environments are actually quite simple. For example, a robot designed to inspect parts as they come by on a conveyer 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 certain kind, and that there are only two actions—accept the part or mark it as a reject.

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 747. The simulator is a very detailed, complex environment, and the software agent must choose from a wide variety of actions in real time. Or imagine a softbot designed to scan online 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 dynamically change its plans when, for example, the connection for one news source crashes or a new one comes online.

Some environments blur the distinction between "real" and "artificial." In the ALIVE environment (Maes *et al.*, 1994), software agents are given as percepts a digitized camera image of a room where a human walks about. The agent processes the camera image and chooses an action. The environment also displays the camera image on a large display screen that the human can watch, and superimposes on the image a computer graphics rendering of the software agent. One such image is a cartoon dog, which has been programmed to move toward the human (unless he points to send the dog away) and to shake hands or jump up eagerly when the human makes certain gestures.

---

[5] For the acronymically minded, we call this the PAGE (Percepts, Actions, Goals, Environment) description. Note that the goals do *not* necessarily have to be represented within the agent; they simply describe the performance measure by which the agent design will be judged.

Agent Type	Percepts	Actions	Goals	Environment
Medical diagnosis system	Symptoms, findings, patient's answers	Questions, tests, treatments	Healthy patient, minimize costs	Patient, hospital
Satellite image analysis system	Pixels of varying intensity, color	Print a categorization of scene	Correct categorization	Images from orbiting satellite
Part-picking robot	Pixels of varying intensity	Pick up parts and sort into bins	Place parts in correct bins	Conveyor belt with parts
Refinery controller	Temperature, pressure readings	Open, close valves; adjust temperature	Maximize purity, yield, safety	Refinery
Interactive English tutor	Typed words	Print exercises, suggestions, corrections	Maximize student's score on test	Set of students

Figure 2.3 Examples of agent types and their PAGE descriptions.

The most famous artificial environment is the Turing Test environment, in which the whole point is that real and artificial agents are on equal footing, but the environment is challenging enough that it is very difficult for a software agent to do as well as a human. Section 2.4 describes in more detail the factors that make some environments more demanding than others.

Agent programs

We will be building intelligent agents throughout the book. They will all have the same skeleton, namely, accepting percepts from an environment and generating actions. The early versions of agent programs will have a very simple form (Figure 2.4). Each will use some internal data structures that will be updated as new percepts arrive. These data structures are operated on by the agent's decision-making procedures to generate an action choice, which is then passed to the architecture to be executed.

There are two things to note about this skeleton program. First, even though we defined the agent mapping as a function from percept *sequences* to actions, the agent program receives only a single percept as its input. It is up to the agent to build up the percept sequence in memory, if it so desires. In some environments, it is possible to be quite successful without storing the percept sequence, and in complex domains, it is infeasible to store the complete sequence.

```
function SKELETON-AGENT(percept) returns action
  static: memory, the agent's memory of the world

  memory ← UPDATE-MEMORY(memory, percept)
  action ← CHOOSE-BEST-ACTION(memory)
  memory ← UPDATE-MEMORY(memory, action)
  return action
```

Figure 2.4 A skeleton agent. On each invocation, the agent's memory is updated to reflect the new percept, the best action is chosen, and the fact that the action was taken is also stored in memory. The memory persists from one invocation to the next.

Second, the goal or performance measure is *not* part of the skeleton program. This is because the performance measure is applied externally to judge the behavior of the agent, and it is often possible to achieve high performance without explicit knowledge of the performance measure (see, e.g., the square-root agent).

Why not just look up the answers?

Let us start with the simplest possible way we can think of to write the agent program—a lookup table. Figure 2.5 shows the agent program. It operates by keeping in memory its entire percept sequence, and using it to index into *table*, which contains the appropriate action for all possible percept sequences.

It is instructive to consider why this proposal is doomed to failure:

1. The table needed for something as simple as an agent that can only play chess would be about 35^{100} entries.
2. It would take quite a long time for the designer to build the table.
3. The agent has no autonomy at all, because the calculation of best actions is entirely built-in. So if the environment changed in some unexpected way, the agent would be lost.

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

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

Figure 2.5 An agent based on a prespecified lookup table. It keeps track of the percept sequence and just looks up the best action.

4. Even if we gave the agent a learning mechanism as well, so that it could have a degree of autonomy, it would take forever to learn the right value for all the table entries.

Despite all this, TABLE-DRIVEN-AGENT *does* do what we want: it implements the desired agent mapping. It is not enough to say, "It can't be intelligent;" the point is to understand why an agent that *reasons* (as opposed to looking things up in a table) can do even better by avoiding the four drawbacks listed here.

An example

At this point, it will be helpful to consider a particular environment, so that our discussion can become more concrete. Mainly because of its familiarity, and because it involves a broad range of skills, we will look at the job of designing an automated taxi driver. We should point out, before the reader becomes alarmed, that such a system is currently somewhat beyond the capabilities of existing technology, although most of the components are available in some form.[6] The full driving task is extremely *open-ended*—there is no limit to the novel combinations of circumstances that can arise (which is another reason why we chose it as a focus for discussion).

We must first think about the percepts, actions, goals and environment for the taxi. They are summarized in Figure 2.6 and discussed in turn.

Agent Type	Percepts	Actions	Goals	Environment
Taxi driver	Cameras, speedometer, GPS, sonar, microphone	Steer, accelerate, brake, talk to passenger	Safe, fast, legal, comfortable trip, maximize profits	Roads, other traffic, pedestrians, customers

Figure 2.6 The taxi driver agent type.

The taxi will need to know where it is, what else is on the road, and how fast it is going. This information can be obtained from the **percepts** provided by one or more controllable TV cameras, the speedometer, and 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; or infrared or sonar sensors to detect distances to other cars and obstacles. Finally, it will need a microphone or keyboard for the passengers to tell it their destination.

The actions available to a taxi driver will be more or less the same ones available to a human driver: control over the engine through the gas pedal and control over steering and braking. In addition, it will need output to a screen or voice synthesizer to talk back to the passengers, and perhaps some way to communicate with other vehicles.

6 See page 26 for a description of an existing driving robot, or look at the conference proceedings on Intelligent Vehicle and Highway Systems (IVHS).

What **performance measure** would we like our automated driver to aspire to? 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 trade-offs involved.

Finally, were this a real project, we would need to decide what kind of driving **environment** the taxi will face. Should it operate on local roads, or also on freeways? Will it be in Southern California, where snow is seldom a problem, or in Alaska, where it seldom is not? Will it always be driving on the right, or might we want it to be flexible enough to drive on the left in case we want to operate taxis in Britain or Japan? Obviously, the more restricted the environment, the easier the design problem.

Now we have to decide how to build a real program to implement the mapping from percepts to action. We will find that different aspects of driving suggest different types of agent program. We will consider four types of agent program:

- Simple reflex agents
- Agents that keep track of the world
- Goal-based agents
- Utility-based agents

Simple reflex agents

The option of constructing an explicit lookup table is out of the question. The visual input from a single camera comes in at the rate of 50 megabytes per second (25 frames per second, 1000 x 1000 pixels with 8 bits of color and 8 bits of intensity information). So the lookup table for an hour would be $2^{60\times 60\times 50M}$ entries.

However, we can summarize portions of the table by noting certain commonly occurring input/output associations. For example, if the car in front brakes, and its brake lights come on, then the driver 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.

Figure 2.7 gives the structure of a simple reflex agent 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

7 Also called **situation-action rules, productions, or if–then rules.** The last term is also used by some authors for logical implications, so we will avoid it altogether.

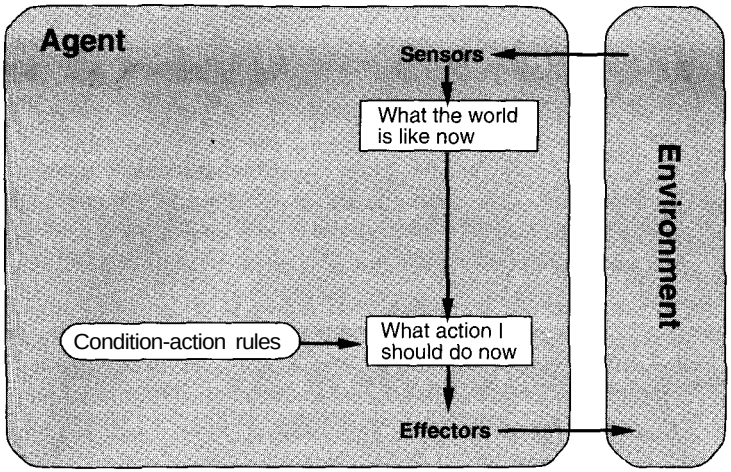


Figure 2.7 Schematic diagram of a simple reflex agent.

```
function SIMPLE-REFLEX-AGENT(percept) returns 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 2.8 A simple reflex agent. It works by finding a rule whose condition matches the current situation (as defined by the percept) and then doing the action associated with that rule.

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 2.8. 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. Although such agents can be implemented very efficiently (see Chapter 10), their range of applicability is very narrow, as we shall see.

Agents that keep track of the world

The simple reflex agent described before will work only if the correct decision can be made on the basis of the current percept. If the car in front is a recent model, and has the centrally mounted brake light now required in the United States, then it will be possible to tell if it is braking from a single image. Unfortunately, older models have different configurations of tail

lights, brake lights, and turn-signal lights, and it is not always possible to tell if the car is braking. Thus, even for the simple braking rule, our driver will have to maintain some sort of **internal state** in order to choose an action. Here, the internal state is not too extensive—it just needs the previous frame from the camera to detect when two red lights at the edge of the vehicle go on or off simultaneously.

INTERNAL STATE

Consider the following more obvious case: from time to time, the driver looks in the rear-view mirror to check on the locations of nearby vehicles. When the driver is not looking in the mirror, the vehicles in the next lane are invisible (i.e., the states in which they are present and absent are indistinguishable); but in order to decide on a lane-change maneuver, the driver needs to know whether or not they are there.

The problem illustrated by this example arises because the sensors do not provide access to the complete state of the world. In such cases, the agent may need to maintain some internal state information in order to distinguish between world states that generate the same perceptual input but nonetheless are significantly different. Here, "significantly different" means that different actions are appropriate in the two states.

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 changes lanes to the right, there is a gap (at least temporarily) in the lane it was in before, 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.

Figure 2.9 gives the structure of the reflex agent, showing how the current percept is combined with the old internal state to generate the updated description of the current state. The agent program is shown in Figure 2.10. 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 7 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, right, or go straight on. The right 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, which describes situations that are desirable—for example, being at the passenger's destination. The agent 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. Sometimes this will be simple, 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 5) and **planning** (Chapters 11 to 13) are the subfields of AI devoted to finding action sequences that do achieve the agent's goals.

GOAL

SEARCH

PLANNING

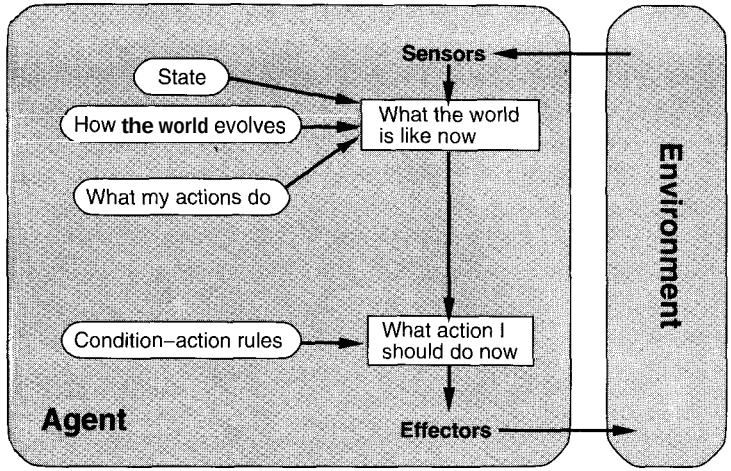


Figure 2.9 A reflex agent with internal state.

```
function REFLEX-AGENT-WITH-STATE(percept) returns action
  static: state, a description of the current world state
          rules, a set of condition-action rules

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

Figure 2.10 A reflex agent with internal state. It works by finding a rule whose condition matches the current situation (as defined by the percept and the stored internal state) and then doing the action associated with that rule.

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 used, because the designer has precomputed the correct action for various cases. 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. From 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 far more flexible. 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 a large number of condition-action

rules. Of course, the goal-based agent is also more flexible with respect to reaching different destinations. Simply by specifying a new destination, we can get the goal-based agent to come up with a new behavior. The reflex agent's rules for when to turn and when to go straight will only work for a single destination; they must all be replaced to go somewhere new.

Figure 2.11 shows the goal-based agent's structure. Chapter 13 contains detailed agent programs for goal-based agents.

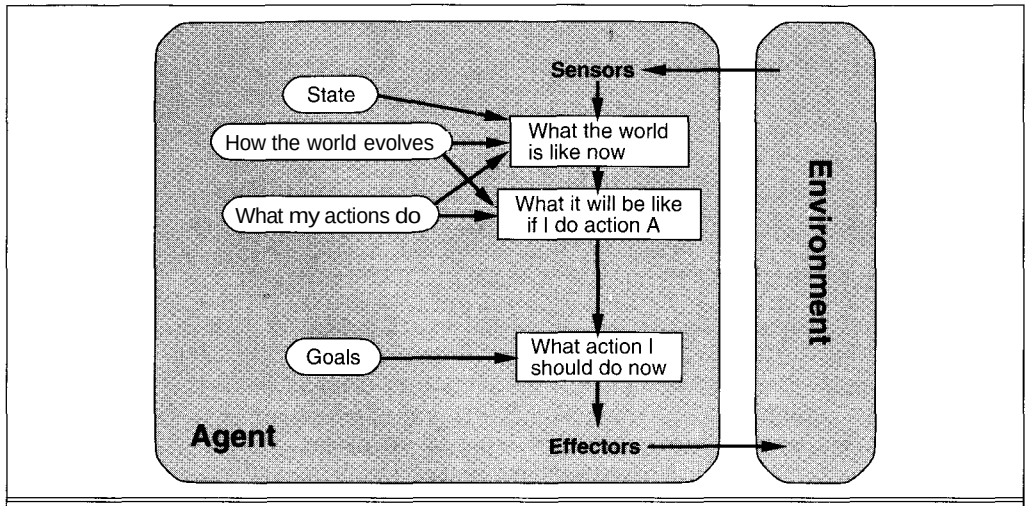


Figure 2.11 An agent with explicit goals.

Utility-based agents

Goals alone are not really enough to generate high-quality behavior. 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 distinction between "happy" and "unhappy" states, whereas a more general performance measure should allow a comparison of different world states (or sequences of 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 is therefore a function that maps a state[9] 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 have trouble. First, when there are conflicting goals, only some of which can be achieved (for example, speed and safety), the utility function specifies the appropriate trade-off. Second, when there are several goals that the agent can aim for, none

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

[9] Or sequence of states, if we are measuring the utility of an agent over the long run.

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 show that any rational agent can be described as possessing a utility function. An agent that possesses an *explicit* utility function therefore can make rational decisions, but may have to compare the utilities achieved by different courses of actions. Goals, although cruder, enable the agent to pick an action right away if it satisfies the goal. In some cases, moreover, a utility function can be translated into a set of goals, such that the decisions made by a goal-based agent using those goals are identical to those made by the utility-based agent.

The overall utility-based agent structure appears in Figure 2.12. Actual utility-based agent programs appeal in Chapter 5, where we examine game-playing programs that must make fine distinctions amoag various board positions; and in Chapter 17, where we tackle the general problem of designing decision-making agents.

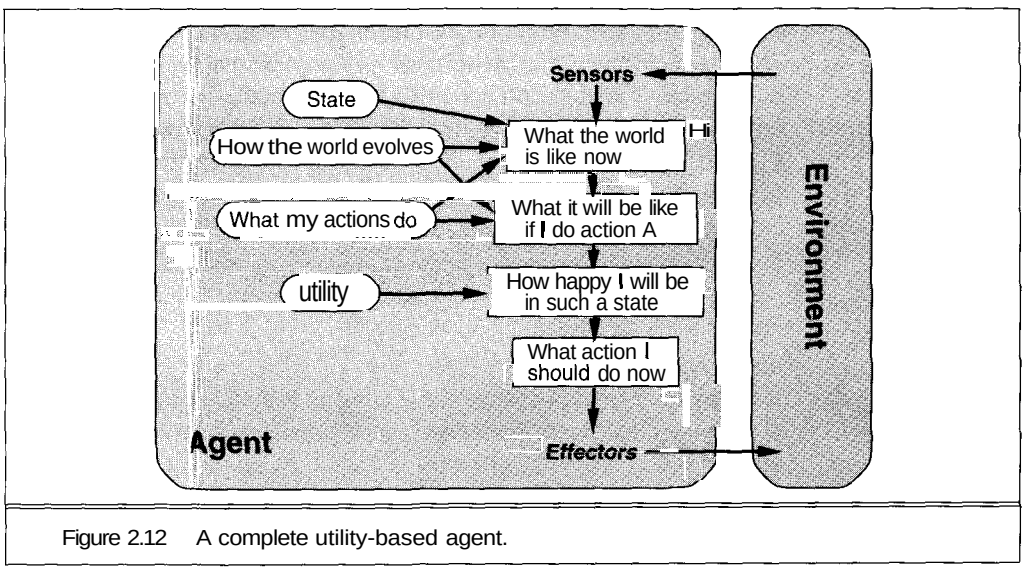


Figure 2.12 A complete utility-based agent.

2.4 ENVIRONMENTS

In this section and in the exercises at the end of the chapter, you will see how to couple an agent to an environment. Section 2.3 introduced several different kinds of agents and environments. In all cases, however, the nature of the connection between them is the same: actions are done by the agent on the environment, which in turn provides percepts to the agent. First, we will describe the different types of environments and how they affect the design of agents. Then we will describe environment programs that can be used as testbeds for agent programs.

Properties of environments

Environments come in several flavors. The principal distinctions to be made are as follows:

ACCESSIBLE

- **Accessible** vs. **inaccessible.**
  If an agent's sensory apparatus gives it access to the complete state of the environment, then we say that the environment is accessible to that agent. An environment is effectively accessible if the sensors detect all aspects that are relevant to the choice of action. An accessible environment is convenient because the agent need not maintain any internal state to keep track of the world.

DETERMINISTIC

- **Deterministic** vs. **nondeterministic**.
  If the next state of the environment is completely determined by the current state and the actions selected by the agents, then we say the environment is deterministic. In principle, an agent need not worry about uncertainty in an accessible, deterministic environment. If the environment is inaccessible, however, then it may *appear* to be nondeterministic. This is particularly true if the environment is complex, making it hard to keep track of all the inaccessible aspects. Thus, it is often better to think of an environment as deterministic or nondeterministic *from the point of view of the agent.*

EPISODIC

- **Episodic** vs. **nonepisodic.**
  In an episodic environment, the agent's experience is divided into "episodes." Each episode consists of the agent perceiving and then acting. The quality of its action depends just on the episode itself, because subsequent episodes do not depend on what actions occur in previous episodes. Episodic environments are much simpler because the agent does not need to think ahead.

STATIC

- **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. If the environment does not change with the passage of time but the agent's performance score does, then we say the environment is **semidynamic**.

SEMIDYNAMIC

DISCRETE

- **Discrete** vs. **continuous.**
  If there are a limited number of distinct, clearly defined percepts and actions we say that the environment is discrete. Chess is discrete—there are a fixed number of possible moves on each turn. Taxi driving is continuous—the speed and location of the taxi and the other vehicles sweep through a range of continuous values.[10]

We will see that different environment types require somewhat different agent programs to deal with them effectively. It will turn out, as you might expect, that the hardest case is *inaccessible, nonepisodic, dynamic,* and *continuous.* 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 nondeterministic.

[10] At a fine enough level of granularity, even the taxi driving environment is discrete, because the camera image is digitized to yield discrete pixel values. But any sensible agent program would have to abstract above this level, up to a level of granularity that is continuous.

Figure 2.13 lists the properties of a number of familiar environments. Note that the answers can change depending on how you conceptualize the environments and agents. For example, poker is deterministic if the agent can keep track of the order of cards in the deck, but it is nondeterministic if it cannot. 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 next game. On the other hand, moves within a single game certainly interact, so the agent needs to look ahead several moves.

Environment	Accessible	Deterministic	Episodic	Static	Discrete
Chess with a clock	Yes	Yes	No	Semi	Yes
Chess without a clock	Yes	Yes	No	Yes	Yes
Poker	No	No	No	Yes	Yes
Backgammon	Yes	No	No	Yes	Yes
Taxi driving	No	No	No	No	No
Medical diagnosis system	No	No	No	No	No
Image-analysis system	Yes	Yes	Yes	Semi	No
Part-picking robot	No	No	Yes	No	No
Refinery controller	No	No	No	No	No
Interactive English tutor	No	No	No	No	Yes

Figure 2.13 Examples of environments and their characteristics.

Environment programs

The generic environment program in Figure 2.14 illustrates the basic relationship between agents and environments. In this book, we will find it convenient for many of the examples and exercises to use an environment simulator that follows this program structure. The simulator takes one or more agents as input and arranges to repeatedly give each agent the right percepts and receive back an action. The simulator then updates the environment based on the actions, and possibly other dynamic processes in the environment that are not considered to be agents (rain, for example). The environment is therefore defined by the initial state and the update function. Of course, an agent that works in a simulator ought also to work in a real environment that provides the same kinds of percepts and accepts the same kinds of actions.

The RUN-ENVIRONMENT procedure correctly exercises the agents in an environment. For some kinds of agents, such as those that engage in natural language dialogue, it may be sufficient simply to observe their behavior. To get more detailed information about agent performance, we insert some performance measurement code. The function RUN-EVAL-ENVIRONMENT, shown in Figure 2.15, does this; it applies a performance measure to each agent and returns a list of the resulting scores. The *scores* variable keeps track of each agent's score.

In general, the performance measure can depend on the entire sequence of environment states generated during the operation of the program. Usually, however, the performance measure

```
procedure RUN-ENVIRONMENT(state, UPDATE-FN, agents, termination)
  inputs: state, the initial state of the environment
          UPDATE-FN, function to modify the environment
          agents, a set of agents
          termination, a predicate to test when we are done

  repeat
    for each agent in agents do
      PERCEPT[agent] ← GET-PERCEPT(agent,state)
    end
    for each agent in agents do
      ACTION[agent] ← PROGRAM[agent](PERCEPT[agent])
    end
    state ← UPDATE-FN(actions, agents, state)
  until termination(state)
```

Figure 2.14 The basic environment simulator program. It gives each agent its percept, gets an action from each agent, and then updates the environment.

```
function RUN-EVAL-ENVIRONMENT(state, UPDATE-FN, agents,
                              termination, PERFORMANCE-FN) returns scores
  local variables: scores, a vector the same size as agents, all 0

  repeat
    for each agent in agents do
      PERCEPT[agent] ← GET-PERCEPT(agent, state)
    end
    for each agent in agents do
      ACTION[agent] ← PROGRAM[agent](PERCEPT[agent])
    end
    state ← UPDATE-FN(actions, agents, state)
    scores ← PERFORMANCE-FN(scores, agents, state)
  until termination(state)
  return scores                                        /* change */
```

Figure 2.15 An environment simulator program that keeps track of the performance measure for each agent.

works by a simple accumulation using either summation, averaging, or taking a maximum. For i example, if the performance measure for a vacuum-cleaning agent is the total amount of dirt] cleaned in a shift, *scores* will just keep track of how much dirt has been cleaned up so far.

RUN-EVAL-ENVIRONMENT returns the performance measure for a a single environment, defined by a single initial state and a particular update function. Usually, an agent is designed to

ENVIRONMENT CLASS

work in an **environment class,** a whole set of different environments. For example, we design a chess program to play against any of a wide collection of human and machine opponents. If we designed it for a single opponent, we might be able to take advantage of specific weaknesses in that opponent, but that would not give us a good program for general play. Strictly speaking, in order to measure the performance of an agent, we need to have an environment generator that selects particular environments (with certain likelihoods) in which to run the agent. We are then interested in the agent's average performance over the environment class. This is fairly straightforward to implement for a simulated environment, and Exercises 2.5 to 2.11 take you through the entire development of an environment and the associated measurement process.

A possible confusion arises between the state variable in the environment simulator and the state variable in the agent itself (see REFLEX-AGENT-WITH-STATE). As a programmer implementing both the environment simulator and the agent, it is tempting to allow the agent to peek at the environment simulator's state variable. This temptation must be resisted at all costs! The agent's version of the state must be constructed from its percepts alone, without access to the complete state information.

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. We split an agent into an architecture and an agent program.
- **An ideal agent** is one that always takes the action that is expected to maximize its performance measure, given the percept sequence it has seen so far.
- An agent is **autonomous** to the extent that its action choices depend on its own experience, rather than on knowledge of the environment that has been built-in by the designer.
- **An agent program** maps from a percept to an action, while updating an internal state.
- There exists a variety of basic agent program designs, depending on 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 percepts, actions, goals, and environment.
- **Reflex agents** respond immediately to percepts, **goal-based agents** act so that they will achieve their goal(s), and **utility-based agents** try to maximize their own "happiness."
- The process of making decisions by reasoning with knowledge is central to AI and to successful agent design. This means that representing knowledge is important.
- Some environments are more demanding than others. Environments that are inaccessible, nondeterministic, nonepisodic, dynamic, and continuous are the most challenging.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

The analysis of rational agency as a mapping from percept sequences to actions probably stems ultimately from the effort to identify rational behavior in the realm of economics and other forms of reasoning under uncertainty (covered in later chapters) and from the efforts of psychological behaviorists such as Skinner (1953) 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. The philosopher Daniel Dennett (1969; 1978b) helped to synthesize these viewpoints into a coherent "intentional stance" toward agents. A high-level, abstract perspective on agency is also taken within the world of AI in (McCarthy and Hayes, 1969). Jon Doyle (1983) proposed that rational agent design is the core of AI, and would remain as its mission while other topics in AI would spin off to form new disciplines. Horvitz *et al.* (1988) specifically suggest the use of rationality conceived as the maximization of expected utility as a basis for AI.

The AI researcher and Nobel-prize-winning economist Herb Simon drew a clear distinction between rationality under resource limitations (procedural rationality) and rationality as making the objectively rational choice (substantive rationality) (Simon, 1958). Cherniak (1986) explores the minimal level of rationality needed to qualify an entity as an agent. Russell and Wefald (1991) deal explicitly with the possibility of using a variety of agent architectures. *Dung Beetle Ecology* (Hanski and Cambefort, 1991) provides a wealth of interesting information on the behavior of dung beetles.

EXERCISES

2.1 What is the difference between a performance measure and a utility function?

2.2 For each of the environments in Figure 2.3, determine what type of agent architecture is most appropriate (table lookup, simple reflex, goal-based or utility-based).

2.3 Choose a domain that you are familiar with, and write a PAGE description of an agent for the environment. Characterize the environment as being accessible, deterministic, episodic, static, and continuous or not. What agent architecture is best for this domain?

2.4 While driving, which is the best policy?

a. Always put your directional blinker on before turning,

b. Never use your blinker,

c. Look in your mirrors and use your blinker only if you observe a car that can observe you?

What kind of reasoning did you need to do to arrive at this policy (logical, goal-based, or utility-based)? What kind of agent design is necessary to carry out the policy (reflex, goal-based, or utility-based)?

The following exercises all concern the implementation of an environment and set of agents in the vacuum-cleaner world.

2.5 Implement a performance-measuring environment simulator for the vacuum-cleaner world. This world can be described as follows:

- ◇ **Percepts:** Each vacuum-cleaner agent gets a three-element percept vector on each turn. The first element, a touch sensor, should be a 1 if the machine has bumped into something and a 0 otherwise. The second comes from a photosensor under the machine, which emits a 1 if there is dirt there and a 0 otherwise. The third comes from an infrared sensor, which emits a 1 when the agent is in its home location, and a 0 otherwise.
- **0 Actions:** There are five actions available: go forward, turn right by 90°, turn left by 90°, suck up dirt, and turn off.
- ◇ **Goals:** The goal for each agent is to clean up and go home. To be precise, the performance measure will be 100 points for each piece of dirt vacuumed up, minus 1 point for each action taken, and minus 1000 points if it is not in the home location when it turns itself off.
- ◇ **Environment:** The environment consists of a grid of squares. Some squares contain obstacles (walls and furniture) and other squares are open space. Some of the open squares contain dirt. Each "go forward" action moves one square unless there is an obstacle in that square, in which case the agent stays where it is, but the touch sensor goes on. A "suck up dirt" action always cleans up the dirt. A "turn off" command ends the simulation.

We can vary the complexity of the environment along three dimensions:

- ◇ **Room shape:** In the simplest case, the room is an n x n square, for some fixed n. We can make it more difficult by changing to a rectangular, L-shaped, or irregularly shaped room, or a series of rooms connected by corridors.
- 0 **Furniture:** Placing furniture in the room makes it more complex than an empty room. To the vacuum-cleaning agent, a piece of furniture cannot be distinguished from a wall by perception; both appear as a 1 on the touch sensor.
- 0 **Dirt placement:** In the simplest case, dirt is distributed uniformly around the room. But it is more realistic for the dirt to predominate in certain locations, such as along a heavily travelled path to the next room, or in front of the couch.

2.6 Implement a table-lookup agent for the special case of the vacuum-cleaner world consisting of a 2 x 2 grid of open squares, in which at most two squares will contain dirt. The agent starts in the upper left corner, facing to the right. Recall that a table-lookup agent consists of a table of actions indexed by a percept sequence. In this environment, the agent can always complete its task in nine or fewer actions (four moves, three turns, and two suck-ups), so the table only needs entries for percept sequences up to length nine. At each turn, there are eight possible percept vectors, so the table will be of size $8^9 = 134,217,728$. Fortunately, we can cut this down by realizing that the touch sensor and home sensor inputs are not needed; we can arrange so that the agent never bumps into a wall and knows when it has returned home. Then there are only two relevant percept vectors, ?0? and ?1?, and the size of the table is at most $2^9 = 512$. Run the environment simulator on the table-lookup agent in all possible worlds (how many are there?). Record its performance score for each world and its overall average score.

2.7 Implement an environment for a $n \times m$ rectangular room, where each square has a 5% chance of containing dirt, and n and m are chosen at random from the range 8 to 15, inclusive.

2.8 Design and implement a pure reflex agent for the environment of Exercise 2.7, ignoring the requirement of returning home, and measure its performance. Explain why it is impossible to have a reflex agent that returns home and shuts itself off. Speculate on what the best possible reflex agent could do. What prevents a reflex agent from doing very well?

2.9 Design and implement several agents with internal state. Measure their performance. How close do they come to the ideal agent for this environment?

2.10 Calculate the size of the table for a table-lookup agent in the domain of Exercise 2.7. Explain your calculation. You need not fill in the entries for the table.

2.11 Experiment with changing the shape and dirt placement of the room, and with adding furniture. Measure your agents in these new environments. Discuss how their performance might be improved to handle more complex geographies.

Part II

PROBLEM-SOLVING

In this part we show how an agent can act by establishing *goals* and considering sequences of actions that might achieve those goals. A goal and a set of means for achieving the goal is called a *problem*, and the process of exploring what the means can do is called *search*. We show what search can do, how it must be modified to account for adversaries, and what its limitations are.

3 SOLVING PROBLEMS BY SEARCHING

In which we look at how an agent can decide what to do by systematically considering the outcomes of various sequences of actions that it might take.

In Chapter 2, we saw that simple reflex agents are unable to plan ahead. They are limited in what they can do because their actions are determined only by the current percept. Furthermore, they have no knowledge of what their actions do nor of what they are trying to achieve.

PROBLEM-SOLVING AGENT

In this chapter, we describe 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 discuss informally how the agent can formulate an appropriate view of the problem it faces. The problem type that results from the formulation process will depend on the knowledge available to the agent: principally, whether it knows the current state and the outcomes of actions. We then define more precisely the elements that constitute a "problem" and its "solution," and give several examples to illustrate these definitions. Given precise definitions of problems, it is relatively straightforward to construct a search process for finding solutions. We cover six different search strategies and show how they can be applied to a variety of problems. Chapter 4 will then cover search strategies that make use of more information about the problem to improve the efficiency of the search process.

This chapter uses concepts from the analysis of algorithms. Readers unfamiliar with the concepts of asymptotic complexity and NP-completeness should consult Appendix A.

3.1 PROBLEM-SOLVING AGENTS

Intelligent agents are supposed to act in such a way that the environment goes through a sequence of states that maximizes the performance measure. In its full generality, this specification is difficult to translate into a successful agent design. As we mentioned in Chapter 2, the task is somewhat simplified if the agent can adopt a **goal** and aim to satisfy it. Let us first look at how and why an agent might do this.

Imagine our agent in the city of Arad, Romania, toward the end of a touring holiday. The agent has a ticket to fly out of Bucharest the following day. The ticket is nonrefundable, the agent's visa is about to expire, and after tomorrow, there are no seats available for six weeks. Now the agent's performance measure contains many other factors besides the cost of the ticket and the undesirability of being arrested aqd deported. For example, it wants to improve its suntan, improve its Romanian, take in the sights, and so on. All these factors might suggest any of a vast array of possible actions. Given the seriousness of the situation, however, it should adopt the goal of driving to Bucharest. Actions that result in a failure to reach Bucharest on time can be rejected without further consideration. Goals such as this help organize behavior by limiting the objectives that the agent is trying to achieve. **Goal formulation,** based on the current situation, is the first step in problem solving. As well as formulating a goal, the agent may wish to decide on some other factors that affect the desirability of different ways of achieving the goal.

GOAL FORMULATION

For the purposes of this chapter, we will consider a goal to be a set of world states—just those states in which the goal is satisfied. Actions can be viewed as causing transitions between world states, so obviously the agent has to find out which 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 18 inches" or "turn the steering wheel six degrees left," it would never find its way out of the parking lot, let alone to Bucharest, because constructing a solution at that level of detail would be an intractable problem. **Problem formulation** is the process of deciding what actions and states to consider, and follows goal formulation. We will discuss this process in more detail. 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 which town to drive to 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 through each of the three towns, to try 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, then, 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 one. 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

SEARCH

SOLUTION

[1] Notice that these states actually correspond to large *sets* of world states, because a 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 as clueless as our agent. We apologize to Romanian readers who are unable to take advantage of this pedagogical device.

EXECUTION

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 the next thing to do, and then removing that step from the sequence. Once the solution has been executed, the agent will find a new goal.

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

  state ← UPDATE-STATE(state, p)
  if s is empty then
      g — FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, g)
      s ← SEARCH(problem)
  action — RECOMMENDATION(s, state)
  s ← REMAINDER(s, state)
  return action
```

Figure 3.1 A simple problem-solving agent.

We will not discuss the UPDATE-STATE and FORMULATE-GOAL functions further in this chapter. The next two sections describe the process of problem formulation, and then the remainder of the chapter is devoted to various versions of the SEARCH function. The execution phase is usually straightforward for a simple problem-solving agent: RECOMMENDATION just takes the first action in the sequence, and REMAINDER returns the rest.

3.2 FORMULATING PROBLEMS

In this section, we will consider the problem formulation process in more detail. First, we will look at the different amounts of knowledge that an agent can have concerning its actions and the state that it is in. This depends on how the agent is connected to its environment through its percepts and actions. We find that there are four essentially different types of problems—single-state problems, multiple-state problems, contingency problems, and exploration problems. We will define these types precisely, in preparation for later sections that address the solution process.

Knowledge and problem types

Let us consider an environment somewhat different from Romania: the vacuum world from Exercises 2.5 to 2.11 in Chapter 2. We will simplify it even further for the sake of exposition. Let the world contain just two locations. Each location may or may not contain dirt, and the agent may be in one location or the other. There are 8 possible world states, as shown in Figure 3.2. The agent has three possible actions in this version of the vacuum world: *Left*,*Right*, and *Suck*. Assume, for the moment, that sucking is 100% effective. The goal is to clean up all the dirt. That is, the goal is equivalent to the state set $\{7, 8\}$.

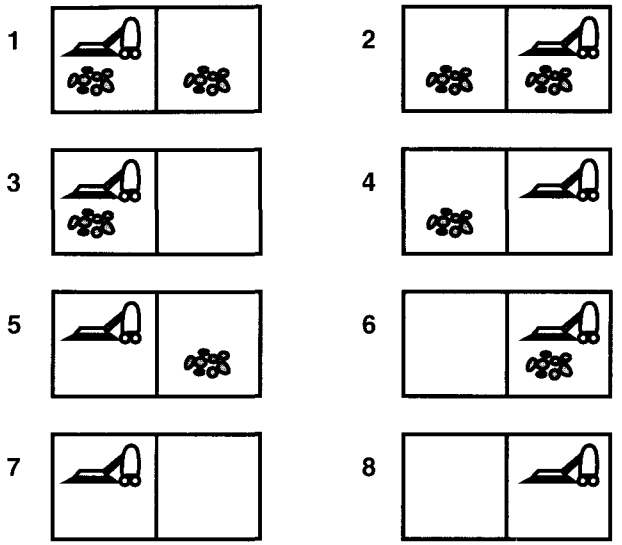


Figure 3.2 The eight possible states of the simplified vacuum world.

First, suppose that the agent's sensors give it enough information to tell exactly which state it is in (i.e., the world is accessible); and suppose that it knows exactly what each of its actions does. Then it can calculate exactly which state it will be in after any sequence of actions. For example, if its initial state is 5, then it can calculate that the action sequence [*Right*,*Suck*]will get to a goal state. This is the simplest case, which we call a **single-state problem.**

SINGLE-STATE PROBLEM

Second, suppose that the agent knows all the effects of its actions, but has limited access to the world state. For example, in the extreme case, it may have no sensors at all. In that case, 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 $\{2, 4, 6, 8\}$. In fact, the agent can discover that the action sequence [*Right*,*Suck*,*Left*,*Suck*] is guaranteed to reach a goal state no matter what the start state. To summarize: when the world is not fully accessible, the agent must reason about sets of states that it might get to, rather than single states. We call this a **multiple-state problem.**

MULTIPLE-STATE PROBLEM

Although it might seem different, the case of ignorance about the effects of actions can be treated similarly. Suppose, for example, that the environment appears to be nondeterministic in that it obeys Murphy's Law: the so-called *Suck* action *sometimes* deposits dirt on the carpet *but only if there is no dirt there already*.[3] For example, if the agent knows it is in state 4, then it knows that if it sucks, it will reach one of the states $\{2, 4\}$. For any *known* initial state, however, there is an action sequence that is guaranteed to reach a goal state (see Exercise 3.2).

Sometimes ignorance prevents the agent from finding a guaranteed solution sequence. Suppose, for example, 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. Suppose further that the sensors tell it that it 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 $\{6, 8\}$. If it is in fact state 6, then the action sequence will succeed, but if it is state 8, the plan will fail. If the agent had chosen the simpler action sequence *[Suck]*, it would also succeed some of the time, but not always. It turns out there is no fixed action sequence that guarantees a solution to this problem.

Obviously, the agent *does* have a way to solve the problem starting from one of $\{1, 3\}$: first suck, then move right, then suck *only if there is dirt there.* Thus, solving this problem requires sensing *during the execution phase.* Notice that the agent must now calculate a whole tree of actions, rather than a single action sequence. In general, each branch of the tree deals with a possible contingency that might arise. For this reason, we call this a **contingency problem.** 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 PROBLEM

Single-state and multiple-state problems can be handled by similar search techniques, which are covered in this chapter and the next. Contingency problems, on the other hand, require more complex algorithms, which we cover in Chapter 13. They 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 possible contingency that might arise during execution, it is often better to actually start executing and see which contingencies *do* arise. The agent can then continue to solve the problem given the additional information. This type of **interleaving** of search and execution is also covered in Chapter 13, and for the limited case of two-player games, in Chapter 5. For the remainder of this chapter, we will only consider cases where guaranteed solutions consist of a single sequence of actions.

INTERLEAVING

Finally, consider the plight of an agent that has no information about the effects of its actions. This is somewhat equivalent to being lost in a strange country with no map at all, and is the hardest task faced by an intelligent agent.[4] The agent must *experiment,* gradually discovering what its actions do and what sorts of states exist. This is a kind of search, but a search in the real world rather than in a model thereof. Taking a step in the real world, rather than in a model, may involve significant danger for an ignorant agent. If it survives, the agent learns a "map" of the environment, which it can then use to solve subsequent problems. We discuss this kind of **exploration problem** in Chapter 20.

EXPLORATION PROBLEM

[3] We assume that most readers face similar problems, and can imagine themselves as frustrated as our agent. We apologize to owners of modern, efficient home appliances who cannot take advantage of this pedagogical device.

[4] It is also the task faced by newborn babies.

Well-defined problems and solutions

PROBLEM

A problem is really a collection of information that the agent will use to decide what to do. We will begin by specifying the information needed to define a single-state problem.

We have seen that the basic elements of a problem definition are the states and actions. To capture these formally, we need the following:

INITIAL STATE

- The **initial state** that the agent knows itself to be in.

OPERATOR

- The set of possible actions available to the agent. The term **operator** is used to denote the description of an action in terms of which state will be reached by carrying out the action in a particular state. (An alternate formulation uses a **successor function** *S*. Given a particular state *x*, *S(x)* returns the set of states reachable from *x* by any single action.)

SUCCESSOR FUNCTION

STATE SPACE

PATH

Together, these define the **state space** of the problem: the set of all states reachable from the initial state by any sequence of actions. A **path** in the state space is simply any sequence of actions leading from one state to another. The next element of a problem is the following:

GOAL TEST

- **The goal test,** which the agent can apply to a single state description to determine if it is a goal state. Sometimes there is an explicit set of possible goal states, and the test simply checks to see if we have reached one of them. 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 can be captured on the next move no matter what the opponent does.

Finally, it may be the case that one solution is preferable to another, even though they both reach the goal. For example, we might prefer paths with fewer or less costly actions.

PATH COST

- **A path cost** function is a function that assigns a cost to a path. In all cases we will consider, the cost of a path is the sum of the costs of the individual actions along the path. The path cost function is often denoted by *g*.

Together, the initial state, operator set, goal test, and path cost function define a problem. Naturally, we can then define a datatype with which to represent problems:

datatype PROBLEM
components: INITIAL-STATE, OPERATORS, GOAL-TEST, PATH-COST-FUNCTION

Instances of this datatype will be the input to our search algorithms. The output of a search algorithm is a **solution,** that is, a path from the initial state to a state that satisfies the goal test.

SOLUTION

To deal with multiple-state problems, we need to make only minor modifications: a problem consists of an initial state *set*; a set of operators specifying for each action the *set* of states reached from any given state; and a goal test and path cost function as before. An operator is applied to a state set by unioning the results of applying the operator to each state in the set. A path now connects *sets of* states, and a solution is now a path that leads to a set of states *all of which* are goal states. The state space is replaced by the **state set space** (see Figure 3.7 for an example). Problems of both types are illustrated in Section 3.3.

STATE SET SPACE

Measuring problem-solving performance

SEARCH COST
TOTAL COST

The effectiveness of a search can be measured in at least three ways. First, does it rind a solution at all? Second, is it a good solution (one with a low path cost)? Third, what is the **search cost** associated with the time and memory required to find a solution? The **total cost** of the search is the sum of the path cost and the search cost.[5]

For the problem of finding a route from Arad to Bucharest, the path cost might be proportional to the total mileage of the path, perhaps with something thrown in for wear and tear on different road surfaces. The search cost will depend on the circumstances. In a static environment, it will be zero because the performance measure is independent of time. If there is some urgency to get to Bucharest, the environment is semidynamic because deliberating longer will cost more. In this case, the search cost might vary approximately linearly with computation time (at least for small amounts of time). Thus, to compute the total cost, it would appear that we have to add miles and milliseconds. This is not always easy, because there is no "official exchange rate" between the two. The agent must somehow decide what resources to devote to search and what resources to devote to execution. For problems with very small state spaces, it is easy to find the solution with the lowest path cost. But for large, complicated problems, there is a trade-off to be made—the agent can search for a very long time to get an optimal solution, or the agent can search for a shorter time and get a solution with a slightly larger path cost. The issue of allocating resources will be taken up again in Chapter 16; for now, we concentrate on the search itself.

Choosing states and actions

Now that we have the definitions out of the way, let us start our investigation of problems with an easy one: "Drive from Arad to Bucharest using the roads in the map in Figure 3.3." An appropriate state space has 20 states, where each state is defined solely by location, specified as a city. Thus, the initial state is "in Arad" and the goal test is "is this Bucharest?" The operators correspond to driving along the roads between cities.

One solution is the path Arad to Sibiu to Rimnicu Vilcea to Pitesti to Bucharest. There are lots of other paths that are also solutions, for example, via Lugoj and Craiova. To decide which of these solutions is better, we need to know what the path cost function is measuring: it could be the total mileage, or the expected travel time. Because our current map does not specify either of these, we will use the number of steps as the cost function. That means that the path through Sibiu and Fagaras, with a path cost of 3, is the best possible solution.

The real art of problem solving is in deciding what goes into the description of the states and operators and what is left out. 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 travelling companions, what is on the radio, what there is to look at out of the window, the vehicle being used for the trip, how fast it is going, whether there are any law enforcement officers nearby, what time it is, whether the driver is hungry or tired or running out of gas, how far it is to the next

[5] In theoretical computer science and in robotics, the search cost (the part you do before interacting with the environment) is called the **offline** cost and the path cost is called the **online** cost.

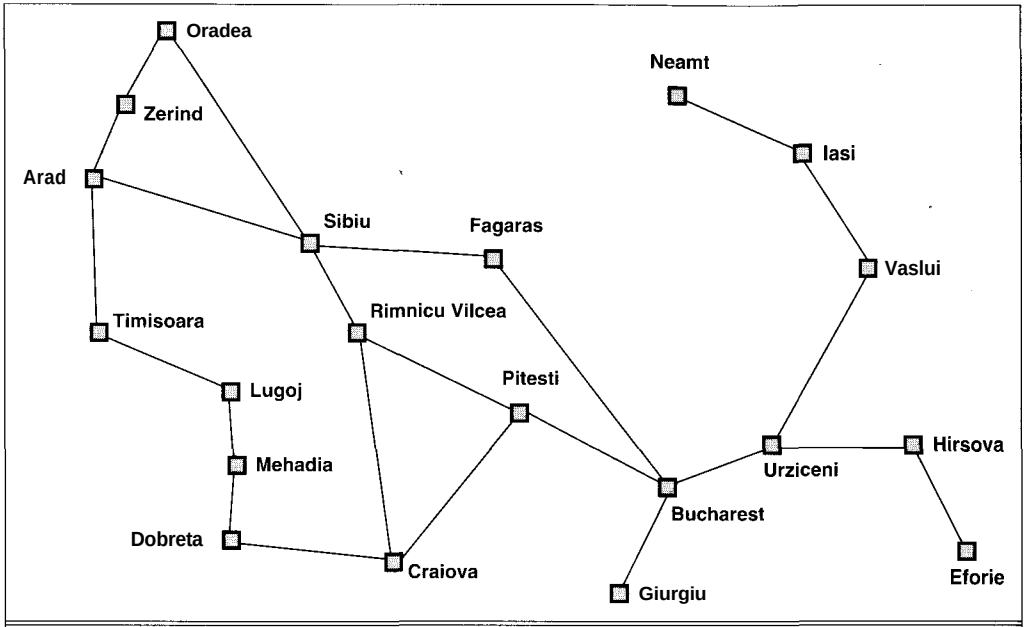


Figure 3.3 A simplified road map of Romania.

rest stop, the condition of the road, the weather, and so on. All these considerations are left out of state descriptions because they are irrelevant to the problem of finding a route to Bucharest.

ABSTRACTION

The process of removing detail from a representation is called **abstraction.**

As well as abstracting the state description, we must abstract the actions themselves. An action—let us say a car trip from Arad to Zerind—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.

Can we be more precise about defining the appropriate level of abstraction? Think of the states and actions we have chosen as corresponding to sets of detailed world states and sets of detailed action sequences. Now consider a solution to the abstract problem: for example, the path Arad to Sibiu to Rimnicu Vilcea to Pitesti to Bucharest. This 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. Each of these more detailed paths is still a solution to the goal, so the abstraction is valid. The abstraction is also useful, because carrying out each of the actions in the solution, such as driving from Pitesti to Bucharest, is somewhat easier than the original problem. 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.

3.3 EXAMPLE PROBLEMS

TOY PROBLEMS
REAL-WORLD PROBLEMS

The range of task environments that can be characterized by well-defined problems is vast. We can distinguish between so-called, toy **problems,** which are intended to illustrate or exercise various problem-solving methods, and so-called **real-world problems,** which tend to be more difficult and whose solutions people actually care about. In this section, we will give examples of both. By nature, toy problems can be given a concise, exact description. This means that they can be easily used by different researchers to compare the performance of algorithms. Real-world problems, on the other hand, tend not to have a single agreed-upon description, but we will attempt to give the general flavor of their formulations.

Toy problems

The 8-puzzle

8-PUZZLE

The **8-puzzle**, an instance of which is shown in Figure 3.4, consists of a 3x3 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 the configuration shown on the right of the figure. One important trick is to notice that rather than use operators such as "move the 3 tile into the blank space," it is more sensible to have operators such as "the blank space changes places with the tile to its left." This is because there are fewer of the latter kind of operator. This leads us to the following formulation:

- ◇ States: a state description specifies the location of each of the eight tiles in one of the nine squares. For efficiency, it is useful to include the location of the blank.
- ◇ **Operators:** blank moves left, right, up, or down.
- 0 **Goal test:** state matches the goal configuration shown in Figure 3.4.
- 0 **Path cost:** each step costs 1, so the path cost is just the length of the path.

SLIDING-BLOCK PUZZLES

The 8-puzzle belongs to the family **of sliding-block puzzles.** This general class is known to be NP-complete, so one does not expect to find methods significantly better than the search

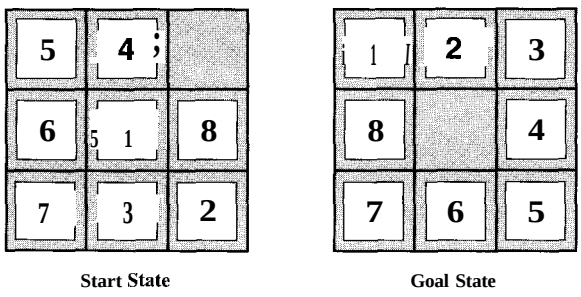


Figure 3.4 A typical instance of the 8-puzzle.

algorithms described in this chapter and the next. The 8-puzzle and its larger cousin, the 15-puzzle, are the standard test problems for new search algorithms in AI.

The 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 top left.

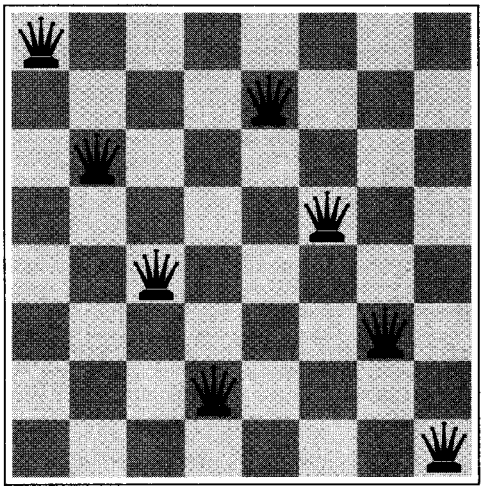


Figure 3.5 Almost a solution to the 8-queens problem. (Solution is left as an exercise.)

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. The *incremental* formulation involves placing queens one by one, whereas the *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; algorithms are thus compared only on search cost. Thus, we have the following goal test and path cost:

- **Goal** test: 8 queens on board, none attacked.
- **Path** cost: zero.

There are also different possible states and operators. Consider the following simple-minded formulation:

- **States:** any arrangement of 0 to 8 queens on board.
- **Operators:** add a queen to any square.

In this formulation, we have 64^8 possible sequences to investigate. A more sensible choice would use the fact that placing a queen where it is already attacked cannot work, because subsequent placings of other queens will not undo the attack. So we might try the following:

- **States:** arrangements of 0 to 8 queens with none attacked.
- **Operators:** place a queen in the left-most empty column such that it is not attacked by any other queen.

It is easy to see that the actions given can generate only states with no attacks; but sometimes no actions will be possible. For example, after making the first seven choices (left-to-right) in Figure 3.5, there is no action available in this formulation. The search process must try another choice. A quick calculation shows that there are only 2057 possible sequences to investigate. *The right formulation makes a big difference to the size of the search space.* Similar considerations apply for a complete-state formulation. For example, we could set the problem up as follows:

- **States:** arrangements of 8 queens, one in each column.
- **Operators:** move any attacked queen to another square in the same column.

This formulation would allow the algorithm to find a solution eventually, but it would be better to move to an unattacked square if possible.

Cryptarithmetic

In cryptarithmetic problems, letters stand for digits and the aim is to find a substitution of digits for letters such that the resulting sum is arithmetically correct. Usually, each letter must stand for a different digit. The following is a well-known example:

```
 FORTY      Solution:   29786      F=2, 0=9, R=7, etc.
+  TEN                    850
+  TEN                    850
 -----                  -----
 SIXTY                  31486
```

The following formulation is probably the simplest:

- **States:** a cryptarithmetic puzzle with some letters replaced by digits.
- **Operators:** replace all occurrences of a letter with a digit not already appearing in the puzzle.
- **Goal test:** puzzle contains only digits, and represents a correct sum.
- **Path cost:** zero. All solutions equally valid.

A moment's thought shows that replacing E by 6 then F by 7 is the same thing as replacing F by 7 then E by 6—order does not matter to correctness, so we want to avoid trying permutations of the same substitutions. One way to do this is to adopt a fixed order, e.g., alphabetical order. A better choice is to do whichever is the most *constrained* substitution, that is, the letter that has the fewest legal possibilities given the constraints of the puzzle.

The vacuum world

Here we will define the simplified vacuum world from Figure 3.2, rather than the full version from Chapter 2. The latter is dealt with in Exercise 3.17.

First, let us review the single-state case with complete information. We assume that the agent knows its location and the locations of all the pieces of dirt, and that the suction is still in good working order.

- ◇ **States:** one of the eight states shown in Figure 3.2 (or Figure 3.6).
- ◇ **Operators:** move left, move right, suck.
- ◇ **Goal test:** no dirt left in any square.
- ◇ **Path cost:** each action costs 1.

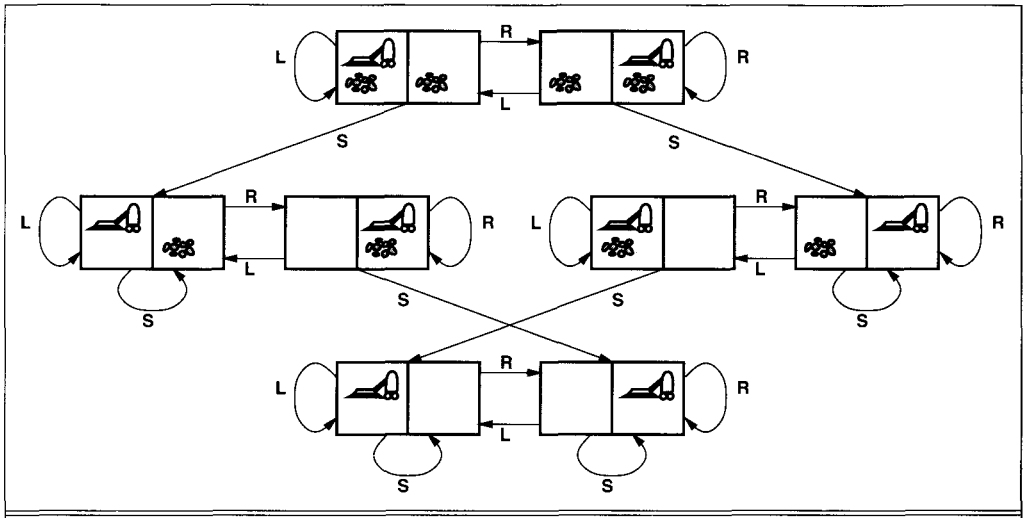


Figure 3.6 Diagram of the simplified vacuum state space. Arcs denote actions. L = move left, R = move right, S = suck.

Figure 3.6 shows the complete state space showing all the possible paths. Solving the problem from any starting state is simply a matter of following arrows to a goal state. This is the case for all problems, of course, but in most, the state space is vastly larger and more tangled.

Now let us consider the case where the agent has no sensors, but still has to clean up all the dirt. Because this is a multiple-state problem, we will have the following:

- **0 State** sets: subsets of states 1-8 shown in Figure 3.2 (or Figure 3.6).
- ◇ **Operators:** move left, move right, suck.
- 0 **Goal test:** all states in state set have no dirt.
- ◇ **Path cost:** each action costs 1.

The start state set is the set of all states, because the agent has no sensors. A solution is any sequence leading from the start state set to a set of states with no dirt (see Figure 3.7). Similar state set spaces can be constructed for the case of uncertainty about actions and uncertainty about both states and actions.

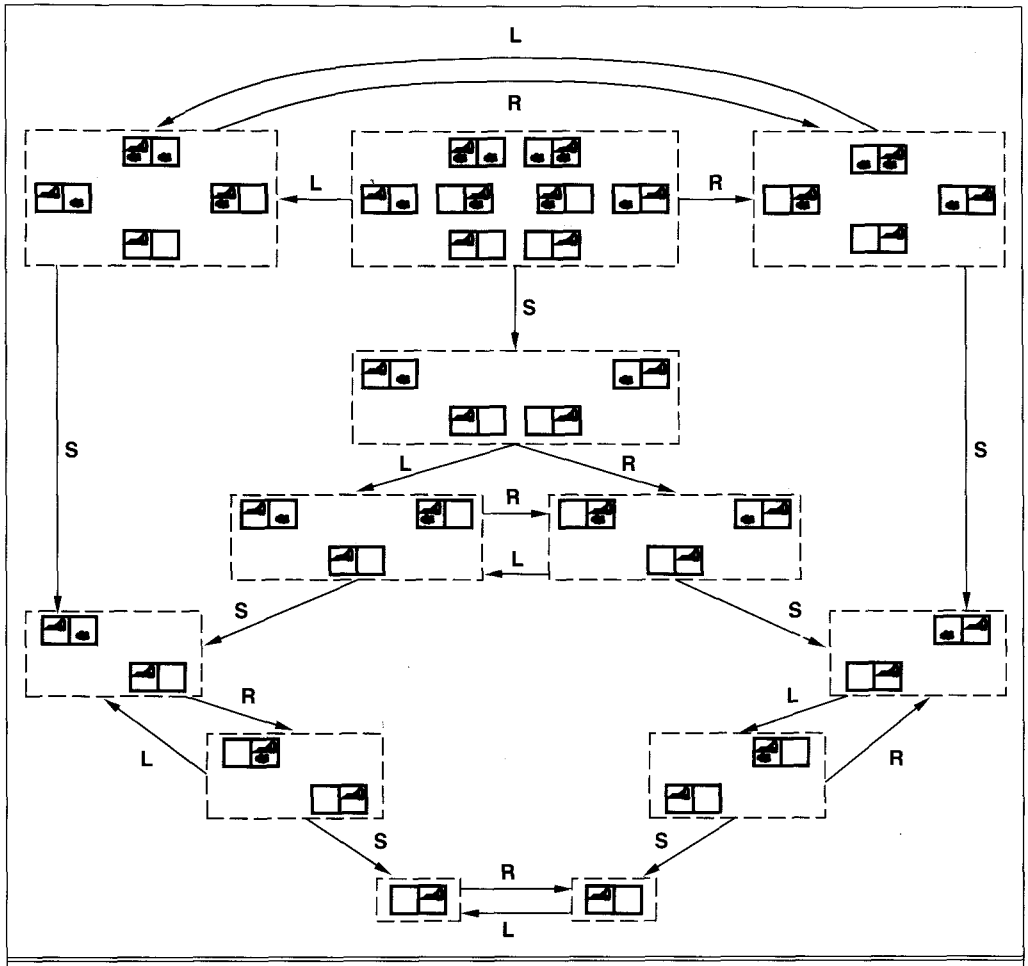


Figure 3.7 State set space for the simplified vacuum world with no sensors. Each dashed-line box encloses a set of states. At any given point, the agent is within a state set but does not know which state of that set it is in. The initial state set (complete ignorance) is the top center box. Actions are represented by labelled arcs. Self-loops are omitted for clarity.

Missionaries and cannibals

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). As with travelling in Romania, the real-life problem must be greatly abstracted before we can apply a problem-solving strategy.

Imagine the scene in real life: three members of the Arawaskan tribe, Alice, Bob, and Charles, stand at the edge of the crocodile-infested Amazon river with their new-found friends, Xavier, Yolanda, and Zelda. All around them birds cry, a rain storm beats down, Tarzan yodels, and so on. The missionaries Xavier, Yolanda, and Zelda are a little worried about what might happen if one of them were caught alone with two or three of the others, and Alice, Bob, and Charles are concerned that they might be in for a long sermon that they might find equally unpleasant. Both parties are not quite sure if the small boat they find tied up by the side of the river is up to making the crossing with two aboard.

To formalize the problem, the first step is to forget about the rain, the crocodiles, and all the other details that have no bearing in the solution. The next step is to decide what the right operator set is. We know that the operators will involve taking one or two people across the river in the boat, but we have to decide if we need a state to represent the time when they are in the boat, or just when they get to the other side. Because the boat holds only two people, no "outnumbering" can occur in it; hence, only the endpoints of the crossing are important. Next, we need to abstract over the individuals. Surely, each of the six is a unique human being, but for the purposes of the solution, when it comes time for a cannibal to get into the boat, it does not matter if it is Alice, Bob, or Charles. Any permutation of the three missionaries or the three cannibals leads to the same outcome. These considerations lead to the following formal definition of the problem:

- **States:** a state consists of an ordered sequence of three numbers representing the number of missionaries, cannibals, and boats on the bank of the river from which they started. Thus, the start state is (3,3,1).
- **Operators:** from each state the possible operators are to take either one missionary, one cannibal, two missionaries, two cannibals, or one of each across in the boat. Thus, there are at most five operators, although most states have fewer because it is necessary to avoid illegal states. Note that if we had chosen to distinguish between individual people then there would be 27 operators instead of just 5.
- **Goal test:** reached state (0,0,0).
- **Path cost:** number of crossings.

This state space is small enough to make it a trivial problem for a computer to solve. People have a hard time, however, because some of the necessary moves appear retrograde. Presumably, humans use some notion of "progress" to guide their search. We will see how such notions are used in the next chapter.

Real-world problems

Route finding

We have already seen how route finding 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, automated travel advisory systems, and airline travel planning systems. The last application is somewhat more complicated, because airline travel has a very complex path cost, in terms of money, seat quality, time of day, type of airplane, frequent-flyer

mileage awards, and so on. Furthermore, the actions in the problem do not have completely known outcomes: flights can be late or overbooked, connections can be missed, and fog or emergency maintenance can cause delays.

Touring and travelling salesperson problems

Consider the problem, "Visit every city in Figure 3.3 at least once, starting and ending in Bucharest." This seems very similar to route finding, because the operators still correspond to trips between adjacent cities. But for this problem, the state space must record more information. In addition to the agent's location, each state must keep track of 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 if the agent is in Bucharest and that all 20 cities have been visited.

TRAVELLING SALESPERSON PROBLEM

The **travelling salesperson problem** (TSP) is a famous touring problem in which each city must be visited exactly once. The aim is to find the *shortest* tour.[6] The problem is NP-hard (Karp, 1972), but an enormous amount of effort has been expended to improve the capabilities of TSP algorithms. In addition to planning trips for travelling salespersons, these algorithms have been used for tasks such as planning movements of automatic circuit board drills.

VLSI layout

The design of silicon chips is one of the most complex engineering design tasks currently undertaken, and we can give only a brief sketch here. A typical VLSI chip can have as many as a million gates, and the positioning and connections of every gate are crucial to the successful operation of the chip. Computer-aided design tools are used in every phase of the process. Two of the most difficult tasks are **cell layout** and **channel routing.** These come after the components and connections of the circuit have been fixed; the purpose is to lay out the circuit on the chip so as to minimize area and connection lengths, thereby maximizing speed. 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 using 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 simple, circular robot moving on a flat surface, the space

[6] Strictly speaking, this is the travelling salesperson optimization problem; the TSP itself asks if a tour exists with cost less than some constant.

is essentially two-dimensional. When the robot has arms and legs 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.

Assembly sequencing

Automatic assembly of complex objects by a robot was first demonstrated by FREDDY the robot (Michie, 1972). Progress since then has been slow but sure, to the point where assembly of objects such as electric motors is economically feasible. In assembly problems, the problem 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 complex geometrical search problem closely related to robot navigation. Thus, the generation of legal successors is the expensive part of assembly sequencing, and the use of informed algorithms to reduce search is essential.

3.4 SEARCHING FOR SOLUTIONS

We have seen how to define a problem, and how to recognize a solution. The remaining part—finding a solution—is done by a search through the state space. The idea is to maintain and extend a set of partial solution sequences. In this section, we show how to generate these sequences and how to keep track of them using suitable data structures.

Generating action sequences

To solve the route-finding problem from Arad to Bucharest, for example, we start off with just the initial state, Arad. The first step is to test if 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 applying the operators to the current state, thereby **generating** a new set of states. The process is called **expanding the** state. In this case, we get three new states, "in Sibiu," "in Timisoara," and "in Zerind," because there is a direct one-step route from Arad to these three cities. If there were only one possibility, we would just take it and continue. But whenever there are multiple possibilities, we must make a choice about which one to consider further.

GENERATING
EXPANDING

This is the essence of search—choosing one option and putting the others aside for later, in case the first choice does not lead to a solution. Suppose we choose Zerind. We check to see if it is a goal state (it is not), and then expand it to get "in Arad" and "in Oradea." We can then choose any of these two, or go back and choose Sibiu or Timisoara. We continue choosing, testing, and expanding until a solution is found, or until there are no more states to be expanded. The choice of which state to expand first is determined by the **search strategy.**

SEARCH STRATEGY

SEARCH TREE
SEARCH NODE

It is helpful to think of the search process as building up a **search tree** that is superimposed over the state space. The root of the search tree is a **search node** corresponding to the initial state. The leaf nodes of the tree correspond to states that do not have successors in the tree, either because they have not been expanded yet, or because they were expanded, but generated the empty set. At each step, the search algorithm chooses one leaf node to expand. Figure 3.8 shows some of the expansions in the search tree for route finding from Arad to Bucharest. The general search algorithm is described informally in Figure 3.9.

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

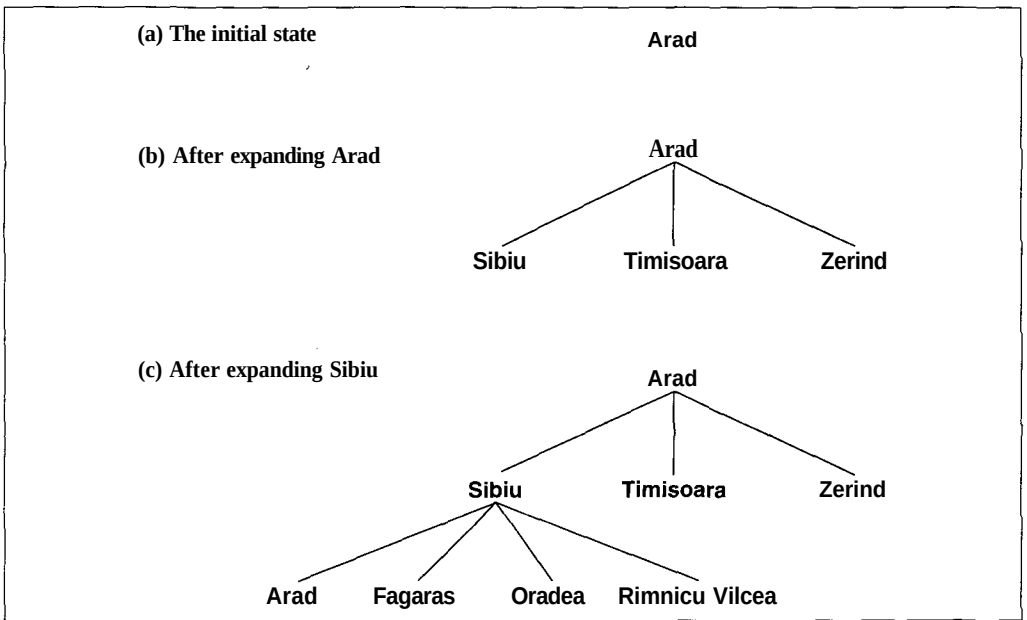


Figure 3.8 Partial search tree for route finding from Arad to Bucharest.

```
function GENERAL-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
  end
```

Figure 3.9 An informal description of the general search algorithm.

infinite number of paths in this state space, so the search tree has an infinite number of nodes. For example, in Figure 3.8, the branch Arad–Sibiu–Arad continues Arad–Sibiu–Arad–Sibiu–Arad, and so on, indefinitely. Obviously, a good search algorithm avoids following such paths. Techniques for doing this are discussed in Section 3.6.

Data structures for search trees

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

- the state in the state space to which the node corresponds;
- PARENT NODE the node in the search tree that generated this node (this is called the **parent node);**
- the operator that was applied to generate the node;
- DEPTH the number of nodes on the path from the root to this node (the **depth** of the node);
- the path cost of the path from the initial state to the node.

The node data type is thus:

datatype node
components: STATE, PARENT-NODE, OPERATOR, DEPTH, PATH-COST

It is important to remember the distinction between nodes and states. A node is a bookkeeping data structure used to represent the search tree for a particular problem instance as generated by a particular algorithm. A state represents a configuration (or set of configurations) of the world. Thus, nodes have depths and parents, whereas states do not. (Furthermore, it is quite possible for two different nodes to contain the same state, if that state is generated via two different sequences of actions.) The EXPAND function is responsible for calculating each of the components of the nodes it generates.

We also need to represent the collection of nodes that are waiting to be expanded—this FRINGE collection is called the **fringe** or **frontier.** The simplest representation would be a set of nodes. FRONTIER 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 QUEUE one. Therefore, we will assume that the collection of nodes is a implemented as a **queue.** The operations on a queue are as follows:

- MAKE-QUEUE(*Elements*) creates a queue with the given elements.
- EMPTY?(*Queue*) returns true only if there are no more elements in the queue.
- REMOVE-FRONT(*Queue*) removes the element at the front of the queue and returns it.
- QUEUING-FN(*Elements*,*Queue*) inserts a set of elements into the queue. Different varieties of the queuing function produce different varieties of the search algorithm.

With these definitions, we can write a more formal version of the general search algorithm. This is shown in Figure 3.10.

```
function GENERAL-SEARCH( problem, QUEUING-FN) returns a solution, or failure

  nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
  loop do
     if nodes is empty then return failure
     node ← REMOVE-FRONT(nodes)
     if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
     nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
  end
```

Figure 3.10 The general search algorithm. (Note that QUEUING-FN is a variable whose value will be a function.)

3.5 SEARCH STRATEGIES

The majority of work in the area of search has gone into finding the right **search strategy** for a problem. In our study of the field we will evaluate strategies in terms of four criteria:

COMPLETENESS
- ◇ **Completeness:** is the strategy guaranteed to find a solution when there is one?

TIME COMPLEXITY
- ◇ **Time complexity:** how long does it take to find a solution?

SPACE COMPLEXITY
- ◇ **Space complexity:** how much memory does it need to perform the search?

OPTIMALITY
- ◇ **Optimality**: does the strategy find the highest-quality solution when there are several different solutions?[7]

UNINFORMED SEARCH

BLIND SEARCH

This section covers six search strategies that come under the heading of **uninformed search.** The term means that they have no information about the number of steps or the path cost from the current state to the goal—all they can do is distinguish a goal state from a nongoal state. Uninformed search is also sometimes called **blind search.**

Consider again the route-finding problem. From the initial state in Arad, there are three actions leading to three new states: Sibiu, Timisoara, and Zerind. An uninformed search has no preference among these, but a more clever agent might notice that the goal, Bucharest, is southeast of Arad, and that only Sibiu is in that direction, so it is likely to be the best choice.

INFORMED SEARCH

HEURISTIC SEARCH

Strategies that use such considerations are called **informed search** strategies or **heuristic search** strategies, and they will be covered in Chapter 4. Not surprisingly, uninformed search is less effective than informed search. Uninformed search is still important, however, because there are many problems for which there is no additional information to consider.

The six uninformed search strategies are distinguished by the *order* in which nodes are expanded. It turns out that this difference can matter a great deal, as we shall shortly see.

---

[7] This is the way "optimality" is used in the theoretical computer science literature. Some AI authors use "optimality" to refer to time of execution and "admissibility" to refer to solution optimality.

Breadth-first search

BREADTH-FIRST SEARCH

One simple search strategy is a **breadth-first search.** In this strategy, the root node is expanded first, then all the nodes generated by the root node are expanded next, and then *their* successors, and so on. In general, all the nodes at depth d in the search tree are expanded before the nodes at depth $d + 1$. Breadth-first search can be implemented by calling the GENERAL-SEARCH algorithm with a queuing function that puts the newly generated states at the end of the queue, after all the previously generated states:

```
function BREADTH-FIRST-SEARCH(problem) returns a solution or failure
  return GENERAL-SEARCH(problem,ENQUEUE-AT-END)
```

Breadth-first search is a very systematic strategy because it considers all the paths of length 1 first, then all those of length 2, and so on. Figure 3.11 shows the progress of the search on a simple binary tree. If there is a solution, breadth-first search is guaranteed to find it, and if there are several solutions, breadth-first search will always find the shallowest goal state first. In terms of the four criteria, breadth-first search is complete, and it is optimal *provided the path cost is a nondecreasing function of the depth of the node.* (This condition is usually satisfied only when all operators have the same cost. For the general case, see the next section.)

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 can be expanded to yield b new states. We say that the **branching factor** of these states (and of the search tree) is b. 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 for this problem has a path length of d. Then the maximum number of nodes expanded before finding a solution is

BRANCHING FACTOR

$$1 + b + b^2 + b^3 + \bullet\bullet\bullet + b^d$$

This is the maximum number, but the solution could be found at any point on the dth level. In the best case, therefore, the number would be smaller.

Those who do complexity analysis get nervous (or excited, if they are the sort of people who like a challenge) whenever they see an exponential complexity bound like $O(b^d)$. Figure 3.12 shows why. It shows the time and memory required for a breadth-first search with branching factor $b = 10$ and for various values of the solution depth d. The space complexity is the same as the time complexity, because all the leaf nodes of the tree must be maintained in memory

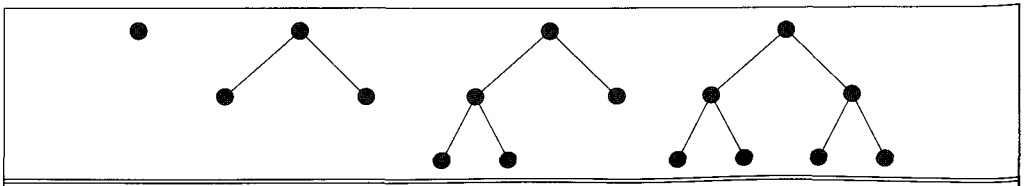


Figure 3.11 Breadth-first search trees after 0, 1, 2, and 3 node expansions.

at the same time. Figure 3.12 assumes that 1000 nodes can be goal-checked and expanded per second, and that a node requires 100 bytes of storage. Many puzzle-like problems fit roughly within these assumptions (give or take a factor of 100) when run on a modern personal computer or workstation.

Depth	Nodes	Time	Memory
0	1	1 millisecond	100 bytes
2	111	.1 seconds	11 kilobytes
4	11,111	11 seconds	1 megabyte
6	10^6	18 minutes	111 megabytes
8	10^8	31 hours	11 gigabytes
10	10^{10}	128 days	1 terabyte
12	10^{12}	35 years	111 terabytes
14	10^{14}	3500 years	11,111 terabytes

Figure 3.12 Time and memory requirements for breadth-first search. The figures shown assume branching factor $b = 10$; 1000 nodes/second; 100 bytes/node.

There are two lessons to be learned from Figure 3.12. First, *the memory requirements are a bigger problem for breadth-first search than the execution time.* Most people have the patience to wait 18 minutes for a depth 6 search to complete, assuming they care about the answer, but not so many have the 111 megabytes of memory that are required. And although 31 hours would not be too long to wait for the solution to an important problem of depth 8, very few people indeed have access to the 11 gigabytes of 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 an uninformed search to find it. Of course, if trends continue then in 10 years, you will be able to buy a computer that is 100 times faster for the same price as your current one. Even with that computer, however, it will still take 128 days to find a solution at depth 12—and 35 years for a solution at depth 14. Moreover, there are no other uninformed search strategies that fare any better. *In general, exponential complexity search problems cannot be solved for any but the smallest instances.*

Uniform cost search

UNIFORM COST SEARCH

Breadth-first search finds the *shallowest* goal state, but this may not always be the least-cost solution for a general path cost function. **Uniform cost search** modifies the breadth-first strategy by always expanding the lowest-cost node on the fringe (as measured by the path cost $g(n)$), rather than the lowest-depth node. It is easy to see that breadth-first search is just uniform cost search with $g(n) = \text{DEPTH}(n)$.

When certain conditions are met, the first solution that is found is guaranteed to be the cheapest solution, because if there were a cheaper path that was a solution, it would have been expanded earlier, and thus would have been found first. A look at the strategy in action will help explain. Consider the route-finding problem in Figure 3.13. The problem is to get from *S* to G,

and the cost of each operator is marked. The strategy first expands the initial state, yielding paths to A, *B,* and *C.* Because the path to A is cheapest, it is expanded next, generating the path *SAG,* which is in fact a solution, though not the optimal one. However, the algorithm does not yet recognize this as a solution, because it has cost 11, and thus is buried in the queue below the path *SB,* which has cost 5. It seems a shame to generate a solution just to bury it deep in the queue, but it is necessary if we want to find the optimal solution rather than just any solution. The next step is to expand *SB,* generating *SBG,* which is now the cheapest path remaining in the queue, so it is goal-checked and returned as the solution.

Uniform cost search finds the cheapest solution provided a simple requirement is met: the cost of a path must never decrease as we go along the path. In other words, we insist that

$$g(\text{SUCCESSOR}(n)) > g(n)$$

for every node *n.*

The restriction to nondecreasing path cost makes sense if the path cost of a node is taken to be the sum of the costs of the operators that make up the path. If every operator has a nonnegative cost, then the cost of a path can never decrease as we go along the path, and uniform-cost search can find the cheapest path without exploring the whole search tree. But if some operator had a negative cost, then nothing but an exhaustive search of all nodes would find the optimal solution, because we would never know when a path, no matter how long and expensive, is about to run into a step with high negative cost and thus become the best path overall. (See Exercise 3.5.)

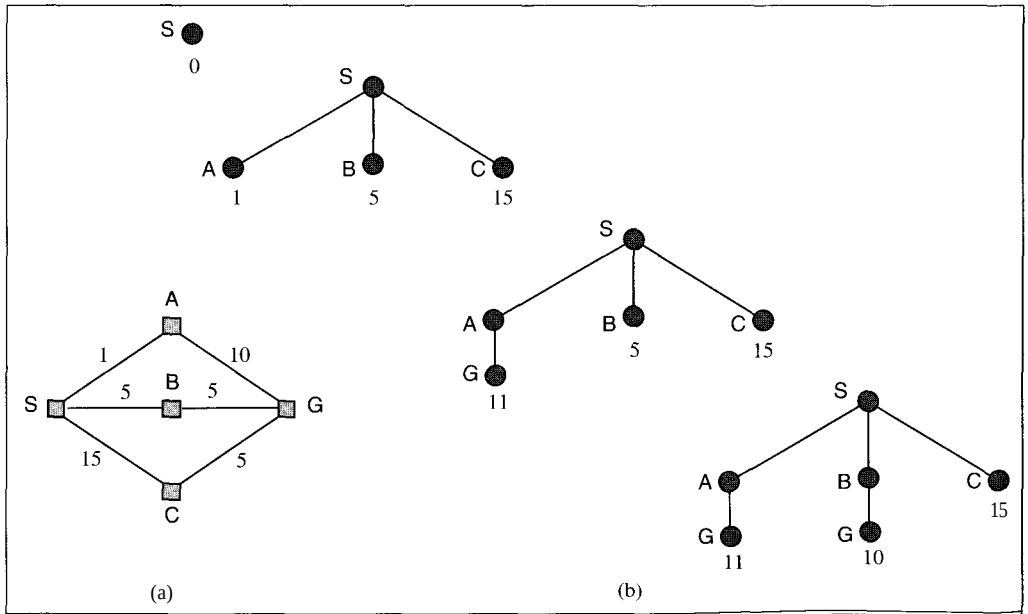


Figure 3.13 A route-finding problem. (a) The state space, showing the cost for each operator. (b) Progression of the search. Each node is labelled with $g(n)$. At the next step, the goal node with $g = 10$ will be selected.

Depth-first search

DEPTH-FIRST SEARCH

Depth-first search always expands one of the nodes at the deepest level of the tree. Only when the search hits a dead end (a nongoal node with no expansion) does the search go back and expand nodes at shallower levels. This strategy can be implemented by GENERAL-SEARCH with a queuing function that always puts the newly generated states at the front of the queue. Because the expanded node was the deepest, its successors will be even deeper and are therefore now the deepest. The progress of the search is illustrated in Figure 3.14.

Depth-first search has very modest memory requirements. As the figure shows, 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. For a state space with branching factor b and maximum depth m, depth-first search requires storage of only bm nodes, in contrast to the b^d that would be required by breadth-first search in the case where the shallowest goal is at depth d. Using the same assumptions as Figure 3.12, depth-first search would require 12 kilobytes instead of 111 terabytes at depth $d = 12$, a factor of 10 billion times less space.

The time complexity for depth-first search is $O(b^m)$. For problems that have very many solutions, depth-first may actually be faster than breadth-first, because it has a good chance of

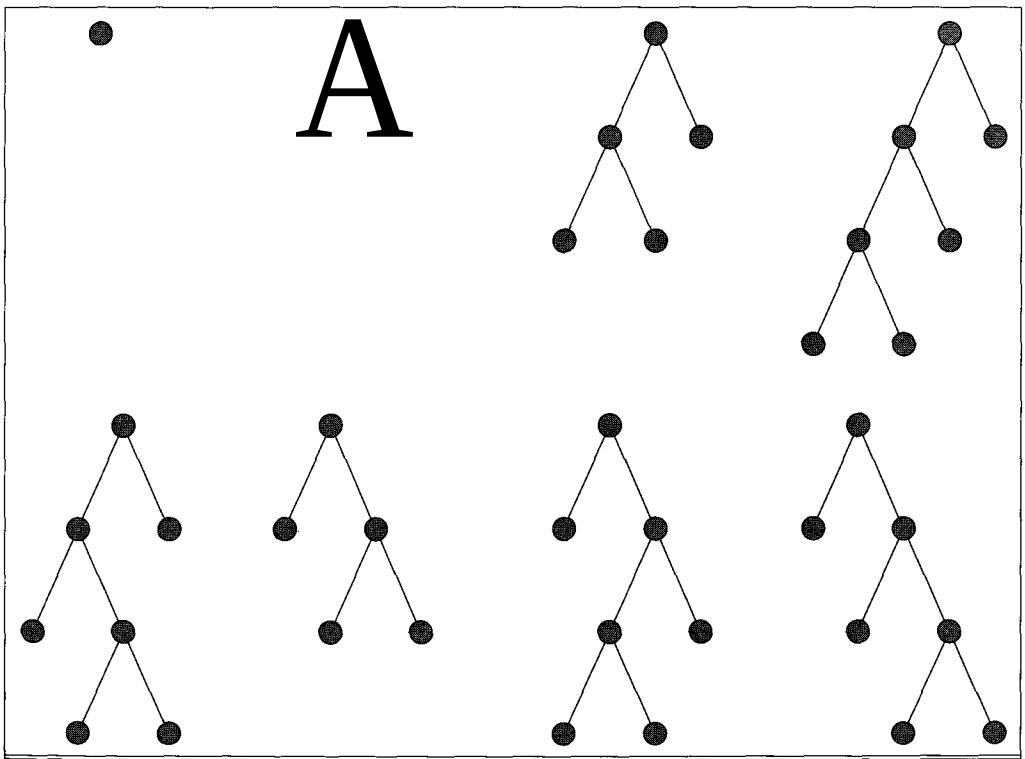


Figure 3.14 Depth-first search trees for a binary search tree. Nodes at depth 3 are assumed to have no successors.

finding a solution after exploring only a small portion of the whole space. Breadth-first search would still have to look at all the paths of length d - 1 before considering any of length d. Depth-first search is still $O(b^m)$ in the worst case.

The drawback of depth-first search is that it can get stuck going down the wrong path. Many problems have very deep or even infinite search trees, so depth-first search will never be able to recover from an unlucky choice at one of the nodes near the top of the tree. The search will always continue downward without backing up, even when a shallow solution exists. Thus, on these problems depth-first search will either get stuck in an infinite loop and never return a solution, or it may eventually find a solution path that is longer than the optimal solution. That means depth-first search is neither complete nor optimal. Because of this, *depth-first search should be avoided for search trees with large or infinite maximum depths.*

It is trivial to implement depth-first search with GENERAL-SEARCH:

```
function DEPTH-FIRST-SEARCH(problem) returns a solution, or failure
  GENERAL-SEARCH(problem, ENQUEUE-AT-FRONT)
```

It is also common to implement depth-first search with a recursive function that calls itself on each of its children in turn. In this case, the queue is stored implicitly in the local state of each invocation on the calling stack.

Depth-limited search

DEPTH-LIMITED SEARCH

Depth-limited search avoids the pitfalls of depth-first search by imposing a cutoff on the maximum depth of a path. This cutoff can be implemented with a special depth-limited search algorithm, or by using the general search algorithm with operators that keep track of the depth. For example, on the map of Romania, there are 20 cities, so we know that if there is a solution, then it must be of length 19 at the longest. We can implement the depth cutoff using operators of the form "If you are in city A and have travelled a path of less than 19 steps, then generate a new state in city B with a path length that is one greater." With this new operator set, we are guaranteed to find the solution if it exists, but we are still not guaranteed to find the shortest solution first: depth-limited search is complete but not optimal. If we choose a depth limit that is too small, then depth-limited search is not even complete. The time and space complexity of depth-limited search is similar to depth-first search. It takes $O(b^l)$ time and $O(bl)$ space, where l is the depth limit.

Iterative deepening search

The hard part about depth-limited search is picking a good limit. We picked 19 as an "obvious" depth limit for the Romania problem, 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. However, for most problems, we will not know a good depth limit until we have solved the problem.

DIAMETER

ITERATIVE DEEPENING SEARCH

Iterative deepening search is a strategy that sidesteps the issue of choosing the best depth limit by trying all possible depth limits: first depth 0, then depth 1, then depth 2, and so on. The algorithm is shown in Figure 3.15. In effect, iterative deepening combines the benefits of depth-first and breadth-first search. It is optimal and complete, like breadth-first search, but has only the modest memory requirements of depth-first search. The order of expansion of states is similar to breadth-first, except that some states are expanded multiple times. Figure 3.16 shows the first four iterations of ITERATIVE-DEEPENING-SEARCH on a binary search tree.

Iterative deepening search may seem wasteful, because so many states are expanded multiple times. For most problems, however, the overhead of this multiple expansion is actually

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution sequence
    inputs: problem, a problem

    for depth ← 0 to ∞ do
      if DEPTH-LIMITED-SEARCH( problem, depth) succeeds then return its result
    end
    return failure
```

Figure 3.15 The iterative deepening search algorithm.

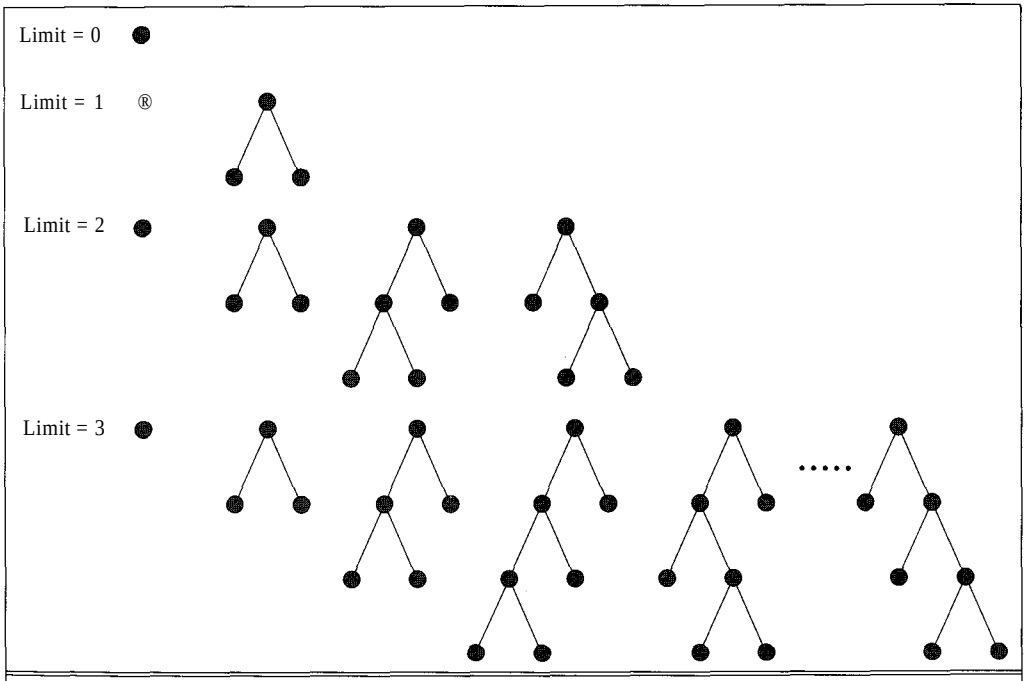


Figure 3.16 Four iterations of iterative deepening search on a binary tree.

rather small. Intuitively, the reason is that in an exponential search tree, almost all of the nodes are in the bottom level, so it does not matter much that the upper levels are expanded multiple times. Recall that the number of expansions in a depth-limited search to depth d with branching factor b is

$$1 + b + b^2 + \cdots + b^{d-2} + b^{d-1} + b^d$$

To make this concrete, for $b = 10$ and $d = 5$. the number is

$$1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111$$

In an iterative deepening search, the nodes on the bottom level are expanded once, those on the next to bottom level are expanded twice, and so on, up to the root of the search tree, which is expanded $d + 1$ times. So the total number of expansions in an iterative deepening search is

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

Again, for $b = 10$ and $d = 5$ the number is

$$6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456$$

All together, an iterative deepening search from depth 1 all the way down to depth d expands only about 11% more nodes than a single breadth-first or depth-limited search to depth d, when $b = 10$. The higher the branching factor, the lower the overhead of repeatedly expanded states, but even when the branching factor is 2, iterative deepening search only takes about twice as long as a complete breadth-first search. This means that the time complexity of iterative deepening is still $O(b^d)$, and the space complexity is $O(bd)$. *In general, iterative deepening is the preferred search method when there is a large search space and the depth of the solution is not known.*

Bidirectional search

The idea behind bidirectional search is to simultaneously search both forward from the initial state and backward from the goal, and stop when the two searches meet in the middle (Figure 3.17). For problems where the branching factor is b in both directions, bidirectional search can make a big difference. If we assume as usual that there is a solution of depth d, then the solution will be found in $O(2b^{d/2}) = O(b^{d/2})$ steps, because the forward and backward searches each have to go only half way. To make this concrete: for $b = 10$ and $d = 6$, breadth-first search generates 1,111,111 nodes, whereas bidirectional search succeeds when each direction is at depth 3, at which point 2,222 nodes have been generated. This sounds great in theory. Several issues need to be addressed before the algorithm can be implemented.

PREDECESSORS

- The main question is, what does it mean to search backwards from the goal? We define the **predecessors** of a node n to be all those nodes that have n as a successor. Searching backwards means generating predecessors successively starting from the goal node.
- When all operators are reversible, the predecessor and successor sets are identical; for some problems, however, calculating predecessors can be very difficult.
- What can be done if there are many possible goal states? If there is an *explicit* list of goal states, such as the two goal states in Figure 3.2, then we can apply a predecessor function to the state set just as we apply the successor function in multiple-state search. If we only

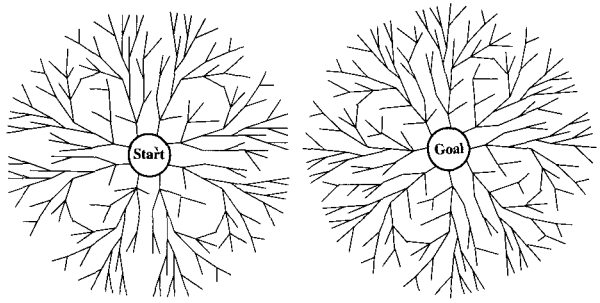


Figure 3.17 A schematic view of a bidirectional breadth-first search that is about to succeed, when a branch from the start node meets a branch from the goal node.

have a *description of* the set, it may be possible to figure out the possible descriptions of "sets of states that would generate the goal set," but this is a very tricky thing to do. For example, what are the states that are the predecessors of the checkmate goal in chess?

- There must be an efficient way to check each new node to see if it already appears in the search tree of the other half of the search.
- We need to decide what kind of search is going to take place in each half. For example, Figure 3.17 shows two breadth-first searches. Is this the best choice?

The $O(b^{d/2})$ complexity figure assumes that the process of testing for intersection of the two frontiers can be done in constant time (that is, is independent of the number of states). This often can be achieved with a hash table. In order for the two searches to meet at all, the nodes of at least one of them must all be retained in memory (as with breadth-first search). This means that the space complexity of uninformed bidirectional search is $O(b^{d/2})$.

Comparing search strategies

Figure 3.18 compares the six search strategies in terms of the four evaluation criteria set forth in Section 3.5.

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Time	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	b^d	bm	bl	bd	$b^{d/2}$
Optimal?	Yes	Yes	No	No	Yes	Yes
Complete?	Yes	Yes	No	Yes, if $l > d$	Yes	Yes

Figure 3.18 Evaluation of search strategies. b is the branching factor; d is the depth of solution; m is the maximum depth of the search tree; l is the depth limit.

3.6 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 on some other path. For some problems, this possibility never comes up; each state can only be reached one way. The efficient formulation of the 8-queens problem is efficient in large part because of this—each state can only be derived through one path.

For many problems, repeated states are unavoidable. This includes all problems where the operators are reversible, such as route-finding problems and the missionaries and cannibals problem. 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. Even when the tree is finite, avoiding repeated states can yield an exponential reduction in search cost. The classic example is shown in Figure 3.19. The space contains only $m + 1$ states, where m is the maximum depth. Because the tree includes each possible path through the space, it has 2^m branches.

There are three ways to deal with repeated states, in increasing order of effectiveness and computational overhead:

- Do not return to the state you just came from. Have the expand function (or the operator set) refuse to generate any successor that is the same state as the node's parent.
- Do not create paths with cycles in them. Have the expand function (or the operator set) refuse to generate any successor of a node that is the same as any of the node's ancestors.
- Do not generate any state that was ever generated before. This requires every state that is generated to be kept in memory, resulting in a space complexity of $O(b^d)$, potentially. It is better to think of this as $O(s)$, where s is the number of states in the entire state space.

To implement this last option, search algorithms often make use of a hash table that stores all the nodes that are generated. This makes checking for repeated states reasonably efficient. The trade-off between the cost of storing and checking and the cost of extra search depends on the problem: the "loopier" the state space, the more likely it is that checking will pay off.

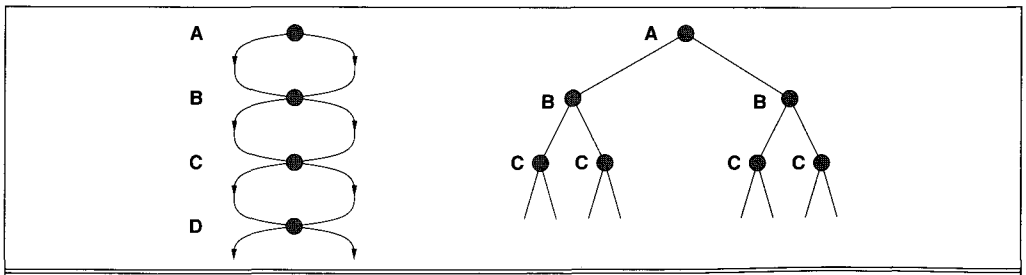


Figure 3.19 A state space that generates an exponentially larger search tree. The left-hand side shows the state space, in which there are two possible actions leading from A to B, two from B to C, and so on. The right-hand side shows the corresponding search tree.

3 7 CONSTRAINT SATISFACTION SEARCH

CONSTRAINT SATISFACTION PROBLEM

VARIABLES

CONSTRAINTS

A constraint satisfaction problem (or CSP) is a special kind of problem that satisfies some additional structural properties beyond the basic requirements for problems in general. In a CSP, the states are defined by the values of a set of **variables** and the goal test specifies a set of **constraints** that the values must obey. For example, the 8-queens problem can be viewed as a CSP in which the variables are the locations of each of the eight queens; the possible values are squares on the board; and the constraints state that no two queens can be in the same row, column or diagonal. A solution to a CSP specifies values for all the variables such that the constraints are satisfied. Cryptarithmetic and VLSI layout can also be described as CSPs (Exercise 3.20). Many kinds of design and scheduling problems can be expressed as CSPs, so they form a very important subclass. CSPs can be solved by general-purpose search algorithms, but because of their special structure, algorithms designed specifically for CSPs generally perform much better.

Constraints come in several varieties. Unary constraints concern the value of a single variable. For example, the variables corresponding to the leftmost digit on any row of a cryptarithmetic puzzle are constrained not to have the value 0. Binary constraints relate pairs of variables. The constraints in the 8-queens problem are all binary constraints. Higher-order constraints involve three or more variables—for example, the columns in the cryptarithmetic problem must obey an addition constraint and can involve several variables. Finally, constraints can be *absolute* constraints, violation of which rules out a potential solution, *or preference* constraints that say which solutions are preferred.

DOMAIN

Each variable V_i in a CSP has a **domain** D_i, which is the set of possible values that the variable can take on. The domain can be *discrete* or *continuous.* In designing a car, for instance, the variables might include component weights (continuous) and component manufacturers (discrete). A unary constraint specifies the allowable subset of the domain, and a binary constraint between two variables specifies the allowable subset of the cross-product of the two domains. In discrete CSPs where the domains are finite, constraints can be represented simply by enumerating the allowable combinations of values. For example, in the 8-queens problem, let $V\backslash$ be the row that the first queen occupies in the first column, and let V_2 be the row occupied by the second queen in the second column. The domains of $V\backslash$ and V_2 are $\{1,2,3,4,5,6,7,8\}$. The no-attack constraint linking $V\backslash$ and V_2 can be represented by a set of pairs of allowable values for $V\backslash$ and V_2: $\{\langle 1,3\rangle,\langle 1,4\rangle,\langle 1,5\rangle,\ldots,\langle 2,4\rangle,\langle 2,5\rangle,\ldots\}$ and so on. Altogether, the no-attack constraint between $V\backslash$ and V_2 rules out 22 of the 64 possible combinations. Using this idea of enumeration, any discrete CSP can be reduced to a binary CSP.

Constraints involving continuous variables cannot be enumerated in this way, and solving continuous CSPs involves sophisticated algebra. In this chapter, we will handle only discrete, absolute, binary (or unary) constraints. Such constraints are still sufficiently expressive to handle a wide variety of problems and to introduce most of the interesting solution methods.

Eet us first consider how we might apply a general-purpose search algorithm to a CSP. The initial state will be the state in which all the variables are unassigned. Operators will assign a value to a variable from the set of possible values. The goal test will check if all variables are assigned and all constraints satisfied. Notice that the maximum depth of the search tree is fixed

at n, the number of variables, and that all solutions are at depth n. We are therefore safe in using depth-first search, as there is no danger of going too deep and no arbitrary depth limit is needed.

In the most naive implementation, any unassigned variable in a given state can be assigned a value by an operator, in which case the branching factor would be as high as $\sum_i |D_i|$, or 64 in the 8-queens problem. A better approach is to take advantage of the fact that the order of variable assignments makes no difference to the final solution. Almost all CSP algorithms therefore generate successors by choosing values for only a single variable at each node. For example, in the 8-queens problem, one can assign a square for the first queen at level 0, for the second queen at level 1, and so on. This results in a search space of size $\prod_i |D_i|$, or 8^8 in the 8-queens problem. A straightforward depth-first search will examine all of these possibilities. Because CSPs include as special cases some well-known NP-complete problems such as 3SAT (see Exercise 6.15 on page 182), we cannot expect to do better than exponential complexity in the worst case. In most real problems, however, we can take advantage of the problem structure to eliminate a large fraction of the search space. The principal source of structure in the problem space is that, *in CSPs, the goal test is decomposed into a set of constraints on variables rather than being a "black box."*

Depth-first search on a CSP wastes time searching when constraints have already been violated. Because of the way that the operators have been defined, an operator can never redeem a constraint that has already been violated. For example, suppose that we put the first two queens in the top row. Depth-first search will examine all 8^6 possible positions for the remaining six queens before discovering that no solution exists in that subtree. Our first improvement is therefore to insert a test before the successor generation step to check whether any constraint has been violated by the variable assignments made up to this point. The resulting algorithm, called **backtracking search,** then backtracks to try something else.

BACKTRACKING SEARCH

Backtracking also has some obvious failings. Suppose that the squares chosen for the first six queens make it impossible to place the eighth queen, because they attack all eight squares in the last column. Backtracking will try all possible placings for the seventh queen, even though the problem is already rendered unsolvable, given the first six choices. **Forward checking** avoids this problem by looking ahead to detect unsolvability. Each time a variable is instantiated, forward checking deletes from the domains of the as-yet-uninstantiated variables all of those values that conflict with the variables assigned so far. If any of the domains becomes empty, then the search backtracks immediately. Forward checking often runs far faster than backtracking and is very simple to implement (see Exercise 3.21).

FORWARD CHECKING

ARC CONSISTENCY

Forward checking is a special case of **arc consistency** checking. A state is arc-consistent if every variable has a value in its domain that is consistent with each of the constraints on that variable. Arc consistency can be achieved by successive deletion of values that are inconsistent with some constraint. As values are deleted, other values may become inconsistent because they relied on the deleted values. Arc consistency therefore exhibits a form **of constraint propagation,** as choices are gradually narrowed down. In some cases, achieving arc consistency is enough to solve the problem completely because the domains of all variables are reduced to singletons. Arc consistency is often used as a preprocessing step, but can also be used during the search.

CONSTRAINT PROPAGATION

Much better results can often be obtained by careful choice of which variable to instantiate and which value to try. We examine such methods in the next chapter.

3.8 SUMMARY

This chapter has introduced methods that an agent can use when it is not clear which immediate action is best. In such cases, the agent can consider possible sequences of actions; 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 **operators, 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.**
- In real life most problems are ill-defined; but with some analysis, many problems can fit into the state space model.
- A single **general 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** expands the shallowest node in the search tree first. It is complete, optimal for unit-cost operators, and has time and space complexity of $O(b^d)$. The space complexity makes it impractical in most cases.
- **Uniform-cost search** expands the least-cost leaf node first. It is complete, and unlike breadth-first search is optimal even when operators have differing costs. Its space and time complexity are the same as for breadth-first search.
- **Depth-first search** expands the deepest node in the search tree first. 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. In search trees of large or infinite depth, the time complexity makes this impractical.
- **Depth-limited search** places a limit on how deep a depth-first search can go. If the limit happens to be equal to the depth of shallowest goal state, then time and space complexity are minimized.
- **Iterative deepening search** calls depth-limited search with increasing limits until a goal is found. It is complete and optimal, and has time complexity of $O(b^d)$ and space complexity of $O(bd)$.
- **Bidirectional search** can enormously reduce time complexity, but is not always applicable. Its memory requirements may be impractical.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Most of the state-space search problems analyzed in this chapter have a long history in the literature, and are far less trivial than they may seem. The missionaries and cannibals problem was analyzed in detail from an artificial intelligence perspective by Amarel (1968), although Amarel's treatment was by no means the first; it had been considered earlier in AI by Simon and Newell (1961), and elsewhere in computer science and operations research by Bellman and Dreyfus (1962). Studies such as these led to the establishment of search algorithms as perhaps the primary tools in the armory of early AI researchers, and the establishment of problem solving as the canonical AI task. (Of course, one might well claim that the latter resulted from the former.)

Amarel's treatment of the missionaries and cannibals problem is particularly noteworthy because it is a classic example of formal analysis of a problem stated informally in natural language. Amarel gives careful attention to *abstracting* from the informal problem statement precisely those features that are necessary or useful in solving the problem, and selecting a formal problem representation that represents only those features. A more recent treatment of problem representation and abstraction, including AI programs that themselves perform these tasks (in part), is to be found in Knoblock (1990).

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. Even Gauss was able to find only 72 of the 92 possible solutions offhand, which gives some indication of the difficulty of this apparently simple problem. (Nauck, who had republished the puzzle, published all 92 solutions later in 1850.) Netto (1901) generalized the problem to "n-queens" (on an n x n chessboard).

The 8-puzzle initially appeared as the more complex 4×4 version, called the 15-puzzle. It was invented by the famous American game designer Sam Loyd (1959) in the 1870s and quickly achieved immense popularity in the United States, comparable to the more recent sensation caused by the introduction of Rubik's Cube. It also quickly attracted the attention of mathematicians (Johnson and Story, 1879; Tait, 1880). The popular reception of the puzzle was so enthusiastic that the Johnson and Story article was accompanied by a note in which the editors of *the American Journal of Mathematics* felt it necessary to state that "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 brief summary of the reasons for the mathematical interest of the 15-puzzle). The 8-puzzle has often been used in AI research in place of the 15-puzzle because the search space is smaller and thus more easily subjected to exhaustive analysis or experimentation. An exhaustive analysis was carried out with computer aid by P. D. A. Schofield (1967). Although very time-consuming, this analysis allowed other, faster search methods to be compared against theoretical perfection for the quality of the solutions found. An in-depth analysis of the 8-puzzle, using heuristic search methods of the kind described in Chapter 4, was carried out by Doran and Michie (1966). The 15-puzzle, like the 8-queens problem, has been generalized to the $n \times n$ case. Ratner and

Warmuth (1986) showed that finding the shortest solution in the generalized $n \times n$ version belongs to the class of NP-complete problems.

"Uninformed" search algorithms for finding shortest paths that rely on current path cost alone, rather than an estimate of the distance to the goal, are a central topic of classical computer science, applied mathematics, and a related field known as *operations research.* Uniform-cost search as a way of finding shortest paths was invented by Dijkstra (1959). A survey of early work in uninformed search methods for shortest paths can be found in Dreyfus (1969); Deo and Pang (1982) give a more recent survey. For the variant of the uninformed shortest-paths problem that asks for shortest paths between *all* pairs of nodes in a graph, the techniques of *dynamic programming* and *memoization* can be used. For a problem to be solved by these techniques, it must be capable of being divided repeatedly into subproblems in such a way that identical subproblems arise again and again. Then dynamic programming or memoization involves systematically recording the solutions to subproblems in a table so that they can be looked up when needed and do not have to be recomputed repeatedly during the process of solving the problem. An efficient dynamic programming algorithm for the all-pairs shortest-paths problem was found by Bob Floyd (1962a; 1962b), and improved upon by Karger *et al.* (1993). Bidirectional search was introduced by Pohl (1969; 1971); it is often used with heuristic guidance techniques of the kind discussed in Chapter 4. Iterative deepening was first used by Slate and Atkin (1977) in the CHESS 4.5 game-playing program.

The textbooks by Nilsson (1971; 1980) are good general sources of information about classical search algorithms, although they are now somewhat dated. A comprehensive, and much more up-to-date, survey can be found in (Korf, 1988).

EXERCISES

3.1 Explain why problem formulation must follow goal formulation.

3.2 Consider the accessible, two-location vacuum world under Murphy's Law. Show that for each initial state, there is a sequence of actions that is guaranteed to reach a goal state.

3.3 Give the initial state, goal test, operators, and path cost function for each of the following. There are several possible formulations for each problem, with varying levels of detail. The main thing is that your formulations should be precise and "hang together" so that they could be implemented.

a. You want to find the telephone number of Mr. Jimwill Zollicoffer, who lives in Alameda, given a stack of directories alphabetically ordered by city.

b. As for part (a), but you have forgotten Jimwill's last name.

c. You are lost in the Amazon jungle, and have to reach the sea. There is a stream nearby.

d. You have to color a complex planar map using only four colors, with no two adjacent regions to have the same color.

e. A monkey is in a room with a crate, with bananas suspended just out of reach on the ceiling. He would like to get the bananas.

f. You are lost in a small country town, and must find a drug store before your hay fever becomes intolerable. There are no maps, and the natives are all locked indoors.

3.4 Implement the missionaries and cannibals problem and use breadth-first search to find the shortest solution. Is it a good idea to check for repeated states? Draw a diagram of the complete state space to help you decide.

3.5 On page 76, we said that we would not consider problems with negative path costs. In this exercise, we explore this in more depth.

a. Suppose that a negative lower bound c is placed on the cost of any given step—that is, negative costs are allowed, but the cost of a step cannot be less than c. Does this allow uniform-cost search to avoid searching the whole tree?

b. 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?

c. 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, 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.

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

3.6 The GENERAL-SEARCH algorithm consists of three steps: goal test, generate, and ordering function, in that order. It seems a shame to generate a node that is in fact a solution, but to fail to recognize it because the ordering function fails to place it first.

a. Write a version of GENERAL-SEARCH that tests each node as soon as it is generated and stops immediately if it has found a goal.

b. Show how the GENERAL-SEARCH algorithm can be used unchanged to do this by giving it the proper ordering function.

3.7 The formulation of problem, solution, and search algorithm given in this chapter explicitly mentions the path to a goal state. This is because the path is important in many problems. For other problems, the path is irrelevant, and only the goal state matters. Consider the problem "Find the square root of 123454321." A search through the space of numbers may pass through many states, but the only one that matters is the goal state, the number 11111. Of course, from a theoretical point of view, it is easy to run the general search algorithm and then ignore all of the path except the goal state. But as a programmer, you may realize an efficiency gain by coding a version of the search algorithm that does not keep track of paths. Consider a version of problem solving where there are no paths and only the states matter. Write definitions of problem and solution, and the general search algorithm. Which of the problems in Section 3.3 would best use this algorithm, and which should use the version that keeps track of paths?

3.8 Given a pathless search algorithm such as the one called for in Exercise 3.7, explain how you can modify the operators to keep track of the paths as part of the information in a state. Show the operators needed to solve the route-finding and touring problems.

3.9 Describe a search space in which iterative deepening search performs much worse than depth-first search.

3.10 Figure 3.17 shows a schematic view of bidirectional search. Why do you think we chose to show trees growing outward from the start and goal states, rather than two search trees growing horizontally toward each other?

3.11 Write down the algorithm for bidirectional search, in pseudo-code or in a programming language. Assume that each search will be a breadth-first search, and that the forward and backward searches take turns expanding a node at a time. Be careful to avoid checking each node in the forward search against each node in the backward search!

3.12 Give the time complexity of bidirectional search when the test for connecting the two searches is done by comparing a newly generated state in the forward direction against all the states generated in the backward direction, one at a time.

3.13 We said that at least one direction of a bidirectional search must be a breadth-first search. What would be a good choice for the other direction? Why?

3.14 Consider the following operator for the 8-queens problem: place a queen in the column with the fewest unattacked squares, in such a way that it does not attack any other queens. How many nodes does this expand before it finds a solution? (You may wish to have a program calculate this for you.)

3.15 The **chain problem** (Figure 3.20) consists of various lengths of chain that must be reconfigured into new arrangements. Operators can open one link and close one link. In the standard form of the problem, the initial state contains four chains, each with three links. The goal state consists of a single chain of 12 links in a circle. Set this up as a formal search problem and find the shortest solution.

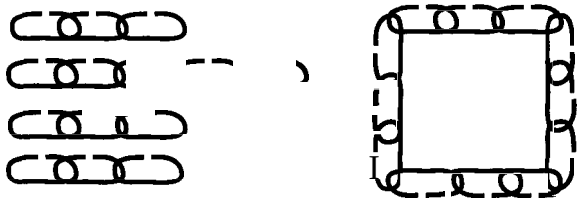


Figure 3.20 The chain problem. Operators can open, remove, reattach, and close a single link at a time.

3.16 Tests of human intelligence often contain **sequence prediction** problems. The aim in such problems is to predict the next member of a sequence of integers, assuming that the number in position n of the sequence is generated using some sequence function $s(n)$, where the first element of the sequence corresponds to $n = 0$. For example, the function $s(n) - 2^n$ generates the sequence $[1, 2, 4, 8, 16, \ldots]$.

In this exercise, you will design a problem-solving system capable of solving such prediction problems. The system will search the space of possible functions until it finds one that matches the observed sequence. The space of sequence functions that we will consider consists of all possible expressions built from the elements 1 and n, and the functions +, x, −, /, and exponentiation. For example, the function 2^n becomes $(1 + 1)^n$ in this language. It will be useful to think of function expressions as binary trees, with operators at the internal nodes and 1's and n's at the leaves.

a. First, write the goal test function. Its argument will be a candidate sequence function s. It will contain the observed sequence of numbers as local state.

b. Now write the successor function. Given a function expression s, it should generate all expressions one step more complex than s. This can be done by replacing any leaf of the expression with a two-leaf binary tree.

c. Which of the algorithms discussed in this chapter would be suitable for this problem? Implement it and use it to find sequence expressions for the sequences [1,2,3,4,5], $[1, 2, 4, 8, 16, \ldots]$, and $[0.5, 2, 4.5, 8]$.

d. If level d of the search space contains all expressions of complexity $d+1$, where complexity is measured by the number of leaf nodes (e.g., $n + (1 \times n)$ has complexity 3), prove by induction that there are roughly $20^d(d+1)!$ expressions at level d.

e. Comment on the suitability of uninformed search algorithms for solving this problem. Can you suggest other approaches?

3.17 The full vacuum world from the exercises in Chapter 2 can be viewed as a search problem in the sense we have defined, provided we assume that the initial state is completely known.

a. Define the initial state, operators, goal test function, and path cost function.

b. Which of the algorithms defined in this chapter would be appropriate for this problem?

c. Apply one of them to compute an optimal sequence of actions for a 3 x 3 world with dirt in the center and home squares.

d. Construct a search agent for the vacuum world, and evaluate its performance in a set of 3x3 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.

e. Compare the performance of your search agent with the performance of the agents constructed for the exercises in Chapter 2. What happens if you include computation time in the performance measure, at various "exchange rates" with respect to the cost of taking a step in the environment?

f. Consider what would happen if the world was enlarged to $n \times n$. How does the performance of the search agent vary with n? Of the reflex agents?

3.18 The search agents we have discussed make use of a complete model of the world to construct a solution that they then execute. Modify the depth-first search algorithm with repeated state checking so that an agent can use it to explore an arbitrary vacuum world even without a model of the locations of walls and dirt. It should not get stuck even with loops or dead ends. You may also wish to have your agent construct an environment description of the type used by the standard search algorithms.

3.19 In discussing the cryptarithmetic problem, we proposed that an operator should assign a value to whichever letter has the least remaining possible values. Is this rule guaranteed to produce the smallest possible search space? Why (not)?

3.20 Define each of the following as constraint satisfaction problems:

a. The cryptarithmetic problem.

b. The channel-routing problem in VLSI layout.

MAP-COLORING

c. The map-coloring problem. In map-coloring, the aim is to color countries on a map using a given set of colors, such that no two adjacent countries are the same color.

FLOOR-PLANNING

d. The rectilinear **floor-planning** problem, which involves finding nonoverlapping places in a large rectangle for a number of smaller rectangles.

3.21 Implement a constraint satisfaction system as follows:

a. Define a datatype for CSPs with finite, discrete domains. You will need to find a way to represent domains and constraints.

b. Implement operators that assign values to variables, where the variables are assigned in a fixed order at each level of the tree.

c. Implement a goal test that checks a complete state for satisfaction of all the constraints.

d. Implement backtracking by modifying DEPTH-FIRST-SEARCH.

e. Add forward checking to your backtracking algorithm.

f. Run the three algorithms on some sample problems and compare their performance.

4 INFORMED SEARCH METHODS

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. It also shows how optimization problems can be solved.

4.1 BEST-FIRST SEARCH

In Chapter 3, we found several ways to apply knowledge to the process of formulating a problem in terms of states and operators. Once we are given a well-defined problem, however, our options are more limited. If we plan to use the GENERAL-SEARCH algorithm from Chapter 3, then the only place where knowledge can be applied is in the queuing function, which determines the node to expand next. Usually, the knowledge to make this determination is provided by an

EVALUATION FUNCTION

evaluation function that returns a number purporting to describe the desirability (or lack thereof) of expanding the node. When the nodes are ordered so that the one with the best evaluation is

BEST-FIRST SEARCH

expanded first, the resulting strategy is called **best-first search.** It can be implemented directly with GENERAL-SEARCH, as shown in Figure 4.1.

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 omniscient, 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.

Just as there is a whole family of GENERAL-SEARCH algorithms with different ordering functions, there is also a whole family of BEST-FIRST-SEARCH algorithms with different evaluation

```
function BEST-FIRST-SEARCH(problem, EVAL-FN) returns a solution sequence
  inputs: problem, a problem
          Eval-Fn, an evaluation function

  Queueing-Fn ← a function that orders nodes by EVAL-FN
  return GENERAL-SEARCH(problem, Queueing-Fn)
```

Figure 4.1 An implementation of best-first search using the general search algorithm.

functions. Because they aim to find low-cost solutions, these algorithms typically use some estimated measure of the cost of the solution and try to minimize it. We have already seen one such measure: the use of the path cost g to decide which path to extend. This measure, however, does not direct search *toward the goal. In order to focus the search, the measure must incorporate some estimate of the cost of the path from a state to the closest goal state.* We look at two basic approaches. The first tries to expand the node closest to the goal. The second tries to expand the node on the least-cost solution path.

Minimize estimated cost to reach a goal: Greedy search

One of the simplest best-first search strategies is to minimize the estimated cost to reach the goal. That is, the node whose state is judged to be closest to the goal state is always expanded first. For most problems, the cost of reaching the goal from a particular state can be estimated but cannot be determined exactly. A function that calculates such cost estimates is called a **heuristic function,** and is usually denoted by the letter h:

HEURISTIC FUNCTION

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

GREEDY SEARCH

A best-first search that uses h to select the next node to expand is called **greedy search,** for reasons that will become clear. Given a heuristic function h, the code for greedy search is just the following:

```
function GREEDY-SEARCH(problem) returns a solution or failure
  return BEST-FIRST-SEARCH(problem, h)
```

Formally speaking, h can be any function at all. We will require only that $h(n) = 0$ if n is a goal.

To get an idea of what a heuristic function looks like, we need to choose a particular problem, because heuristic functions are problem-specific. Let us return to the route-finding problem from Arad to Bucharest. The map for that problem is repeated in Figure 4.2.

STRAIGHT-LINE DISTANCE

A good heuristic function for route-finding problems like this is the **straight-line distance** to the goal. That is,

$h_{SLD}(n) =$ straight-line distance between n and the goal location.

HISTORY OF "HEURISTIC"

> By now the space aliens had mastered my own language, but they still made simple mistakes like using "hermeneutic" when they meant "heuristic."
> — a Louisiana factory worker in Woody Alien's *The UFO Menace*

The word "heuristic" is derived from the Greek verb *heuriskein*, meaning "to find" or "to discover." Archimedes is said to have run naked down the street shouting "*Heureka*" (I have found it) after discovering the principle of flotation in his bath. Later generations converted this to Eureka.

The technical meaning of "heuristic" has undergone several changes in the history of AI. In 1957, George Polya wrote an influential book called *How to Solve It* that used "heuristic" to refer to the study of methods for discovering and inventing problem-solving techniques, particularly for the problem of coming up with mathematical proofs. Such methods had often been deemed not amenable to explication.

Some people use heuristic as the opposite of algorithmic. For example, Newell, Shaw, and Simon stated in 1963, "A process that may solve a given problem, but offers no guarantees of doing so, is called a heuristic for that problem." But note that there is nothing random or nondeterministic about a heuristic search algorithm: it proceeds by algorithmic steps toward its result. In some cases, there is no guarantee how long the search will take, and in some cases, the quality of the solution is not guaranteed either. Nonetheless, it is important to distinguish between "nonalgorithmic" and "not precisely characterizable."

Heuristic techniques dominated early applications of artificial intelligence. The first "expert systems" laboratory, started by Ed Feigenbaum, Brace Buchanan, and Joshua Lederberg at Stanford University, was called the Heuristic Programming Project (HPP). Heuristics were viewed as "rules of thumb" that domain experts could use to generate good solutions without exhaustive search. Heuristics were initially incorporated directly into the structure of programs, but this proved too inflexible when a large number of heuristics were needed. Gradually, systems were designed that could accept heuristic information expressed as "rules," and rule-based systems were born.

Currently, heuristic is most often used as an adjective, referring to any technique that improves the average-case performance on a problem-solving task, but does not necessarily improve the worst-case performance. In the specific area of search algorithms, it refers to a function that provides an estimate of solution cost.

A good article on heuristics (and one on hermeneutics!) appears in the *Encyclopedia of AI* (Shapiro, 1992).

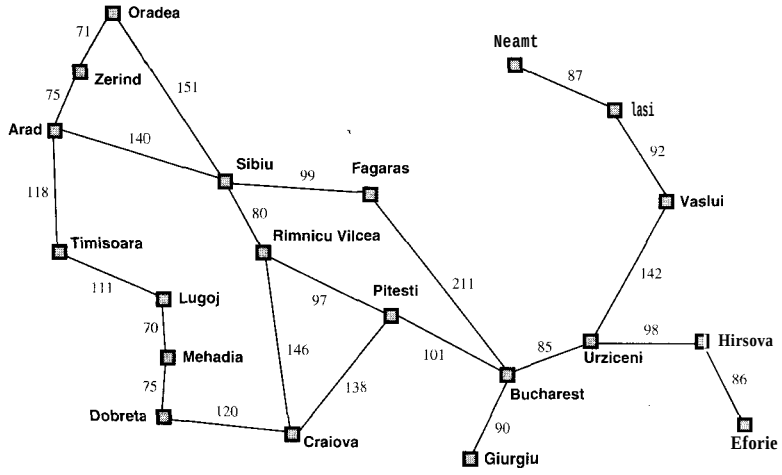


Figure 4.2 Map of Romania with road distances in km, and straight-line distances to Bucharest.

Notice that we can only calculate the values of h_{SLD} if we know the map coordinates of the cities in Romania. Furthermore, h_{SLD} is only useful because a road from A to B usually tends to head in more or less the right direction. This is the sort of extra information that allows heuristics to help in reducing search cost.

Figure 4.3 shows the progress of a greedy search to find a path from Arad to Bucharest. With the straight-line-distance heuristic, 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, the heuristic leads to minimal search cost: it finds a solution without ever expanding a node that is not on the solution path. However, it is not perfectly optimal: the path it found via Sibiu and Fagaras to Bucharest is 32 miles longer than the path through Rimnicu Vilcea and Pitesti. This path was not found because Fagaras is closer to Bucharest in straight-line distance than Rimnicu Vilcea, so it was expanded first. The strategy prefers to take the biggest bite possible out of the remaining cost to reach the goal, without worrying about whether this will be best in the long run—hence the name "greedy search." Although greed is considered one of the seven deadly sins, it turns out that greedy algorithms often perform quite well. They tend to find solutions quickly, although as shown in this example, they do not always find the optimal solutions: that would take a more careful analysis of the long-term options, not just the immediate best choice.

Greedy search is susceptible to false starts. Consider the problem of getting from Iasi to Fagaras. The heuristic suggests that Neamt be expanded first, 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. Hence, in this case, 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.

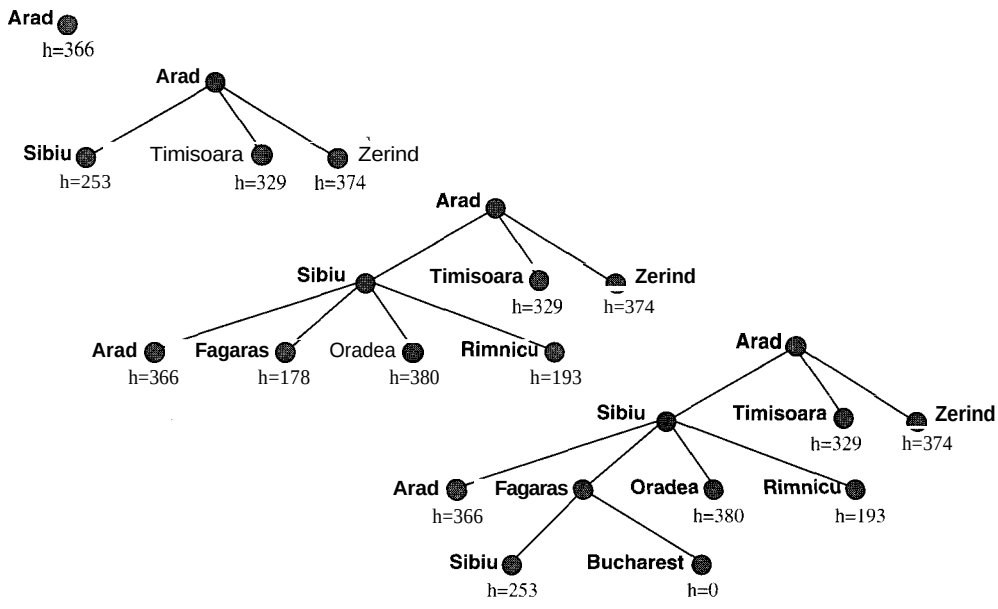


Figure 4.3 Stages in a greedy search for Bucharest, using the straight-line distance to Bucharest as the heuristic function h_{SLD}. Nodes are labelled with their h-values.

Greedy 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 complexity for greedy search is $O(b^m)$, where m is the maximum depth of the search space. Because greedy search retains all nodes in memory, its space complexity is the same as its time complexity. With a good heuristic function, the space and time complexity can be reduced substantially. The amount of the reduction depends on the particular problem and quality of the h function.

Minimizing the total path cost: A* search

Greedy search minimizes the estimated cost to the goal, $h(n)$, and thereby cuts the search cost considerably. Unfortunately, it is neither optimal nor complete. Uniform-cost search, on the other hand, minimizes the cost of the path so far, $g(n)$; it is optimal and complete, but can be very inefficient. It would be nice if we could combine these two strategies to get the advantages of both. Fortunately, we can do exactly that, combining the two evaluation functions simply by summing them:

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

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 f. The pleasant thing about this strategy is that it is more than just reasonable. We can actually prove that it is complete and optimal, given a simple restriction on the h function.

ADMISSIBLE HEURISTIC

The restriction is to choose an h function that *never overestimates* the cost to reach the goal. Such an h is called an **admissible heuristic.** Admissible heuristics are by nature optimistic, because they think the cost of solving the problem is less than it actually is. This optimism transfers to the f function as well: *If his admissible, $f(n)$ never overestimates the actual cost of the best solution through n.* Best-first search using f as the evaluation function and an admissible h function is known as A* **search**

A* SEARCH

```
function A*-SEARCH(problem) returns a solution or failure
  return BEST-FIRST-SEARCH(problem,g + h)
```

Perhaps the most 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. In Figure 4.4, we show the first few steps of an A* search for Bucharest using the h_{SLD} heuristic. Notice that the A* search prefers to expand from Rimnicu Vilcea rather than from Fagaras. Even though Fagaras is closer to Bucharest, the path taken to get to Fagaras is not as *efficient* in getting close to Bucharest as the path taken to get to Rimnicu. The reader may wish to continue this example to see what happens next.

The behavior of A* search

Before we prove the completeness and optimality of A*, it will be useful to present an intuitive picture of how it works. A picture is not a substitute for a proof, but it is often easier to remember and can be used to generate the proof on demand. First, a preliminary observation: if you examine the search trees in Figure 4.4, you will notice an interesting phenomenon. Along any path from the root, the f-cost never decreases. This is no accident. It holds true for almost all admissible heuristics. A heuristic for which it holds is said to exhibit **monotonicity**.[1]

MONOTONICITY

If the heuristic is one of those odd ones that is not monotonic, it turns out we can make a minor correction that restores monotonicity. Let us consider two nodes n and n', where n is the parent of n'. Now suppose, for example, that $g(n) = 3$ and $h(n) = 4$. Then $f(n) = g(n)+h(n)- 7$—that is, we know that the true cost of a solution path through n is at least 7. Suppose also that $g(n') = 4$ and $h(n')- 2$, so that $f(n') = 6$. Clearly, this is an example of a nonmonotonic heuristic. Fortunately, from the fact that *any path through n' is also a path through* n, we can see that the value of 6 is meaningless, because we already know the true cost is at least 7. Thus, we should

[1] It can be proved (Pearl, 1984) that a heuristic is monotonic if and only if it obeys the triangle inequality. The triangle inequality says that two sides of a triangle cannot add up to less than the third side (see Exercise 4.7). Of course, straight-line distance obeys the triangle inequality and is therefore monotonic.

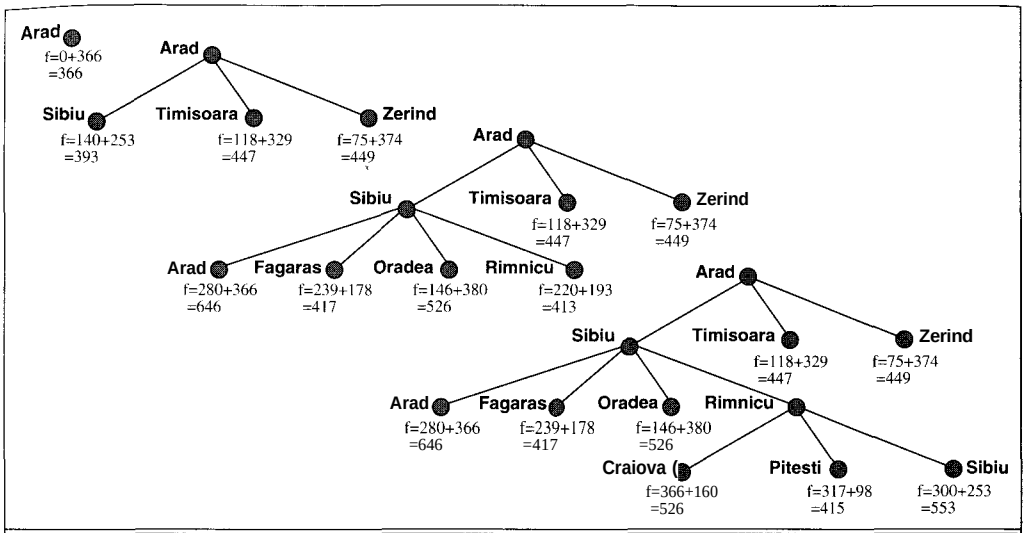


Figure 4.4 Stages in an A* search for Bucharest. Nodes are labelled with / = g + *h*. The *h* values are the straight-line distances to Bucharest taken from Figure 4.1.

check, each time we generate a new node, to see if its/-cost is less than its parent's f-cost; if it is, we use the parent's f-cost instead:

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

In this way, we ignore the misleading values that may occur with a nonmonotonic heuristic. This equation is called the **pathmax** equation. If we use it, then f will always be nondecreasing along any path from the root, provided h is admissible.

PATHMAX

The purpose of making this observation is to legitimize a certain picture of what A* does. If f never decreases along any path out from the root, we can conceptually draw **contours** in the state space. Figure 4.5 shows an example. Inside the contour labelled 400, all nodes have $f(n)$ less than or equal to 400, and so on. Then, because A* expands the leaf node of lowest f, we can see that an A* search fans out from the start node, adding nodes in concentric bands of increasing f-cost.

CONTOURS

With uniform-cost search (A* search using $h = 0$), 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 we define f^* to be the cost of the optimal solution path, then we can say the following:

- A* expands all nodes with $f(n) < f^*$.
- A* may then expand some of the nodes right on the "goal contour," for which $f(n) =/*$, before selecting a goal node.

Intuitively, it is obvious that the first solution found must be the optimal one, because nodes in all subsequent contours will have higher f-cost, and thus higher g-cost (because all goal states have $h(n) = 0$). Intuitively, it is also obvious that A* search is complete. As we add bands of

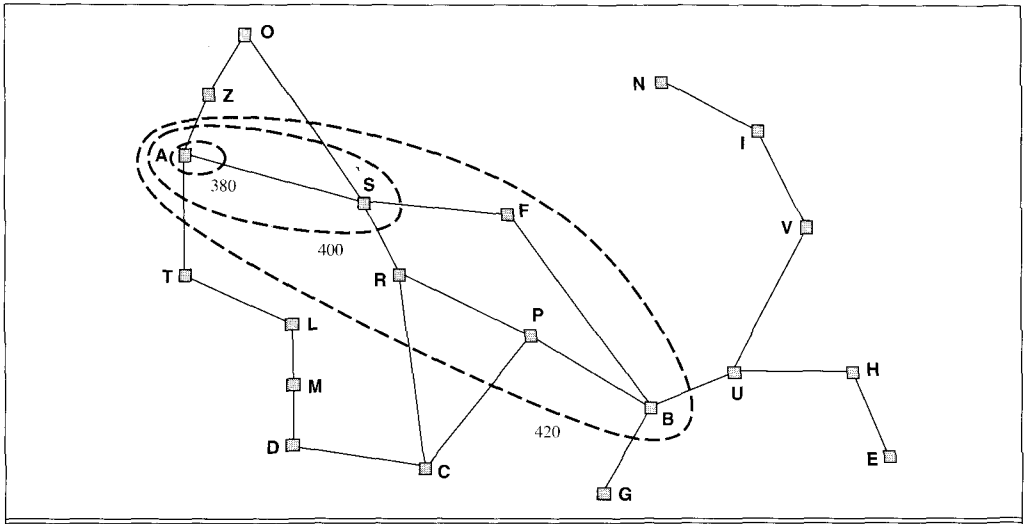


Figure 4.5 Map of Romania showing contours at $f = 380$, $f = 400$ and $f = 420$, with Arad as the start state. Nodes inside a given contour have f-costs lower than the contour value.

increasing f, we must eventually reach a band where f is equal to the cost of the path to a goal state. We will turn these intuitions into proofs in the next subsection.

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*. We can explain this as follows: any algorithm that *does not* expand all nodes in the contours between the root and the goal contour runs the risk of missing the optimal solution. A long and detailed proof of this result appears in Dechter and Pearl (1985).

Proof of the optimality of A*

Let G be an optimal goal state, with path cost f^*. Let G_2 be a suboptimal goal state, that is, a goal state with path cost $g(G_2) > f^*$. The situation we imagine is that A* has selected G_2 from the queue. Because G_2 is a goal state, this would terminate the search with a suboptimal solution (Figure 4.6). We will show that this is not possible.

Consider a node n that is currently a leaf node on an optimal path to G (there must be some such node, unless the path has been completely expanded—in which case the algorithm would have returned G). Because h is admissible, we must have

$$\Gamma \geq f(n).$$

Furthermore, if n is not chosen for expansion over G_2, we must have

$$f(n) \geq f(G_2).$$

Combining these two together, we get

$$\Gamma > f(G_2).$$

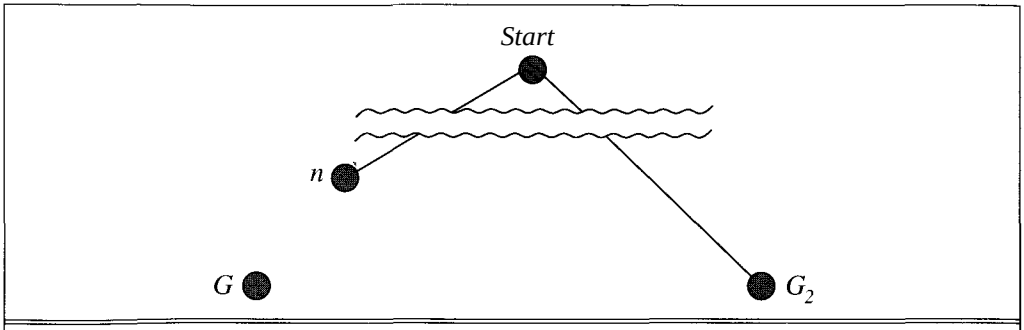


Figure 4.6 Situation at the point where a suboptimal goal state G_2 is about to be expanded. Node *n* is a leaf node on an optimal path from the start node to the optimal goal state G.

But because G_2 is a goal state, we have $h(G_2) = 0$; hence $f(G_2) = g(G_2)$. Thus, we have proved, from our assumptions, that

$$f^* \geq g(G_2).$$

This contradicts the assumption that G_2 is suboptimal, so it must be the case that A* never selects a suboptimal goal for expansion. Hence, because it only returns a solution after selecting it for expansion, A* is an optimal algorithm.

Proof of the completeness of A*

We said before that because A* expands nodes in order of increasing f, it must eventually expand to reach a goal state. This is true, of course, unless there are infinitely many nodes with $f(n) < f^*$. The only way there could be an infinite number of nodes is either (a) there is a node with an infinite branching factor, or (b) there is a path with a finite path cost but an infinite number of nodes along it.[2]

LOCALLY FINITE GRAPHS

Thus, the correct statement is that A* is complete on **locally finite graphs** (graphs with a finite branching factor) provided there is some positive constant δ such that every operator costs at least δ.

Complexity of A*

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

[2] Zeno's paradox, which purports to show that a rock thrown at a tree will never reach it, provides an example that violates condition (b). The paradox is created by imagining that the trajectory is divided into a series of phases, each of which covers half the remaining distance to the tree; this yields an infinite sequence of steps with a finite total cost.

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. Of course, the use of a good heuristic still provides enormous savings compared to an uninformed search. In the next section, 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, A* usually runs out of space long before it runs out of time. Recently developed algorithms have overcome the space problem without sacrificing optimality or completeness. These are discussed in Section 4.3.

4.2 HEURISTIC FUNCTIONS

So far we have seen just one example of a heuristic: straight-line distance for route-finding problems. In this section, we will look at heuristics for the 8-puzzle. This will shed light on the nature of heuristics in general.

The 8-puzzle was one of the earliest heuristic search problems. As mentioned in Section 3.3, the object of the puzzle is to slide the tiles horizontally or vertically into the empty space until the initial configuration matches the goal configuration (Figure 4.7).

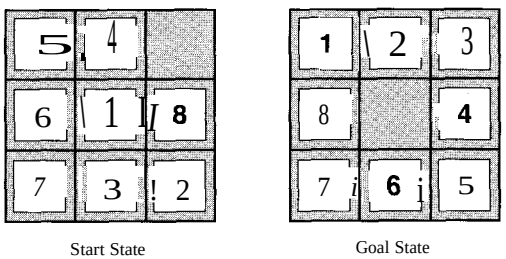


Figure 4.7 A typical instance of the 8-puzzle.

The 8-puzzle is just the right level of difficulty to be interesting. A typical solution is about 20 steps, although this of course varies depending on the initial state. 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 20 would look at about $3^{20} = 3.5 \times 10^9$ states. By keeping track of repeated states, we could cut this down drastically, because there are only $9! = 362,880$ different arrangements of 9 squares. This is still a large number of states, so the next order of business is to find a good

heuristic function. If we want to find the shortest solutions, we need a heuristic function that never overestimates the number of steps to the goal. Here are two candidates:

- $h\backslash$ = the number of tiles that are in the wrong position. For Figure 4.7, none of the 8 tiles is in the goal 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 any move can only move one tile one step closer to the goal. The 8 tiles in the start state give a Manhattan distance of

MANHATTAN DISTANCE

$$h_2 = 2 + 3 + 2 + 1 + 2 + 2 + 1 + 2 = 15$$

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 expanded 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 nodes. Thus,

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

For example, if A* finds a solution at depth 5 using 52 nodes, then the effective branching factor is 1.91. Usually, the effective branching factor exhibited by a given heuristic is fairly constant over a large range of problem instances, and 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 $h\backslash$ and h_2, we randomly generated 100 problems each with solution lengths $2, 4, \ldots, 20$, and solved them using A* search with $h\backslash$ and h_2, as well as with uninformed iterative deepening search. Figure 4.8 gives the average number of nodes expanded by each strategy, and the effective branching factor. The results show that h_2 is better than h_1, and that uninformed search is much worse.

DOMINATES

One might ask if 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, $h_2(n) > h_1(n)$. We say that h_2 **dominates** h_1. Domination translates directly into efficiency: A* using h_2 will expand fewer nodes, on average, than A* using h_1. We can show this using the following simple argument. Recall the observation on page 98 that every node with $f(n) < f^*$ will be expanded. This is the same as saying that every node with $h(n) <$/* — $g(n)$ will be expanded. But because h_2 is at least as big as $h\backslash$ for all nodes, every node that is expanded by A* search with h_2 will also be expanded with h_1, and h_1 may also l cause other nodes to be expanded as well. *Hence, it is always better to use a heuristic function 1 with higher values, as long as it does not overestimate.*

	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	364404	227	73	2.78	1.42	1.24
14	3473941	539	113	2.83	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 $h1$, h_2. Data are averaged over 100 instances of the 8-puzzle, for various solution lengths.

Inventing heuristic functions

We have seen that both $h1$ and h_2 are fairly good heuristics for the 8-puzzle, and that h_2 is better. But we do not know how to invent a heuristic function. How might one have come up with h_2? Is it possible for a computer to mechanically invent such a heuristic?

h_1 and h_2 are estimates to the remaining path length for the 8-puzzle, but they can also be considered to be 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 $h1$ would accurately give the number of steps to 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 less restrictions on the operators is called a **relaxed problem.** *It is often the case that the cost of an exact solution to a relaxed problem is a good heuristic for the original problem.*

RELAXED PROBLEM

If a problem definition is written down in a formal language, it is possible to construct relaxed problems automatically.[3] For example, if the 8-puzzle operators are described as

A tile can move from square A to square B if A is adjacent to B and B is blank,

we can generate three relaxed problems by removing one or more 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.

Recently, a program called ABSOLVER was written that 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 existing heuristic, and found the first useful heuristic for the famous Rubik's cube puzzle.

[3] In Chapters 7 and 11, we will describe formal languages suitable for this task. For now, we will use English.

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 also admissible. Furthermore, h dominates all of the individual heuristics from which it is composed.

Another way to invent a good heuristic is to use statistical information. This can be gathered by running a search over a number of training problems, such as the 100 randomly chosen 8-puzzle configurations, and gathering statistics. For example, we might find that when $h_2(n) = 14$, it turns out that 90% of the time the real distance to the goal is 18. Then when faced with the "real" problem, we can use 18 as the value whenever $h_2(n)$ reports 14. Of course, if we use probabilistic information like this, we are giving up on the guarantee of admissibility, but we are likely to expand fewer nodes on average.

FEATURES

Often it is possible to pick out **features** of a state that contribute to its heuristic evaluation function, even if it is hard to say exactly what the contribution should be. For example, the goal in chess is to checkmate the opponent, and relevant features include the number of pieces of each kind belonging to each side, the number of pieces that are attacked by opponent pieces, and so on. Usually, the evaluation function is assumed to be a linear combination of the feature values. Even if we have no idea how important each feature is, or even if a feature is good or bad, it is still possible to use a learning algorithm to acquire reasonable coefficients for each feature, as demonstrated in Chapter 18. In chess, for example, a program could learn that one's own queen should have a large positive coefficient, whereas an opponent's pawn should have a small negative coefficient.

Another factor that we have not considered so far is the search cost of actually running the heuristic function on a node. We have been assuming that the cost of computing the heuristic function is about the same as the cost of expanding a node, so that minimizing the number of nodes expanded is a good thing. But if the heuristic function is so complex that computing its value for one node takes as long as expanding hundreds of nodes, then we need to reconsider. After all, it is easy to have a heuristic that is perfectly accurate—if we allow the heuristic to do, say, a full breadth-first search "on the sly." That would minimize the number of nodes expanded by the real search, but it would not minimize the overall search cost. A good heuristic function must be efficient as well as accurate.

Heuristics for constraint satisfaction problems

In Section 3.7, we examined a class of problems called **constraint satisfaction problems** (CSPs). A constraint satisfaction problem consists of a set of variables that can take on values from a given domain, together with a set of constraints that specify properties of the solution. Section 3.7 examined uninformed search methods for CSPs, mostly variants of depth-first search. Here, we extend the analysis by considering heuristics for selecting a variable to instantiate and for choosing a value for the variable.

To illustrate the basic idea, we will use the map-coloring problem shown in Figure 4.9. (The idea of map coloring is to avoid coloring adjacent countries with the same color.) Suppose that we can use at most three colors (red, green, and blue), and that we have chosen green for country A and red for country B. Intuitively, it seems obvious that we should color E next, because the only possible color for E is blue. All the other countries have a choice of colors, and we might make the wrong choice and have to backtrack. In fact, once we have colored E blue, then we are forced to color C red and F green. After that, coloring D either blue or red results in a solution. In other words, we have solved the problem with no search at all.

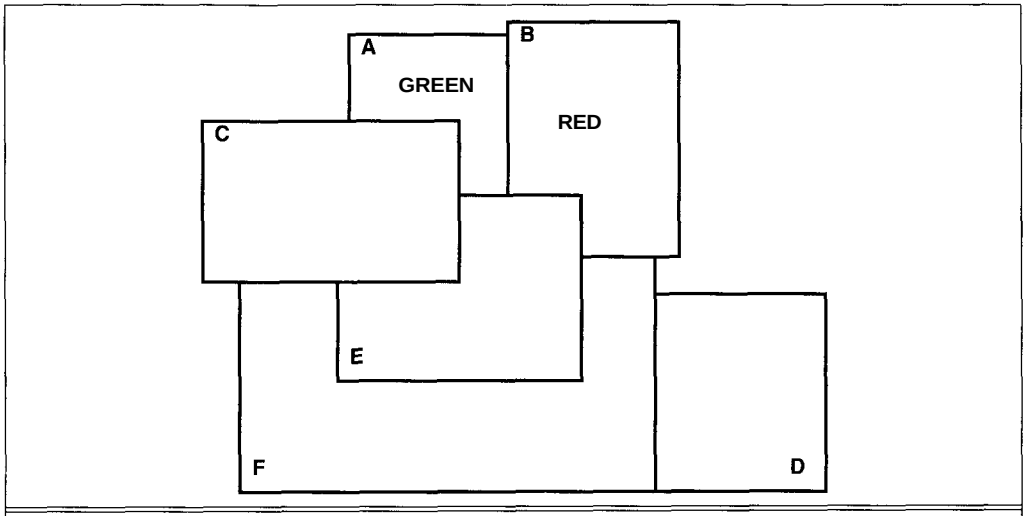


Figure 4.9 A map-coloring problem after the first two variables (A and B) have been selected. Which country should we color next?

MOST-CONSTRAINED-VARIABLE

This intuitive idea is called the **most-constrained-variable** heuristic. It is used with forward checking (see Section 3.7), which keeps track of which values are still allowed for each variable, given the choices made so far. At each point in the search, the variable with the *fewest* possible values is chosen to have a value assigned. In this way, the branching factor in the search tends to be minimized. For example, when this heuristic is used in solving n-queens problems, the feasible problem size is increased from around 30 for forward checking to approximately 100. The **most-constraining-variable** heuristic is similarly effective. It attempts to reduce the branching factor on future choices by assigning a value to the variable that is involved in the largest number of constraints on other unassigned variables.

MOST-CONSTRAINING-VARIABLE

Once a variable has been selected, we still need to choose a value for it. Suppose that we decide to assign a value to country C after A and B. One's intuition is that red is a better choice than blue, because it leaves more freedom for future choices. This intuition is the **least-constraining-value** heuristic—choose a value that rules out the smallest number of values in variables connected to the current variable by constraints. When applied to the n-queens problem, it allows problems up to n=1000 to be solved.

LEAST-CONSTRAINING-VALUE

4.3 MEMORY BOUNDED SEARCH

Despite all the clever search algorithms that have been invented, the fact remains that some problems are intrinsically difficult, by the nature of the problem. When we run up against these exponentially complex problems, something has to give. Figure 3.12 shows that *the first thing to give is usually the available memory.* In this section, we investigate two algorithms that are designed to conserve memory. The first, IDA*, is a logical extension of ITERATIVE-DEEPENING-SEARCH to use heuristic information. The second, SMA*, is similar to A*, but restricts the queue size to fit into the available memory.

Iterative deepening A* search (IDA*)

IDA*

In Chapter 3, we showed that iterative deepening is a useful technique for reducing memory requirements. We can try the same trick again, turning A* search into iterative deepening A*, or IDA* (see Figure 4.10). In this algorithm, each iteration is a depth-first search, just as in regular iterative deepening. The depth-first search is modified to use an f-cost limit rather than a depth limit. Thus, each iteration expands all nodes inside the contour for the current f-cost, peeping over the contour to find out where the next contour lies. (See the DFS-CONTOUR function in Figure 4.10.) Once the search inside a given contour has been completed, a new iteration is started using a new f-cost for the next contour.

IDA* is complete and optimal with the same caveats as A* search, but because it is depth-first, it only requires space proportional to the longest path that it explores. If b is the smallest operator cost and f^* the optimal solution cost, then in the worst case, IDA* will require bf^*/δ nodes of storage. In most cases, bd is a good estimate of the storage requirements.

The time complexity of IDA* depends strongly on the number of different values that the heuristic function can take on. The Manhattan distance heuristic used in the 8-puzzle takes on one of a small number of integer values. Typically, f only increases two or three times along any solution path. Thus, IDA* only goes through two or three iterations, and its efficiency is similar to that of A*—in fact, the last iteration of IDA* usually expands roughly the same number of nodes as A*. Furthermore, because IDA* does not need to insert and delete nodes on a priority queue, its overhead per node can be much less than that for A*. Optimal solutions for many practical problems were first found by IDA*, which for several years was the only optimal, memory-bounded, heuristic algorithm.

Unfortunately, IDA* has difficulty in more complex domains. In the travelling salesperson problem, for example, the heuristic value is different for every state. This means that each contour only includes one more state than the previous contour. If A* expands N nodes, IDA* will have to go through N iterations and will therefore expand $1 + 2 + \cdots + N = O(N^2)$ nodes. Now if N is too large for the computer's memory, then N^2 is almost certainly too long to wait!

One way around this is to increase the/-cost limit by a fixed amount ϵ on each iteration, so that the total number of iterations is proportional to $1/\epsilon$. This can reduce the search cost, at the expense of returning solutions that can be worse than optimal by at most ϵ. Such an algorithm is called **ϵ-admissible.**

ϵ-ADMISSIBLE

```
function IDA*(problem) returns a solution sequence
  inputs: problem, a problem
  static: f-limit, the current f-COST limit
          mot, a node

  root ← MAKE-NODE(INITIAL-STATE[problem])
  f-limit ← f-COST(root)
  loop do
     solution, f-limit — DFS-CONTOUR(root,f-limit)
     if solution is non-null then return solution
     itf-limit = ∞ then return failure; end

function DFS-CONTOUR(node,f-limit) returns a solution sequence and a new f-COST limit
  inputs: node, a node
          f-limit, the current f-COST limit
  static: next-f, the f-COST limit for the next contour, initially ∞

  if f-COST[node] > f-limit then return null, f-COST[node]
  if GOAL-TEST[problem](STATE[node]) then return node, f-limit
  for each node s in SUCCESSORS(node) do
     solution, new-f — DFS-CONTOUR(s,f-limit)
     if solution is non-null then return solution, f-limit
     next-f ← MIN(next-f, new-f); end
  return null, next-f
```

Figure 4.10 The IDA* (Iterative Deepening A*) search algorithm.

SMA* search

IDA*'s difficulties in certain problem spaces can be traced to using *too little* memory. Between iterations, it retains only a single number, the current f-cost limit. Because it cannot remember its history, IDA* is doomed to repeat it. This is doubly true in state spaces that are graphs rather than trees (see Section 3.6). IDA* can be modified to check the current path for repeated states, but is unable to avoid repeated states generated by alternative paths.

SMA*

In this section, we describe the SMA* (Simplified Memory-Bounded A*) algorithm, which can make use of all available memory to carry out the search. Using more memory can only improve search efficiency—onecould always ignore the additional space, but usually it is better to remember a node than to have to regenerate it when needed. SMA* has the following properties:

- It will utilize whatever memory is made available to it.
- It avoids repeated states as far as its memory allows.
- It is complete if the available memory is sufficient to store the *shallowest* solution path.
- It is optimal if enough memory is available to store the shallowest optimal solution path. Otherwise, it returns the best solution that can be reached with the available memory.
- When enough memory is available for the entire search tree, the search is optimally efficient.

The one unresolved question is whether SMA* is always optimally efficient among all algorithms given the same heuristic information and the same memory allocation.

The design of SMA* is simple, at least in overview. When it needs to generate a successor but has no memory left, it will need to make space on the queue. To do this, it drops a node from the queue. Nodes that are dropped from the queue in this way are called **forgotten nodes.** It prefers to drop nodes that are unpromising—that is, nodes with high/-cost. To avoid reexploring subtrees that it has dropped from memory, it retains in the ancestor nodes information about the quality of the best path in the forgotten subtree. In this way, it *only* regenerates the subtree 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.

SMA* is best explained by an example, which is illustrated in Figure 4.11. The top of the figure shows the search space. Each node is labelled with $g + h$ —f values, and the goal nodes (D, F, I, J) are shown in squares. The aim is to find the lowest-cost goal node with enough memory for only *three* nodes. The stages of the search are shown in order, left to right, with each stage numbered according to the explanation that follows. Each node is labelled with its current f-cost, which is continuously maintained to reflect the least f-cost of any of its descendants.[4] Values in parentheses show the value of the best forgotten descendant. The algorithm proceeds as follows:

1. At each stage, one successor is added to the deepest lowest-/-cost node that has some successors not currently in the tree. The left child B is added to the root A.
2. Now $f(A)$ is still 12, so we add the right child G (f = 13). Now that we have seen all the children of A, we can update its f-cost to the minimum of its children, that is, 13. The memory is now full.
3. G is now designated for expansion, but we must first drop a node to make room. We drop the shallowest highest-/-cost leaf, that is, B. When we have done this, we note that A's best forgotten descendant has f = 15, as shown in parentheses. We then add H, with $f(H)$ = 18. Unfortunately, H is not a goal node, but the path to H uses up all the available memory. Hence, there is no way to find a solution through H, so we set $f(H) = \infty$.
4. G is expanded again. We drop H, and add I, with $f(I)$ = 24. Now we have seen both successors of G, with values of oo and 24, so $f(G)$ becomes 24. $f(A)$ becomes 15, the minimum of 15 (forgotten successor value) and 24. Notice that I is a goal node, but it might not be the best solution because A's/-cost is only 15.
5. A is once again the most promising node, so B is generated for the second time. We have found that the path through G was not so great after all.
6. C, the first successor of B, is a nongoal node at the maximum depth, so $f(C)$ = oo.
7. To look at the second successor, D, we first drop C. Then $f(D)$ = 20, and this value is inherited by B and A.
8. Now the deepest, lowest-/-cost node is D. D is therefore selected, and because it is a goal node, the search terminates.

[4] Values computed in this way are called **backed-up values.** Because $f(n)$ is supposed to be an estimate of the least-cost solution path through n, and a solution path through n is bound to go through one of n's descendants, backing up the least f-cost among the descendants is a sound policy.

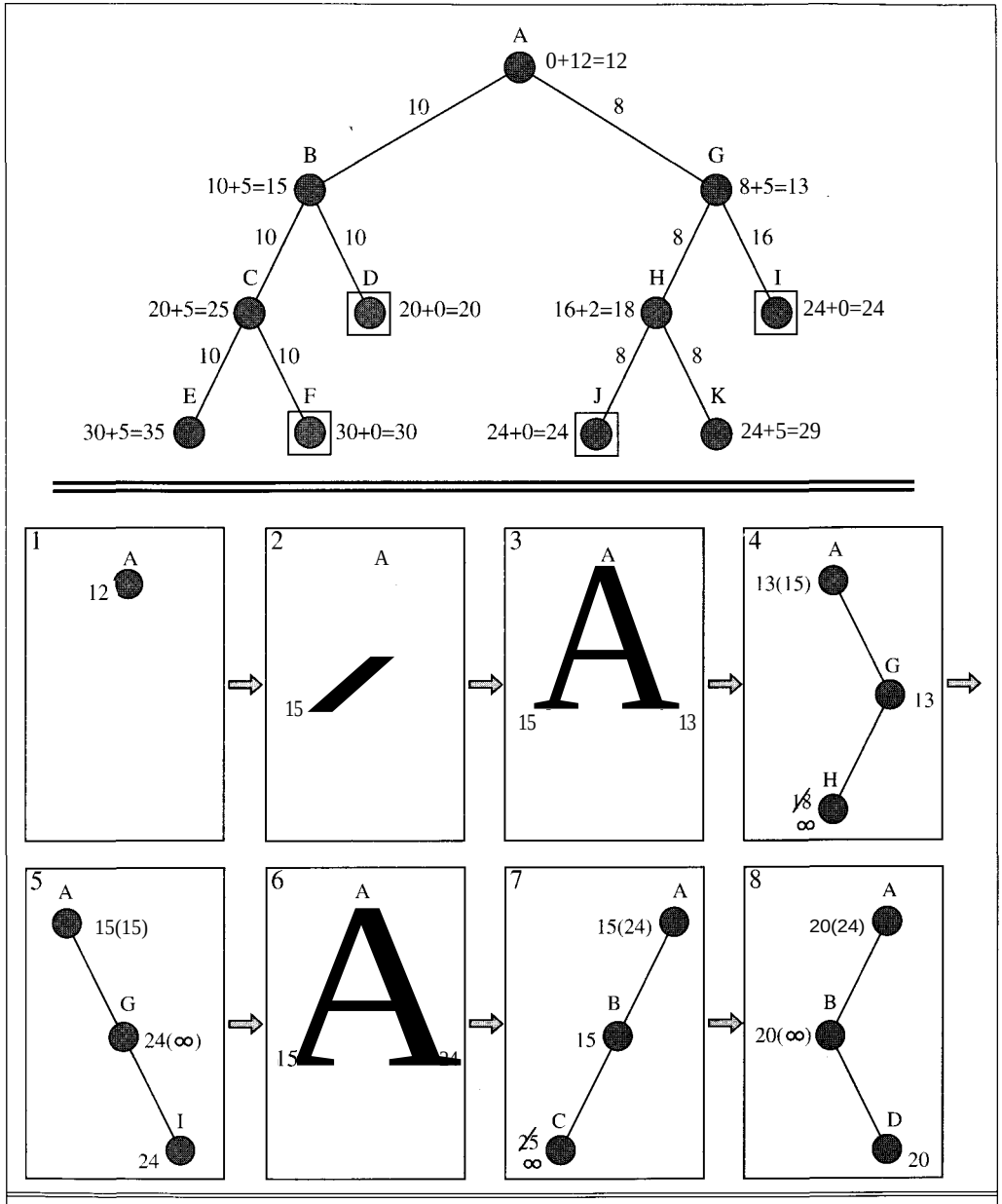


Figure 4.11 Progress of an SMA* search with a memory size of three nodes, on the state space shown at the top. Each node is labelled with its *current f*-cost. Values in parentheses show the value of the best forgotten descendant.

In this case, there is enough memory for the shallowest optimal solution path. If J had had a cost of 19 instead of 24, however, SMA* still would not have been able to find it because the solution path contains four nodes. In this case, SMA* would have returned D, which would be the best reachable solution. It is a simple matter to have the algorithm signal that the solution found may not be optimal.

A rough sketch of SMA* is shown in Figure 4.12. In the actual program, some gymnastics are necessary to deal with the fact that nodes sometimes end up with some successors in memory and some forgotten. When we need to check for repeated nodes, things get even more complicated. SMA* is the most complicated search algorithm we have seen yet.

```
function SMA*(problem) returns a solution sequence
  inputs: problem, a problem
  static: Queue, a queue of nodes ordered by f-cost

  Queue — MAKE-QUEUE({MAKE-NODE(INITIAL-STATE[problem])})
  loop do
    if Queue is empty then return failure
    n — deepest least-f-cost node in Queue
    if GOAL-TEST(n) then return success
    s — NEXT-SUCCESSOR(n)
    if s is not a goal and is at maximum depth then
        f(s) - ∞
    else
        f(s) ← MAX(f(n), g(s)+h(s))
    if all of n's successors have been generated then
        update n's f-cost and those of its ancestors if necessary
    if SUCCESSORS(n) all in memory then remove n from Queue
    if memory is full then
        delete shallowest, highest-f-cost node in Queue
        remove it from its parent's successor list
        insert its parent on Queue if necessary
    insert s on Queue
  end
```

Figure 4.12 Sketch of the SMA* algorithm. Note that numerous details have been omitted in the interests of clarity.

Given a reasonable amount of memory, SMA* can solve significantly more difficult problems than A* without incurring significant overhead in terms of extra nodes generated. It performs well on problems with highly connected state spaces and real-valued heuristics, on which IDA* has difficulty. On very hard problems, however, it will often be the case that SMA* is forced to continually switch back and forth between a set of candidate solution paths. 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 trade-off between time and memory, it seems that this is an inescapable problem. The only way out is to drop the optimality requirement.

4A ITERATIVE IMPROVEMENT ALGORITHMS

ITERATIVE IMPROVEMENT

We saw in Chapter 3 that several well-known problems (for example, 8-queens and VLSI layout) have the property that the state description itself contains all the information needed for a solution. The path by which the solution is reached is irrelevant. In such cases, **iterative improvement** algorithms often provide the most practical approach. For example, we start with all 8 queens on the board, or all wires routed through particular channels. Then, we might move queens around trying to reduce the number of attacks; or we might move a wire from one channel to another to reduce congestion. *The general idea is to start with a complete configuration and to make modifications to improve its quality.*

The best way to understand iterative improvement algorithms is to consider all the states laid out on the surface of a landscape. The height of any point on the landscape corresponds to the evaluation function of the state at that point (Figure 4.13). The idea of iterative improvement is to move around the landscape trying to find the highest peaks, which are the optimal solutions. Iterative improvement algorithms usually keep track of only the current state, and do not look ahead beyond the immediate neighbors of that state. This resembles trying to find the top of Mount Everest in a thick fog while suffering from amnesia. Nonetheless, sometimes iterative improvement is the method of choice for hard, practical problems. We will see several applications in later chapters, particularly to neural network learning in Chapter 19.

HILL-CLIMBING
GRADIENT DESCENT
SIMULATED ANNEALING

Iterative improvement algorithms divide into two major classes. **Hill-climbing** (or, alternatively, **gradient descent** if we view the evaluation function as a cost rather than a quality) algorithms always try to make changes that improve the current state. **Simulated annealing** algorithms can sometimes make changes that make things worse, at least temporarily.

Hill-climbing search

The hill-climbing search algorithm is shown in Figure 4.14. It is simply a loop that continually moves in the direction of increasing value. The algorithm does not maintain a search tree, so the node data structure need only record the state and its evaluation, which we denote by VALUE. One important refinement is that when there is more than one best successor to choose from, the algorithm can select among them at random. This simple policy has three well-known drawbacks:

- ◇ **Local maxima:** a local maximum, as opposed to a global maximum, is a peak that is lower than the highest peak in the state space. Once on a local maximum, the algorithm will halt even though the solution may be far from satisfactory.
- ◇ **Plateaux:** a plateau is an area of the state space where the evaluation function is essentially flat. The search will conduct a random walk.

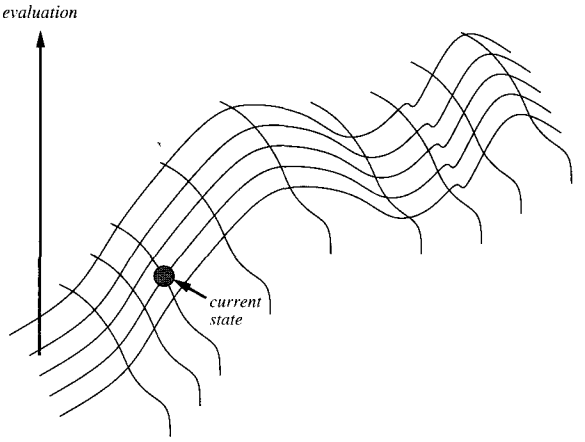


Figure 4.13 Iterative improvement algorithms try to find peaks on a surface of states where height is defined by the evaluation function.

```
function HILL-CLIMBING(problem) returns a solution state
  inputs: problem, a problem
  static: current, a node
          next, a node

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

Figure 4.14 The hill-climbing search algorithm.

◇ **Ridges:** a ridge may have steeply sloping sides, so that the search reaches the top of the ridge with ease, but the top may slope only very gently toward a peak. Unless there happen to be operators that move directly along the top of the ridge, the search may oscillate from side to side, making little progress.

RANDOM-RESTART HILL-CLIMBING

In each case, the algorithm reaches a point at which no progress is being made. If this happens, an obvious thing to do is start again from a different starting point. **Random-restart hill-climbing** does just this: it conducts a series of hill-climbing searches from randomly generated initial states, running each until it halts or makes no discernible progress. It saves the best result found so far from any of the searches. It can use a fixed number of iterations, or can continue until the best saved result has not been improved for a certain number of iterations.

Clearly, if enough iterations are allowed, random-restart hill-climbing will eventually find the optimal solution. The success of hill-climbing depends very much on the shape of the state-space "surface": if there are only a few local maxima, random-restart hill-climbing will find a good solution very quickly. A realistic problems has surface that looks more like a porcupine. If the problem is NP-complete, then in all likelihood we cannot do better than exponential time. It follows that there must be an exponential number of local maxima to get stuck on. Usually, however, a reasonably good solution can be found after a small number of iterations.

Simulated annealing

Instead of starting again randomly when stuck on a local maximum, we could allow the search to take some downhill steps to escape the local maximum. This is roughly the idea **of simulated annealing** (Figure 4.15). The innermost loop of simulated annealing is quite similar to hill-climbing. Instead of picking the *best* move, however, it picks a *random* move. If the move actually improves the situation, it is always executed. Otherwise, the algorithm makes the move with some probability less than 1. The probability decreases exponentially with the "badness" of the move—the amount AE by which the evaluation is worsened.

A second parameter T is also used to determine the probability. At higher values of T, "bad" moves are more likely to be allowed. As T tends to zero, they become more and more unlikely, until the algorithm behaves more or less like hill-climbing. The *schedule* input determines the value of T as a function of how many cycles already have been completed.

The reader by now may have guessed that the name "simulated annealing" and the parameter names ΔE and T were chosen for a good reason. The algorithm was developed from an explicit analogy with **annealing**—the process of gradually cooling a liquid until it freezes. The VALUE function corresponds to the total energy of the atoms in the material, and T corresponds to the

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"
  static: 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 4.15 The simulated annealing search algorithm.

temperature. The *schedule* determines the rate at which the temperature is lowered. Individual moves in the state space correspond to random fluctuations due to thermal noise. One can prove that if the temperature is lowered sufficiently slowly, the material will attain a lowest-energy (perfectly ordered) configuration. This corresponds to the statement that if *schedule* lowers T slowly enough, the algorithm will find a global optimum.

Simulated annealing was first used extensively to solve VLSI layout problems in the early 1980s. Since then, it has been applied widely to factory scheduling and other large-scale optimization tasks. In Exercise 4.12, you are asked to compare its performance to that of random-restart hill-climbing on the n-queens puzzle.

Applications in constraint satisfaction problems

Constraint satisfaction problems (CSPs) can be solved by iterative improvement methods by first assigning values to all the variables, and then applying modification operators to move the configuration toward a solution. Modification operators simply assign a different value to a variable. For example, in the 8-queens problem, an initial state has all eight queens on the board, and an operator moves a queen from one square to another.

HEURISTIC REPAIR

Algorithms that solve CSPs in this fashion are often called **heuristic repair** methods, because they repair inconsistencies in the current configuration. 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. This is illustrated in Figure 4.16 for an 8-queens problem, which it solves in two steps.

MIN-CONFLICTS

Min-conflicts is surprisingly effective for many CSPs, and is able to solve the million-queens problem in an average of less than 50 steps. It has also 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 ten minutes. Min-conflicts is closely related to the GSAT algorithm described on page 182, which solves problems in propositional logic.

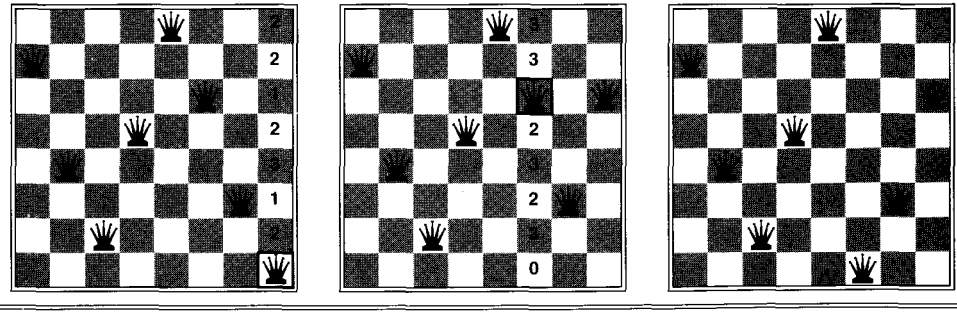


Figure 4.16 A two-step solution for an 8-queens problem using min-conflicts. At each stage, a queen is chosen for reassignment in its column. The number of conflicts (in this case, the number of attacking queens) is shown in each square. The algorithm moves the queen to the min-conflict square, breaking ties randomly.

4.5 SUMMARY

This chapter has examined the application **of heuristics** to reduce search costs. We have looked at 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 GENERAL-SEARCH where the minimum-cost nodes (according to some measure) are expanded first.
- If we minimize the estimated cost to reach the goal, $h(n)$, we get **greedy search.** The search time is usually decreased compared to an uninformed algorithm, but the algorithm is neither optimal nor complete.
- Minimizing $f(n) = g(n) + h(n)$ combines the the advantages of uniform-cost search and greedy search. If we handle repeated states and guarantee that $h(n)$ never overestimates, we get A* **search.**
- A* is complete, optimal, and optimally efficient among all optimal search algorithms. Its space complexity is still prohibitive.
- The time complexity of heuristic algorithms depends on the quality of the heuristic function. Good heuristics can sometimes be constructed by examining the problem definition or by generalizing from experience with the problem class.
- We can reduce the space requirement of A* with memory-bounded algorithms such as IDA* (iterative deepening A*) and SMA* (simplified memory-bounded A*).
- **Iterative improvement** algorithms keep only a single state in memory, but can get stuck on local maxima. Simulated annealing provides a way to escape local maxima, and is complete and optimal given a long enough cooling schedule.
- For constraint satisfaction problems, variable and value ordering heuristics can provide huge performance gains. Current algorithms often solve very large problems very quickly.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

The same paper that introduced the phrase "heuristic search" (Newell and Ernst, 1965) also introduced the concept of an evaluation function, understood as an estimate of the distance to the goal, to guide search; this concept was also proposed in the same year by Lin (1965). Doran and Michie (1966) did 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 used their heuristic functions as the sole guiding element in the search, ignoring the information provided by current path length that is used by uniform-cost search and by A*.

The A* algorithm, incorporating the current path length into heuristic search, was developed by Hart, Nilsson, and Raphael (1968). Certain subtle technical errors in the original presentation

of A* were corrected in a later paper (Hart *et al.*, 1972). An excellent summary of early work in search is provided by Nilsson (1971).

The original A* paper introduced a property of heuristics called "consistency." The monotonicity property of heuristics was introduced by Pohl (1977) as a simpler replacement for the consistency property. Pearl (1984) showed that consistency and monotonicity were equivalent properties. The pathmax equation was first used in A* search by Mero (1984).

Pohl (1970; 1977) pioneered the study of the relationship between the error in A*'s heuristic function 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).

A* and other state-space search algorithms are closely related to the *branch-and-bound* techniques that are widely used in operations research. An early survey of branch-and-bound techniques is given by Lawler and Wood (1966). The seminal paper by Held and Karp (1970) considers the use of the minimum-spanning-tree heuristic (see Exercise 4.11) for the travelling salesperson problem, showing how such admissible heuristics can be derived by examining relaxed problems. Generation of effective new heuristics by problem relaxation was successfully implemented by Prieditis (1993), building on earlier work with Jack Mostow (Mostow and Prieditis, 1989). The probabilistic interpretation of heuristics was investigated in depth by Hansson and Mayer (1989).

The relationships between state-space search and branch-and-bound have been investigated in depth by Dana Nau, Laveen Kanal, and Vipin Kumar (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." More material along these lines is found in Kumar (1991).

There are a large number of minor and major 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*, and dynamically adjusts the weights w_g and w_h according to certain criteria as the search progresses. Dynamic weighting usually cannot guarantee that optimal solutions will be found, as A* can, but under certain circumstances dynamic weighting can find solutions much more efficiently than A*.

The most-constrained-variable heuristic was introduced by Bitner and Reingold (1975), and further investigated by Purdom (1983). Empirical results on the n-queens problem were obtained by Stone and Stone (1986). Brelaz (1979) used the most-constraining-variable heuristic as a tie-breaker after applying the most-constrained-variable heuristic. The resulting algorithm, despite its simplicity, is still the best method for k-coloring arbitrary graphs. The least-constraining-value heuristic was developed and analyzed in Haralick and Elliot (1980). The min-conflicts heuristic was first proposed by Gu (1989), and was developed independently by Steve Minton (Minton *et al.*, 1992). Minton explains the remarkable performance of min-conflicts by modelling the search process as a random walk that is biased to move toward solutions. The effectiveness of algorithms such as min-conflicts and the related GSAT algorithm (see Exercise 6.15) in solving randomly

generated problems almost "instantaneously," despite the NP-completeness of the associated problem classes, has prompted an intensive investigation. It turns out that almost all randomly generated problems are either 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 (Kirkpatrick and Selman, 1994).

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. Doran and Michie's (1966) Graph Traverser, one of the earliest search programs, commits to an operator after searching best-first up to the memory limit. As with other "staged search" algorithms, optimality cannot be ensured because until the best path has been found the optimality of the first step remains in doubt. IDA* was the first widely used optimal, memory-bounded, heuristic search algorithm, and a large number of variants have been developed. The first reasonably public paper dealing specifically with IDA* was Korf's (1985b), although Korf credits the initial idea to a personal communication from Judea Pearl and also mentions Berliner and Goetsch's (1984) technical report describing their implementation of IDA* concurrently with Korf's own work. A more comprehensive exposition of IDA* can be found in Korf (1985a). A thorough analysis of the efficiency of IDA*, and its difficulties with real-valued heuristics, appears in Patrick *et al.* (1992). The SMA* algorithm described in the text was based on an earlier algorithm called MA* (Chakrabarti *et al,* 1989), and first appeared in Russell (1992). The latter paper also introduced the "contour" representation of search spaces. Kaindl and Khorsand (1994) apply SMA* to produce a bidirectional search algorithm that exhibits significant performance improvements over previous algorithms.

Three other memory-bounded algorithms deserve mention. RBFS (Korf, 1993) and IE (Russell, 1992) are two very similar algorithms that use linear space and a simple recursive formulation, like IDA*, but retain information from pruned branches to improve efficiency. Particularly in tree-structured search spaces with discrete-valued heuristics, they appear to be competitive with SMA* because of their reduced overhead. RBFS is also able to carry out a best-first search when the heuristic is inadmissible. Finally, **tabu search** algorithms (Glover, 1989), which maintain a bounded list of states that must not be revisited, have proved effective for optimization problems in operations research.

TABU SEARCH

Simulated annealing was first described by Kirkpatrick, Gelatt, and Vecchi (1983), who borrowed the algorithm directly from the **Metropolis algorithm** used to simulate complex systems in statistical physics (Metropolis *et al.*, 1953). Simulated annealing is now a subfield in itself, with several hundred papers published every year.

PARALLEL SEARCH

The topic of **parallel search** algorithms was not covered in the chapter, partly because it requires a lengthy discussion of parallel architectures. As parallel computers are becoming widely available, 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).

By far the most comprehensive source on 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 substantive and important articles on heuristic search. New results on search algorithms appear regularly in the journal *Artificial Intelligence.*

EXERCISES

4.1 Suppose that we run a greedy search algorithm with $h(n) = -g(n)$. What sort of search will the greedy search emulate?

4.2 Come up with heuristics for the following problems. Explain whether they are admissible, and whether the state spaces contain local maxima with your heuristic:

a. The general case of the chain problem (i.e., with an arbitrary goal state) from Exercise 3.15.
b. Algebraic equation solving (e.g., "solve $x^2y^3 = 3 - xy$ for x").
c. Path planning in the plane with rectangular obstacles (see also Exercise 4.13).
d. Maze problems, as defined in Chapter 3.

4.3 Consider the problem of constructing crossword puzzles: fitting words into a grid of intersecting horizontal and vertical squares. Assume that a list of words (i.e., a dictionary) is provided, and that the task is to fill in the squares using any subset of this list. Go through a complete goal and problem formulation for this domain, and choose a search strategy to solve it. Specify the heuristic function, if you think one is needed.

4.4 Sometimes there is no good evaluation function for a problem, but there is a good comparison method: a way to tell if one node is better than another, without assigning numerical values to either. Show that this is enough to do a best-first search. What properties of best-first search do we give up if we only have a comparison method?

4.5 We saw on page 95 that the straight-line distance heuristic is misleading 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?

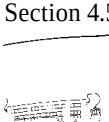
4.6 Invent a heuristic function for the 8-puzzle that sometimes overestimates, and show how it can lead to a suboptimal solution on a particular problem.

4.7 Prove that if the heuristic function h obeys the triangle inequality, then the f-cost along any path in the search tree is nondecreasing. (The triangle inequality says that the sum of the costs from A to B and B to C must not be less than the cost from A to C directly.)

4.8 We showed in the chapter that an admissible heuristic heuristic (when combined with pathmax) leads to monotonically nondecreasing f values along any path (i.e., $f(successor(n)) \geq f(n)$). Does the implication go the other way? That is, does monotonicity in f imply admissibility of the associated h?

4.9 We gave two simple heuristics for the 8-puzzle: Manhattan distance and misplaced tiles. Several heuristics in the literature purport to be better than either of these. (See, for example, Nilsson (1971) for additional improvements on Manhattan distance, and Mostow and Prieditis (1989) for heuristics derived by semimechanical methods.) Test these claims by implementing the heuristics and comparing the performance of the resulting algorithms.

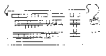
4.10 Would a bidirectional A* search be a good idea? Under what conditions would it be applicable? Describe how the algorithm would work.



4.11 The travelling salesperson problem (TSP) can be solved using 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 using 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.

4.12 Implement the n-queens problem and solve it using hill-climbing, hill-climbing with random restart, and simulated annealing. Measure the search cost and percentage of solved problems using randomly generated start states. Graph these against the difficulty of the problems (as measured by the optimal solution length). Comment on your results.



4.13 Consider the problem of finding the shortest path between two points on a plane that has convex polygonal obstacles as shown in Figure 4.17. 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. Implement any of the admissible algorithms discussed in the chapter. Keep the implementation of the algorithm *completely* independent of the specific domain. Then apply it to solve various problem instances in the domain.

4.14 In this question, we will turn the geometric scene from a simple data structure into a complete environment. Then we will put the agent in the environment and have it find its way to the goal.

a. Implement an evaluation environment as described in Chapter 2. The environment should behave as follows:
   - The percept at each cycle provides the agent with a list of the places that it can see from its current location. Each place is a position vector (with an x and y component) giving the coordinates of the place *relative to the agent.* Thus, if the agent is at (1,1) and there is a visible vertex at (7,3), the percept will include the position vector (6,2). (It therefore will be up to the agent to find out where it is! It can assume that all locations have a different "view.")

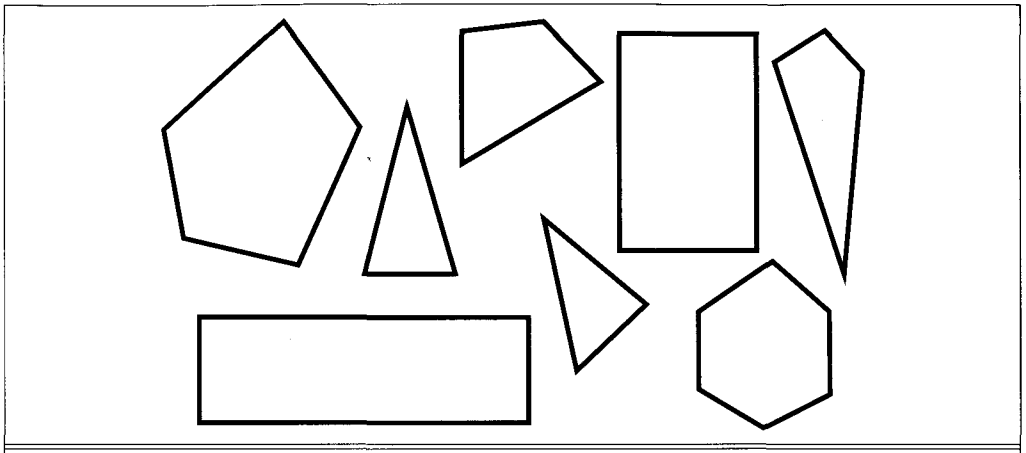


Figure 4.17 A scene with polygonal obstacles.

- The action returned by the agent will be the vector describing the straight-line path it wishes to follow (thus, the relative coordinates of the place it wishes to go). If the move does not bump into anything, the environment should "move" the agent and give it the percept appropriate to the next place; otherwise it stays put. If the agent wants to move (0,0) and is at the goal, then the environment should move the agent to a *random vertex in the scene.* (First pick a random polygon, and then a random vertex on that polygon.)

b. Implement an agent function that operates in this environment as follows:

- If it does not know where it is, it will need to calculate that from the percept.
- If it knows where it is and does not have a plan, it must calculate a plan to get home to the goal, using a search algorithm.
- Once it knows where it is and has a plan, it should output the appropriate action from the plan. It should say (0,0) once it gets to the goal.

c. Show the environment and agent operating together. The environment should print out some useful messages for the user showing what is going on.

d. 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 actual motion errors from which both humans and robots suffer. Modify the agent so that it always tries to get back on track when this happens. What it should do is this: if such an error is detected, first find out where it is and then modify its plan to first go back to where it was and resume the old plan. Remember that sometimes getting back to where it was may fail also! Show an example of the agent successfully overcoming two successive motion errors and still reaching the goal.

e. Now try two different recovery schemes after an error: head for the closest vertex on the original route; and replan a route to the goal from the new location. Compare the

performance of the three recovery schemes using a variety of exchange rates between search cost and path cost.

4.15 In this exercise, we will examine hill-climbing in the context of robot navigation, using the environment in Figure 4.17 as an example.

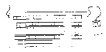
a. Repeat Exercise 4.14 using hill-climbing. Does your agent ever get stuck on a local maximum? Is it *possible* for it to get stuck with convex obstacles?

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 maxima?

4.16 Prove that IDA* returns optimal solutions whenever there is sufficient memory for the longest path with cost $< f^*$. Could it be modified along the lines of SMA* to succeed even with enough memory for only the shortest solution path?



4.17 Compare the performance of A*, SMA*, and IDA* on a set of randomly generated problems in the 8-puzzle (with Manhattan distance) and TSP (with minimum spanning tree) domains. Discuss your results. What happens to the performance of IDA* when a small random number is added to the heuristic values in the 8-puzzle domain?

4.18 Proofs of properties of SMA*:

a. SMA* abandons paths that fill up memory by themselves but do not contain a solution. Show that without this check, SMA" will get stuck in an infinite loop whenever it does not have enough memory for the shortest solution path.

b. Prove that SMA* terminates in a finite space or if there is a finite path to a goal. The proof will work by showing that it can never generate the same tree twice. This follows from the fact that between any two expansions of the same node, the node's parent must increase its f-cost. We will prove this fact by a downward induction on the depth of the node.

(i) Show that the property holds for any node at depth $d = MAX$.

(ii) Show that if it holds for all nodes at depth $d + 1$ or more, it must also hold for all nodes at depth d.

5 GAME PLAYING

In which we examine the problems that arise when we try to plan ahead in a world that includes a hostile agent.

5.1 INTRODUCTION: GAMES AS SEARCH PROBLEMS

Games have engaged the intellectual faculties of humans—sometimes to an alarming degree—for as long as civilization has existed. Board games such as chess and Go are interesting in part because they offer pure, abstract competition, without the fuss and bother of mustering up two armies and going to war. It is this abstraction that makes game playing an appealing target of AI research. The state of a game is easy to represent, and agents are usually restricted to a fairly small number of well-defined actions. *That makes game playing an idealization of worlds in which hostile agents act so as to diminish one's well-being.* Less abstract games, such as croquet or football, have not attracted much interest in the AI community.

Game playing is also one of the oldest areas of endeavor in artificial intelligence. In 1950, almost as soon as computers became programmable, the first chess programs were written by Claude Shannon (the inventor of information theory) and by Alan Turing. Since then, there has been steady progress in the standard of play, to the point where current systems can challenge the human world champion without fear of gross embarrassment.

Early researchers chose chess for several reasons. A chess-playing computer would be an existence proof of a machine doing something thought to require intelligence. Furthermore, the simplicity of the rules, and the fact that the world state is fully accessible to the program[1] means that it is easy to represent the game as a search through a space of possible game positions. The computer representation of the game actually can be correct in every relevant detail—unlike the representation of the problem of fighting a war, for example.

[1] Recall from Chapter 2 that **accessible** means that the agent can perceive everything there is to know about the environment. In game theory, chess is called a game of **perfect information.**

The presence of an opponent makes the decision problem somewhat more complicated than the search problems discussed in Chapter 3. The opponent introduces *uncertainty,* because one never knows what he or she is going to do. In essence, all game-playing programs must deal with the **contingency problem** defined in Chapter 3. The uncertainty is not like that introduced, say, by throwing dice or by the weather. The opponent will try as far as possible to make the least benign move, whereas the dice arid the weather are assumed (perhaps wrongly!) to be indifferent to the goals of the agent. This complication is discussed in Section 5.2.

But what makes games *really* different is that they are usually much too hard to solve. Chess, for example, 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} nodes (although there are "only" about 10^{40} *different* legal positions). Tic-Tac-Toe (noughts and crosses) is boring for adults precisely because it is easy to determine the right move. The complexity of games introduces a completely new kind of uncertainty that we have not seen so far; the uncertainty arises not because there is missing information, but because one does not have time to calculate the exact consequences of any move. Instead, one has to make one's best guess based on past experience, and act before one is sure of what action to take. In this respect, *games are much more like the real world than the standard search problems we have looked at so far.*

Because they usually have time limits, games also penalize inefficiency very severely. Whereas an implementation of A* search that is 10% less efficient will simply cost a little bit extra to run to completion, a chess program that is 10% less effective 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 use of time to reach good decisions, when reaching optimal decisions is impossible. These ideas should be kept in mind throughout the rest of the book, because the problems of complexity arise in every area of AI. We will return to them in more detail in Chapter 16.

We begin our discussion by analyzing how to find the theoretically best move. 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 5.5 discusses games such as backgammon that include an element of chance. Finally, we look at how state-of-the-art game-playing programs succeed against strong human opposition.

5.2 PERFECT DECISIONS IN TWO-PERSON GAMES

We will consider the general case of a game with two players, whom we will call MAX and M1N, 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 (or sometimes 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 an indication of whose move it is.
- **A set** of **operators,** which define the legal moves that a player can make.

TERMINAL TEST

- A **terminal test,** which determines when the game is over. States where the game has ended are called **terminal states.**

PAYOFF FUNCTION

- A **utility function** (also called a **payoff function),** which gives a numeric value for the outcome of a game. In chess, the outcome is a win, loss, or draw, which we can represent by the values +1, −1, or 0. Some games have a wider variety of possible outcomes; for example, the payoffs in backgammon range from +192 to −192.

If this were a normal search problem, then all MAX would have to do is search for a sequence of moves that leads to a terminal state that is a winner (according to the utility function), and then go ahead and make the first move in the sequence. Unfortunately, MIN has something to say about it. MAX therefore must find a **strategy** that will lead to a winning terminal state regardless of what MIN does, where the strategy includes the correct move for MAX for each possible move by MIN. We will begin by showing how to find the optimal (or rational) strategy, even though normally we will not have enough time to compute it.

STRATEGY

Figure 5.1 shows part of the search tree for the game of Tic-Tac-Toe. From the initial state, MAX has a choice of nine possible moves. Play alternates between MAX placing x's and MIN placing o's until we reach leaf nodes corresponding to terminal states: states where 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.

Even a simple game like Tic-Tac-Toe is too complex to show the whole search tree, so we will switch to the absolutely trivial game in Figure 5.2. The possible moves for MAX are labelled A_1, A_2, and A_3. The possible replies to A_1 for MIN are A_{11}, $A\backslash 2$, A_{13}, and so on. This particular game ends after one move each by MAX and MIN. (In game parlance, we say this tree is one move deep, consisting of two half-moves or two **ply.)** The utilities of the terminal states in this game range from 2 to 14.

PLY

MINIMAX

The **minimax** algorithm is designed to determine the optimal strategy for MAX, and thus to decide what the best first move is. The algorithm consists of five steps:

- Generate the whole game tree, all the way down to the terminal states.
- Apply the utility function to each terminal state to get its value.
- Use the utility of the terminal states to determine the utility of the nodes one level higher up in the search tree. Consider the leftmost three leaf nodes in Figure 5.2. In the V node above it, MIN has the option to move, and the best MIN can do is choose A_{11}, which leads to the minimal outcome, 3. Thus, even though the utility function is not immediately applicable to this V node, we can assign it the utility value 3, under the assumption that MIN will do the right thing. By similar reasoning, the other two V nodes are assigned the utility value 2.
- Continue backing up the values from the leaf nodes toward the root, one layer at a time.
- Eventually, the backed-up values reach the top of the tree; at that point, MAX chooses the move that leads to the highest value. In the topmost A node of Figure 5.2, MAX has a choice of three moves that will lead to states with utility 3, 2, and 2, respectively. Thus, MAX's best opening move is A_1. This is called the **mininiax decision,** because it maximizes the utility under the assumption that the opponent will play perfectly to minimize it.

MINIMAX DECISION

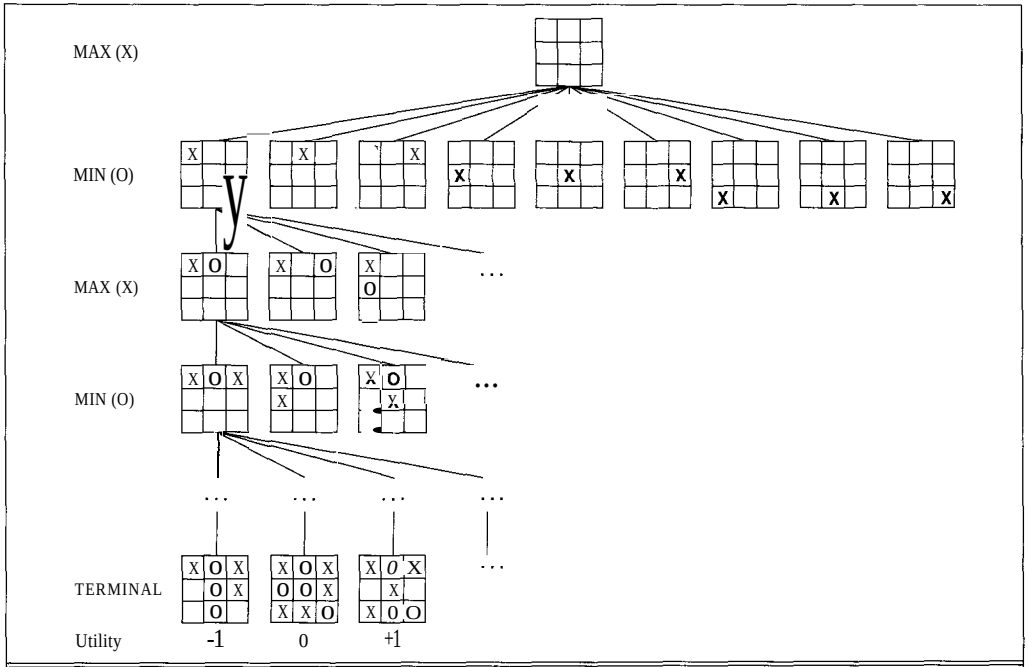


Figure 5.1 A (partial) search tree for the game of Tic-Tac-Toe. The top node is the initial state, and MAX moves first, placing an x in some square. We show part of the search tree, giving alternating moves by MIN (o) and MAX, until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

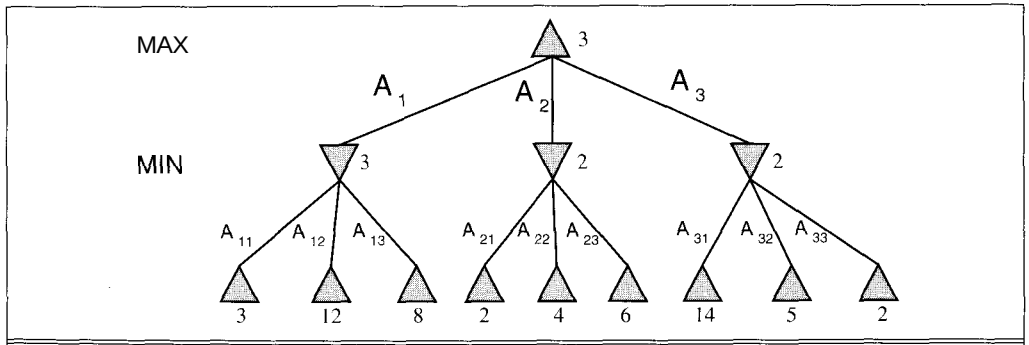


Figure 5.2 A two-ply game *tree* as generated by the minimax algorithm. The A nodes *ars* moves by MAX and the V nodes aremoves by MIN. The terminal nodes show the utility valuefor MAX computed by the utility function (i.e., by the rules of the game), whereas the utilities of the other nodes are computed by the minimax algorithm from the utilities of their successors. MAX'S best move is *A*\, and MIN's best reply is *A*\\.

Figure 5.3 shows a more formal description of the minimax algorithm. The top level function, MINIMAX-DECISION, selects from the available moves, which are evaluated in turn by the MINIMAX-VALUE function.

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 algorithm is a depth-first search (although here the implementation is through recursion rather than using a queue of nodes), so its space requirements are only linear in m and b. For real games, of course, the time cost is totally impractical, but this algorithm serves as the basis for more realistic methods and for the mathematical analysis of games.

```
function MINIMAX-DECISION(game) returns an operator

  for each op in OPERATORS[game] do
      VALUE[op] — MINIMAX-VALUE(APPLY(op, game), game)
  end
  return the op with the highest VALUE[op]

function MINIMAX-VALUE(state, game) returns a utility value

  if TERMINAL-TEST[game](state) then
      return  UTILITY[game](state)
  else if MAX is to move in state then
      return the highest MINIMAX-VALUE of SUCCESSORS(state)
  else
      return the lowest MINIMAX-VALUE of SUCCESSORS(state)
```

Figure 5.3 An algorithm for calculating minimax decisions. It returns the operator that 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 function MINIMAX-VALUE goes through the whole game tree, all the way to the leaves, to determine the backed-up value of a state.

5.3 IMPERFECT DECISIONS

The minimax algorithm assumes that the program has time to search all the way to terminal states, which is usually not practical. Shannon's original paper on chess proposed that instead of going all the way to terminal states and using the utility function, the program should cut off the search earlier and apply a heuristic **evaluation function** to the leaves of the tree. In other words, the suggestion is to alter minimax in two ways: the utility function is replaced by an evaluation function EVAL, and the terminal test is replaced by a cutoff test CUTOFF-TEST.

Evaluation functions

An evaluation function returns an *estimate* of the expected utility of the game from a given position. The idea was not new when Shannon proposed it. For centuries, chess players (and, of course, aficionados of other games) have developed ways of judging the winning chances of each side based on easily calculated features of a position. 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. All other things being equal, a side that has a secure material advantage of a pawn or more will probably win the game, and a 3-point advantage is sufficient for near-certain victory. Figure 5.4 shows four positions with their evaluations.

MATERIAL VALUE

It should be clear that the performance of a game-playing program is extremely dependent on the quality of its evaluation function. If it is inaccurate, then it will guide the program toward positions that are apparently "good," but in fact disastrous. How exactly do we measure quality?

First, the evaluation function must agree with the utility function on terminal states. Second, it must not take too long! (As mentioned in Chapter 4, if we did not limit its complexity, then it could call minimax as a subroutine and calculate the exact value of the position.) Hence, there is a trade-off between the accuracy of the evaluation function and its time cost. Third, an evaluation function should accurately reflect the actual chances of winning.

One might well wonder about the phrase "chances of winning." After all, chess is not a game of chance. But if we have cut off the search at a particular nonterminal state, then we do not know what will happen in subsequent moves. For concreteness, assume the evaluation function counts only material value. Then, in the opening position, the evaluation is 0, because both sides have the same material. All the positions up to the first capture will also have an evaluation of 0. If MAX manages to capture a bishop without losing a piece, then the resulting position will have an evaluation value of 3. The important point is that a given evaluation value covers many different positions—all the positions where MAX is up by a bishop are grouped together into a *category* that is given the label "3." Now we can see how the word "chance" makes sense: the evaluation function should reflect the chance that a position chosen at random from such a category leads to a win (or draw or loss), based on previous experience.[2]

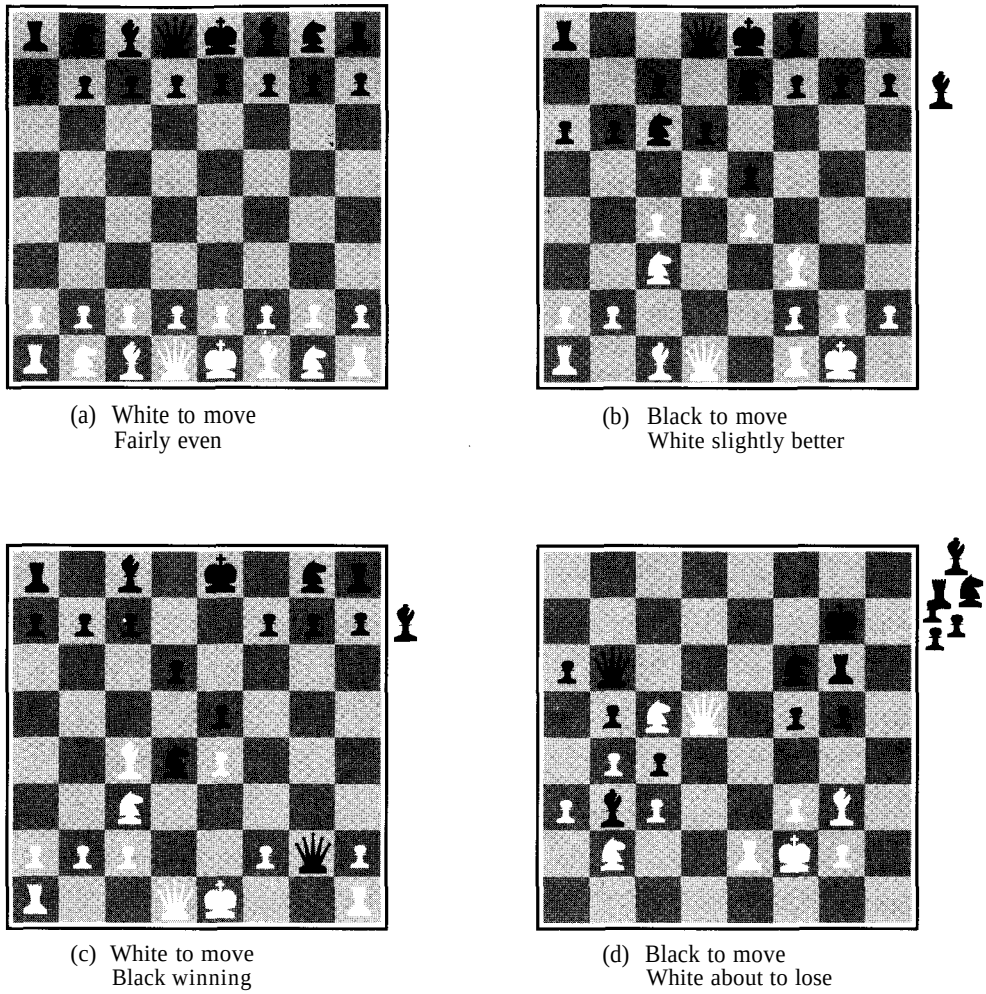
This suggests that the evaluation function should be specified by the rules of probability: if position A has a 100% chance of winning, it should have the evaluation 1.00, and if position B has a 50% chance of winning, 25% of losing, and 25% of being a draw, its evaluation should be $+1 \times .50 + -1 \times .25 + 0 \times .25 = .25$. But in fact, we need not be this precise; the actual numeric values of the evaluation function are not important, as long as A is rated higher than B.

The material advantage evaluation function assumes that the value of a piece can be judged independently of the other pieces present on the board. This kind of evaluation function is called a **weighted linear function,** because it can be expressed as

$$w_1 f_1 + w_2 f_2 + \cdots + w_n f_n$$

where the w's are the weights, and the f's are the features of the particular position. The w's would be the values of the pieces (1 for pawn, 3 for bishop, etc.), and the f's would be the numbers

2 Techniques for automatically constructing evaluation functions with this property are discussed in Chapter 18. In assessing the value of a category, more normally occurring positions should be given more weight.



(a) White to move
Fairly even

(b) Black to move
White slightly better

(c) White to move
Black winning

(d) Black to move
White about to lose

Figure 5.4 Some chess positions and their evaluations.

of each kind of piece on the board. Now we can see where the particular piece values come from: they give the best approximation to the likelihood of winning in the individual categories.

Most game-playing programs use a linear evaluation function, although recently nonlinear functions have had a good deal of success. (Chapter 19 gives an example of a neural network that is trained to learn a nonlinear evaluation function for backgammon.) In constructing the linear formula, one has to first pick the features, and then adjust the weights until the program plays well. The latter task can be automated by having the program play lots of games against itself, but at the moment, no one has a good idea of how to pick good features automatically.

Cutting off search

The most straightforward approach to controlling the amount of search is to set a fixed depth limit, so that the cutoff test succeeds for all nodes at or below depth *d*. The depth is chosen so that the amount of time used will not exceed what the rules of the game allow. A slightly 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.

These approaches can have some disastrous consequences because of 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 shown in Figure 5.4(d). According to the material function, white is ahead by a knight and therefore almost certain to win. However, because it is black's move, white's queen is lost because the black knight can capture it without any recompense for white. Thus, in reality the position is won for black, but this can only be seen by looking ahead one more ply.

QUIESCENT

Obviously, a more sophisticated cutoff test is needed. The evaluation function should only be applied 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

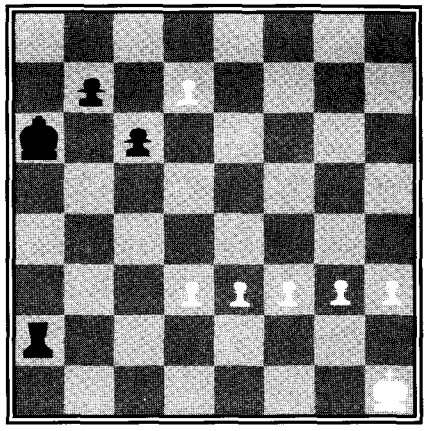
HORIZON PROBLEM

The horizon problem 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 5.5. Black is slightly ahead in material, but if white can advance its pawn from the seventh row to the eighth, it will become a queen and be an easy win for white. Black can forestall this for a dozen or so 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 horizon" to a place where it cannot be detected. At present, no general solution has been found for the horizon problem.

5.4 ALPHA-BETA PRUNING

Let us assume we have implemented a minimax search with a reasonable evaluation function for chess, and a reasonable cutoff test with a quiescence search. With a well-written program on an ordinary computer, one can probably search about 1000 positions a second. How well will our program play? In tournament chess, one gets about 150 seconds per move, so we can look at 150,000 positions. In chess, the branching factor is about 35, so our program will be able to look ahead only three or four ply, and will play at the level of a complete novice! Even average human players can make plans six or eight ply ahead, so our program will be easily fooled.

Fortunately, it is possible to compute the correct minimax decision without looking at every node in the search tree. The process of eliminating a branch of the search tree from consideration



Black to move

Figure 5.5 The horizon problem. 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 slight advantage for black, when it is really a sure win for white.

PRUNING
ALPHA-BETA PRUNING

without examining it is called **pruning** the search tree. 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 the two-ply game tree from Figure 5.2, shown again in Figure 5.6. The search proceeds as before: $A\backslash$, then $A\backslash\backslash$, $A\backslash 2$, A_{13}, and the node under A_1 gets minimax value 3. Now we follow $A2$, and A_{21}, which has value 2. At this point, we realize that if MAX plays A_2, MIN has the option of reaching a position worth 2, and some other options besides. Therefore, we can say already that move A_2 is worth *at most 2* to MAX. Because we already know that move $A\backslash$ is worth 3, there is no point at looking further under A_2. In other words, we can prune the search tree at this point and be confident that the pruning will have no effect on the outcome.

The general principle is this. Consider a node *n* somewhere in the tree (see Figure 5.7), 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. Let a be the value of the best choice we have found so far at any choice point along the path for MAX, and β be 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 value of a and 8 as it goes along, and prunes a subtree (i.e., terminates the recursive call) as soon as it is known to be worse than the current α or β value.

The algorithm description in Figure 5.8 is divided into a MAX-VALUE function and a MIN-VALUE function. These apply to MAX nodes and MIN nodes, respectively, but each does the same thing: return the minimax value of the node, except for nodes that are to be pruned (in

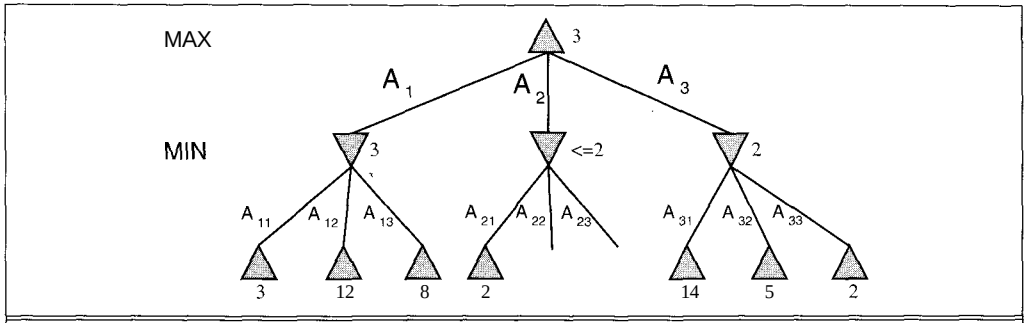


Figure 5.6 The two-ply game tree as generated by alpha-beta.

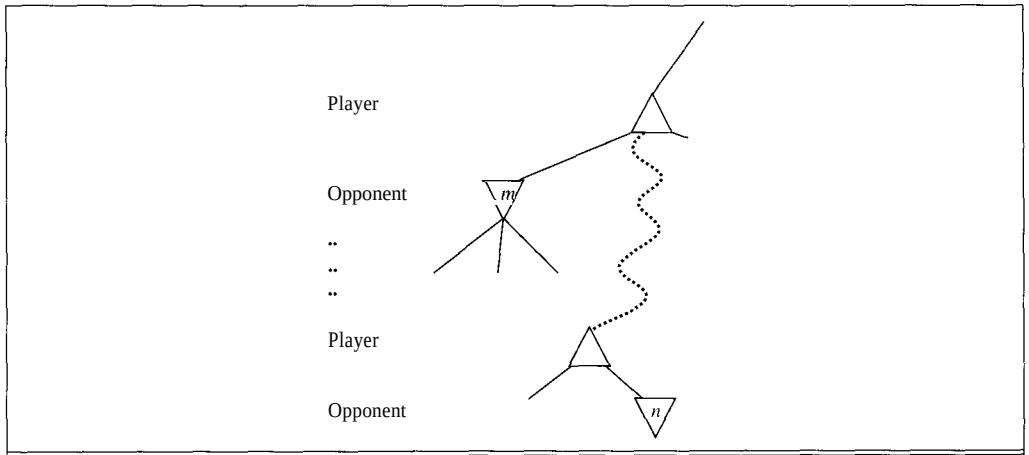


Figure 5.7 Alpha-beta pruning: the general case. If m is better than n for Player, we will never get to n in play.

which case the returned value is ignored anyway). The alpha-beta search function itself is just a copy of the MAX-VALUE function with extra code to remember and return the best move found.

Effectiveness of alpha-beta pruning

The effectiveness of alpha-beta depends on the ordering in which the successors are examined. This is clear from Figure 5.6, where we could not prune $A3$ at all because A_{31} and A_{32} (the worst moves from the point of view of MIN) were generated first. This suggests 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,[3] then it turns out that alpha-beta only needs to examine $O(b^{d/2})$ nodes to pick the best move, instead of $O(b^d)$ with minimax. This means that the effective branching factor is $\sqrt{b}$ instead of b—for chess, 6 instead of 35. Put another way, this means

[3] Obviously, it cannot be done perfectly, otherwise the ordering function could be used to play a perfect game!

```
function MAX-VALUE(stategame, a, β) returns the minimax value of state
  inputs: state, current state in game
          game, game description
          a, the best score for MAX along the path to state
          β, the best score for MIN along the path to state

  if CUTOFF-TEST(state) then return EVAL(state)
  for each s in SUCCESSORS(state) do
     a ← MAX(α , MIN-VALUE(s, game, a, β))
     if a ≥ ft then return ft
  end
  return a

function MIN-VALUE(state, game, a, ft) returns the minimax value of state

  if CUTOFF-TEST(state) then return EVAL(state)
  for each s in SUCCESSORS(state) do
     ft — MIN( ft, MAX-VALUE(s, game, a, β))
     if ft < a then return a
  end
  return ft
```

Figure 5.8 The alpha-beta search algorithm. It does the same computation as a normal minimax, but prunes the search tree.

that alpha-beta can look ahead twice as far as minimax for the same cost. Thus, by generating 150,000 nodes in the time allotment, a program can look ahead eight ply instead of four. By thinking carefully about *which computations actually affect the decision,* we are able to transform a program from a novice into an expert.

The effectiveness of alpha-beta pruning was first analyzed in depth by Knuth and Moore (1975). As well as the best case described in the previous paragraph, they analyzed the case in which successors are ordered randomly. It turns out that the asymptotic complexity is $O((b/\log b)^d)$,which seems rather dismal because the effective branching factor $b/\log b$ is not much less than b itself. On the other hand, the asymptotic formula is only accurate for $b > 1000$ or so—in other words, not for any games we can reasonably play using these techniques. For reasonable b, the total number of nodes examined will be roughly $O(b^{3d/4})$. In practice, a fairly simple ordering function (such as trying captures first, then threats, then forward moves, then backward moves) gets you fairly close to the best-case result rather than the random result. Another popular approach is to do an iterative deepening search, and use the backed-up values from one iteration to determine the ordering of successors in the next iteration.

TREE MODEL

It is also worth noting that all complexity results on games (and, in fact, on search problems in general) have to assume an idealized **tree model** in order to obtain their results. For example, the model used for the alpha-beta result in the previous paragraph assumes that all nodes have the same branching factor b; that all paths reach the fixed depth limit d; and that the leaf evaluations

are randomly distributed across the last layer of the tree. This last assumption is seriously flawed: for example, if a move higher up the tree is a disastrous blunder, then most of its descendants will look bad for the player who made the blunder. The value of a node is therefore likely to be highly correlated with the values of its siblings. The amount of correlation depends very much on the particular game and indeed the particular position at the root. Hence, there is an unavoidable component of *empirical science* involved in game-playing research, eluding the power of mathematical analysis.

5.5 GAMES THAT INCLUDE AN ELEMENT OF CHANCE

In real life, unlike chess, there are many unpredictable external events that put us into unforeseen situations. Many games mirror this unpredictability by including a random element such as throwing 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 set of legal moves that is available to the player. In the backgammon position of Figure 5.9, 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 complete game tree of the sort we saw in chess and Tic-Tac-Toe. A game tree in backgammon must include **chance nodes** in addition to MAX and MIN nodes. Chance nodes are shown as circles in Figure 5.10. The branches leading from each chance node denote the possible dice rolls, and each is labelled 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 fifteen distinct rolls a 1/18 chance.

CHANCE NODES

The next step is to understand how to make correct decisions. Obviously, we still want to pick the move from $A_1, \ldots, A_n$ that leads to the best position. However, each of the possible positions no longer has a definite minimax value (which in deterministic games was the utility of the leaf reached by best play). Instead, we can only calculate an average or **expected value,** where the average is taken over all the possible dice rolls that could occur.

EXPECTED VALUE

It is straightforward to calculate expected values of nodes. For terminal nodes, we use the utility function, just like in deterministic games. Going one step up in the search tree, we hit a chance node. In Figure 5.10, the chance nodes are circles; we will consider the one labelled C. Let *di* be a possible dice roll, and $P(d_i)$ be the chance or probability of obtaining that roll. For each dice roll, we calculate the utility of the best move for MIN, and then add up the utilities, weighted by the chance that the particular dice roll is obtained. If we let *S(C, di)* denote the set of positions generated by applying the legal moves for dice roll $P(d_i)$ to the position at *C*, then we can calculate the so-called **expectimax value** of *C* using the formula

EXPECTIMAX VALUE

$$expectimax(C) = \sum_i P(d_i) \max_{s \in S(C,d_i)} (utility(s))$$

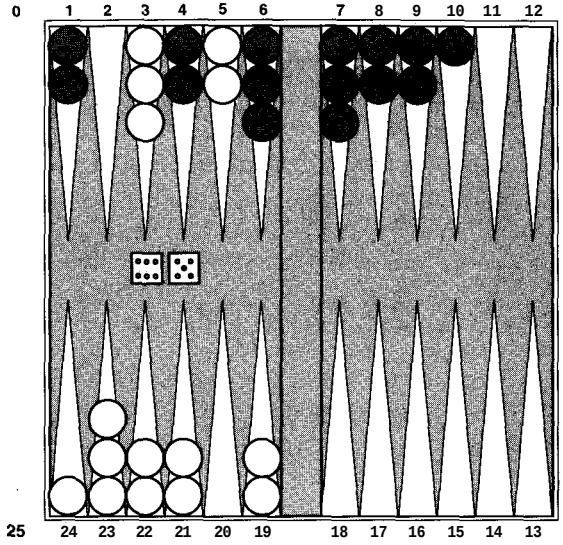


Figure 5.9 A typical backgammon position. The aim of the game is to move all one's pieces off the board. White moves clockwise toward 25, and black moves counterclockwise toward 0. A piece can move to any position except one where there are two or more of the opponent's pieces. If it moves to a position with one opponent piece, that piece is captured and has to start its journey again from the beginning. In this position, white has just rolled 6-5 and has four legal moves: (5-10,5-11), (5-11,19-24), (5-10,10-16), and (5-11,11-16).

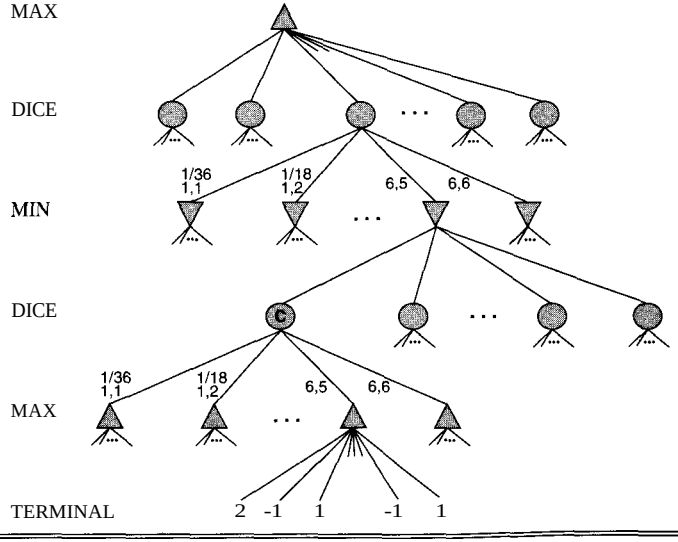


Figure 5.10 Schematic game tree for a backgammon position.

This gives us the expected utility of the position at C assuming best play. Going up one more level to the MIN nodes (∇ in Figure 5.10), we can now apply the normal minimax-value formula, because we have assigned utility values to all the chance nodes. We then move up to chance node B, where we can compute the **expectimin value** using a formula that is analogous to expectimax.

EXPECTIMIN VALUE

This process can be applied recursively all the way up the tree, except at the top level where the dice roll is already known. To calculate the best move, then, we simply replace MINIMAX-VALUE in Figure 5.3 by EXPECTIMINIMAX-VALUE, the implementation of which we leave as an exercise.

Position evaluation in games with chance nodes

As with minimax, the obvious approximation to make with expectiminimax is to cut off search at some point and apply an evaluation function to the leaves. One might think that evaluation functions for games such as backgammon are no different, in principle, from evaluation functions for chess—they should just give higher scores to better positions.

In fact, the presence of chance nodes means one has to be more careful about what the evaluation values mean. Remember that for minimax, any order-preserving transformation of the leaf values does not affect the choice of move. Thus, we can use either the values 1, 2, 3, 4 or the values 1, 20, 30, 400, and get the same decision. This gives us a good deal of freedom in designing the evaluation function: it will work fine as long as positions with higher evaluations lead to wins more often, on average.

With chance nodes, we lose this freedom. Figure 5.11 shows what happens: with leaf values 1, 2, 3, 4, move A_1 is best; with leaf 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 evaluation values! It turns out that to avoid this sensitivity, the evaluation function can be only *a positive linear* transformation of the likelihood 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 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 depth of the tree is limited to some small depth d, the extra cost compared to minimax makes it unrealistic to consider looking ahead very far in games such as backgammon, where n is 21 and b is usually around 20, but in some situations can be as high as 4000. Two ply 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.

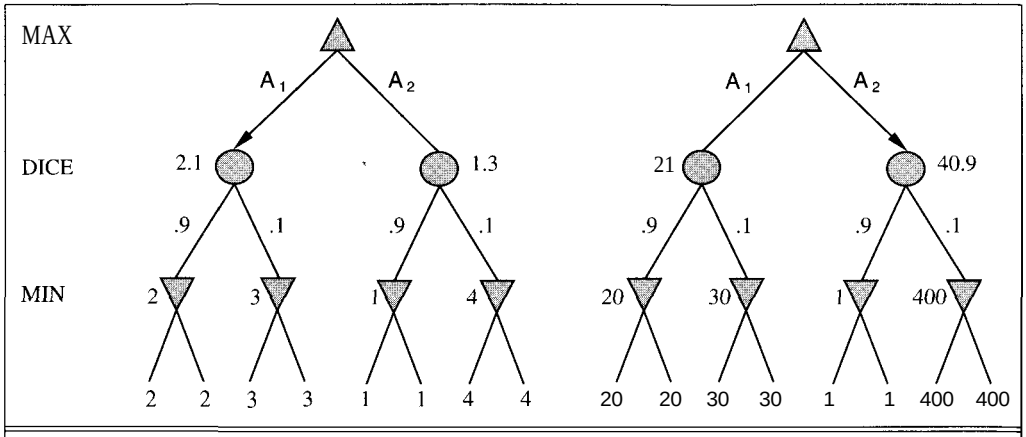


Figure 5.11 An order-preserving transformation on leaf values changes the best move.

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, with a bit of ingenuity. Consider the chance node *C* in Figure 5.10, and what happens to its value as we examine and evaluate its children; the question is, is it possible to find an upper bound on the value of *C* before we have looked at all its children? (Recall that this is what alpha-beta needs in order to prune a node and its subtree.) At first sight, it might seem impossible, because the value of C is the *average* of its children's values, and 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 boundaries on the possible values of the utility function, then we can arrive at boundaries for the average. For example, if we say that all utility values are between +1 and −1, 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. Designing the pruning process is a little bit more complicated than for alpha-beta, and we leave it as an exercise.

5.6 STATE-OF-THE-ART GAME PROGRAMS

Designing game-playing programs has a dual purpose: both to better understand how to choose actions in complex domains with uncertain outcomes and to develop high-performance systems for the particular game studied. In this section, we examine progress toward the latter goal.

Chess

Chess has received by far the largest share of attention in game playing. Although not meeting the promise made by Simon in 1957 that within 10 years, computers would beat the human world champion, they are now within reach of that goal. In speed chess, computers have defeated the world champion, Gary Kasparov, In both 5-minute and 25-minute games, but in full tournament games are only ranked among the top 100 players worldwide at the time of writing. Figure 5.12 shows the ratings of human and computer champions over the years. It is tempting to try to extrapolate and see where the lines will cross.

Progress beyond a mediocre level was initially very slow: some programs in the early 1970s became extremely complicated, with various kinds of tricks for eliminating some branches of search, generating plausible moves, and so on, but the programs that won the ACM North American Computer Chess Championships (initiated in 1970) tended to use straightforward alpha-beta search, augmented with book openings and infallible endgame algorithms. (This offers an interesting example of how high performance requires a hybrid decision-making architecture to implement the agent function.)

The first real jump in performance came not from better algorithms or evaluation functions, but from hardware. Belle, the first special-purpose chess computer (Condon and Thompson, 1982), used custom integrated circuits to implement move generation and position evaluation, enabling it to search several million positions to make a single move. Belle's rating was around 2250, on a scale where beginning humans are 1000 and the world champion around 2750; it became the first master-level program.

The HITECH system, also a special-purpose computer, was designed by former world correspondence champion Hans Berliner and his student Carl Ebeling to allow rapid calculation of very sophisticated evaluation functions. Generating about 10 million positions per move and using probably the most accurate evaluation of positions yet developed, HITECH became

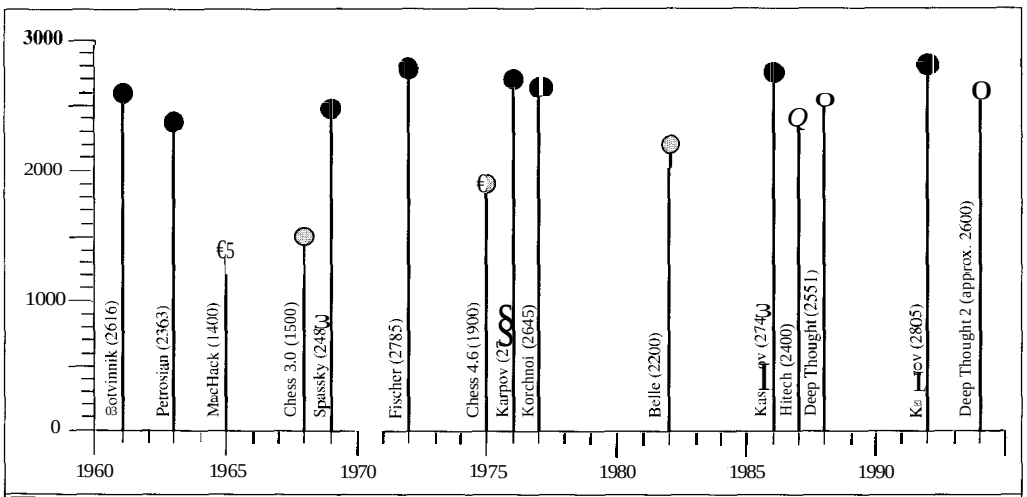


Figure 5.12 Ratings of human and machine chess champions.

computer world champion in 1985, and was the first program to defeat a human grandmaster, Arnold Denker, in 1987. At the time it ranked among the top 800 human players in the world.

The best current system is Deep Thought 2. It is sponsored by IBM, which hired part of the team that built the Deep Thought system at Carnegie Mellon University. Although Deep Thought 2 uses a simple evaluation function, it examines about half a billion positions per move, allowing it to reach depth 10 or 11, with a special provision to follow lines of forced moves still further (it once found a 37-move checkmate). In February 1993, Deep Thought 2 competed against the Danish Olympic team and won, 3–1, beating one grandmaster and drawing against another. Its FIDE rating is around 2600, placing it among the top 100 human players.

The next version of the system, Deep Blue, will use a parallel array of 1024 custom VLSI chips. This will enable it to search the equivalent of one billion positions per second (100-200 billion per move) and to reach depth 14. A 10-processor version is due to play the Israeli national team (one of the strongest in the world) in May 1995, and the full-scale system will challenge the world champion shortly thereafter.

Checkers or Draughts

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 20. Samuel's program began as a novice, but after only a few days' self-play was able to compete on equal terms in some very strong human tournaments. 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 cycle time of almost a millisecond, this remains one of the great feats of AI.

Few other people attempted to do better until Jonathan Schaeffer and colleagues developed Chinook, which runs on ordinary computers using alpha-beta search, but uses several techniques, including perfect solution databases for all six-piece positions, that make its endgame play devastating. Chinook won the 1992 U.S. Open, and became the first program to officially challenge for a real 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 21.5–18.5. More recently, 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.

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. Even so, Othello programs on normal computers are far better than humans, who generally refuse direct challenges in tournaments.

Backgammon

As mentioned before, the inclusion of uncertainty from dice rolls makes search an expensive luxury in backgammon. The first program to make a serious impact, BKG, used only a one-ply search but a very complicated evaluation function. In an informal match in 1980, it defeated the human world champion 5-1, but was quite lucky with the dice. Generally, it plays at a strong amateur level.

More recently, Gerry Tesauro (1992) combined Samuel's learning method with neural network techniques (Chapter 19) to develop a new evaluation function. His program is reliably ranked among the top three players in the world.

Go

Go is the most popular board game in Japan, requiring at least as much discipline from its professionals as chess. The branching factor approaches 360, so that regular search methods are totally lost. Systems based on large knowledge bases of rules for suggesting plausible moves seem to have some hope, but still play very poorly. Particularly given the $2,000,000 prize for the first program to defeat a top-level player, Go seems like an area likely to benefit from intensive investigation using more sophisticated reasoning methods.

5.7 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, it has 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, considering how to relax the assumptions and perhaps derive new insights.

First, let us consider minimax. Minimax is an optimal method for selecting a move from a given search tree *provided the leaf node evaluations are exactly correct.* In reality, evaluations are usually crude estimates of the value of a position, and can be considered to have large errors associated with them. Figure 5.13 shows a two-ply game tree for which minimax seems inappropriate. Minimax suggests taking the right-hand branch, whereas it is quite likely that true value of the left-hand branch is higher. The minimax choice relies on the assumption that *all* of the nodes labelled with values 100, 101, 102, and 100 are *actually* better than the node labelled with value 99. 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 and may require detailed correlation information that is hard to obtain.

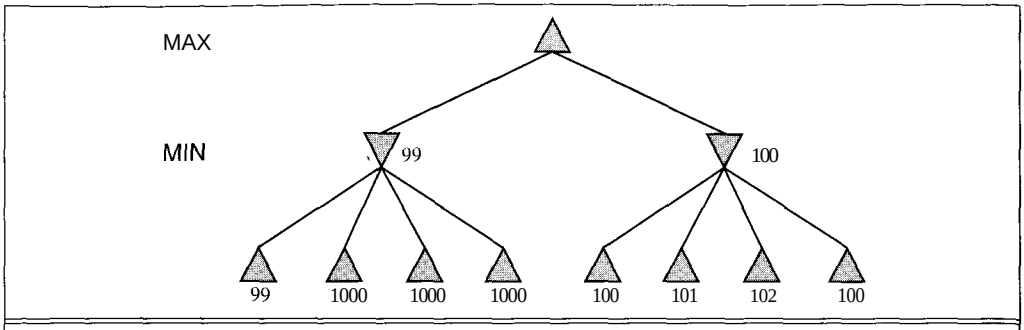


Figure 5.13 A two-ply game tree for which minimax may be inappropriate.

Next, we consider the search algorithm that generates the tree. The aim of an algorithm designer is to specify a computation that completes in a timely manner and results in a good move choice. 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 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 better used later for 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, where 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 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, however, there is no good understanding of how to *combine* the two kinds of algorithm into a robust and efficient system. Such a system would be a significant achievement not just for game-playing research, but also for AI research in general, because it would be much more likely to apply to the problem faced by a general intelligent agent.

5.8 SUMMARY

Games are fascinating, and writing game-playing programs perhaps even more so. We might say that game playing is to AI as Grand Prix motor racing is to the car industry: although the specialized task and extreme competitive pressure lead one to design systems that do not look much like your garden-variety, general-purpose intelligent system, a lot of leading-edge concepts and engineering ideas come out of it. On the other hand, just as you would not expect a Grand Prix racing car to perform well on a bumpy dirt road, you should not expect advances in game playing to translate immediately into advances in less abstract domains.

The most important ideas are as follows:

- A game can be defined by the initial state (how the board is set up), the operators (which define the legal moves), a terminal test (which says when the game is over), and a utility or payoff function (which says who won, and by how much).
- In two-player games with perfect information, the **minimax** algorithm can determine the best move for a player (assuming the opponent plays perfectly) by enumerating the entire game tree.
- **The alpha-beta** algorithm does the same calculation as minimax, but is more efficient because it prunes away branches of the search tree that it can prove are irrelevant to the final outcome.
- Usually, it is not feasible to consider the whole game tree (even with alpha-beta), so we need to cut off the search 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 chance nodes by taking the average utility of all its children nodes, weighted by the probability of the child.

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 "Turk," exhibited in 1769, a supposed chess-playing automaton whose cabinet actually concealed a diminutive human chess expert during play. The Turk is described in Harkness and Battell (1947). In 1846, Charles Babbage appears to have contributed the first serious discussion of the feasibility of computer game playing

(Morrison and Morrison, 1961). He believed that if his most ambitious design for a mechanical digital computer, the Analytical Engine, were ever completed, it could be programmed to play checkers and chess. He also designed, but did not build, a special-purpose machine for playing Tic-Tac-Toe. Ernst Zermelo, the designer of modern axiomatic set theory, later speculated on the rather quixotic possibility of searching the entire game tree for chess in order to determine a perfect strategy (Zermelo, 1976). The first functioning (and nonfraudulent) game-playing machine was designed and built around 1890 by the Spanish engineer Leonardo Torres y Quevedo. It specialized in the "KRK" chess endgame (king and rook vs. king), playing the side with the king and rook against a human opponent attempting to defend with the lone king. Its play was correct and it was capable of forcing mate from any starting position (with the machine moving first). The "Nimotron" (Condon *et al.*, 1940) demonstrated perfect play for the very simple game of Nim. Significantly, a completely optimal strategy for Nim and an adequate strategy for the KRK chess endgame (i.e., one which will always win when given the first move, although not necessarily in the minimal number of moves) are both simple enough to be memorized and executed algorithmically by humans.

Torres y Quevedo's achievement, and even Babbage's and Zermelo's speculations, remained relatively isolated until the mid-1940s—the era when programmable electronic digital computers were first being developed. The comprehensive theoretical analysis of game strategy in *Theory of Games and Economic Behavior* (Von Neumann and Morgenstern, 1944) placed emphasis on minimaxing (without any depth cutoff) as a way to define mathematically the game-theoretic value of a position in a game. Konrad Zuse (1945), the first person to design a programmable computer, developed ideas as to how mechanical chess play might be accomplished. Adriaan de Groot (1946) carried out in-depth psychological analysis of human chess strategy, which was useful to designers of computer chess programs. Norbert Wiener's (1948) book *Cybernetics* included a brief sketch of the functioning of a possible computer chess-playing program, including the idea of using minimax search with a depth cutoff and an evaluation function to select a move. Claude Shannon (1950) wrote a highly influential article that laid out the basic principles underlying modern computer game-playing programs, although the article did not actually include a program of his own. Shannon described minimaxing with a depth cutoff and evaluation function more clearly and in more detail than had Wiener, and introduced the notion of quiescence of a position. Shannon also described the possibility of using nonexhaustive ("type B") as opposed to exhaustive ("type A") minimaxing. Slater (1950) and the commentators on his article in the same volume also explored the possibilities for computer chess play. In particular, Good (1950) developed the notion of quiescence independently of Shannon.

In 1951, Alan Turing wrote the first actual computer program capable of playing a full game of chess. (The program was published in Turing (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.

Checkers, rather than chess, was the first of the classic games for which a program actually running on a computer was capable of playing out a full game. Christopher Strachey (1952) was the first to publish such research, although Slagle (1971) mentions a checkers program written by Arthur Samuel as early as 1947. Chinook, the checkers program that recently took over the world title from Marion Tinsley, is described by Schaeffer et al. (1992).

A group working at Los Alamos (Kister *et al.*, 1957) designed and ran a program that played a full game of a variant of chess using a 6 x 6 board. Alex Bernstein wrote the first program to play a full game of standard chess (Bernstein and Roberts, 1958; Bernstein *et al.*, 1958), unless possibly this feat was accomplished by the Russian BESM program mentioned in Newell *et al.* (1958), about which little information is available.

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-beta; it was the first chess program to do so. According to Nilsson (1971), Arthur Samuel's checkers program (Samuel, 1959; Samuel, 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) and by the MacHack 6 chess program (Greenblatt *et al.*, 1967). MacHack 6 was the first chess program to compete successfully with humans, although it fell considerably short of Herb Simon's prediction in 1957 that a computer program would be world chess champion within 10 years (Simon and Newell, 1958). Knuth and Moore (1975) provide a history of alpha-beta, along with a proof of its correctness and a time complexity analysis. Further analysis of the effective branching factor and time complexity of alpha-beta is given by Pearl (1982b). Pearl shows alpha-beta to be asymptotically optimal among all game-searching algorithms.

It would be a mistake to infer that alpha-beta's asymptotic optimality has completely suppressed interest in other game-searching algorithms. The best-known alternatives are probably the B* algorithm (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 as minimax and alpha-beta do, and SSS* (Stockman, 1979), which dominates alpha-beta in the sense that the set of nodes in the tree that it examines is a (sometimes proper) subset of those examined by alpha-beta. Palay (1985) uses probability distributions in place of the point values of alpha-beta or the intervals of B*. David McAllester's (1988) conspiracy number search is an interesting generalization of alpha-beta. MGSS* (Russell and Wefald, 1989) uses the advanced decision-theoretic techniques of Chapter 16 to decide which nodes to examine next, and was able to outplay an alpha-beta algorithm at Othello despite searching an order of magnitude fewer nodes. Individual games are subject to ad hoc mathematical analysis; a fascinating study of a huge number of games is given by Berlekamp et al. (1982).

D. F. Beal (1980) and Dana Nau (1980; 1983) independently and simultaneously showed that under certain assumptions about the game being analyzed, any form of minimaxing, including alpha-beta, using an evaluation function, yields estimates that are actually *less* reliable than the direct use of the evaluation function, without any search at all! *Heuristics* (Pearl, 1984) gives a thorough analysis of alpha-beta and describes B*, SSS*, and other alternative game search algorithms. It also explores the reasons for the Beal/Nau paradox, and why it does not apply to chess and other games commonly approached via automated game-tree search. Pearl also describes AND/OR graphs (Slagle, 1963a), which generalize game-tree search but can be applied to other types of problems as well, and the AO* algorithm (Martelli and Montanari, 1973; Martelli and Montanari, 1978) for searching them. Kaindl (1990) gives another survey of sophisticated search algorithms.

The first two computer chess programs to play a match against each other were the Kotok-McCarthy program and the "ITEP" program written at Moscow's Institute of Theoretical and Experimental Physics (Adelson-Velsky *et al.*, 1970). This intercontinental match was played by telegraph. It ended in 1967 with a 3-1 victory for the ITEP program. The first ACM North American Computer Chess Championship tournament was held in New York City in 1970. The first World Computer Chess Championship was held in 1974 in Stockholm (Hayes and Levy, 1976). It was won by Kaissa (Adelson-Velsky *et al.*, 1975), another program from ITEP.

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). Most of the strongest recent programs, such as HITECH (Ebeling, 1987; Berliner and Ebeling, 1989) and Deep Thought (Hsu *et al.*, 1990) have run on custom hardware. Major exceptions are Cray Blitz (Hyatt *et al.*, 1986), which runs on a general-purpose Cray supercomputer, and Socrates II, winner of the 23rd ACM North American Computer Chess Championship in 1993, which runs on an Intel 486-based microcomputer. It should be noted that Deep Thought was not there to defend its title. Deep Thought 2 regained the championship in 1994. It should also be noted that even custom-hardware machines can benefit greatly from improvements purely at the software level (Berliner, 1989).

The Fredkin Prize, established in 1980, offered $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, and the $10,000 prize by Deep Thought in 1989. The $100,000 prize remains unclaimed, in view of convincing wins in extended play by world champion Gary Kasparov over Deep Thought (Hsu *et al.*, 1990).

The literature for computer chess is far better developed than for any other game played by computer programs. Aside from the tournaments already mentioned, the rather misleadingly named conference proceedings *Heuristic Programming in Artificial Intelligence* report on the Computer Chess Olympiads. The International Computer Chess Association (ICCA), founded in 1977, publishes the quarterly *ICCA Journal.* Important papers have been published in the numbered serial anthology *Advances in Computer Chess,* starting with (Clarke, 1977). Some early general AI textbooks (Nilsson, 1971; Slagle, 1971) include extensive material on game-playing programs, including chess programs. David Levy's *Computer Chess Compendium* (Levy, 1988a) anthologizes many of the most important historical papers in the field, together with the scores of important games played by computer programs. The edited volume by Marsland and Schaeffer (1990) contains interesting historical and theoretical papers on chess and Go along with descriptions of Cray Blitz, HITECH, and Deep Thought. Several important papers on chess, along with material on almost all games for which computer game-playing programs have been written (including checkers, backgammon, Go, Othello, and several card games) can be found in Levy (1988b). There is even a textbook on how to write a computer game-playing program, by one of the major figures in computer chess (Levy, 1983).

The expectimax algorithm described in the text 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 backgammon program BKG (Berliner, 1977; Berliner, 1980b) was

the first program to defeat a human world champion at a major classic game (Berliner, 1980a), although Berliner was the first to acknowledge that this was a very short exhibition match (not a world championship match) and that BKG was very lucky with the dice.

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 search-based approach similar to that taken by most chess programs but with more selectivity to overcome the enormous branching factor. Zobrist (1970) used a pattern-recognition approach. Reitman and Wilcox (1979) used condition-action rules based on complex patterns, combined with highly selective localized search. The Go Explorer and its successors (Kierulf *et al.*, 1990) continue to evolve along these lines. YUGO (Shirayanagi, 1990) places heavy emphasis on knowledge representation and pattern knowledge. The *Computer Go Newsletter*, published by the Computer Go Association, describes current developments.

EXERCISES

5.1 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 thus 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. We will use a linear evaluation function defined as

$$Eval = 3X_2 + X_1 - (3O_2 + O_1)$$

a. Approximately how many possible games of Tic-Tac-Toe are there?

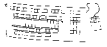
b. Show the whole game tree starting from an empty board down to depth 2, (i.e., one X and one O on the board), taking symmetry into account. You should have 3 positions at level 1 and 12 at level 2.

c. Mark on your tree the evaluations of all the positions at level 2.

d. Mark on your tree the backed-up values for the positions at levels 1 and 0, using the minimax algorithm, and use them to choose the best starting move.

e. Circle the nodes at level 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.*

5.2 Implement a general game-playing agent for two-player deterministic games, using alpha-beta search. You can assume the game is accessible, so the input to the agent is a complete description of the state.



5.3 Implement move generators and evaluation functions for one or more of the following games: kalah, Othello, checkers, chess. Exercise your game-playing agent using the 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?

5.4 The algorithms described in this chapter construct a search tree for each move from scratch. Discuss the advantages and disadvantages of retaining the search tree from one move to the next and extending the appropriate portion. How would tree retention interact with the use of selective search to examine "useful" branches of the tree?

5.5 Develop a formal proof of correctness of alpha-beta pruning. To do this, consider the situation shown in Figure 5.14. 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

$$n_1 = \min(n_2, n_{21}, \ldots, n_{2b_2})$$

By writing a similar expression for the value of n_2, find an expression for n_1 in terms of n_j.

b. Let l_i be the minimum (or maximum) of the node values to the left of node n_i at depth i. These are the nodes whose minimax value is already known. Similarly, let r_i be the minimum (or maximum) of the node values to the right of n_i at depth i. These nodes have not yet been explored. 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.

You might want to consult Wand (1980), who shows how the alpha-beta algorithm can be automatically synthesized from the minimax algorithm, using some general program-transformation techniques.

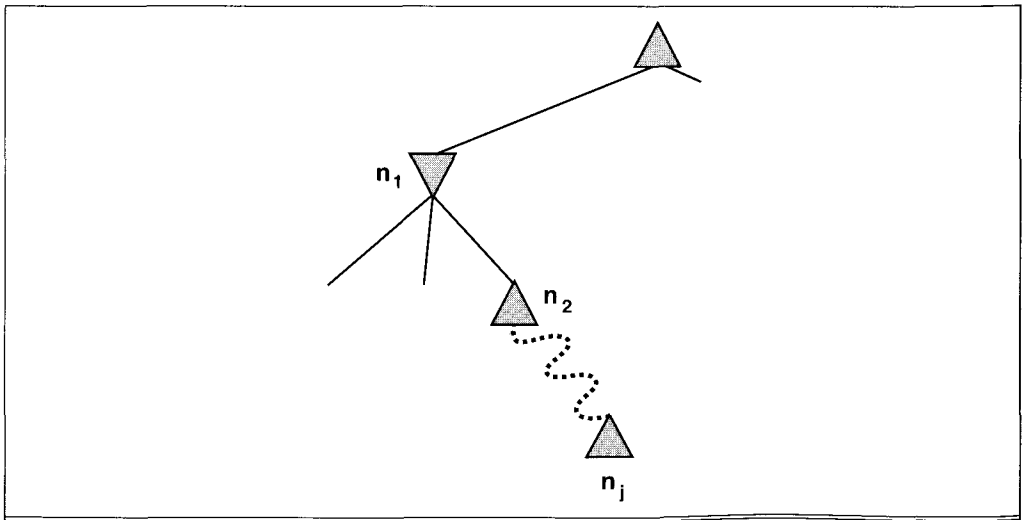


Figure 5.14 Situation when considering whether to prune node n_j.

5.6 Prove that with a positive linear transformation of leaf values, the move choice remains unchanged in a game tree with chance nodes.

5.7 Consider the following procedure for choosing moves in games with chance nodes:

- Generate a suitable number (say, 50) dice-roll sequences down to a suitable depth (say, 8).
- With known dice rolls, the game tree becomes deterministic. For each dice-roll sequence, solve the resulting deterministic game tree using alpha-beta.
- Use the results to estimate the value of each move and choose the best.

Will this procedure work correctly? Why (not)?

5.8 Let us consider the problem of search in a *three-player* game. (You can assume no alliances are allowed for now.) We will call the players 0, 1, and 2 for convenience. The first change is that the evaluation function will return a list of three values, indicating (say) the likelihood of winning for players 0, 1, and 2, respectively.

a. Complete the following game tree by filling in the backed-up value triples for all remaining nodes, including the root:

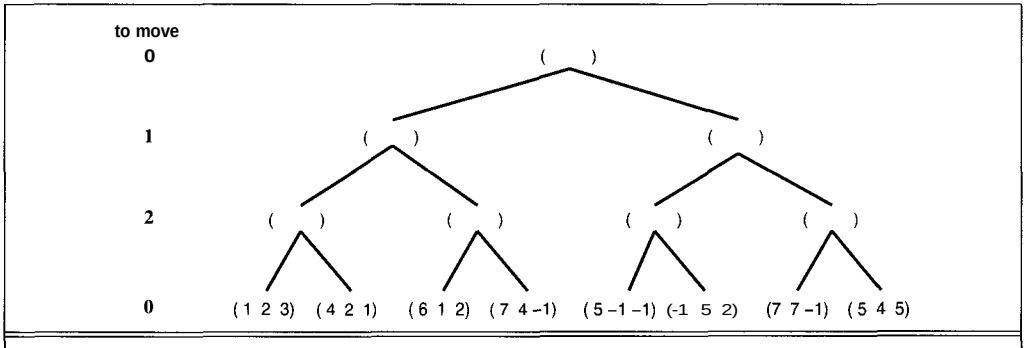


Figure 5.15 The first three ply of a game tree with three players (0, 1, and 2).

b. Rewrite MINIMAX-DECISION and MINIMAX-VALUE so that they work correctly for the three-player game.

c. Discuss the problems that might arise if players could form and terminate alliances as well as make moves "on the board." Indicate briefly how these problems might be addressed.

5.9 Describe and implement a general game-playing environment for an arbitrary number of players. Remember that time is part of the environment state, as well as the board position.

5.10 Suppose we play a variant of Tic-Tac-Toe in which each player sees only his or her own moves. If the player makes a move on a square occupied by an opponent, the board "beeps" and the player gets another try. Would the backgammon model suffice for this game, or would we need something more sophisticated? Why?

5.11 Describe and/or implement state descriptions, move generators, and evaluation functions for one or more of the following games: backgammon, Monopoly, Scrabble, bridge (declarer play is easiest).

5.12 Consider carefully the interplay of chance events and partial information in each of the games in Exercise 5.11.

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 5.7 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.

5.13 The Chinook checkers program makes extensive use of endgame databases, which provide exact values for every position within 6 moves of the end of the game. How might such databases be generated efficiently?

5.14 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.

5.15 For a game with which you are familiar, describe how an agent could be defined with condition-action rules, subgoals (and their conditions for generation), and action-utility rules, instead of by minimax search.

5.16 The minimax algorithm returns the best move for MAX under the assumption that MIN plays optimally. What happens when MIN plays suboptimally?

5.17 We have assumed that the rules of each game define a utility function that is used by both players, and that a utility of x for MAX means a utility of $-x$ for MIN. Games with this property are called **zero-sum** games. Describe how the minimax and alpha-beta algorithms change when we have nonzero-sum games—that is, when each player has his or her own utility function. You may assume that each player knows the other's utility function.