

# 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



© 1995 by Prentice-Hall, Inc.  
A Simon & Schuster Company  
Englewood Cliffs, New Jersey 07632

The author and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The author and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher.

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

<b>I</b>	<b>Artificial Intelligence</b>	<b>1</b>
1	<b>Introduction</b> .....	3
2	<b>Intelligent Agents</b> .....	31
<b>II</b>	<b>Problem-solving</b>	<b>53</b>
3	Solving Problems by Searching .....	55
4	Informed Search Methods .....	92
5	Game <b>Playing</b> .....	122
<b>III</b>	<b>Knowledge and reasoning</b>	<b>149</b>
6	Agents that Reason <b>Logically</b> .....	151
7	First-Order <b>Logic</b> .....	185
8	Building a Knowledge Base .....	217
9	Inference in First-Order <b>Logic</b> .....	265
10	Logical Reasoning <b>Systems</b> .....	297
<b>IV</b>	<b>Acting logically</b>	<b>335</b>
11	<b>Planning</b> .....	337
12	Practical Planning .....	367
13	Planning and <b>Acting</b> .....	392
<b>V</b>	<b>Uncertain knowledge and reasoning</b>	<b>413</b>
14	<b>Uncertainty</b> .....	415
15	Probabilistic Reasoning <b>Systems</b> .....	436
16	Making Simple Decisions .....	471
17	Making Complex Decisions .....	498
<b>VI</b>	<b>Learning</b>	<b>523</b>
18	Learning from <b>Observations</b> .....	525
19	Learning in Neural and Belief <b>Networks</b> .....	563
20	Reinforcement <b>Learning</b> .....	598
21	Knowledge in <b>Learning</b> .....	625
<b>VII</b>	<b>Communicating, perceiving, and acting</b>	<b>649</b>
22	Agents that Communicate .....	651
23	Practical Natural Language Processing .....	691
24	Perception .....	724
25	<b>Robotics</b> .....	773
<b>VIII</b>	<b>Conclusions</b>	<b>815</b>
26	Philosophical Foundations .....	817
27	<b>AI:</b> Present and Future .....	842
A	Complexity analysis and O() <b>notation</b> .....	851
B	Notes on Languages and <b>Algorithms</b> .....	854
	<b>Bibliography</b>	<b>859</b>
	<b>Index</b>	<b>905</b>

# Contents

<b>I Artificial Intelligence</b>	<b>1</b>
<b>1 Introduction</b>	<b>3</b>
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
<b>2 Intelligent Agents</b>	<b>31</b>
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
<b>II Problem-solving</b>	<b>53</b>
<b>3 Solving Problems by Searching</b>	<b>55</b>
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
<b>4 Informed Search Methods</b>	<b>92</b>
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
<b>5</b>	<b>Game Playing</b>	<b>122</b>
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
<b>III</b>	<b>Knowledge and reasoning</b>	<b>149</b>
<b>6</b>	<b>Agents that Reason Logically</b>	<b>151</b>
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
<b>7 First-Order Logic</b> . . . . .	<b>185</b>
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 $\exists!$ . . . . .	196
The uniqueness operator $\iota$ . . . . .	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
<b>8 Building a Knowledge Base</b>	<b>217</b>
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
<b>9 Inference in First-Order Logic</b>	<b>265</b>
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
<b>10</b>	<b>Logical Reasoning Systems</b>	<b>297</b>
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
<b>IV</b>	<b>Acting logically</b>	<b>335</b>
<b>11</b>	<b>Planning</b>	<b>337</b>
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
	<i>Shakey's</i> world . . . . .	360
11.9	Summary . . . . .	362
	Bibliographical and Historical Notes . . . . .	363
	Exercises . . . . .	364
<b>12</b>	<b>Practical Planning</b>	<b>367</b>
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
<b>13 Planning and Acting</b> . . . . .	<b>392</b>
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
<b>V Uncertain knowledge and reasoning</b> . . . . .	<b>413</b>
<b>14 Uncertainty</b> . . . . .	<b>415</b>
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
<b>15 Probabilistic Reasoning Systems</b>	<b>436</b>
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
<b>16 Making Simple Decisions</b>	<b>471</b>
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
<b>17 Making Complex Decisions</b>	<b>498</b>
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
<b>VI Learning</b>	<b>523</b>
<b>18 Learning from Observations</b>	<b>525</b>
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
<b>19</b>	<b>Learning in Neural and Belief Networks</b>	<b>563</b>
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
<b>20</b>	<b>Reinforcement Learning</b>	<b>598</b>
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
<b>21</b>	<b>Knowledge in Learning</b>	<b>625</b>
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
<b>VII</b>	<b>Communicating, perceiving, and acting</b>	<b>649</b>
<b>22</b>	<b>Agents that Communicate</b>	<b>651</b>
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
<b>23</b>	<b>Practical Natural Language Processing</b>	<b>691</b>
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
<b>24 Perception</b>	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: $P(\text{words})$ . . . . .	760
The acoustic model: $P(\text{signals} \text{words})$ . . . . .	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
<b>25 Robotics</b>	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
<b>VIII Conclusions</b>	<b>815</b>
<b>26 Philosophical Foundations</b>	<b>817</b>
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
<b>27 AI: Present and Future</b>	<b>842</b>
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
<b>A Complexity analysis and O() notation</b>	<b>851</b>
A.1 Asymptotic Analysis . . . . .	851
A.2 Inherently Hard Problems . . . . .	852
Bibliographical and Historical Notes . . . . .	853

<b>B Notes on Languages and Algorithms</b>	<b>854</b>
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
<b>Bibliography</b>	<b>859</b>
<b>Index</b>	<b>905</b>

# 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.<sup>11</sup> 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.

<sup>11</sup> 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.*

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.<sup>1</sup> 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 modern 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**. An AI 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.

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

RATIONALITY

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

<sup>2</sup> 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 . . . <i>machines with minds</i> , in the full and literal sense" (Haugeland, 1985)	"The study of mental faculties through the use of computational models" (Charniak and McDermott, 1985)				
"[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning . . ." (Bellman, 1978)	"The study of the computations that make it possible to perceive, reason, and act" (Winston, 1992)				
"The art of creating machines that perform functions that require intelligence when performed by people" (Kurzweil, 1990)	"A field of study that seeks to explain and emulate intelligent behavior in terms of computational processes" (Schalkoff, 1990)				
"The study of how to make computers do things at which, at the moment, people are better" (Rich and Knight, 1991)					
<p>Figure 1.1 Some definitions of AI. They are organized into four categories:</p> <table border="1"> <tr> <td>Systems that think like humans.</td> <td>Systems that think rationally.</td> </tr> <tr> <td>Systems that act like humans.</td> <td>Systems that act rationally.</td> </tr> </table>		Systems that think like humans.	Systems that think rationally.	Systems that act like humans.	Systems that act rationally.
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:

- 0 **natural language processing** to enable it to communicate successfully in English (or some other human language);
- ◇ **knowledge representation** to store information provided before or during the interrogation;
- ◇ **automated reasoning** to use the stored information to answer questions and to draw new conclusions;
- ◇ **machine learning** to adapt to new circumstances and to detect and extrapolate patterns.

### NATURAL LANGUAGE PROCESSING

### KNOWLEDGE REPRESENTATION AUTOMATED REASONING

### MACHINE LEARNING

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** 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, 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;

LOGIC

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**.

LOGICIST

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.

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.

AGENT

## Acting rationally: The rational agent approach

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

LIMITED  
RATIONALITY

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.

## 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,<sup>3</sup> "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."<sup>4</sup> 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

EMPIRICIST

INDUCTION

An alternative to dualism is **materialism**, which holds that all the world (including the brain and mind) operate according to physical law.<sup>5</sup> 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*,<sup>6</sup> 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**:

<sup>3</sup> 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.

<sup>4</sup> Note that other translations have "goodness/good" instead of "piety/pious."

<sup>5</sup> 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.

<sup>6</sup> An update of Aristotle's *organon*, or instrument of thought.

LOGICAL POSITIVISM  
OBSERVATION SENTENCES  
CONFIRMATION THEORY

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.<sup>7</sup> 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.

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.

<sup>7</sup> 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.

ALGORITHM

## 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.

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.<sup>8</sup> 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 Gödel (1906-1978) showed that there exists an effective procedure to prove any true statement in the first-order logic of Frege and Russell; but 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.

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-

INCOMPLETENESS  
THEOREM

INTRACTABILITY

<sup>8</sup> To understand why Frege's notation was not universally adopted, see the cover of this book.

REDUCTION

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. 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!

BEH

COG  
PSY

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*

## BEHAVIORISM

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.

## COGNITIVE PSYCHOLOGY

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<sup>9</sup> (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:

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 is 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

<sup>9</sup> 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,<sup>10</sup> 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 \$150 billion/year. In the United States, the computer industry (including software and services) now accounts for about 10% of the gross national product.

Each generation of computer hardware has brought an increase in speed and capacity, and a decrease in price. Computer engineering has been remarkably successful, regularly doubling performance every two years, with no immediate end in sight for this rate of increase. Massively 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

<sup>10</sup> 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.<sup>11</sup> 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.

<sup>11</sup> 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.<sup>12</sup> 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.

<sup>12</sup> 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,<sup>13</sup> 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."<sup>14</sup> 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.

<sup>13</sup> Now Carnegie Mellon University (CMU).

<sup>14</sup> 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**, which was to become the dominant AI programming language. Lisp is the second-oldest language in current use.<sup>15</sup> 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

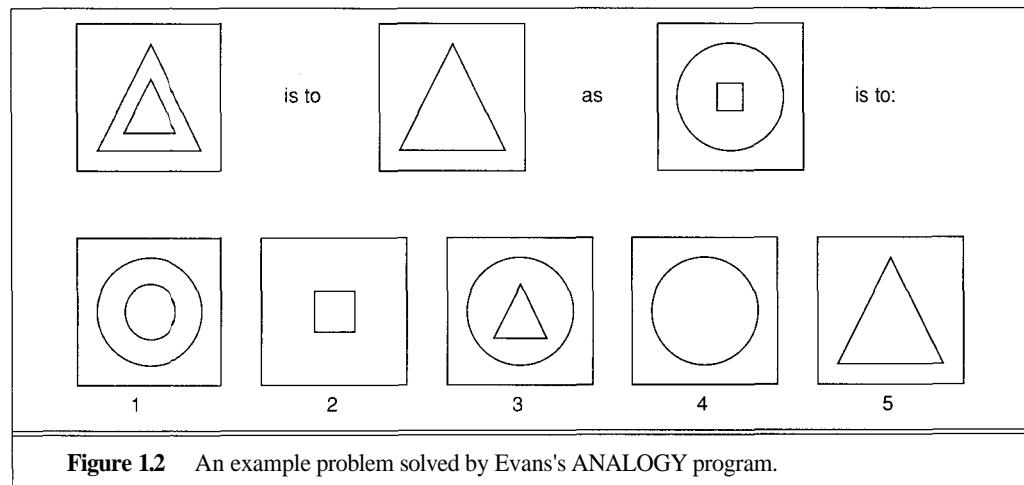
<sup>15</sup> FORTRAN is one year older than Lisp.

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

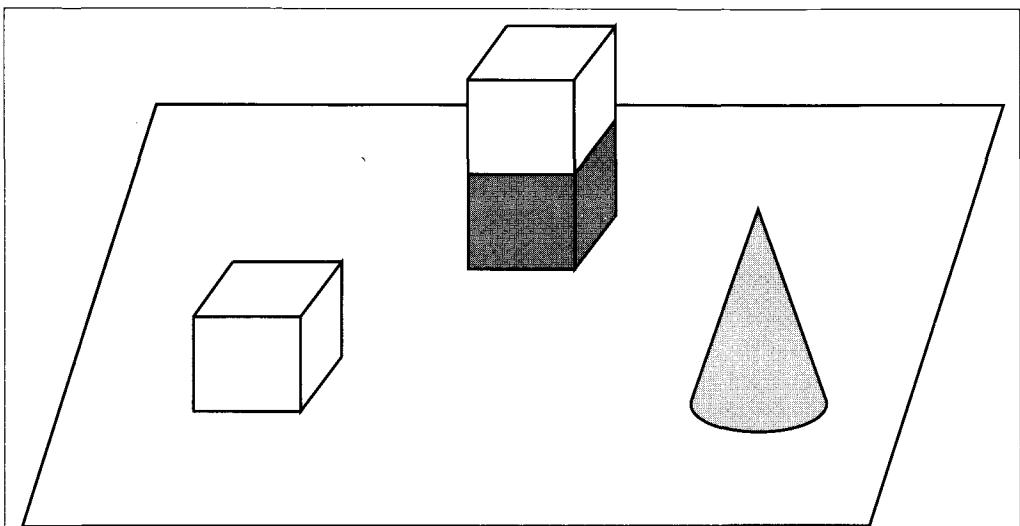
If the number of customers Tom gets is twice the square of 20 percent of the number of advertisements he runs, and the number of advertisements he runs is 45, what is the number of customers Tom gets?



**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 damped 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.

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

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.

## 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.<sup>16</sup> 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

<sup>16</sup> 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.<sup>17</sup> 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.

<sup>17</sup> 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 Allen Newell, John Laird, and Paul Rosenbloom on SOAR (Newell, 1990; Laird *et al.*, 1987) is the best-known example of a complete agent architecture 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,<sup>18</sup> 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 successfully 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

---

<sup>18</sup> 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—but rather science, engineering, and mathematics, to which this book provides an introduction.

## 1.5 SUMMARY

---

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

- Different people think of AI differently. Two important questions to ask are: Are you concerned with thinking or behavior? Do you want to model humans, or work from an ideal standard?
- In this book, we adopt the view that intelligence is concerned mainly with **rational action**. Ideally, an **intelligent agent** takes the best possible action in a situation. We will study the problem of building agents that are intelligent in this sense.
- Philosophers (going back to 400 B.C.) made AI conceivable by considering the ideas that the mind is in some ways like a machine, that it operates on knowledge encoded in some internal language, and that thought can be used to 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 *PCAI*. 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?

STRONG AI  
WEAK AI

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.

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.

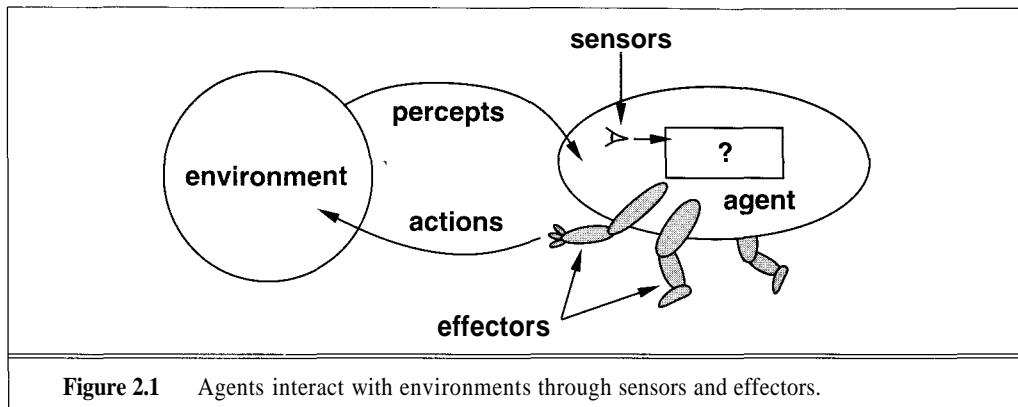
Our aim in this book is to design agents that do a good job of acting on their environment. First, we will be a little more precise about what we mean by a good job. Then we will talk about different designs for successful agents—filling in the question mark in Figure 2.1. We discuss some of the general 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.<sup>1</sup>

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,<sup>2</sup> 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

<sup>1</sup> 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.

<sup>2</sup> 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.<sup>3</sup>

<sup>3</sup> One of the authors still gets a small thrill when his computer successfully resets itself at daylight savings time.

MAPPING

IDEAL MAPPINGS



## 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.*

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

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.



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<sup>4</sup> 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

AGENT PROGRAM

ARCHITECTURE

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

<sup>4</sup> 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:

$$\text{agent} = \text{architecture} + \text{program}$$

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.<sup>5</sup> 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.

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.

<sup>5</sup> 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  $\leftarrow$  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  $\leftarrow$  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.<sup>6</sup> 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.

<sup>6</sup> 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**<sup>7</sup> written as

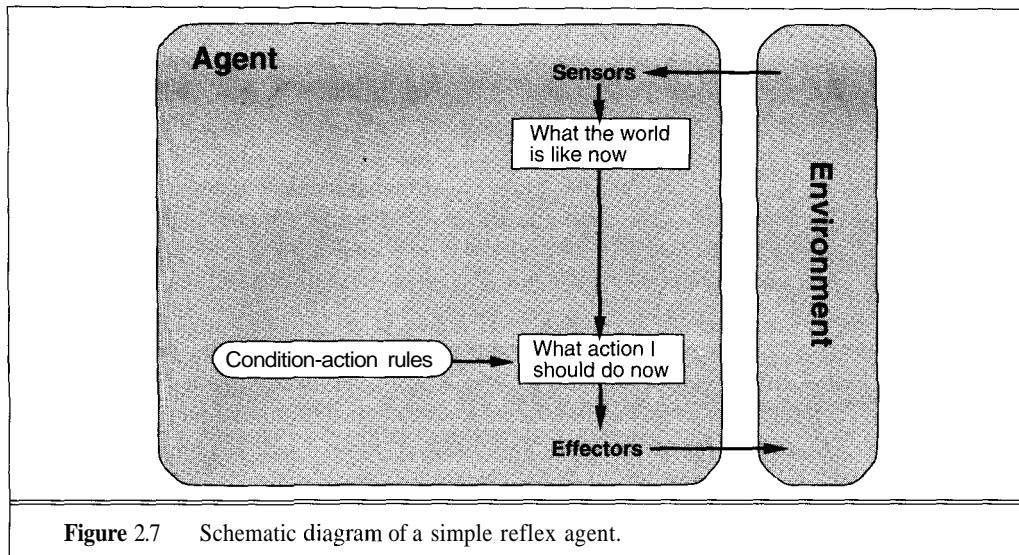
**if** *car-in-front-is-braking* **then** *initiate-braking*

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

CONDITION-ACTION  
RULE

<sup>7</sup> 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  $\leftarrow$  INTERPRET-INPUT(percept)
    rule  $\leftarrow$  RULE-MATCH(state, rules)
    action  $\leftarrow$  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

## INTERNAL STATE

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.

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.

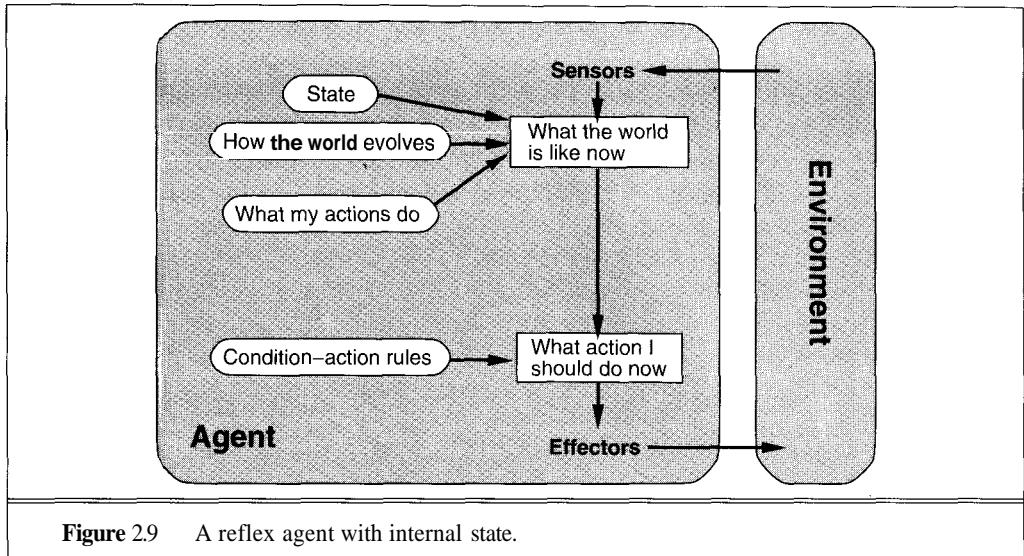
## Goal-based agents

## GOAL

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.

## 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  $\leftarrow$  UPDATE-STATE(state, percept)
  rule  $\leftarrow$  RULE-MATCH(state, rules)
  action  $\leftarrow$  RULE-ACTION[rule]
  state  $\leftarrow$  UPDATE-STATE(state, action)
  return action

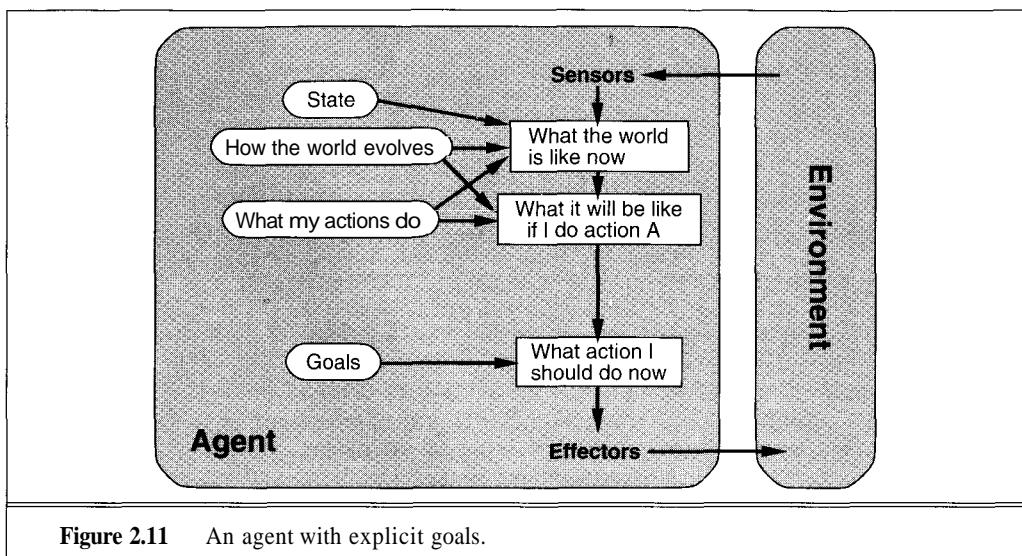
```

**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.<sup>8</sup>

Utility is therefore a function that maps a state<sup>9</sup> 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

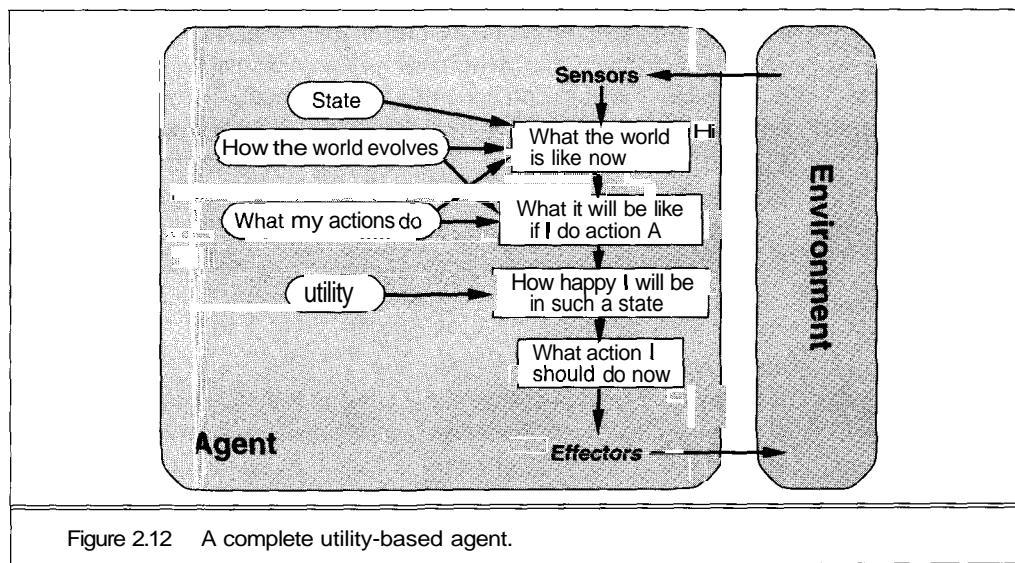
<sup>8</sup> The word "utility" here refers to "the quality of being useful," not to the electric company or water works.

<sup>9</sup> 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 appear in Chapter 5, where we examine game-playing programs that must make fine distinctions among various board positions; and in Chapter 17, where we tackle the general problem of designing decision-making agents.



## 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.

ACCESSIBLE

## Properties of environments

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

DETERMINISTIC

### 0 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.

EPISODIC

### 0 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*.

STATIC

### 0 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.

SEMDYNAMIC

### 0 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**.

DISCRETE

### 0 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.<sup>10</sup>

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.

<sup>10</sup> 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]  $\leftarrow$  GET-PERCEPT(agent, state)
    end
    for each agent in agents do
      ACTION[agent]  $\leftarrow$  PROGRAM[agent](PERCEPT[agent])
    end
    state  $\leftarrow$  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]  $\leftarrow$  GET-PERCEPT(agent, state)
    end
    for each agent in agents do
      ACTION[agent]  $\leftarrow$  PROGRAM[agent](PERCEPT[agent])
    end
    state  $\leftarrow$  UPDATE-FN(actions, agents, state)
    scores  $\leftarrow$  PERFORMANCE-FN(scores, agents, state)
  until termination(state)
  return scores                                I * 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 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 single environment, defined by a single initial state and a particular update function. Usually, an agent is designed to

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  $90^\circ$ , turn left by  $90^\circ$ , 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  $n \times n$  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  $2 \times 2$  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 and 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.

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.<sup>1</sup>

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.<sup>2</sup> 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

<sup>1</sup> 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.

<sup>2</sup> 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  $\leftarrow$  UPDATE-STATE(state, p)
  if s is empty then
    g  $\leftarrow$  FORMULATE-GOAL(state)
    problem  $\leftarrow$  FORMULATE-PROBLEM(state, g)
    s  $\leftarrow$  SEARCH(problem)
    action  $\leftarrow$  RECOMMENDATION(s, state)
    s  $\leftarrow$  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  $\{7, 8\}$ .

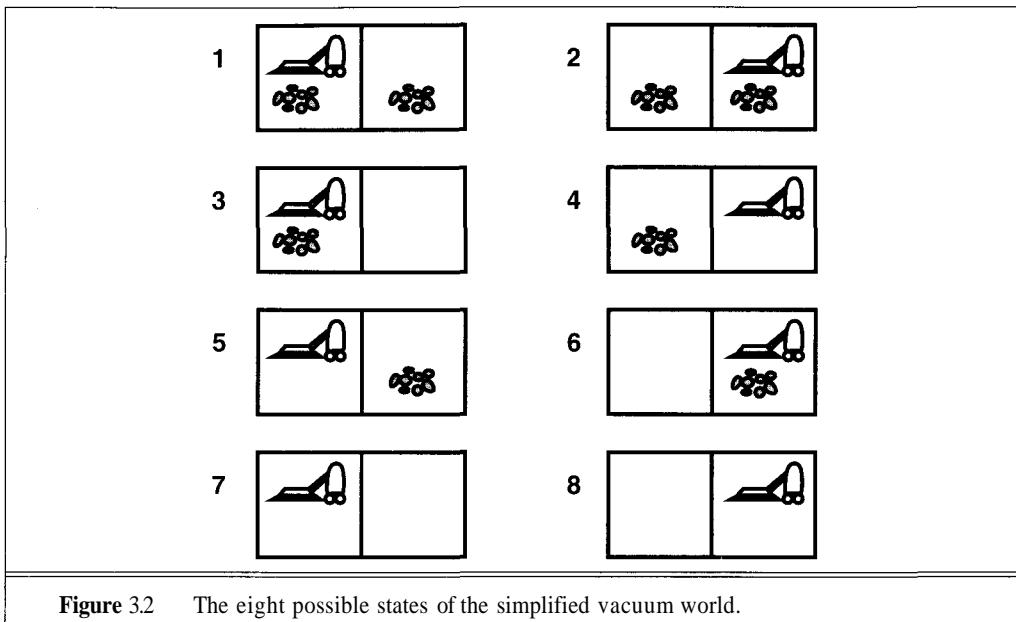


Figure 3.2 The eight possible states of the simplified vacuum world.

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**.

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**.

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*.<sup>3</sup> 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.

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.

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.<sup>4</sup> 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.

CONTINGENCY  
PROBLEM

INTERLEAVING

EXPLORATION  
PROBLEM

<sup>3</sup> 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.

<sup>4</sup> 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**

**OPERATOR**

**SUCCESSOR FUNCTION**

**STATE SPACE**

**PATH**

**GOALTEST**

**PATH COST**

**SOLUTION**

**STATE SET SPACE**

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

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:

- 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.

- 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:

<b>datatype PROBLEM</b> <b>components:</b> 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.

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.

SEARCH COST  
TOTAL COST

## Measuring problem-solving performance

The effectiveness of a search can be measured in at least three ways. First, does it find 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.<sup>5</sup>

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.

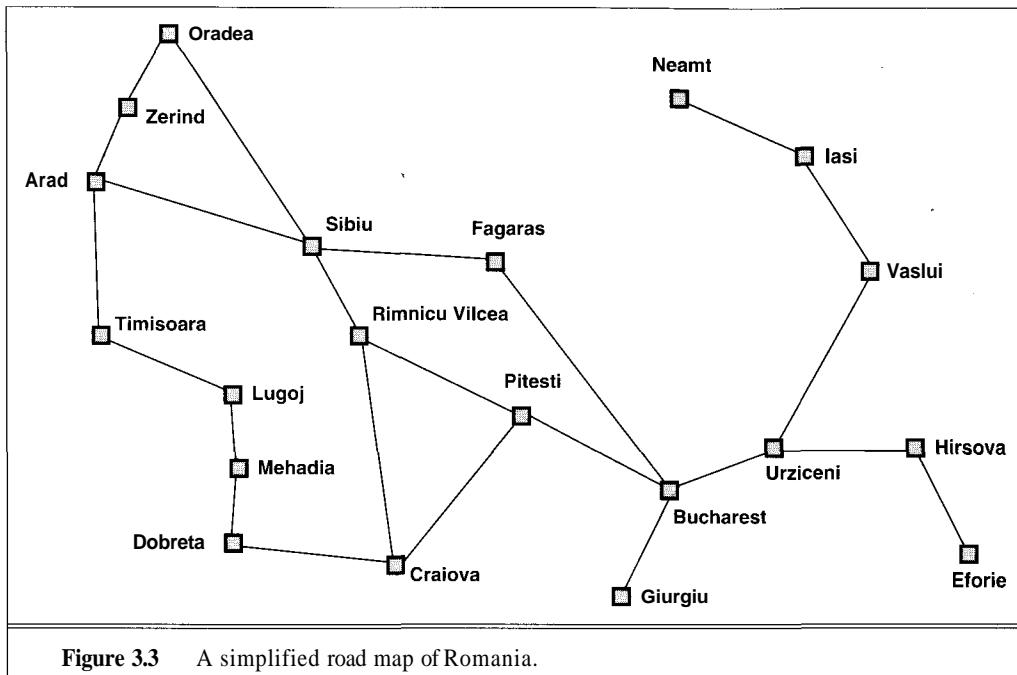
## Choosing states and actions

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

<sup>5</sup> 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.

ABSTRACTION

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. 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

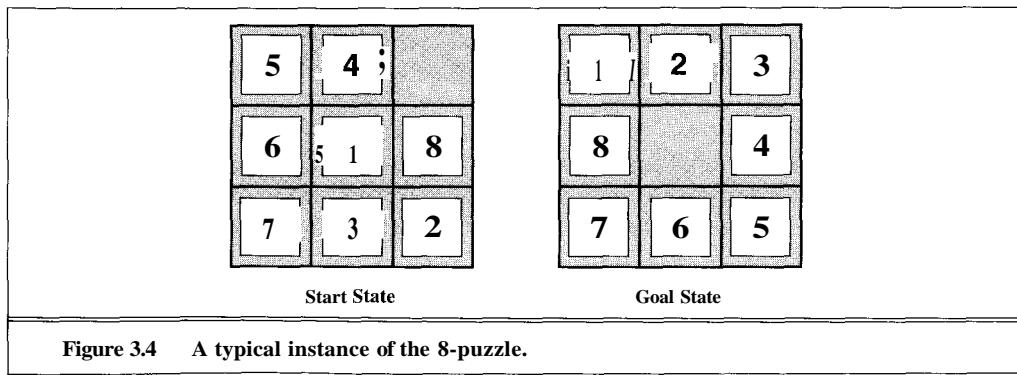
8-PUZZLE

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.
- ◊ **Goal test:** state matches the goal configuration shown in Figure 3.4.
- ◊ **Path cost:** each step costs 1, so the path cost is just the length of the path.

SLIDING-BLOCK PUZZLES

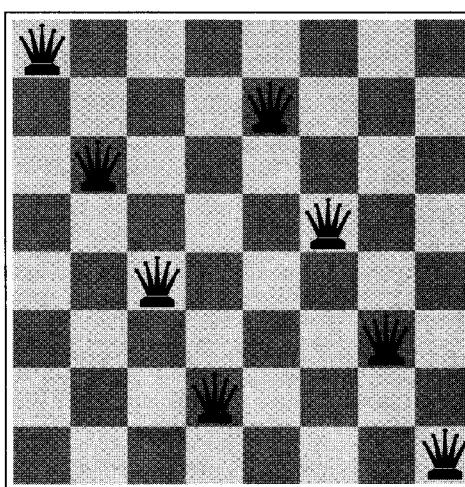
The 8-puzzle belongs to the family of **sliding-block puzzles**. This general class is known to be NP-complete, so one does not expect to find methods significantly better than the search



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:

0 **Goal test:** 8 queens on board, none attacked.

0 **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:

**0 States:** arrangements of 0 to 8 queens with none attacked.

**0 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:

**0 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, O=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.

**0 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.

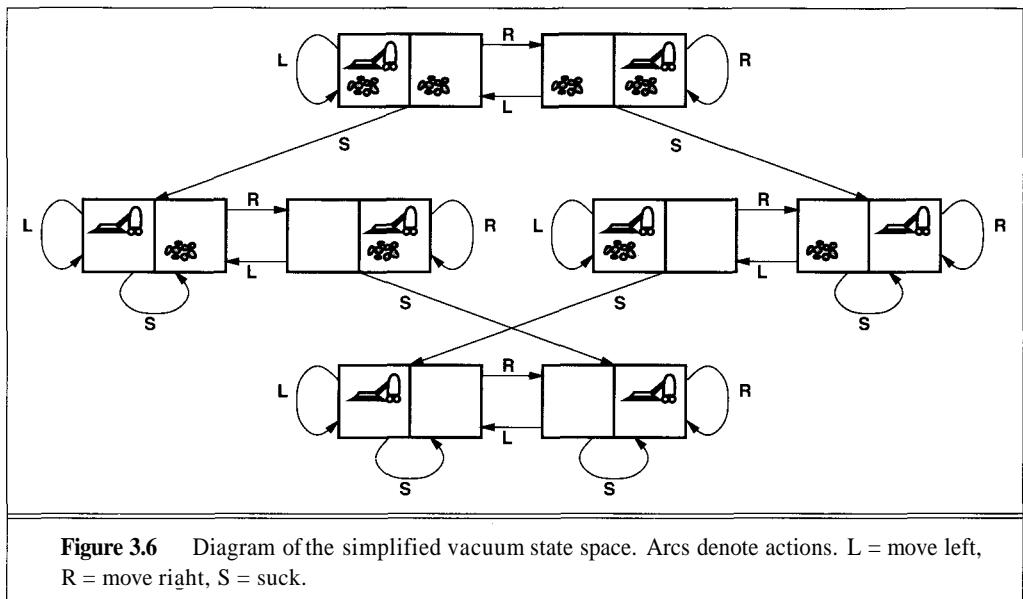
A moment's thought shows that replacing E by 6 then F by 7 is the same thing as replacing F by 7 then E by 6—order does not matter to correctness, so we want to avoid trying permutations of the same substitutions. One way to do this is to adopt a fixed order, e.g., alphabetical order. A better choice 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.



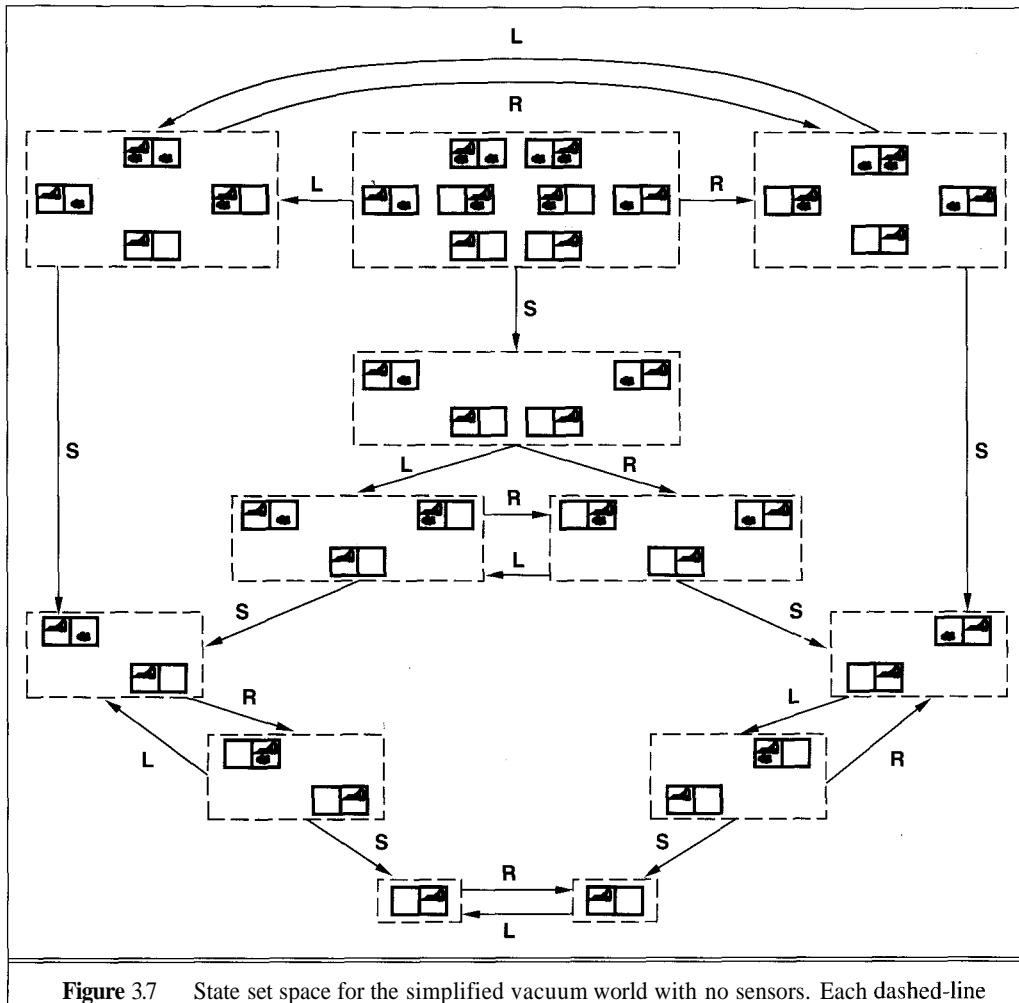
**Figure 3.6** Diagram of the simplified vacuum state space. Arcs denote actions. L = move left, R = move right, S = suck.

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:

- 0 **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).
- 0 **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.
- 0 **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.<sup>6</sup> 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

<sup>6</sup> 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.

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 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

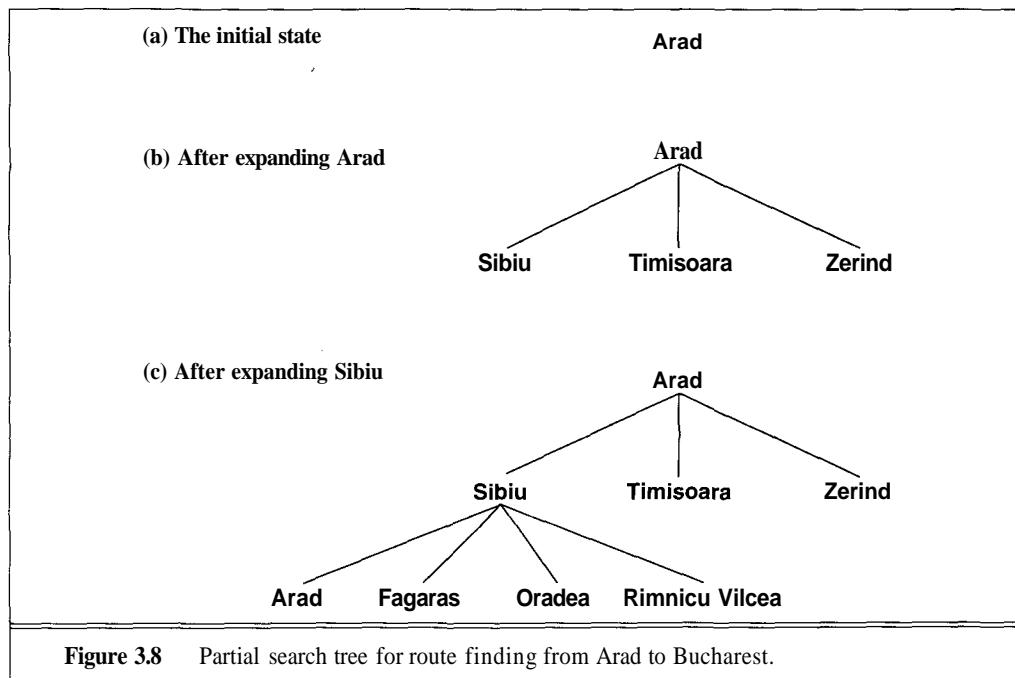


Figure 3.8 Partial search tree for route finding from Arad to Bucharest.

```

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:

PARENT NODE

DEPTH

- the state in the state space to which the node corresponds;
- 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;
- 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

FRINGE  
FRONTIER

QUEUE

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 collection is called the **fringe** or **frontier**. The simplest representation would be a set of nodes. The search strategy then would be a function that selects the next node to be expanded from this set. Although this is conceptually straightforward, it could be computationally expensive, because the strategy function might have to look at every element of the set to choose the best one. Therefore, we will assume that the collection of nodes is implemented as a **queue**. The operations on a queue are as follows:

- **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.
- **QUEUEING-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  $\leftarrow$  MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
  loop do
    if nodes is empty then return failure
    node  $\leftarrow$  REMOVE-FRONT(nodes)
    if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
    nodes  $\leftarrow$  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     | ◊ <b>Completeness:</b> is the strategy guaranteed to find a solution when there is one?   |
| TIME COMPLEXITY  | ◊ <b>Time complexity:</b> how long does it take to find a solution?   |
| SPACE COMPLEXITY | ◊ <b>Space complexity:</b> how much memory does it need to perform the search?  |
| OPTIMALITY       | ◊ <b>Optimality:</b> does the strategy find the highest-quality solution when there are several different solutions? <sup>7</sup> |

UNINFORMED  
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**.

BLIND SEARCH

INFORMED SEARCH  
HEURISTIC 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. 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.

<sup>7</sup> 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

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

This is the maximum number, but the solution could be found at any point on the  $d$ th 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

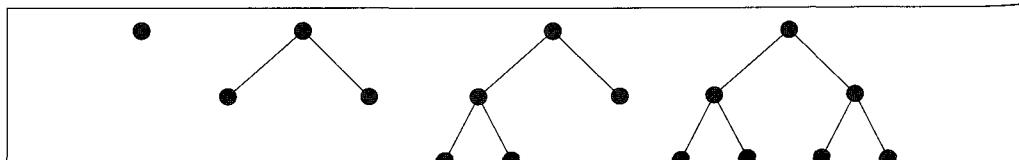


Figure 3.11 Breadth-first search trees after 0, 1, 2, and 3 node expansions.

## BRANCHING FACTOR

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

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,

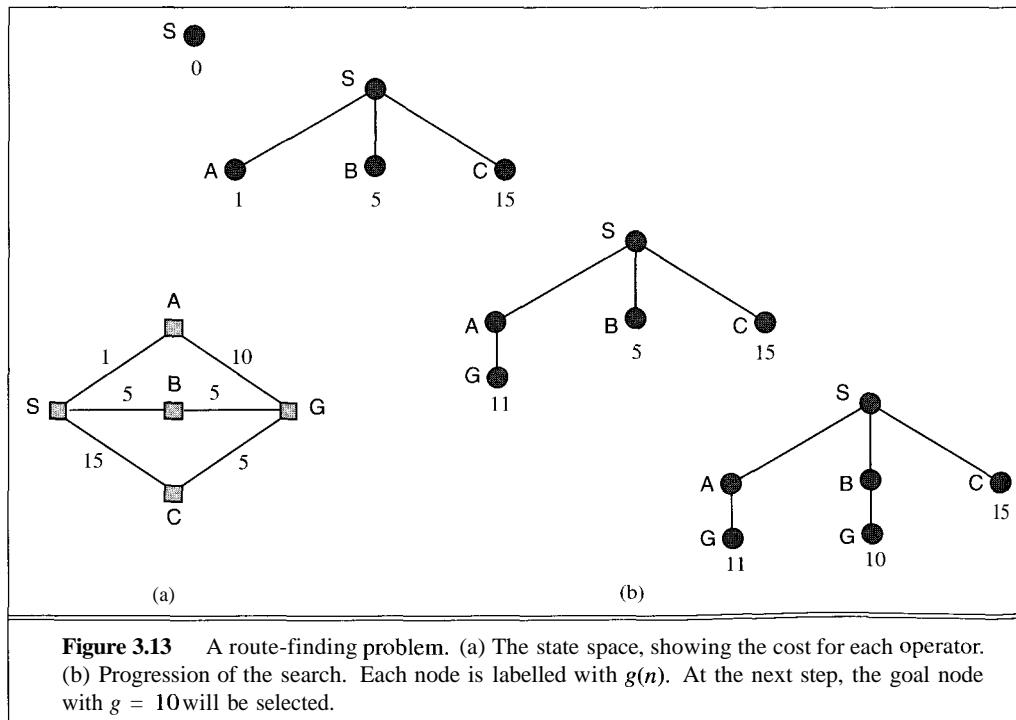
and the cost of each operator is marked. The strategy first expands the initial state, yielding paths to A, B, and C. Because the path to A is cheapest, it is expanded next, generating the path SAG, which is in fact a solution, though not the optimal one. However, the algorithm does not yet recognize 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.)



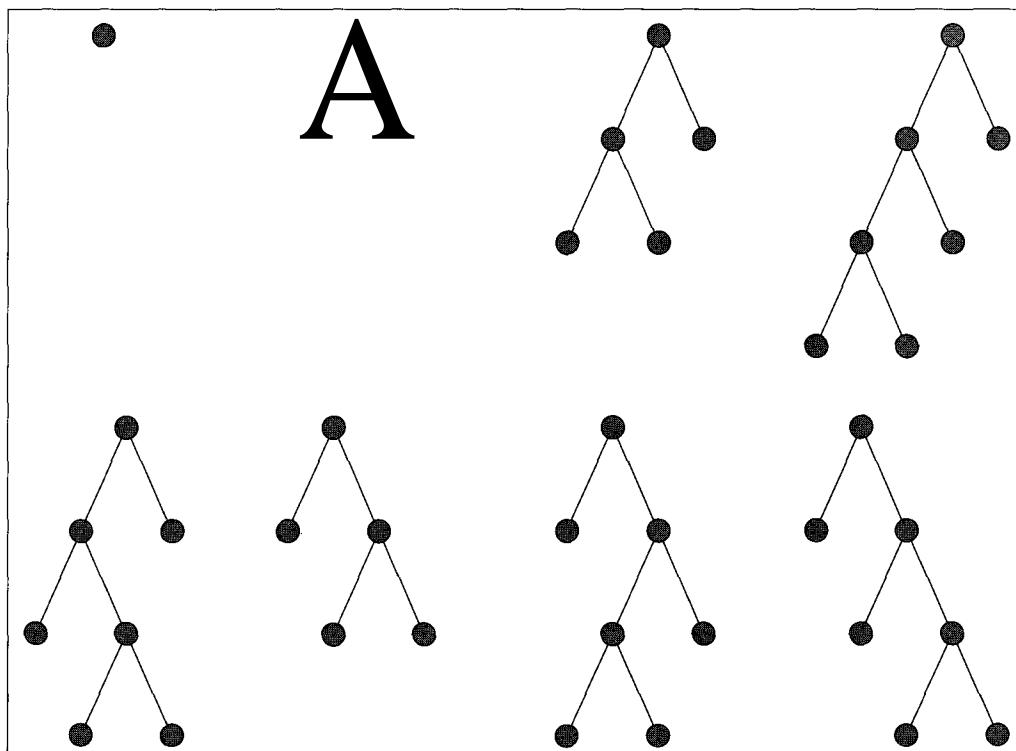
**Figure 3.13** A route-finding problem. (a) The state space, showing the cost for each operator. (b) Progression of the search. Each node is labelled with  $g(n)$ . At the next step, the goal node with  $g = 10$  will be selected.

## Depth-first search

**Depth-first search** 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.

DIAMETER

## 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.

## 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  $\leftarrow$  0 to  $\infty$  do
    if DEPTH-LIMITED-SEARCH(problem, depth) succeeds then return its result
  end
  return failure

```

Figure 3.15 The iterative deepening search algorithm.

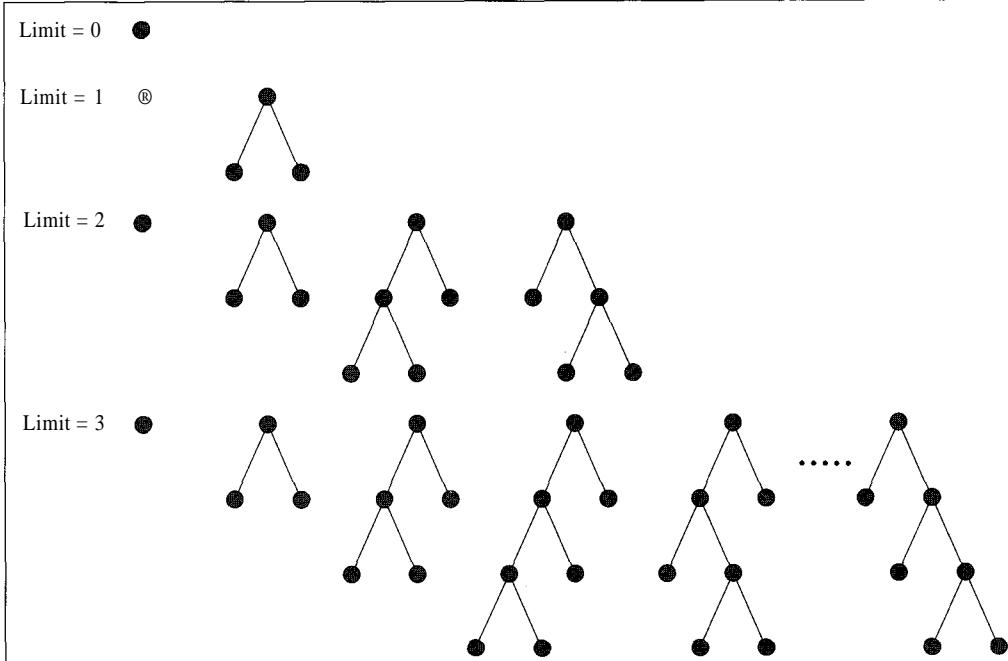


Figure 3.16 Four iterations of iterative deepening search on a binary tree.

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

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

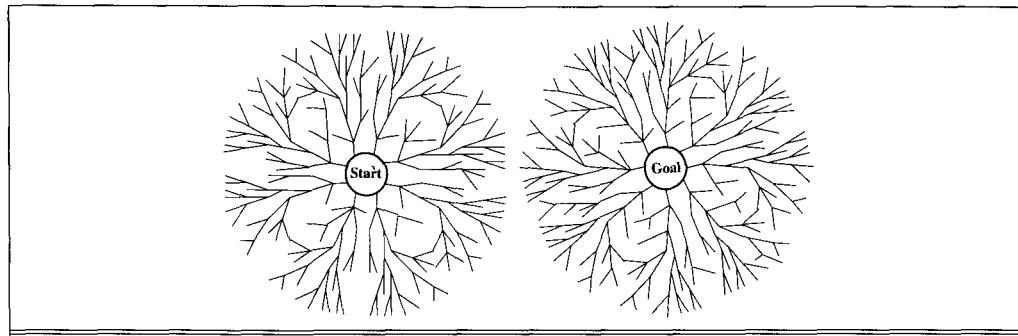
All together, an iterative deepening search from depth 1 all the way down to depth  $d$  expands only about 11% more nodes than a single breadth-first or depth-limited search to depth  $d$ , when  $b = 10$ . The higher the branching factor, the lower the overhead of repeatedly expanded states, but even 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 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.

- 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

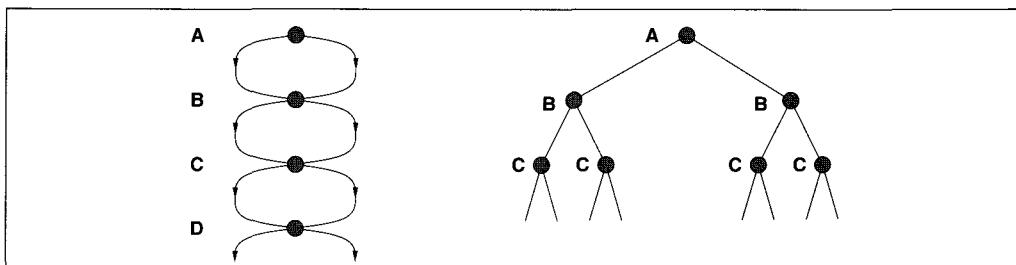
Up to this point, we have all but ignored one of the most important complications to the search process: the possibility of wasting time by expanding states that have already been encountered and expanded before on some other path. For some problems, this possibility never comes up; each state can only 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

DOMAIN

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.

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  $V_1$  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  $V_1$  and  $V_2$  are  $\{1, 2, 3, 4, 5, 6, 7, 8\}$ . The no-attack constraint linking  $V_1$  and  $V_2$  can be represented by a set of pairs of allowable values for  $V_1$  and  $V_2$ :  $\{\langle 1, 3 \rangle, \langle 1, 4 \rangle, \langle 1, 5 \rangle, \dots, \langle 2, 4 \rangle, \langle 2, 5 \rangle, \dots\}$  and so on. Altogether, the no-attack constraint between  $V_1$  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.

Let 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 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 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.

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.



BACKTRACKING  
SEARCH

FORWARD  
CHECKING

ARC CONSISTENCY

CONSTRAINT  
PROPAGATION

## 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  $n \times n$  chessboard).

The 8-puzzle initially appeared as the more complex  $4 \times 4$  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 I 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?

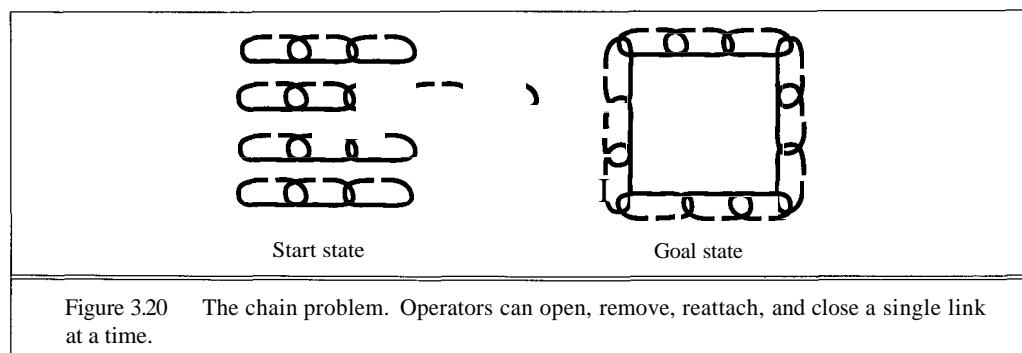
**3.11** Write down the algorithm for bidirectional search, in pseudo-code or in a programming language. Assume that each search will be a breadth-first search, and that the forward and backward searches take turns expanding a node at a time. Be careful to avoid checking each node in the forward search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





**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  $[1, 2, 4, 8, 16, \dots]$ .

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  $+$ ,  $\times$ ,  $-$ ,  $/$ , 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  $[1, 2, 3, 4, 5]$ ,  $[1, 2, 4, 8, 16, \dots]$ , 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  $3 \times 3$  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  $3 \times 3$  worlds with probability 0.2 of dirt in each square. Include the search cost as well as path cost in the performance measure, using a reasonable exchange rate.
- 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.
- 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.
- d. The rectilinear **floor-planning** problem, which involves finding nonoverlapping places in a large rectangle for a number of smaller rectangles.

MAP-COLORING

FLOOR-PLANNING



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

EVALUATION FUNCTION

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** 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 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  $h$ :

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

HEURISTIC FUNCTION

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.

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.

STRAIGHT-LINE DISTANCE

## 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, Bruce 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).

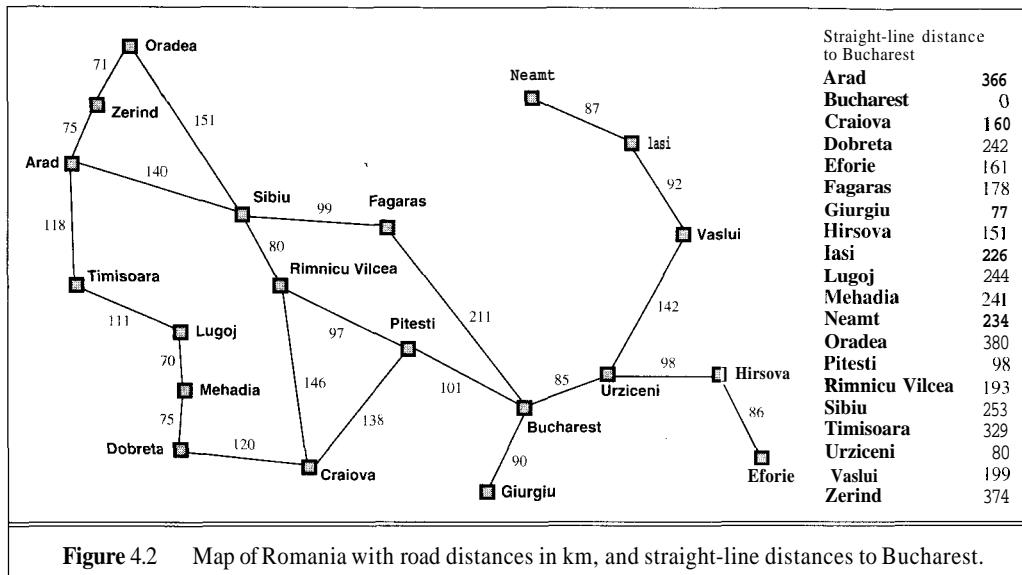
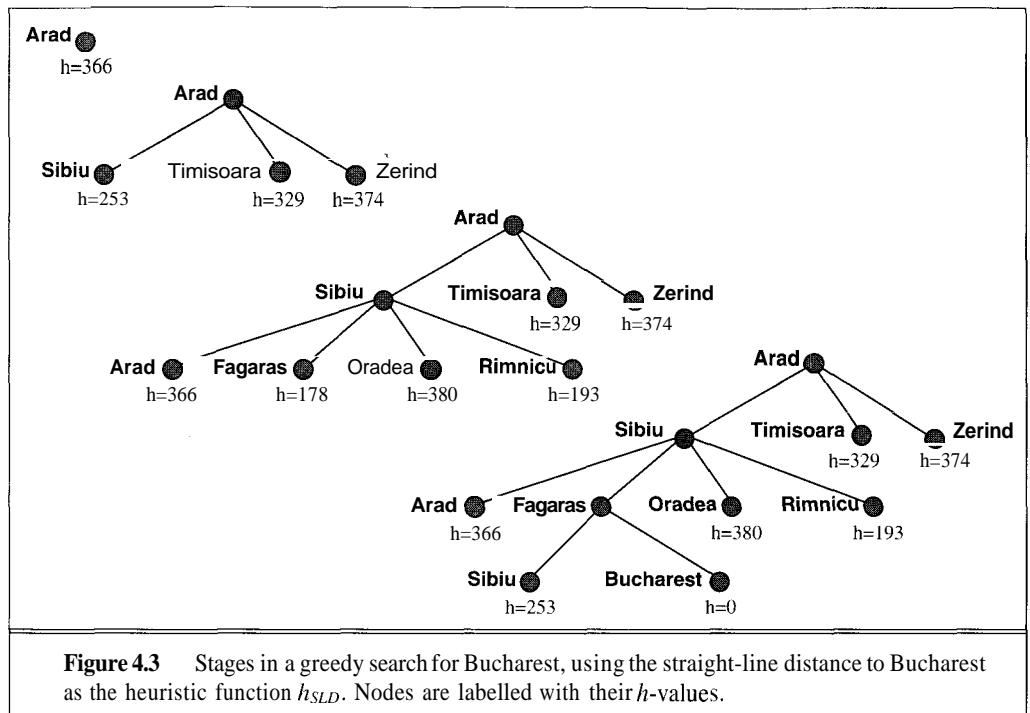


Figure 4.2 Map of Romania with road distances in km, and straight-line distances to Bucharest.

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.

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  $h$  is 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**

```
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**.<sup>1</sup>

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

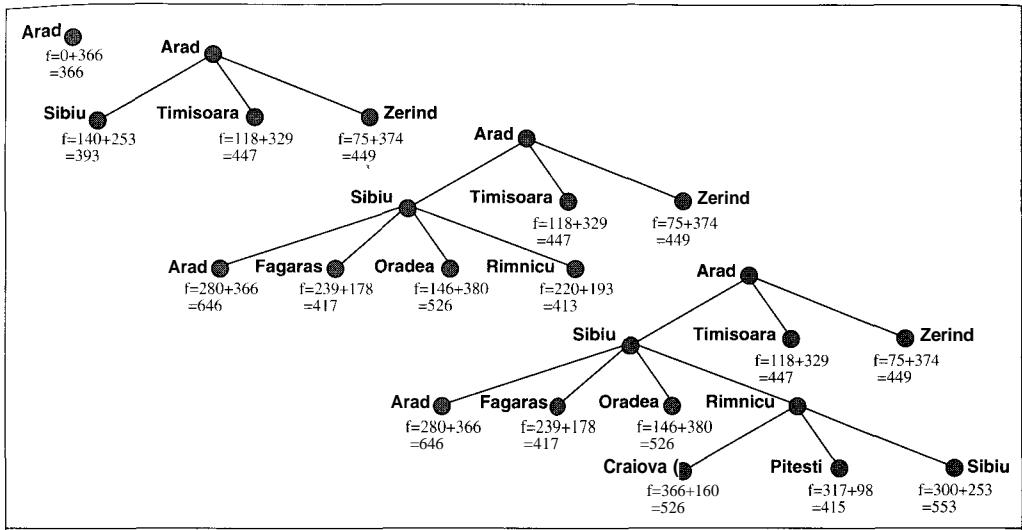
<sup>1</sup> It can be proved (Pearl, 1984) that a heuristic is monotonic if and only if it obeys the triangle inequality. The triangle inequality says that two sides of a triangle cannot add up to less than the third side (see Exercise 4.7). Of course, straight-line distance obeys the triangle inequality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

ADMISSIBLE  
HEURISTIC



A\* SEARCH

MONOTONICITY



**Figure 4.4** Stages in an A\* search for Bucharest. Nodes are labelled with  $f = g + h$ . 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  $f$ -cost is less than its parent's  $f$ -cost; if it is, we use the parent's  $f$ -cost instead:

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

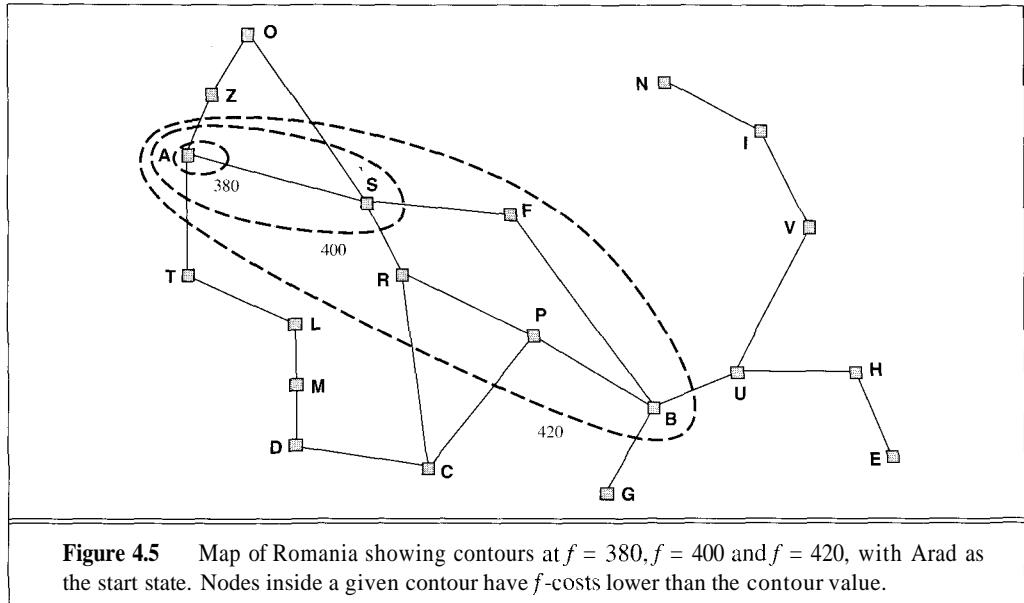
In this way, we ignore the misleading values that may occur with a nonmonotonic heuristic. This equation is called the **pathmax** equation. If we use it, then  $f$  will always be nondecreasing along any path from the root, provided  $h$  is admissible.

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.

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  $f(n) = f^*$ , 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.

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).

OPTIMALLY  
EFFICIENT

### 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

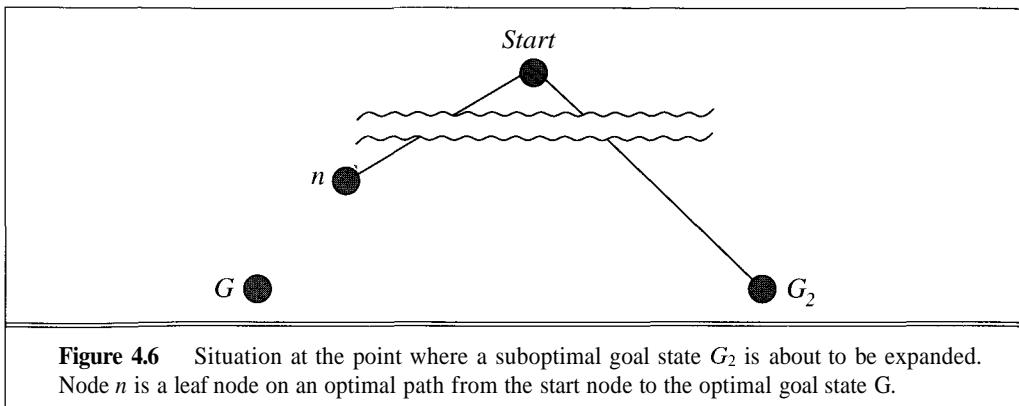
$$f \geq f(n).$$

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

$$f > f(G_2).$$



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.<sup>2</sup>

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  $\delta$  such that every operator costs at least  $\delta$ .

### 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

<sup>2</sup> 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

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

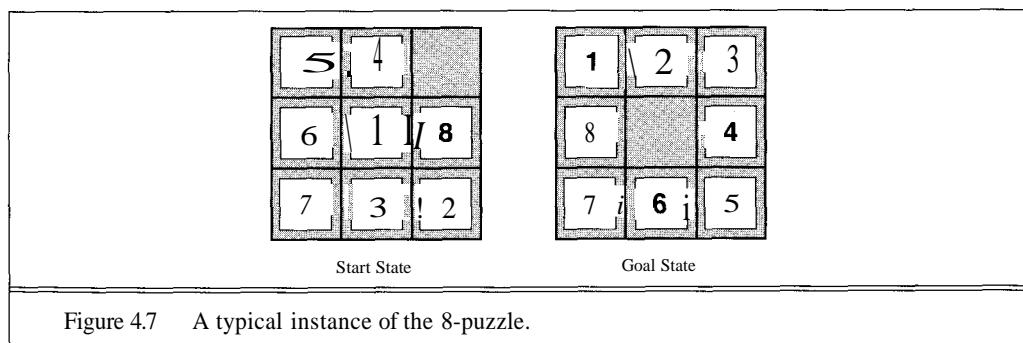
where  $h^*(n)$  is the *true* cost of getting from  $n$  to the goal. For almost all heuristics in practical use, the error is at least proportional to the path cost, and the resulting exponential growth eventually overtakes any computer. Of course, the use of a good heuristic still provides enormous 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).



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:

- $h_1$  = 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

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

MANHATTAN DISTANCE

EFFECTIVE BRANCHING FACTOR

## The effect of heuristic accuracy on performance

One way to characterize the quality of a heuristic is the **effective branching factor**  $b^*$ . If the total number of nodes 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,

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

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  $h_1$  and  $h_2$ , we randomly generated 100 problems each with solution lengths 2, 4, ..., 20, and solved them using A\* search with  $h_1$  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  $h(n) < f^* - g(n)$  will be expanded. But because  $h_2$  is at least as big as  $h_1$  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 cause other nodes to be expanded as well. Hence, it is always better to use a heuristic function with higher values, as long as it does not overestimate.



$d$	Search Cost			Effective Branching Factor		
	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  $h_1$ ,  $h_2$ . Data are averaged over 100 instances of the 8-puzzle, for various solution lengths.

## Inventing heuristic functions

We have seen that both  $h_1$  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  $h_1$  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.*

If a problem definition is written down in a formal language, it is possible to construct relaxed problems automatically.<sup>3</sup> 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.

<sup>3</sup> 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  $h_1 \dots h_m$  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

$$h(n) = \max(h_1(n), \dots, h_m(n)).$$

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

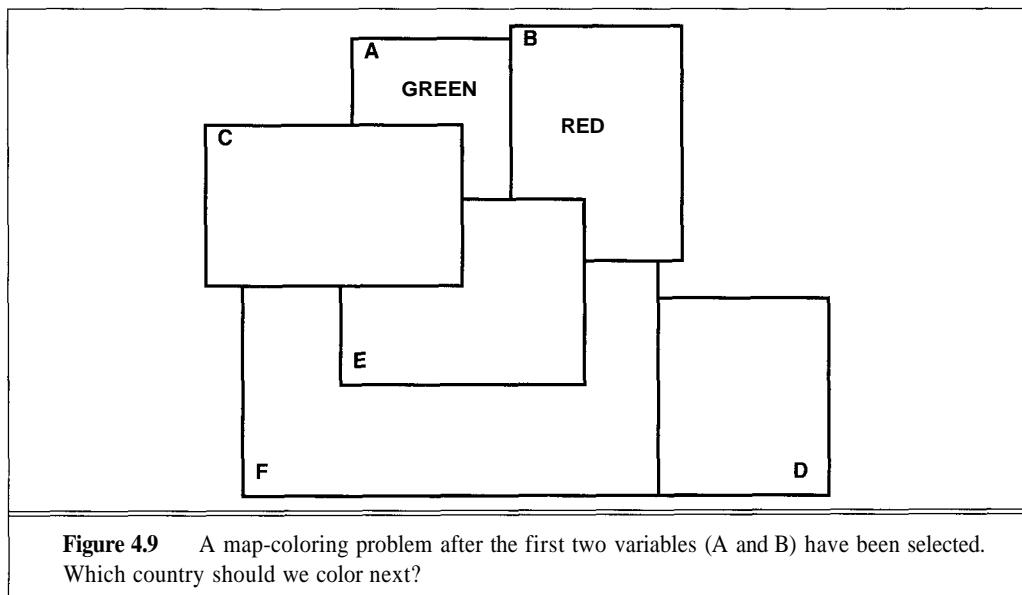
Often it is possible to pick out **features** of a state that contribute to its heuristic evaluation function, even if it is hard to say exactly what the contribution should be. For example, the goal in chess is to checkmate the opponent, and relevant features include the number of pieces of each kind 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.



MOST-CONSTRAINED-VARIABLE

MOST-CONSTRAINING-VARIABLE

LEAST-CONSTRAINING-VALUE

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.

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.

## 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  $bf^*/\delta$  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  $1 + 2 + \dots + N = O(N^2)$  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  $f$ -cost limit by a fixed amount  $\epsilon$  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  $\epsilon$ . Such an algorithm is called  $\epsilon$ -admissible.

```

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

  root  $\leftarrow$  MAKE-NODE(INITIAL-STATE[problem])
  f-limit  $\leftarrow$  f- COST(root)
  loop do
    solution, f-limit  $\leftarrow$  DFS-CONTOUR(root,f-limit)
    if solution is non-null then return solution
    if f-limit =  $\infty$  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  $\infty$ 

  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  $\leftarrow$  DFS-CONTOUR(s,f-limit)
    if solution is non-null then return solution, f-limit
    next-f  $\leftarrow$  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—one could 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.

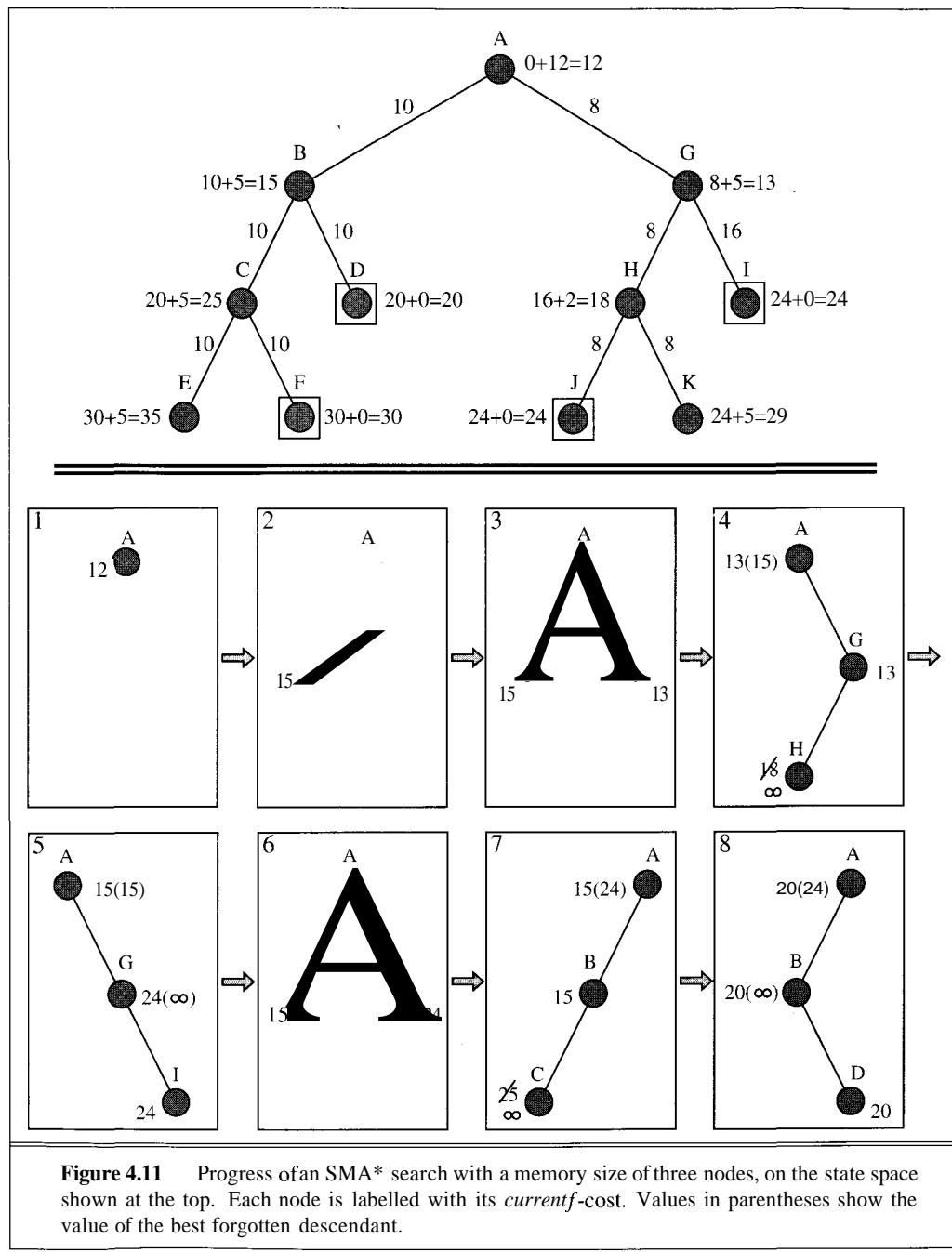
The one unresolved question is whether SMA\* is always optimally efficient among all algorithms given the same heuristic information and the same memory allocation.

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.<sup>4</sup> 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  $f(C) = \infty$ .
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.

<sup>4</sup> 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.



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) =  $\infty$ 
    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.

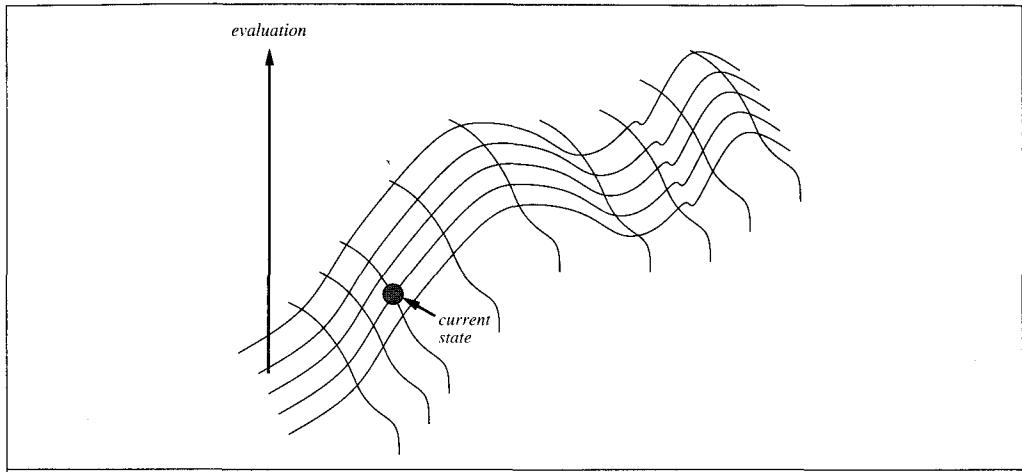
HILL-CLIMBING  
GRADIENT DESCENT  
SIMULATED  
ANNEALING

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  $\leftarrow$  MAKE-NODE(INITIAL-STATE[problem])
  loop do
    next  $\leftarrow$  a highest-valued successor of current
    if VALUE[next] < VALUE[current] then return current
    current  $\leftarrow$  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.

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  $\Delta E$  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  $\Delta E$  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  $\leftarrow$  MAKE-NODE(INITIAL-STATE[problem])
  for t  $\leftarrow$  1 to  $\infty$  do
    T  $\leftarrow$  schedule[t]
    if T=0 then return current
    next  $\leftarrow$  a randomly selected successor of current
     $\Delta E \leftarrow$  VALUE[next] - VALUE[current]
    if  $\Delta E > 0$  then current  $\leftarrow$  next
    else current  $\leftarrow$  next only with probability  $e^{\Delta E/T}$ 
```

**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

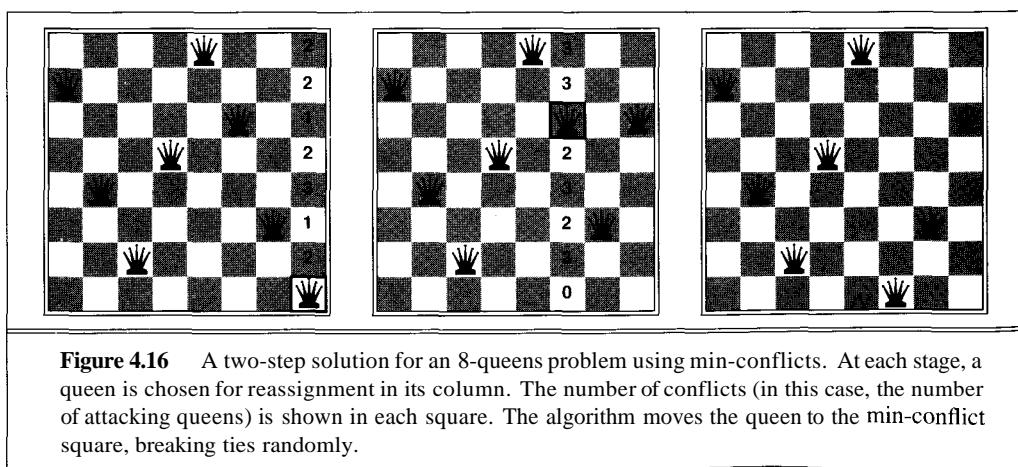
Constraint satisfaction problems (CSPs) can be solved by iterative improvement methods by first assigning values to all the variables, and then applying modification operators to move the configuration toward a solution. Modification operators simply assign a different value to a variable. For example, 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

MIN-CONFLICTS

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 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.



## 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 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.

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.

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:
- The general case of the chain problem (i.e., with an arbitrary goal state) from Exercise 3.15.
  - Algebraic equation solving (e.g., "solve  $x^2y^3 = 3 - xy$  for  $x$ ").
  - Path planning in the plane with rectangular obstacles (see also Exercise 4.13).
  - 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 (when combined with pathmax) leads to monotonically nondecreasing  $f$  values along any path (i.e.,  $f(\text{successor}(n)) \geq f(n)$ ). 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.

- Show how this heuristic can be derived using a relaxed version of the TSP.
- Show that the MST heuristic dominates straight-line distance.
- Write a problem generator for instances of the TSP where cities are represented by random points in the unit square.
- 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.



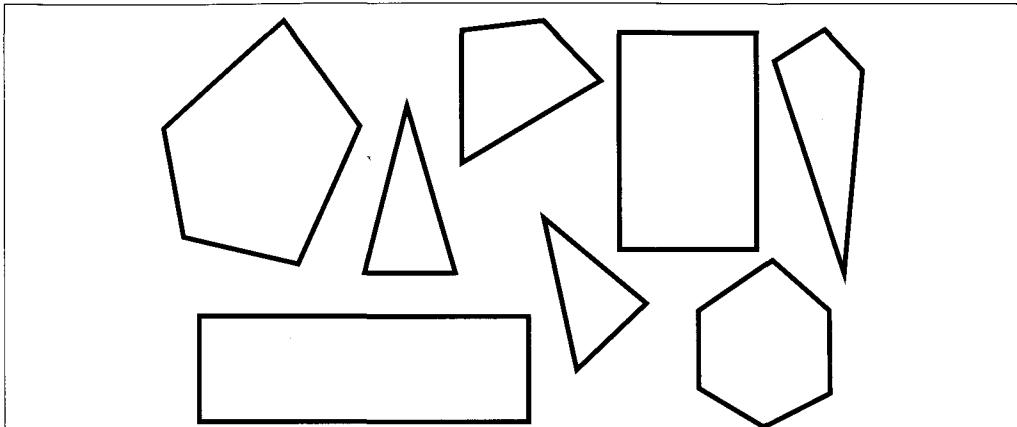
**4.13** Consider the problem of finding the shortest path between two points on a plane that has convex polygonal obstacles as shown in Figure 4.17. This is an idealization of the problem that a robot has to solve to navigate its way around a crowded environment.

- 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?
- 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?
- 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.
- 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.

- 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  $d = \text{MAX}$ .
  - (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<sup>1</sup> 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.

<sup>1</sup> 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 and 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 MIN, for reasons that will soon become obvious. MAX moves first, and then they take turns moving until the game is over. At the end of the game, points are awarded to the winning player (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.

STRATEGY

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.

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.

PLY

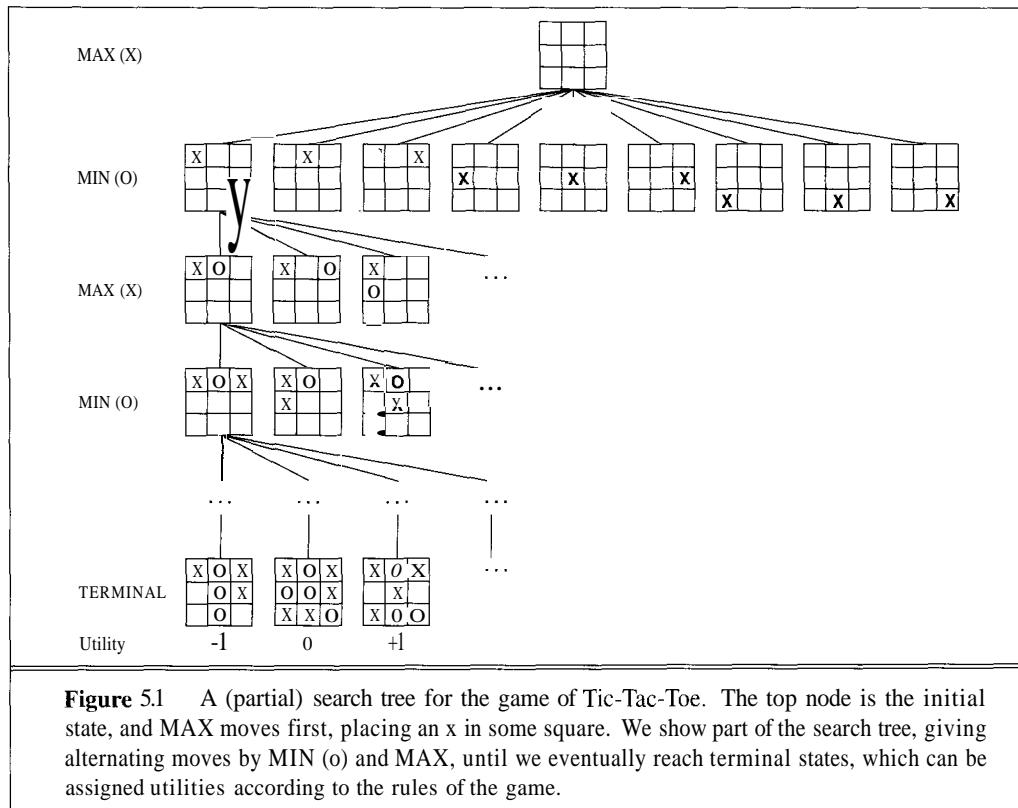
Even a simple game like Tic-Tac-Toe is too complex to show the whole search tree, so we will switch to the absolutely trivial game in Figure 5.2. The possible moves for MAX are labelled  $A_1$ ,  $A_2$ , and  $A_3$ . The possible replies to  $A_1$  for MIN are  $A_{11}$ ,  $A_{12}$ ,  $A_{13}$ , and 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.

MINIMAX

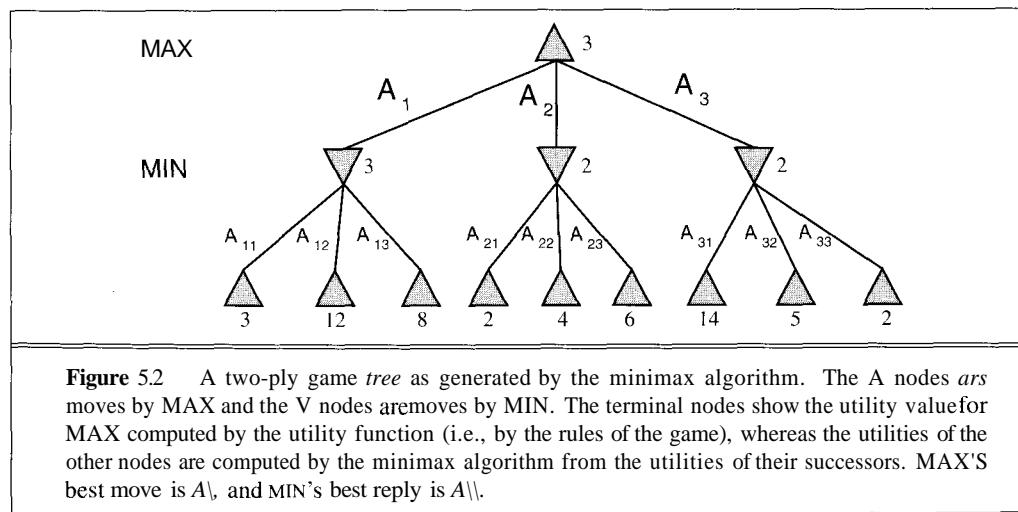
The **minimax** algorithm is designed to determine the optimal strategy for MAX, and thus to decide what the best first move is. The algorithm consists of five steps:

MINIMAX DECISION

- 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  $A_{11}$ . This is called the **minimax decision**, because it maximizes the utility under the assumption that the opponent will play perfectly to minimize it.



**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 are moves by MAX and the V nodes are moves by MIN. The terminal nodes show the utility value for 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<sub>1</sub>**, and MIN's best reply is **A<sub>11</sub>**.

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 corresponds 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.

MATERIAL VALUE

## 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.

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.<sup>2</sup>

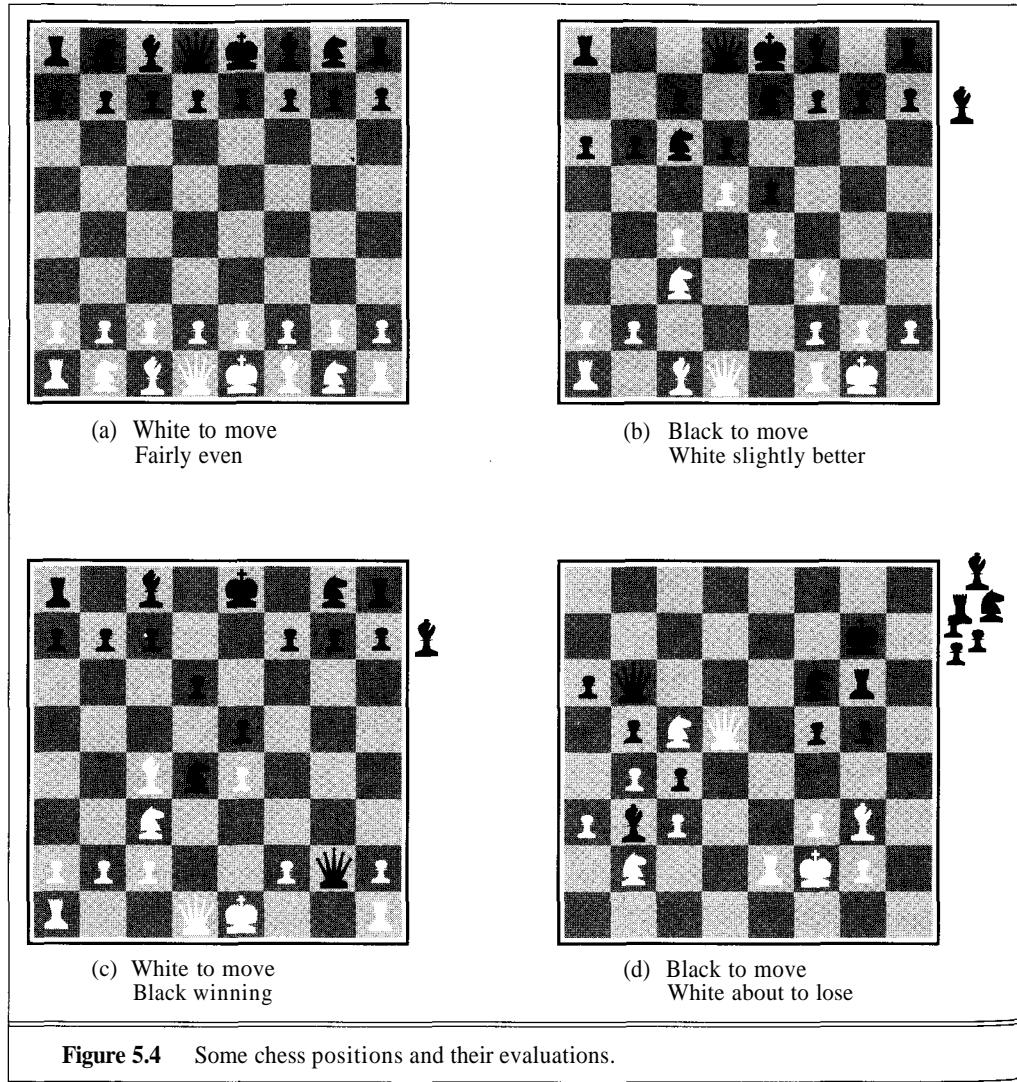
This suggests that the evaluation function should be specified by the rules of probability: if position A has a 100% chance of winning, it should have the evaluation 1.00, and if position B has a 50% chance of winning, 25% of losing, and 25% of being a draw, its evaluation should be  $+1 \times .50 + -1 \times .25 + 0 \times .25 = .25$ . 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

$$w_1f_1 + w_2f_2 + \dots + w_nf_n$$

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

<sup>2</sup> 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.



**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.

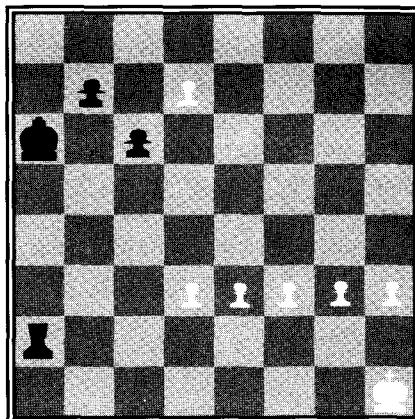
HORIZON PROBLEM

**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-BETAPRUNING

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



**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

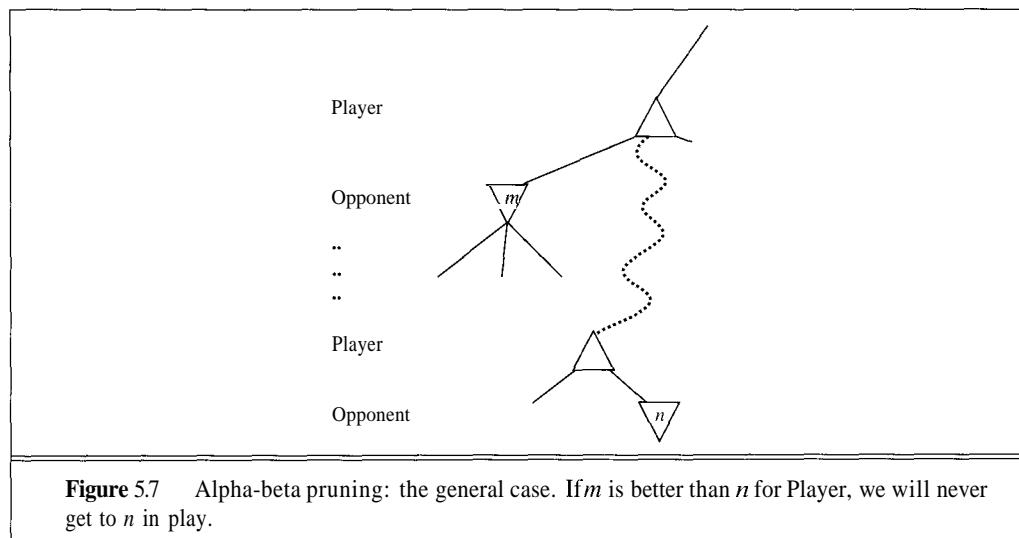
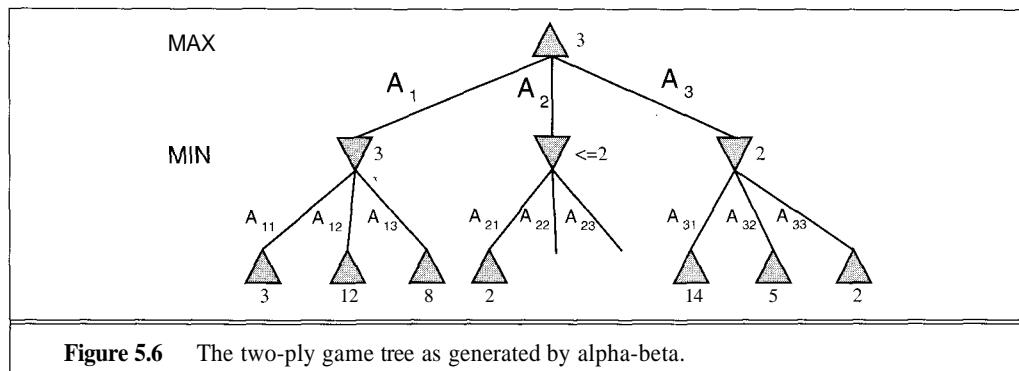
without examining it is called **pruning** the search tree. The particular technique we will examine is called **alpha-beta pruning**. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

Consider the two-ply game tree from Figure 5.2, shown again in Figure 5.6. The search proceeds as before:  $A_1$ , then  $A_1\backslash$ ,  $A_1\backslash 2$ ,  $A_{13}$ , and the node under  $A_1$  gets minimax value 3. Now we follow  $A_2$ , 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  $A_1$  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  $\alpha$  be the value of the best choice we have found so far at any choice point along the path for MAX, and  $\beta$  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  $\alpha$  and  $\beta$  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  $\alpha$  or  $\beta$  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



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  $A_3$  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,<sup>3</sup> 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

<sup>3</sup> Obviously, it cannot be done perfectly, otherwise the ordering function could be used to play a perfect game!

```

function MAX-VALUE(stategame, a,  $\beta$ ) 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
     $\beta$ , 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  $\leftarrow$  MAX(a, MIN-VALUE(s, game, a,  $\beta$ ))
    if a  $\geq$  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  $\leftarrow$  MIN(ft, MAX-VALUE(s, game, a,  $\beta$ ))
    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 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.

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.

The next step is to understand how to make correct decisions. Obviously, we still want to pick the move from  $A_1, \dots, A_n$  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.

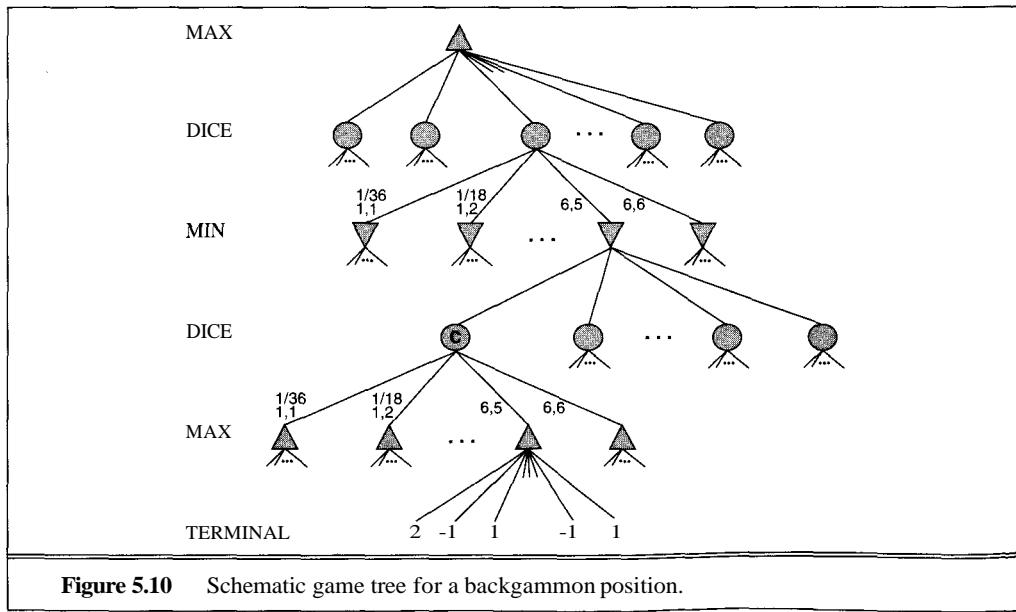
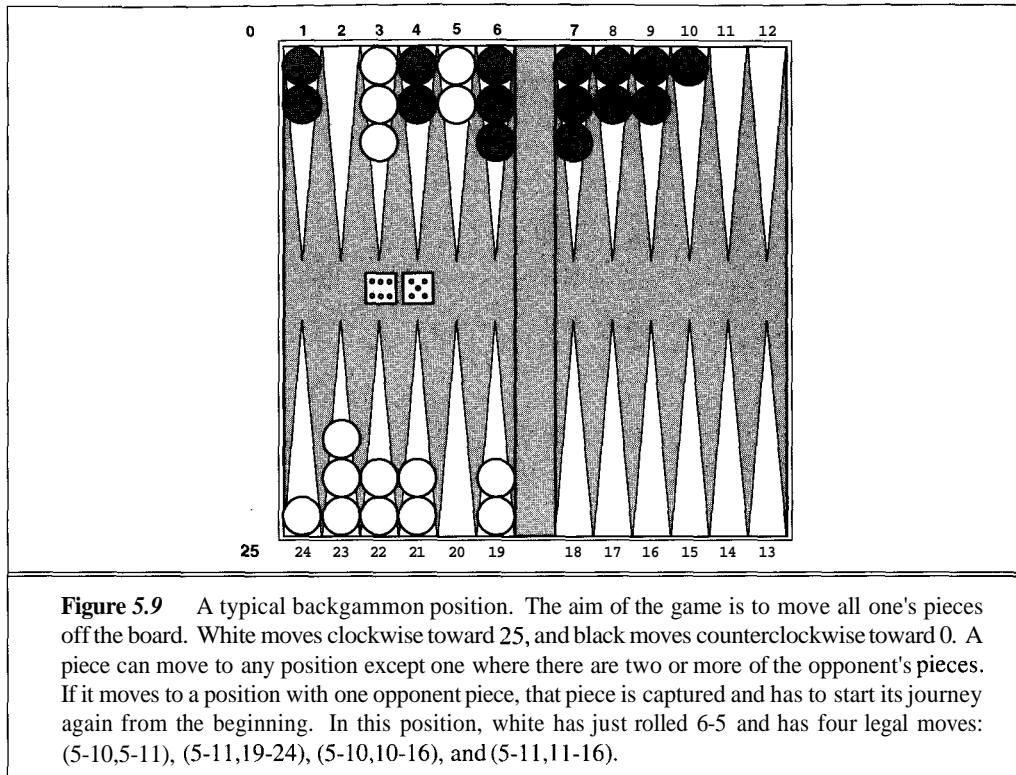
It is straightforward to calculate expected values of nodes. For terminal nodes, we use the utility function, just like in deterministic games. Going one step up in the search tree, we hit a chance node. In Figure 5.10, the chance nodes are circles; we will consider the one labelled C. Let  $d_i$  be 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  $S(C, d_i)$  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

$$\text{expectimax}(C) = \sum_i P(d_i) \max_{s \in S(C, d_i)} (\text{utility}(s))$$

CHANCE NODES

EXPECTED VALUE

EXPECTIMAX VALUE



EXPECTIMIN VALUE

This gives us the expected utility of the position at  $C$  assuming best play. Going up one more level to the MIN nodes ( $\nabla$  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.

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.

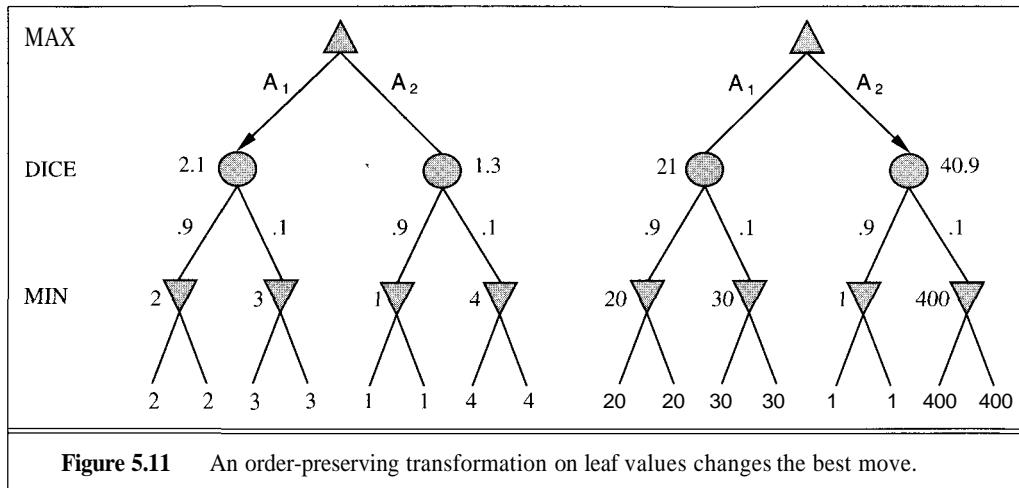
With chance nodes, we lose this freedom. Figure 5.11 shows what happens: with leaf values 1, 2, 3, 4, move  $A_1$  is best; with leaf values 1, 20, 30, 400, move  $A_2$  is best. Hence, the program behaves totally differently if we make a change in the scale of evaluation values! It turns out that to 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.



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

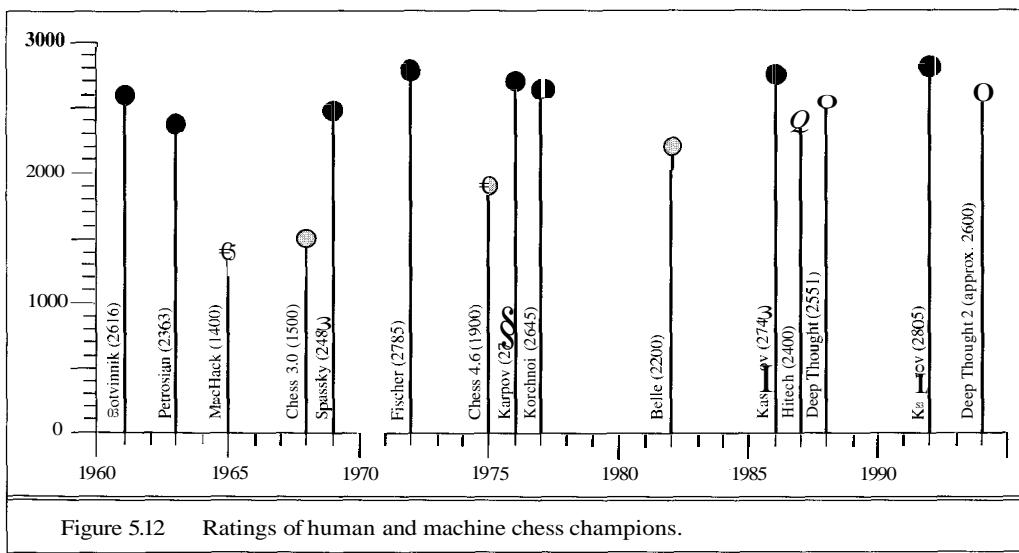


Figure 5.12 Ratings of human and machine chess champions.

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.

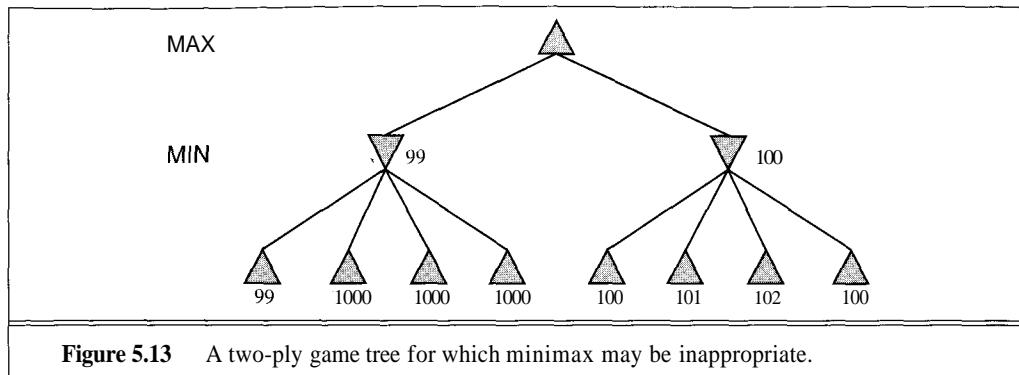
## Go

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 leafnode 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.



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 (Lefkowitz, 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

$$\text{Eval} = 3X_2 + X_1 - (3O_2 + O_1)$$

- 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  $n_j$ .

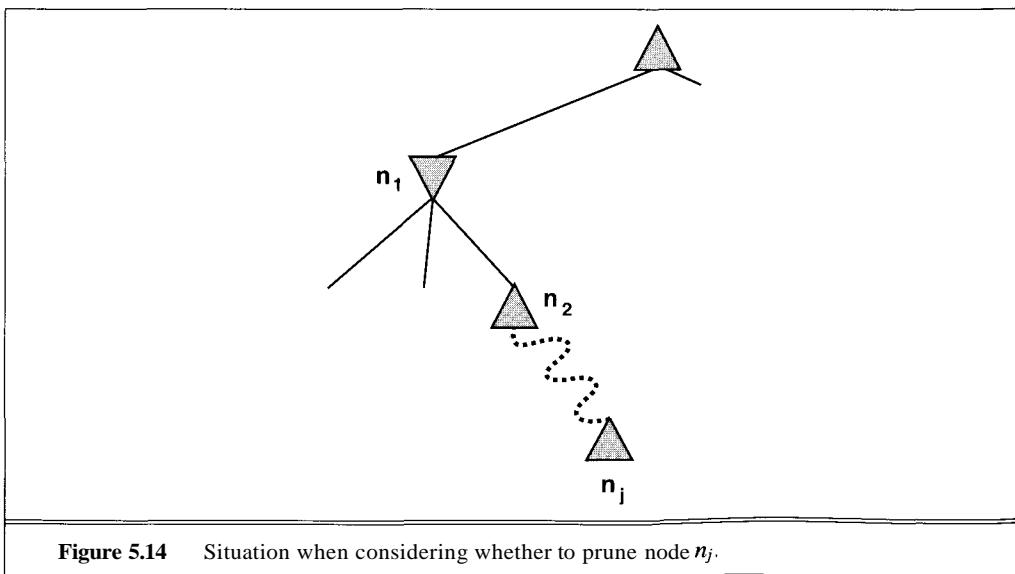
- The value of  $n_1$  is given by

$$n_1 = \min(n_2, n_{21}, \dots, n_{2b_2})$$

By writing a similar expression for the value of  $n_2$ , find an expression for  $n_1$  in terms of  $n_j$ .

- 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.
- 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.
- 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.



5.6 Prove that with a positive linear transformation of leaf values, the move choice remains unchanged in a game tree with chance nodes.

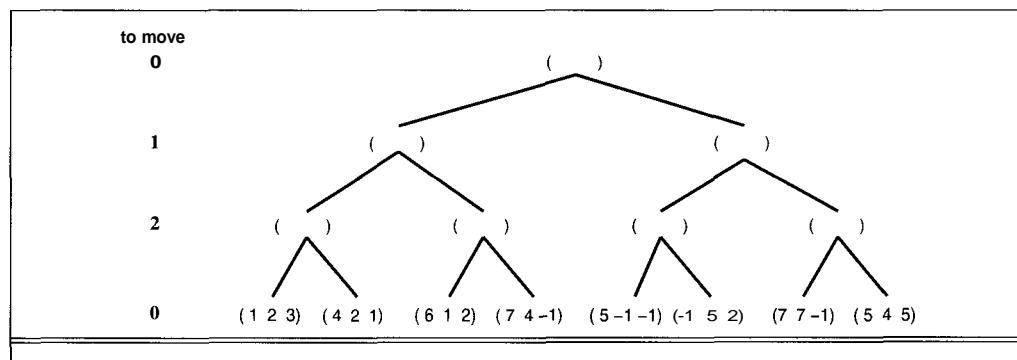
5.7 Consider the following procedure for choosing moves in games with chance nodes:

- 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.

# Part III

## KNOWLEDGE AND REASONING

In Part II, we showed that an agent that has goals and searches for solutions to the goals can do better than one that just reacts to its environment. We focused mainly on the question of how to carry out the search, leaving aside the question of general methods for describing states and actions.

In this part, we extend the capabilities of our agents by endowing them with the capacity for general logical reasoning. A logical, knowledge-based agent begins with some knowledge of the world and of its own actions. It uses logical reasoning to maintain a description of the world as new percepts arrive, and to deduce a course of action that will achieve its goals.

In Chapter 6, we introduce the basic design for a knowledge-based agent. We then present a simple logical language for expressing knowledge, and show how it can be used to draw conclusions about the world and to decide what to do. In Chapter 7, we augment the language to make it capable of expressing a wide variety of knowledge about complex worlds. In Chapter 8, we exercise this capability by expressing a significant fragment of commonsense knowledge about the real world, including time, change, objects, categories, events, substances, and of course money. In Chapters 9 and 10 we discuss the theory and practice of computer systems for logical reasoning.

# 6

# AGENTS THAT REASON LOGICALLY

*In which we design agents that can form representations of the world, use a process of inference to derive new representations about the world, and use these new representations to deduce what to do.*

In this chapter, we introduce the basic design for a knowledge-based agent. As we discussed in Part I (see, for example, the statement by Craik on page 13), the knowledge-based approach is a particularly powerful way of constructing an agent program. It aims to implement a view of agents in which they can be seen as *knowing* about their world, and *reasoning* about their possible courses of action. Knowledge-based agents are able to accept new tasks in the form of explicitly described goals; they can achieve competence quickly by being told or learning new knowledge about the environment; and they can adapt to changes in the environment by updating the relevant knowledge. A knowledge-based agent needs to know many things: the current state of the world; how to infer unseen properties of the world from percepts; how the world evolves over time; what it wants to achieve; and what its own actions do in various circumstances.

We begin in Section 6.1 with the overall agent design. Section 6.2 introduces a simple new environment, the wumpus world, where a knowledge-based agent can easily attain a competence that would be extremely difficult to obtain by other means. Section 6.3 discusses the basic elements of the agent design: a formal language in which knowledge can be expressed, and a means of carrying out reasoning in such a language. These two elements constitute what we call a **logic**. Section 6.4 gives an example of how these basic elements work in a logic called **propositional logic**, and Section 6.5 illustrates the use of propositional logic to build a logical agent for the wumpus world.

LOGIC

## 6.1 A KNOWLEDGE-BASED AGENT

KNOWLEDGE BASE  
SENTENCE

The central component of a knowledge-based agent is its **knowledge base**, or KB. Informally, a knowledge base is a set of representations of facts about the world. Each individual representation is called a **sentence**. (Here "sentence" is used as a technical term. It is related to the sentences

KNOWLEDGE  
REPRESENTATION  
LANGUAGE

INFERENCE

BACKGROUND  
KNOWLEDGE

of English and other natural languages, but is not identical.) The sentences are expressed in a language called a **knowledge representation language**.

There must be a way to add new sentences to the knowledge base, and a way to query what is known. The standard names for these tasks are TELL and ASK, respectively. The fundamental requirement that we will impose on TELL and ASK is that when one ASKS a question of the knowledge base, the answer should follow from what has been told (or rather, TELLed) to the knowledge base previously. Later in the chapter, we will be more precise about the crucial word "follow." For now, take it to mean that the knowledge base should not just make up things as it goes along. Determining what follows from what the KB has been TELLed is the job of the **inference** mechanism, the other main component of a knowledge-based agent.

Figure 6.1 shows the outline of a knowledge-based agent program. Like all our agents, it takes a percept as input and returns an action. The agent maintains a knowledge base, *KB*, which may initially contain some **background knowledge**. Each time the agent program is called, it does two things. First, it TELLS the knowledge base what it perceives.<sup>1</sup> Second, it ASKS the knowledge base what action it should perform. In the process of answering this query, logical reasoning is used to prove which action is better than all others, given what the agent knows and what its goals are. The agent then performs the chosen action.

```
function KB-AGENT(percept) returns an action
  static: KB, a knowledge base
    t, a counter, initially 0, indicating time
    TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
    action — ASK(KB, MAKE-ACTION-QUERY(t))
    TELL(KB, MAKE-ACTION-SENTENCE(action, t))
    t — t + 1
  return action
```

**Figure 6.1** A generic knowledge-based agent.

The details of the representation language are hidden inside two functions that implement the interface between the agent program "shell" and the core representation and reasoning system. MAKE-PERCEPT-SENTENCE takes a percept and a time and returns a sentence representing the fact that the agent perceived the percept at the given time, and MAKE-ACTION-QUERY takes a time as input and returns a sentence that is suitable for asking what action should be performed at that time. The details of the inference mechanism are hidden inside TELL and ASK. Later sections will reveal these details.

A careful examination of Figure 6.1 reveals that it is quite similar to the design of agents with internal state described in Chapter 2. Because of the definitions of TELL and ASK, however, the knowledge-based agent is not an arbitrary program for calculating actions based on the internal state variable. At any point, we can describe a knowledge-based agent at three levels:

<sup>1</sup> You might think of TELL and ASK as procedures that humans can use to communicate with knowledge bases. Don't be confused by the fact that here it is the agent that is TELLING things to its own knowledge base.

KNOWLEDGE LEVEL  
EPISTEMOLOGICAL LEVEL

LOGICAL LEVEL

IMPLEMENTATION LEVEL



DECLARATIVE

## 6.2 THE WUMPUS WORLD ENVIRONMENT

WUMPUS WORLD

Before launching into a full exposition of knowledge representation and reasoning, we will describe a simple environment class—the **wumpus world**—that provides plenty of motivation for logical reasoning. Wumpus was an early computer game, based on an agent who explores a cave consisting of rooms connected by passageways. Lurking somewhere in the cave is the wumpus, a beast that eats anyone who enters its room. To make matters worse, some rooms contain bottomless pits that will trap anyone who wanders into these rooms (except for the wumpus, who is too big to fall in). The only mitigating feature of living in this environment is the occasional heap of gold.

It turns out that the wumpus game is rather tame by modern computer game standards. However, it makes an excellent testbed environment for intelligent agents. Michael Genesereth was the first to suggest this.

- **The knowledge level or epistemological level** is the most abstract; we can describe the agent by saying what it knows. For example, an automated taxi might be said to know that the Golden Gate Bridge links San Francisco and Marin County. If TELL and ASK work correctly, then most of the time we can work at the knowledge level and not worry about lower levels.

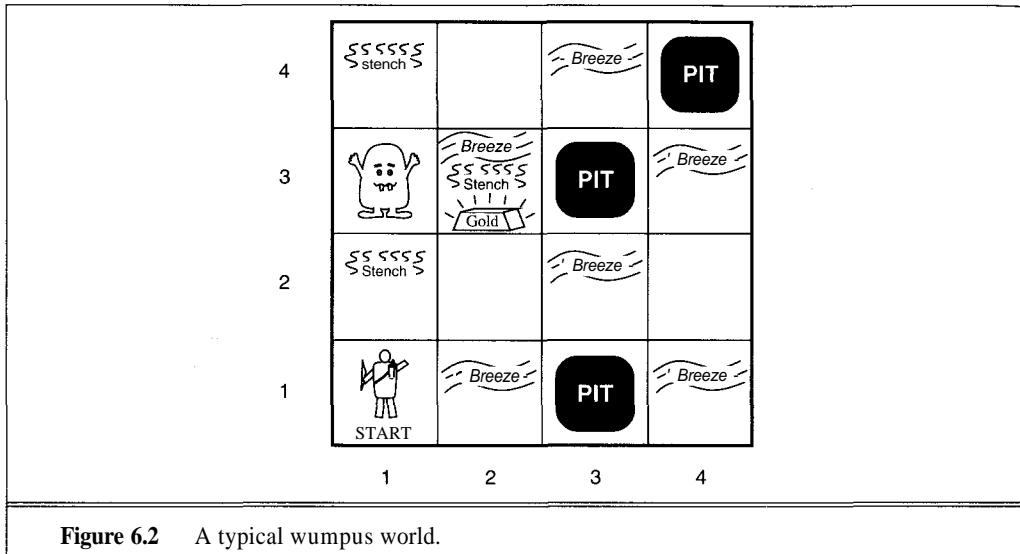
- **The logical level** is the level at which the knowledge is encoded into sentences. For example, the taxi might be described as having the logical sentence *Links(GGBridge, SF, Marin)* in its knowledge base.

- **The implementation level** is the level that runs on the agent architecture. It is the level at which there are physical representations of the sentences at the logical level. A sentence such as *Links(GGBridge, SF, Marin)* could be represented in the KB by the string "Links(GGBridge, SF, Marin)" contained in a list of strings; or by a "1" entry in a three-dimensional table indexed by road links and location pairs; or by a complex set of pointers connecting machine addresses corresponding to the individual symbols. The choice of implementation is very important to the efficient performance of the agent, but it is irrelevant to the logical level and the knowledge level.

We said that it is possible to understand the operation of a knowledge-based agent in terms of what it knows. *It is possible to construct a knowledge-based agent by TELLing it what it needs to know.* The agent's initial program, before it starts to receive percepts, is built by adding one by one the sentences that represent the designer's knowledge of the environment. Provided that the representation language makes it easy to express this knowledge in the form of sentences, this simplifies the construction problem enormously. This is called the **declarative** approach to system building. Also, one can design **learning** mechanisms that output general knowledge about the environment given a series of percepts. By hooking up a learning mechanism to a knowledge-based agent, one can make the agent fully autonomous.

## Specifying the environment

Like the vacuum world, the wumpus world is a grid of squares surrounded by walls, where each square can contain agents and objects. The agent always starts in the lower left corner, a square that we will label [1,1]. The agent's task is to find the gold, return to [1,1] and climb out of the cave. An example wumpus world is shown in Figure 6.2.



**Figure 6.2** A typical wumpus world.

To specify the agent's task, we specify its percepts, actions, and goals. In the wumpus world, these are as follows:

- In the square containing the wumpus and in the directly (not diagonally) adjacent squares the agent will perceive a stench.
- In the squares directly adjacent to a pit, the agent will perceive a breeze.
- In the square where the gold is, the agent will perceive a glitter.
- When an agent walks into a wall, it will perceive a bump.
- When the wumpus is killed, it gives out a woeful scream that can be perceived anywhere in the cave.
- The percepts will be given to the agent in the form of a list of five symbols; for example, if there is a stench, a breeze, and a glitter but no bump and no scream, the agent will receive the percept *[Stench, Breeze, Glitter, None, None]*. The agent *cannot* perceive its own location.
- Just as in the vacuum world, there are actions to go forward, turn right by 90°, and turn left by 90°. In addition, the action *Grab* can be used to pick up an object that is in the same square as the agent. The action *Shoot* can be used to fire an arrow in a straight line in the direction the agent is facing. The arrow continues until it either hits and kills the wumpus or hits the wall. The agent only has one arrow, so only the first *Shoot* action has any effect.

Finally, the action *Climb* is used to leave the cave; it is effective only when the agent is in the start square.

- The agent dies a miserable death if it enters a square containing a pit or a live wumpus. It is safe (but smelly) to enter a square with a dead wumpus.
- The agent's goal is to find the gold and bring it back to the start as quickly as possible, without getting killed. To be precise, 1000 points are awarded for climbing out of the cave while carrying the gold, but there is a 1-point penalty for each action taken, and a 10,000-point penalty for getting killed.

As we emphasized in Chapter 2, an agent can do well in a single environment merely by memorizing the sequence of actions that happens to work in that environment. To provide a real test, we need to specify a complete class of environments, and insist that the agent do well, on average, over the whole class. We will assume a  $4 \times 4$  grid surrounded by walls. The agent always starts in the square labeled (1,1), facing toward the right. The locations of the gold and the wumpus are chosen randomly, with a uniform distribution, from the squares other than the start square. In addition, each square other than the start can be a pit, with probability 0.2.

In most of the environments in this class, there is a way for the agent to safely retrieve the gold. In some environments, the agent must choose between going home empty-handed or taking a chance that could lead either to death or to the gold. And in about 21% of the environments (the ones where the gold is in a pit or surrounded by pits), there is no way the agent can get a positive score. Sometimes life is just unfair.

After gaining experience with this class of environments, we can experiment with other classes. In Chapter 22 we consider worlds where two agents explore together and can communicate with each other. We could also consider worlds where the wumpus can move, or where there are multiple troves of gold, or multiple wumpuses.<sup>2</sup>

## Acting and reasoning in the wumpus world

We now know the rules of the wumpus world, but we do not yet have an idea of how a wumpus world agent should act. An example will clear this up and will show why a successful agent will need to have some kind of logical reasoning ability. Figure 6.3(a) shows an agent's state of knowledge at the start of an exploration of the cave in Figure 6.2, after it has received its initial percept. To emphasize that this is only a representation, we use letters such as *A* and *OK* to represent sentences, in contrast to Figure 6.2, which used (admittedly primitive) pictures of the wumpus and pits.

From the fact that there was no stench or breeze in [1,1], the agent can infer that [1,2] and [2,1] are free of dangers. They are marked with an *OK* to indicate this. From the fact that the agent is still alive, it can infer that [1,1] is also *OK*. A cautious agent will only move into a square that it knows is *OK*. Let us suppose the agent decides to move forward to [2,1], giving the scene in Figure 6.3(b).

The agent detects a breeze in [2,1], so there must be a pit in a neighboring square, either [2,2] or [3,1]. The notation *P?* indicates a possible pit. The pit cannot be in [1,1], because the

<sup>2</sup> Or is it wumpi?

<table border="1" style="border-collapse: collapse; width: 100%; height: 100%;"> <tr><td>1,4</td><td>2,4</td><td>3,4</td><td>4,4</td></tr> <tr><td>1,3</td><td>2,3</td><td>3,3</td><td>4,3</td></tr> <tr><td>1,2</td><td>2,2</td><td>3,2</td><td>4,2</td></tr> <tr><td style="text-align: center;">OK</td><td></td><td></td><td></td></tr> <tr><td>1,1 A OK</td><td>2,1 OK</td><td>3,1 OK</td><td>4,1</td></tr> </table>	1,4	2,4	3,4	4,4	1,3	2,3	3,3	4,3	1,2	2,2	3,2	4,2	OK				1,1 A OK	2,1 OK	3,1 OK	4,1	<b>A</b> = Agent <b>B</b> = Breeze <b>G</b> = Glitter, Gold <b>OK</b> = Safe square <b>P</b> = Pit <b>S</b> = Stench <b>V</b> = Visited <b>W</b> = Wumpus	<table border="1" style="border-collapse: collapse; width: 100%; height: 100%;"> <tr><td>1,4</td><td>2,4</td><td>3,4</td><td>4,4</td></tr> <tr><td>1,3</td><td>2,3</td><td>3,3</td><td>4,3</td></tr> <tr><td>1,2</td><td>2,2 P?</td><td>3,2</td><td>4,2</td></tr> <tr><td style="text-align: center;">OK</td><td></td><td></td><td></td></tr> <tr><td>1,1 V OK</td><td>2,1 B OK</td><td>3,1 P? B</td><td>4,1</td></tr> </table>	1,4	2,4	3,4	4,4	1,3	2,3	3,3	4,3	1,2	2,2 P?	3,2	4,2	OK				1,1 V OK	2,1 B OK	3,1 P? B	4,1
1,4	2,4	3,4	4,4																																							
1,3	2,3	3,3	4,3																																							
1,2	2,2	3,2	4,2																																							
OK																																										
1,1 A OK	2,1 OK	3,1 OK	4,1																																							
1,4	2,4	3,4	4,4																																							
1,3	2,3	3,3	4,3																																							
1,2	2,2 P?	3,2	4,2																																							
OK																																										
1,1 V OK	2,1 B OK	3,1 P? B	4,1																																							
(a)		(b)																																								

Figure 6.3 The first step taken by the agent in the wumpus world. (a) The initial situation, after percept [None, None, None, None, None]. (b) After one move, with percept [None, Breeze, None, None, None].

<table border="1" style="border-collapse: collapse; width: 100%; height: 100%;"> <tr><td>1,4</td><td>2,4</td><td>3,4</td><td>4,4</td></tr> <tr><td>1,3 W!</td><td>2,3</td><td>3,3</td><td>4,3</td></tr> <tr><td>1,2 A S OK</td><td>2,2</td><td>3,2</td><td>4,2</td></tr> <tr><td style="text-align: center;">OK</td><td></td><td></td><td></td></tr> <tr><td>1,1 V OK</td><td>2,1 B V OK</td><td>3,1 P! V OK</td><td>4,1</td></tr> </table>	1,4	2,4	3,4	4,4	1,3 W!	2,3	3,3	4,3	1,2 A S OK	2,2	3,2	4,2	OK				1,1 V OK	2,1 B V OK	3,1 P! V OK	4,1	<b>A</b> = Agent <b>B</b> = Breeze <b>G</b> = Glitter, Gold <b>OK</b> = Safe square <b>P</b> = Pit <b>S</b> = Stench <b>V</b> = Visited <b>W</b> = Wumpus	<table border="1" style="border-collapse: collapse; width: 100%; height: 100%;"> <tr><td>1,4</td><td>2,4 P?</td><td>3,4</td><td>4,4</td></tr> <tr><td>1,3 W!</td><td>2,3 A S G B</td><td>3,3 P? B</td><td>4,3</td></tr> <tr><td>1,2 S V OK</td><td>2,2 V OK</td><td>3,2</td><td>4,2</td></tr> <tr><td style="text-align: center;">OK</td><td></td><td></td><td></td></tr> <tr><td>1,1 V OK</td><td>2,1 B V OK</td><td>3,1 P! V OK</td><td>4,1</td></tr> </table>	1,4	2,4 P?	3,4	4,4	1,3 W!	2,3 A S G B	3,3 P? B	4,3	1,2 S V OK	2,2 V OK	3,2	4,2	OK				1,1 V OK	2,1 B V OK	3,1 P! V OK	4,1
1,4	2,4	3,4	4,4																																							
1,3 W!	2,3	3,3	4,3																																							
1,2 A S OK	2,2	3,2	4,2																																							
OK																																										
1,1 V OK	2,1 B V OK	3,1 P! V OK	4,1																																							
1,4	2,4 P?	3,4	4,4																																							
1,3 W!	2,3 A S G B	3,3 P? B	4,3																																							
1,2 S V OK	2,2 V OK	3,2	4,2																																							
OK																																										
1,1 V OK	2,1 B V OK	3,1 P! V OK	4,1																																							
(a)		(b)																																								

Figure 6.4 Two later stages in the progress of the agent. (a) After the third move, with percept [Stench, None, None, None, None]. (b) After the fifth move, with percept [Stench, Breeze, Glitter, None, None].

agent was already there and did not fall in. At this point, there is only one known square that is *OK* and has not been visited yet. So the prudent agent will turn around, go back to [1,1], and then proceed to [1,2], giving the state of knowledge in Figure 6.4(a).

The agent detects a stench in [1,2], which means that there must be a wumpus nearby. But the wumpus cannot be in [1,1] (or it would have eaten the agent at the start), and it cannot be in

[2,2] (or the agent would have detected a stench when it was in [2,1]). Therefore, the agent can infer that the wumpus is in [1,3]. The notation  $W!$  indicates this. More interesting is that the lack of a *Breeze* percept in [1,2] means that there must be a pit in [3,1]. The reasoning is that no breeze in [1,2] means there can be no pit in [2,2]. But we already inferred that there must be a pit in either [2,2] or [3,1], so this means it must be in [3,1]. This is a fairly difficult inference, because it combines knowledge gained at different times in different places, and relies on the lack of a percept to make one crucial step. The inference is beyond the abilities of most animals, but it is typical of the kind of reasoning that a logical agent does.

After these impressive deductions, there is only one known unvisited *OK* square left, [2,2], so the agent will move there. We will not show the agent's state of knowledge at [2,2]; we just assume the agent turns and moves to [2,3], giving us Figure 6.4(b). In [2,3], the agent detects a glitter, so it should grab the gold and head for home, making sure its return trip only goes through squares that are known to be *OK*.

In the rest of this chapter, we describe how to build a logical agent that can represent beliefs such as "there is a pit in [2,2] or [3,1]" and "there is no wumpus in [2,2]," and that can make all the inferences that were described in the preceding paragraphs.

## 6.3 REPRESENTATION, REASONING, AND LOGIC

In this section, we will discuss the nature of representation languages, and of logical languages in particular, and explain in detail the connection between the language and the reasoning mechanism that goes with it. Together, representation and reasoning support the operation of a knowledge-based agent.

KNOWLEDGE  
REPRESENTATION

The object of **knowledge representation** is to express knowledge in computer-tractable form, such that it can be used to help agents perform well. A knowledge representation language is defined by two aspects:

SYNTAX

- **The syntax** of a language describes the possible configurations that can constitute sentences. Usually, we describe syntax in terms of how sentences are represented on the printed page, but the real representation is inside the computer: each sentence is implemented by a physical configuration or physical property of some part of the agent. For now, think of this as being a physical pattern of electrons in the computer's memory.

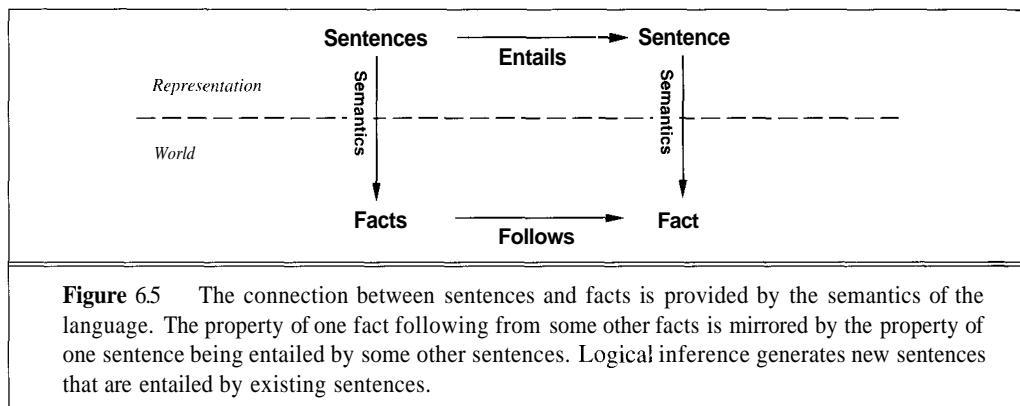
SEMANTICS

- **The semantics** determines the facts in the world to which the sentences refer. Without semantics, a sentence is just an arrangement of electrons or a collection of marks on a page. With semantics, each sentence makes a claim about the world. And with semantics, we can say that when a particular configuration exists within an agent, the agent **believes** the corresponding sentence.

For example, the syntax of the language of arithmetic expressions says that if  $x$  and  $y$  are expressions denoting numbers, then  $x > y$  is a sentence about numbers. The semantics of the language says that  $x > y$  is false when  $y$  is a bigger number than  $x$ , and true otherwise.

Provided the syntax and semantics are defined precisely, we can call the language a **logic**.<sup>3</sup> From the syntax and semantics, we can derive an inference mechanism for an agent that uses the language. We now explain how this comes about.

First, recall that the semantics of the language determine the fact to which a given sentence refers (see Figure 6.5). It is important to distinguish between facts and their representations. Facts are part of the world,<sup>4</sup> whereas their representations must be encoded in some way that can be physically stored within an agent. We cannot put the world inside a computer (nor can we put it inside a human), so all reasoning mechanisms must operate on representations of facts, rather than on the facts themselves. *Because sentences are physical configurations of parts of the agent, reasoning must be a process of constructing new physical configurations from old ones. Proper reasoning should ensure that the new configurations represent facts that actually follow from the facts that the old configurations represent.*



Consider the following example. From the fact that the solar system obeys the laws of gravitation, and the fact of the current arrangement of the sun, planets, and other bodies, it follows (so the astronomers tell us) that Pluto will eventually spin off into the interstellar void. But if our agent reasons improperly, it might start with representations of the first two facts and end with a representation that means that Pluto will shortly arrive in the vicinity of Bucharest. Or we might end up with "logical" reasoning like that in Figure 6.6.

We want to generate new sentences that are necessarily true, given that the old sentences are true. This relation between sentences is called **entailment**, and mirrors the relation of one fact following from another (Figure 6.5). In mathematical notation, the relation of entailment between a knowledge base  $KB$  and a sentence  $\alpha$  is pronounced " $KB$  entails  $\alpha$ " and written as

$$KB \models \alpha.$$

An inference procedure can do one of two things: given a knowledge base  $KB$ , it can generate

<sup>3</sup> This is perhaps a rather broad interpretation of the term "logic," one that makes "representation language" and "logic" synonymous. However, most of the principles of logic apply at this general level, rather than just at the level of the particular languages most often associated with the term.

<sup>4</sup> As Wittgenstein (1922) put it in his famous *Tractatus Logico-Philosophicus*: "The world is everything that is the case." We are using the word "fact" in this sense: as an "arrangement" of the world that may or may not be the case.



FIRST VILLAGER: We have found a witch. May we burn her?

ALL: A witch! Burn her!

BEDEVERE: Why do you think she is a witch?

SECOND VILLAGER: She turned *me* into a newt.

BEDEVERE: A newt?

SECOND VILLAGER (*after looking at himself for some time*): I got better.

ALL: Burn her anyway.

BEDEVERE: Quiet! Quiet! There are ways of telling whether she is a witch.

BEDEVERE: Tell me ... what do you do with witches?

ALL: Burn them.

BEDEVERE: And what do you burn, apart from witches?

FOURTH VILLAGER: ... Wood?

BEDEVERE: So why do witches burn?

SECOND VILLAGER: (*pianissimo*) Because they're made of wood?

BEDEVERE: Good.

ALL: I see. Yes, of course.

BEDEVERE: So how can we tell if she is made of wood?

FIRST VILLAGER: Make a bridge out of her.

BEDEVERE: Ah ... but can you not also make bridges out of stone?

ALL: Yes, of course ... um ... er ...

BEDEVERE: Does wood sink in water?

ALL: No, no, it floats. Throw her in the pond.

BEDEVERE: Wait. Wait ... tell me, what also floats on water?

ALL: Bread? No, no no. Apples ... gravy ... very small rocks ...

BEDEVERE: No, no no,

KING ARTHUR: A duck!

(*They all turn and look at ARTHUR. BEDEVERE looks up very impressed.*)

BEDEVERE: Exactly. So ... logically ...

FIRST VILLAGER (*beginning to pick up the thread*): If she ... weighs the same as a duck ... she's made of wood.

BEDEVERE: And therefore?

ALL: A witch!

**Figure 6.6** An example of "logical" reasoning gone wrong. (Excerpted with permission from *Monty Python and the Holy Grail*, © 1977, Reed Consumer Books.)

new sentences  $a$  that purport to be entailed by  $KB$ . Or, given a knowledge base  $KB$  and another sentence  $a$ , it can report whether or not  $a$  is entailed by  $KB$ . An inference procedure that generates only entailed sentences is called **sound or truth-preserving**.

An inference procedure  $\iota$  can be described by the sentences that it can derive. If  $\iota$  can derive  $a$  from  $KB$ , a logician would write

$$KB \vdash_i \alpha,$$

PROOF

which is pronounced "Alpha is derived from  $KB$  by  $i$ " or " $i$  derives alpha from  $KB$ ." Sometimes the inference procedure is implicit and the  $i$  is omitted. The record of operation of a sound inference procedure is called a **proof**.

COMPLETE

In understanding entailment and proof, it may help to think of the set of all consequences of  $KB$  as a haystack and  $a$  as a needle. Entailment is like the needle being in the haystack; proof is like finding it. For real haystacks, which are finite in extent, it seems obvious that a systematic examination can always decide whether the needle is in the haystack. This is the question of completeness: an inference procedure is **complete** if it can find a proof for any sentence that is entailed. But for many knowledge bases, the haystack of consequences is infinite, and completeness becomes an important issue.<sup>5</sup>



PROOF THEORY

We have said that sound inference is desirable. How is it achieved? *The key to sound inference is to have the inference steps respect the semantics of the sentences they operate upon. That is, given a knowledge base,  $KB$ , the inference steps should only derive new sentences that represent facts that follow from the facts represented by  $KB$ .* By examining the semantics of logical languages, we can extract what is called the **proof theory** of the language, which specifies the reasoning steps that are sound. Consider the following familiar example from mathematics, which illustrates syntax, semantics, and proof theory. Suppose we have the following sentence:

$$E = mc^2$$

The syntax of the "equation language" allows two expressions to be connected by an "=" sign. An expression can be a simple symbol or number, a concatenation of two expressions, two expressions joined by a "+" sign, and so on. The semantics of the language says that the two expressions on each side of "=" refer to the same quantity; that the concatenation of two expressions refers to the quantity that is the product of the quantities referred to by each of the expressions; and so on. From the semantics, we can show that a new sentence can be generated by, for example, concatenating the same expression to both sides of the equation:

$$ET = mc^2 T$$

Most readers will have plenty of experience with inference of this sort. Logical languages are like this simple equation language, but rather than dealing with algebraic properties and numerical quantities, they must deal with more or less everything we might want to represent and about which we might want to reason.

## Representation

We will now look a little more deeply into the nature of knowledge representation languages, with the aim of designing an appropriate syntax and semantics. We will begin with two familiar classes of languages, programming languages and natural languages, to see what they are good at representing and where they have problems.

Programming languages (such as C or Pascal or Lisp) are good for describing algorithms and concrete data structures. We could certainly imagine using an  $4 \times 4$  array to represent the contents of the wumpus world, for example. Thus, the programming language statement `World[2,2]—Pit` is a fairly natural way to say that there is a pit in square [2,2]. However, most

<sup>5</sup> Compare with the case of infinite search spaces in Chapter 3, where depth-first search is not complete.

programming languages do not offer any easy way to say "there is a pit in [2,2] or [3,1]" or "there is a wumpus in *some* square." The problem is that programming languages are designed to completely describe the state of the computer and how it changes as the program executes. But we would like our knowledge representation language to support the case where we do not have complete information—where we do not know for certain how things are, but only know some possibilities for how they might or might not be. A language that does not let us do this is not expressive enough.

Natural languages (such as English or Spanish) are certainly expressive—we managed to write this whole book using natural language with only occasional lapses into other languages (including logic, mathematics, and the language of diagrams). But natural languages have evolved more to meet the needs of **communication** rather than representation. When a speaker points and says, "Look!" the listener comes to know that, say, Superman has finally appeared over the rooftops. But we would not want to say that the sentence "Look!" encoded that fact. Rather, the meaning of the sentence depends both on the sentence itself and on the context in which the sentence was spoken. A natural language is a good way for a speaker to get a listener to come to know something, but often this sharing of knowledge is done without explicit representation of the knowledge itself. Natural languages also suffer from ambiguity—in a phrase such as "small dogs and cats," it is not clear whether the cats are small. Contrast this to the programming language construct " $-d + c$ ," where the precedence rules for the language tell us that the minus sign applies to  $d$ , not to  $d + c$ .

A good knowledge representation language should combine the advantages of natural languages and formal languages. It should be expressive and concise so that we can say everything we need to say succinctly. It should be unambiguous and independent of context, so that what we say today will still be interpretable tomorrow. And it should be effective in the sense that there should be an inference procedure that can make new inferences from sentences in our language.

Many representation languages have been designed to try to achieve these criteria. In this book, we concentrate on first-order logic as our representation language because it forms the basis of most representation schemes in AI. Just as it would be overoptimistic to believe that one can make real progress in physics without understanding and using equations, it is important to develop a talent for working with logical notation if one is to make progress in artificial intelligence. However, it is also important *not* to get too concerned with the *specifics* of logical notation—after all, there are literally dozens of different versions, some with  $x$ 's and  $>$ 's and exotic mathematical symbols, and some with rather visually appealing diagrams with arrows and bubbles. The main thing to keep hold of is how a precise, formal language can represent knowledge, and how mechanical procedures can operate on expressions in the language to perform reasoning. The fundamental concepts remain the same no matter what language is being used to represent the knowledge.

## Semantics

In logic, the **meaning** of a sentence is what it states about the world, that the world is *this* way and not *that* way. So how does a sentence get its meaning? How do we establish the correspondence between sentences and facts? Essentially, this is up to the person who wrote the sentence. In order to say what it means, the writer has to provide an **interpretation** for it; to say what fact

## THE LANGUAGE OF THOUGHT

Philosophers and psychologists have long pondered how it is that humans and other animals represent knowledge. It is clear that the evolution of natural language has played an important role in developing this ability in humans. But it is also true that humans seem to represent much of their knowledge in a nonverbal form. Psychologists have done studies to confirm that humans remember the "gist" of something they have read rather than the exact words. You could look at Anderson's (1980, page 96) description of an experiment by Wanner, or you could perform your own experiment by deciding which of the following two phrases formed the opening of Section 6.3:

"In this section, we will discuss the nature of representation languages ..."

"This section covers the topic of knowledge representation languages ..."

In Wanner's experiment, subjects made the right choice at chance level—about 50% of the time—but remembered the overall idea of what they read with better than 90% accuracy. This indicates that the exact words are not part of the representations they formed. A similar experiment (Sachs, 1967) showed that subjects remember the words for a short time (seconds to tens of seconds), but eventually forget the words and remember only the meaning. This suggests that people process the words to form some kind of nonverbal representation which they maintain as memories.

The exact mechanism by which language enables and shapes the representation of ideas in humans remains a fascinating question. The famous **Sapir-Whorf hypothesis** claims that the language we speak profoundly influences the way in which we think and make decisions, in particular by setting up the category structure by which we divide up the world into different sorts of objects. Whorf (1956) claimed that Eskimos have many words for snow, and thus experience snow in a different way from speakers of other languages. His analysis has since been discredited (Pullum, 1991); Inuit, Yupik, and other related languages seem to have about the same number of words for snow-related concepts as English (consider blizzard, sprinkling, flurries, powder, slush, snowbank, snowdrift, etc.). Of course, different languages *do* carve up the world differently. Spanish has two words for "fish," one for the live animal and one for the food. English does not make this distinction, but it does have the cow/beef distinction. There is no evidence that this means that English and Spanish speakers think about the world in fundamentally different ways.

For our purposes, it is important to remember that the language used to represent an agent's internal knowledge is quite different from the external language used to communicate with other agents. (See Chapter 22 for the study of communication.)

it corresponds to. A sentence does not mean something *by itself*. This is a difficult concept to accept, because we are used to languages like English where the interpretation of most things was fixed a long time ago.

The idea of interpretation is easier to see in made-up languages. Imagine that one spy wants to pass a message to another, but worries that the message may be intercepted. The two spies could agree in advance on a nonstandard interpretation in which, say, the interpretation of "Pope" is a particular piece of microfilm and the interpretation of "Denver" is the pumpkin left on the porch, and so forth. Then, when the first spy sends a newspaper clipping with the headline "The Pope is in Denver," the second spy will know that the microfilm is in the pumpkin.

COMPOSITIONAL

It is possible, in principle, to define a language in which every sentence has a completely arbitrary interpretation. But in practice, all representation languages impose a *systematic* relationship between sentences and facts. The languages we will deal with are all **compositional**—the meaning of a sentence is a function of the meaning of its parts. Just as the meaning of the mathematical expression  $x^2 + y^2$  is related to the meanings of  $x^2$  and  $y^2$ , we would like the meaning of the sentence " $S_{1,4}$  and  $S_{1,2}$ " to be related to the meanings of " $S_{1,4}$ " and " $S_{1,2}$ ". It would be very strange if " $S_{1,4}$ " meant there is a stench in square [1,4] and " $S_{1,2}$ " meant there is a stench in square [1,2], but " $S_{1,4}$  and  $S_{1,2}$ " meant that France and Poland drew 1-1 in last week's ice-hockey qualifying match. In Section 6.4, we describe the semantics of a simple language, the language of propositional logic, that obeys constraints like these. Such constraints make it easy to specify a proof theory that respects the semantics.



Once a sentence is given an interpretation by the semantics, the sentence says that the world is *this way* and not *that way*. Hence, it can be true or false. *A sentence is true under a particular interpretation if the state of affairs it represents is the case.* Note that truth depends both on the interpretation of the sentence and on the actual state of the world. For example, the sentence " $S_{1,2}$ " would be true under the interpretation in which it means that there is a stench in [1,2], in the world described in Figure 6.2. But it would be false in worlds that do not have a stench in [1,2], and it would be false in Figure 6.2 under the interpretation in which it means that there is a breeze in [1,2].

LOGICAL INFERENCE  
DEDUCTION

## Inference

The terms "reasoning" and "inference" are generally used to cover any process by which conclusions are reached. In this chapter, we are mainly concerned with sound reasoning, which we will call **logical inference or deduction**. Logical inference is a process that implements the entailment relation between sentences. There are a number of ways to approach the design of logical inference systems. We will begin with the idea of a **necessarily true sentence**.

VALID

## Validity and satisfiability

A sentence is **valid** or necessarily true if and only if it is true under all possible interpretations in all possible worlds, that is, regardless of what it is supposed to mean and regardless of the state of affairs in the universe being described. For example, the sentence

"There is a stench at [1,1] or there is not a stench at [1,1]."

is valid, because it is true whether or not "there is a stench in [1,1]" is true, and it is true regardless of the interpretation of "there is a stench in [1,1]." In contrast,

"There is an open area in the square in front of me or there is a wall in the square in front of me."

is not valid by itself. It is only valid under the assumption that every square has either a wall or an open area in it. So the sentence

"If every square has either a wall or an open area in it, then there is an open area in the square in front of me, or there is a wall in the square in front of me."

is valid.<sup>6</sup> There are several synonyms for valid sentences. Some authors use the terms **analytic sentences** or **tautologies** for valid sentences.

SATISFIABLE

A sentence is **satisfiable** if and only if there is some interpretation in some world for which it is true. The sentence "there is a wumpus at [1,2]" is satisfiable because there might well be a wumpus in that square, even though there does not happen to be one in Figure 6.2. A sentence that is not satisfiable is **unsatisfiable**. Self-contradictory sentences are unsatisfiable, if the contradictoriness does not depend on the meanings of the symbols. For example, the sentence

"There is a wall in front of me and there is no wall in front of me"  
is unsatisfiable.

### Inference in computers

It might seem that valid and unsatisfiable sentences are useless, because they can only express things that are obviously true or false. In fact, we will see that validity and unsatisfiability are crucial to the ability of a computer to reason.

The computer suffers from two handicaps: it does not necessarily know the interpretation you are using for the sentences in the knowledge base, and it knows nothing at all about the world except what appears in the knowledge base. Suppose we ask the computer if it is OK to move to square [2,2]. The computer does not know what OK means, nor does it know what a wumpus or a pit is. So it cannot reason informally as we did on page 155. All it can do is see if its knowledge base entails the sentence "[2,2] is OK." In other words, the inference procedure has to show that the sentence "If *KB* is true then [2,2] is OK" is a valid sentence. If it is valid, then it does not matter that the computer does not know the interpretation you are using or that it does not know much about the world—the conclusion is guaranteed to be correct under all interpretations in all worlds in which the original *KB* is true. In Section 6.4, we will give an example of a formal procedure for deciding if a sentence is valid.

What makes formal inference powerful is that there is no limit to the complexity of the sentences it can handle. When we think of valid sentences, we usually think of simple examples like "The wumpus is dead or the wumpus is not dead." But the formal inference mechanism can just as well deal with valid sentences of the form "If *KB* then *P*," where *KB* is a conjunction of thousands of sentences describing the laws of gravity and the current state of the solar system, and *P* is a long description of the eventual departure of Pluto from the system.

<sup>6</sup> In these examples, we are assuming that words like "if," "then," "every," "or" and "not" are part of the standard syntax of the language, and thus are not open to varying interpretation.

To reiterate, the great thing about formal inference is that it can be used to derive valid conclusions even when the computer does not know the interpretation you are using. The computer only reports valid conclusions, which must be true regardless of your interpretation. Because you know the interpretation, the conclusions will be meaningful to you, and they are guaranteed to follow from your premises. *The word “you” in this paragraph can be applied equally to human and computer agents.*

## Logics

To summarize, we can say that a logic consists of the following:

1. A formal system for describing states of affairs, consisting of
  - (a) the **syntax** of the language, which describes how to make sentences, and
  - (b) the **semantics** of the language, which states the systematic constraints on how sentences relate to states of affairs.
2. **The proof theory**—a set of rules for deducing the entailments of a set of sentences.

We will concentrate on two kinds of logic: propositional or Boolean logic, and first-order logic (more precisely, first-order predicate calculus with equality).

In **propositional logic**, symbols represent whole propositions (facts); for example, *D* might have the interpretation “the wumpus is dead.” which may or may not be a true proposition. Proposition symbols can be combined using **Boolean connectives** to generate sentences with more complex meanings. Such a logic makes very little commitment to how things are represented, so it is not surprising that it does not give us much mileage as a representation language.

First-order logic commits to the representation of worlds in terms of **objects** and **predicates** on objects (i.e., properties of objects or relations between objects), as well as using **connectives** and **quantifiers**, which allow sentences to be written about everything in the universe at once. First-order logic seems to be able to capture a good deal of what we know about the world, and has been studied for about a hundred years. We will spend therefore a good deal of time looking at how to do representation and deduction using it.

It is illuminating to consider logics in the light of their ontological and epistemological commitments. **Ontological commitments** have to do with the nature of *reality*. For example, propositional logic assumes that there are facts that either hold or do not in the world. Each fact can be in one of two states: true or false. First-order logic assumes more: namely, that the world consists of objects with certain relations between them that do or do not hold. Special-purpose logics make still further ontological commitments; for example, **temporal logic** assumes that the world is ordered by a set of time points or intervals, and includes built-in mechanisms for reasoning about time.

**Epistemological commitments** have to do with the possible states of *knowledge* an agent can have using various types of logic. In both propositional and first-order logic, a sentence represents a fact and the agent either believes the sentence to be true, believes it to be false, or is unable to conclude either way. These logics therefore have three possible states of belief regarding any sentence. Systems using probability theory, on the other hand, can have any *degree* of belief, ranging from 0 (total disbelief) to 1 (total belief). For example, a probabilistic

PROPOSITIONAL  
LOGIC

BOOLEAN  
CONNECTIVES

ONTOLOGICAL  
COMMITMENTS

TEMPORAL LOGIC

EPISTEMOLOGICAL  
COMMITMENTS

## FUZZY LOGIC

wumpus-world agent might believe that the wumpus is in [1,3] with probability 0.75. Systems based on **fuzzy logic** can have degrees of belief in a sentence, and also allow *degrees of truth*: a fact need not be true or false in the world, but can be true to a certain degree. For example, "Vienna is a large city" might be true only to degree 0.6. The ontological and epistemological commitments of various logics are summarized in Figure 6.7.

Language	Ontological Commitment (What exists in the world)	Epistemological Commitment (What an agent believes about facts)
Propositional logic	facts	true/false/unknown
First-order logic	facts, objects, relations	true/false/unknown
Temporal logic	facts, objects, relations, times	true/false/unknown
Probability theory	facts	degree of belief 0...1
Fuzzy logic	degree of truth	degree of belief 0...1

Figure 6.7 Formal languages and their ontological and epistemological commitments.

## 6.4 PROPOSITIONAL LOGIC: A VERY SIMPLE LOGIC

Despite its limited expressiveness, propositional logic serves to illustrate many of the concepts of logic just as well as first-order logic. We will describe its syntax, semantics, and associated inference procedures.

### Syntax

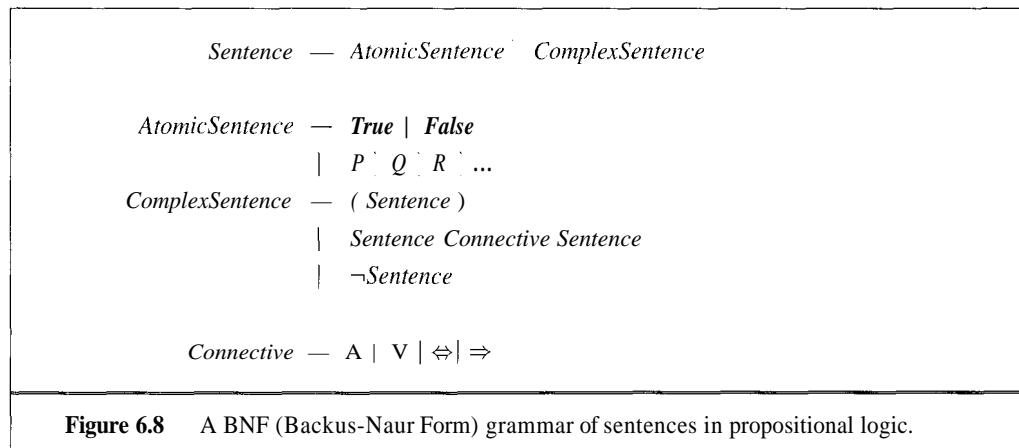
The **syntax** of propositional logic is simple. The symbols of propositional logic are the logical constants *True* and *False*, proposition symbols such as *P* and *Q*, the logical connectives *A*, *V*,  $\Leftrightarrow$ ,  $\Rightarrow$ , and  $\neg$ , and parentheses,  $( )$ . All sentences are made by putting these symbols together using the following rules:

- The logical constants *True* and *False* are sentences by themselves.
- A propositional symbol such as *P* or *Q* is a sentence by itself.
- Wrapping parentheses around a sentence yields a sentence, for example,  $(P \wedge Q)$ .
- A sentence can be formed by combining simpler sentences with one of the five logical connectives:

A (and). A sentence whose main connective is *A*, such as  $P \wedge (Q \vee R)$ , is called a **conjunction (logic)**; its parts are the **conjuncts**. (The *A* looks like an "A" for "And.")  
 V (or). A sentence using *V*, such as  $A \vee (P \wedge Q)$ , is a **disjunction** of the **disjuncts** *A* and  $(P \wedge Q)$ . (Historically, the *V* comes from the Latin "vel," which means "or." For most people, it is easier to remember as an upside-down and.)

IMPLICATION	$\Rightarrow$ (implies). A sentence such as $(P \wedge Q) \Rightarrow R$ is called an <b>implication</b> (or conditional).
PREMISE	Its <b>premise or antecedent</b> is $P \wedge Q$ , and its <b>conclusion or consequent</b> is $R$ . Implications are also known as <b>rules or if-then</b> statements. The implication symbol is sometimes written in other books as $\supset$ or $\rightarrow$ .
CONCLUSION	
EQUIVALENCE	$\Leftrightarrow$ (equivalent). The sentence $(P \wedge Q) \Leftrightarrow (Q \wedge P)$ is an <b>equivalence</b> (also called a <b>biconditional</b> ).
NEGATION	$\neg$ (not). A sentence such as $\neg P$ is called the <b>negation</b> of $P$ . All the other connectives combine two sentences into one; $\neg$ is the only connective that operates on a single sentence.
ATOMIC SENTENCES	
COMPLEX SENTENCES	
LITERAL	

Figure 6.8 gives a formal grammar of propositional logic; see page 854 if you are not familiar with the BNF notation. The grammar introduces **atomic sentences**, which in propositional logic consist of a single symbol (e.g.,  $P$ ), and **complex sentences**, which contain connectives or parentheses (e.g.,  $P \wedge Q$ ). The term literal is also used, meaning either an atomic sentence or a negated atomic sentence.



**Figure 6.8** A BNF (Backus-Naur Form) grammar of sentences in propositional logic.

Strictly speaking, the grammar is ambiguous — a sentence such as  $P \wedge Q \vee R$  could be parsed as either  $(P \wedge Q) \vee R$  or as  $P \wedge (Q \vee R)$ . This is similar to the ambiguity of arithmetic expressions such as  $P + Q \times R$ , and the way to resolve the ambiguity is also similar: we pick an order of precedence for the operators, but use parentheses whenever there might be confusion. The order of precedence in propositional logic is (from highest to lowest):  $\neg$ ,  $\wedge$ ,  $\vee$ ,  $\Rightarrow$ , and  $\Leftrightarrow$ . Hence, the sentence

$$\neg P \vee Q \wedge R \Rightarrow S$$

is equivalent to the sentence

$$((\neg P) \vee (Q \wedge R)) \Rightarrow S.$$

## Semantics

The **semantics** of propositional logic is also quite straightforward. We define it by specifying the interpretation of the proposition symbols and constants, and specifying the meanings of the logical connectives.

A proposition symbol can mean whatever you want. That is, its interpretation can be any arbitrary fact. The interpretation of  $P$  might be the fact that Paris is the capital of France or that the wumpus is dead. A sentence containing just a proposition symbol is satisfiable but not valid: it is true just when the fact that it refers to is the case.

With logical constants, you have no choice; the sentence *True* always has as its interpretation the way the world actually is—the true fact. The sentence *False* always has as its interpretation the way the world is not.

A complex sentence has a meaning derived from the meaning of its parts. Each connective can be thought of as a function. Just as addition is a function that takes two numbers as input and returns a number, so *and* is a function that takes two truth values as input and returns a truth value. We know that one way to define a function is to make a table that gives the output value for every possible input value. For most functions (such as addition), this is impractical because of the size of the table, but there are only two possible truth values, so a logical function with two arguments needs a table with only four entries. Such a table is called a **truth table**. We give truth tables for the logical connectives in Figure 6.9. To use the table to determine, for example, the value of *True V False*, first look on the left for the row where  $P$  is *true* and  $Q$  is *false* (the third row). Then look in that row under the  $P \vee Q$  column to see the result: *True*.

TRUTH TABLE

$P$	$Q$	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
<i>False</i>	<i>False</i>	<i>True</i>	<i>False</i>	<i>False</i>	<i>True</i>	<i>True</i>
<i>False</i>	<i>True</i>	<i>True</i>	<i>False</i>	<i>True</i>	<i>True</i>	<i>False</i>
<i>True</i>	<i>False</i>	<i>False</i>	<i>False</i>	<i>True</i>	<i>False</i>	<i>False</i>
<i>True</i>	<i>True</i>	<i>False</i>	<i>True</i>	<i>True</i>	<i>True</i>	<i>True</i>

Figure 6.9 Truth tables for the five logical connectives.

Truth tables define the semantics of sentences such as *True A True*. Complex sentences such as  $(P \vee Q) \wedge \neg S$  are defined by a process of decomposition: first, determine the meaning of  $(P \wedge Q)$  and of  $\neg S$ , and then combine them using the definition of the  $\wedge$  function. This is exactly analogous to the way a complex arithmetic expression such as  $(px) + -s$  is evaluated.

The truth tables for "and," "or," and "not" are in close accord with our intuitions about the English words. The main point of possible confusion is that  $P \vee Q$  is true when either *or both*  $P$  and  $Q$  are true. There is a different connective called "exclusive or" ("xor" for short) that gives false when both disjuncts are true.<sup>7</sup> There is no consensus on the symbol for exclusive or; two choices are  $\vee\bar{\wedge}$  and  $\oplus$ .

In some ways, the implication connective  $\Rightarrow$  is the most important, and its truth table might seem puzzling at first, because it does not quite fit our intuitive understanding of "P implies Q"

<sup>7</sup> Latin has a separate word, *aut*, for exclusive or.

or "if  $P$  then  $Q$ ." For one thing, propositional logic does not require any relation of causation or relevance between  $P$  and  $Q$ . The sentence "5 is odd implies Tokyo is the capital of Japan" is a true sentence of propositional logic (under the normal interpretation), even though it is a decidedly odd sentence of English. Another point of confusion is that any implication is true whenever its antecedent is false. For example, "5 is even implies Sam is smart" is true, regardless of whether Sam is smart. This seems bizarre, but it makes sense if you think of " $P \Rightarrow Q$ " as saying, "If  $P$  is true, then I am claiming that  $Q$  is true. Otherwise I am making no claim."

## Validity and inference

Truth tables can be used not only to define the connectives, but also to test for valid sentences. Given a sentence, we make a truth table with one row for each of the possible combinations of truth values for the proposition symbols in the sentence. For each row, we can calculate the truth value of the entire sentence. If the sentence is true in every row, then the sentence is valid. For example, the sentence

$$((P \vee H) \wedge \neg H) \Rightarrow P$$

is valid, as can be seen in Figure 6.10. We include some intermediate columns to make it clear how the final column is derived, but it is not important that the intermediate columns are there, as long as the entries in the final column follow the definitions of the connectives. Suppose  $P$  means that there is a wumpus in [1,3] and  $H$  means there is a wumpus in [2,2]. If at some point we learn  $(P \vee H)$  and then we also learn  $\neg H$ , then we can use the valid sentence above to conclude that  $P$  is true—that the wumpus is in [1,3].

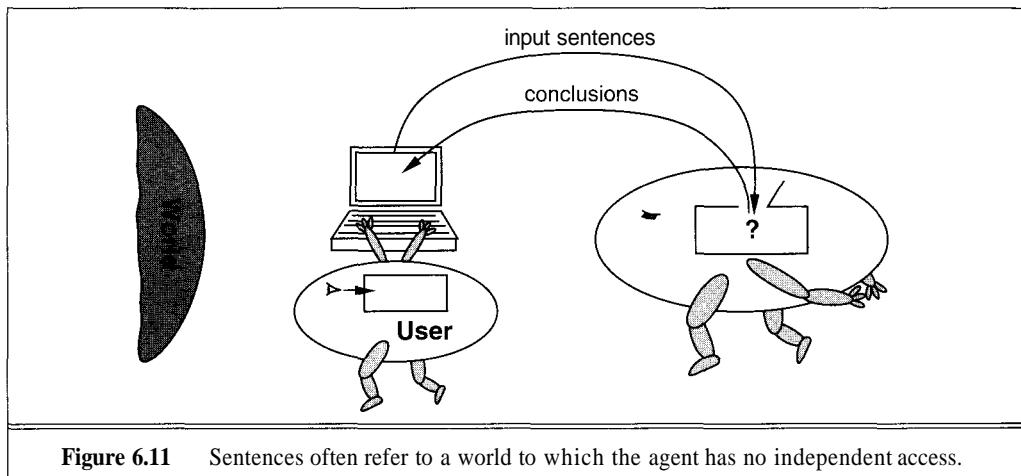
$P$	$H$	$P \vee H$	$(P \vee H) \wedge \neg H$	$((P \vee H) \wedge \neg H) \Rightarrow P$
<i>False</i>	<i>False</i>	<i>False</i>	<i>False</i>	<i>True</i>
<i>False</i>	<i>True</i>	<i>True</i>	<i>False</i>	<i>True</i>
<i>True</i>	<i>False</i>	<i>True</i>	<i>True</i>	<i>True</i>
<i>True</i>	<i>True</i>	<i>True</i>	<i>False</i>	<i>True</i>

Figure 6.10 Truth table showing validity of a complex sentence.

This is important. It says that if a machine has some premises and a possible conclusion, it can determine if the conclusion is true. It can do this by building a truth table for the sentence  $\text{Premises} \Rightarrow \text{Conclusion}$  and checking all the rows. If every row is true, then the conclusion is entailed by the premises, which means that the fact represented by the conclusion follows from the state of affairs represented by the premises. Even though the machine has no idea what the conclusion means, the user could read the conclusions and use his or her interpretation of the proposition symbols to see what the conclusions mean—in this case, that the wumpus is in [1,3]. Thus, we have fulfilled the promise made in Section 6.3.

It will often be the case that the sentences input into the knowledge base by the user refer to a world to which the computer has no independent access, as in Figure 6.11, where it is the user who observes the world and types sentences into the computer. It is therefore essential

that a reasoning system be able to draw conclusions that follow from the premises, regardless of the world to which the sentences are intended to refer. But it is a good idea for a reasoning system to follow this principle in any case. Suppose we replace the "user" in Figure 6.11 with a camera-based visual processing system that sends input sentences to the reasoning system. It makes no difference! Even though the computer now has "direct access" to the world, inference can still take place through direct operations on the syntax of sentences, without any additional information as to their intended meaning.



**Figure 6.11** Sentences often refer to a world to which the agent has no independent access.

## Models

MODEL

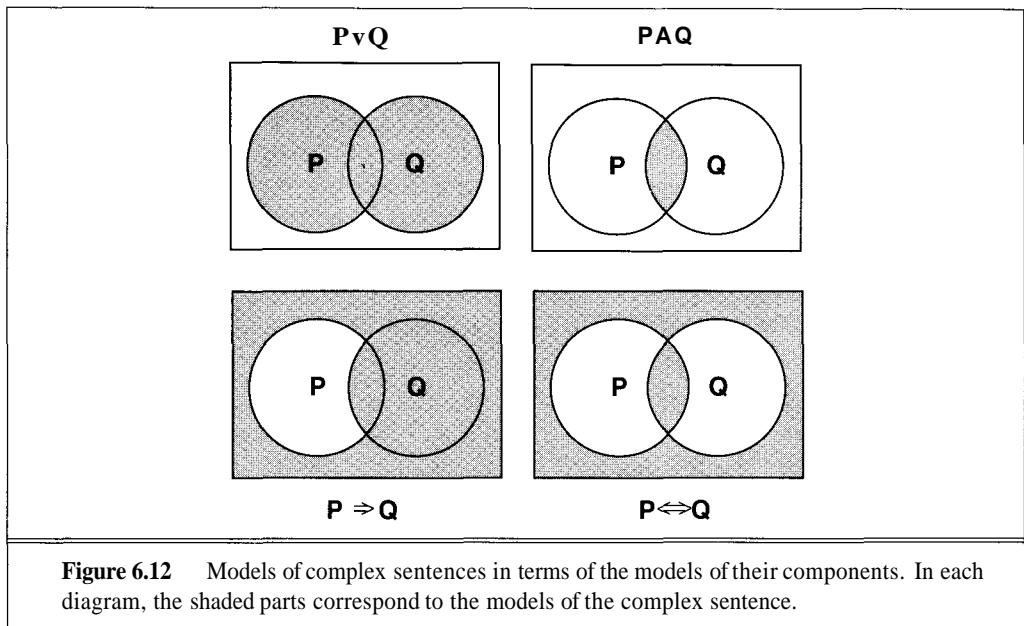
Any world in which a sentence is true under a particular interpretation is called a **model** of that sentence under that interpretation. Thus, the world shown in Figure 6.2 is a model of the sentence " $S_{1,2}$ " under the interpretation in which it means that there is a stench in [1,2]. There are many other models of this sentence—you can make up a world that does or does not have pits and gold in various locations, and as long as the world has a stench in [1,2], it is a model of the sentence. The reason there are so many models is because " $S_{1,2}$ " makes a very weak claim about the world. The more we claim (i.e., the more conjunctions we add into the knowledge base), the fewer the models there will be.



Models are very important in logic, because, to restate the definition of **entailment**, a sentence  $a$  is entailed by a knowledge base  $KB$  if the models of  $KB$  are all models of  $a$ . If this is the case, then whenever  $KB$  is true,  $a$  must also be true.

In fact, we could define the meaning of a sentence by means of set operations on sets of models. For example, the set of models of  $P \wedge Q$  is the intersection of the models of  $P$  and the models of  $Q$ . Figure 6.12 diagrams the set relationships for the four binary connectives.

We have said that models are worlds. One might feel that real worlds are rather messy things on which to base a formal system. Some authors prefer to think of models as *mathematical objects*. In this view, a model in propositional logic is simply a mapping from proposition symbols



directly to truth and falsehood, that is, the label for a row in a truth table. Then the models of a sentence are just those mappings that make the sentence true. The two views can easily be reconciled because each possible assignment of true and false to a set of proposition symbols can be viewed as an equivalence class of worlds that, under a given interpretation, have those truth values for those symbols. There may of course be many different "real worlds" that have the same truth values for those symbols. The only requirement to complete the reconciliation is that each proposition symbol be *either true or false* in each world. This is, of course, the basic ontological assumption of propositional logic, and is what allows us to expect that manipulations of symbols lead to conclusions with reliable counterparts in the actual world.

## Rules of inference for propositional logic

The process by which the soundness of an inference is established through truth tables can be extended to entire *classes* of inferences. There are certain patterns of inferences that occur over and over again, and their soundness can be shown once and for all. Then the pattern can be captured in what is called an **inference rule**. Once a rule is established, it can be used to make inferences without going through the tedious process of building truth tables.

We have already seen the notation  $a \vdash \beta$  to say that  $\beta$  can be derived from  $a$  by inference. There is an alternative notation,

$$\frac{a}{\beta}$$

which emphasizes that this is not a sentence, but rather an inference rule. Whenever something in the knowledge base matches the pattern above the line, the inference rule concludes the premise

below the line. The letters  $\alpha$ ,  $\beta$ , etc., are intended to match any sentence, not just individual proposition symbols. If there are several sentences, in either the premise or the conclusion, they are separated by commas. Figure 6.13 gives a list of seven commonly used inference rules.

An inference rule is sound if the conclusion is true in all cases where the premises are true. To verify soundness, we therefore construct a truth table with one line for each possible model of the proposition symbols in the premise, and show that in all models where the premise is true, the conclusion is also true. Figure 6.14 shows the truth table for the resolution rule.

$\diamond$  **Modus Ponens or Implication-Elimination:** (From an implication and the premise of the implication, you can infer the conclusion.)

$$\frac{a \Rightarrow \beta, \quad a}{\beta}$$

$\diamond$  **And-Elimination:** (From a conjunction, you can infer any of the conjuncts.)

$$\frac{\alpha_1 \wedge \alpha_2 \wedge \dots \wedge \alpha_n}{\alpha_i}$$

$\diamond$  **And-Introduction:** (From a list of sentences, you can infer their conjunction.)

$$\frac{\alpha_1, \alpha_2, \dots, \alpha_n}{\alpha_1 \wedge \alpha_2 \wedge \dots \wedge \alpha_n}$$

**0 Or-Introduction:** (From a sentence, you can infer its disjunction with anything else at all.)

$$\frac{\alpha_i}{\alpha_1 \vee \alpha_2 \vee \dots \vee \alpha_n}$$

$\diamond$  **Double-Negation Elimination:** (From a doubly negated sentence, you can infer a positive sentence.)

$$\frac{\neg\neg a}{a}$$

$\diamond$  **Unit Resolution:** (From a disjunction, if one of the disjuncts is false, then you can infer the other one is true.)

$$\frac{a \vee \beta, \quad \neg\beta}{a}$$

$\diamond$  **Resolution:** (This is the most difficult. Because 0 cannot be both true and false, one of the other disjuncts must be true in one of the premises. Or equivalently, implication is transitive.)

$$\frac{a \vee \beta, \quad \neg\beta \vee \gamma}{a \vee \gamma} \quad \text{or equivalently} \quad \frac{\neg\alpha \Rightarrow \beta, \quad \beta \Rightarrow \gamma}{\neg\alpha \Rightarrow \gamma}$$

**Figure 6.13** Seven inference rules for propositional logic. The unit resolution rule is a special case of the resolution rule, which in turn is a special case of the full resolution rule for first-order logic discussed in Chapter 9.

$\alpha$	$\beta$	7	$\alpha \vee \beta$	$\neg\beta \vee 7$	$\alpha \vee 7$
<u>False</u>	<u>False</u>	<u>False</u>	<u>False</u>	<u>True</u>	<u>False</u>
<u>False</u>	<u>False</u>	<u>True</u>	<u>False</u>	<u>True</u>	<u>True</u>
<u>False</u>	<u>True</u>	<u>False</u>	<u>True</u>	<u>False</u>	<u>False</u>
<u>False</u>	<u>True</u>	<u>True</u>	<u>True</u>	<u>True</u>	<u>True</u>
<u>True</u>	<u>False</u>	<u>False</u>	<u>True</u>	<u>True</u>	<u>True</u>
<u>True</u>	<u>False</u>	<u>True</u>	<u>True</u>	<u>True</u>	<u>True</u>
<u>True</u>	<u>True</u>	<u>False</u>	<u>True</u>	<u>False</u>	<u>True</u>
<u>True</u>	<u>True</u>	<u>True</u>	<u>True</u>	<u>True</u>	<u>True</u>

Figure 6.14 A truth table demonstrating the soundness of the resolution inference rule. We have underlined the rows where both premises are true.

As we mentioned above, a logical proof consists of a sequence of applications of inference rules, starting with sentences initially in the KB, and culminating in the generation of the sentence whose proof is desired. To prove that  $P$  follows from  $(P \vee H)$  and  $\neg H$ , for example, we simply require one application of the resolution rule, with  $a$  as  $P$ ,  $\beta$  as  $H$ , and 7 empty. The job of an inference procedure, then, is to construct proofs by finding appropriate sequences of applications of inference rules.

## Complexity of propositional inference

The truth-table method of inference described on page 169 is complete, because it is always possible to enumerate the  $2^n$  rows of the table for any proof involving  $n$  proposition symbols. On the other hand, the computation time is exponential in  $n$ , and therefore impractical. One might wonder whether there is a polynomial-time proof procedure for propositional logic based on using the inference rules from Section 6.4.

In fact, a version of this very problem was the first addressed by Cook (1971) in his theory of NP-completeness. (See also the appendix on complexity.) Cook showed that checking a set of sentences for satisfiability is NP-complete, and therefore unlikely to yield to a polynomial-time algorithm. However, this does not mean that all instances of propositional inference are going to take time proportional to  $2^n$ . In many cases, the proof of a given sentence refers only to a small subset of the KB and can be found fairly quickly. In fact, as Exercise 6.15 shows, really hard problems are quite rare.

The use of inference rules to draw conclusions from a knowledge base relies implicitly on a general property of certain logics (including propositional and first-order logic) called **monotonicity**. Suppose that a knowledge base  $KB$  entails some set of sentences. A logic is monotonic if when we add some new sentences to the knowledge base, all the sentences entailed by the original  $KB$  are still entailed by the new larger knowledge base. Formally, we can state the property of monotonicity of a logic as follows:

$$\text{if } KB_1 \models a \text{ then } (KB_1 \cup KB_2) \models a$$

This is true regardless of the contents of  $KB_2$ —it can be irrelevant or even contradictory to  $KB_1$ .

LOCAL

HORN SENTENCES

It is fairly easy to show that propositional and first-order logic are monotonic in this sense; one can also show that probability theory is not monotonic (see Chapter 14). An inference rule such as Modus Ponens is **local** because its premise need only be compared with a small portion of the KB (two sentences, in fact). Were it not for monotonicity, we could not have any local inference rules because the rest of the KB might affect the soundness of the inference. This would potentially cripple any inference procedure.

There is also a useful class of sentences for which a polynomial-time inference procedure exists. This is the class called **Horn sentences**.<sup>8</sup> A Horn sentence has the form:

$$P_1 \wedge P_2 \wedge \dots \wedge P_n \Rightarrow Q$$

where the  $P_i$  and  $Q$  are nonnegated atoms. There are two important special cases: -First, when  $Q$  is the constant *False*, we get a sentence that is equivalent to  $\neg P_1 \vee \dots \vee \neg P_n$ . Second, when  $n = 1$  and  $P_1 = \text{True}$ , we get *True*  $\Rightarrow Q$ , which is equivalent to the atomic sentence  $Q$ . Not every knowledge base can be written as a collection of Horn sentences, but for those that can, we can use a simple inference procedure: apply Modus Ponens wherever possible until no new inferences remain to be made. We discuss Horn sentences and their associated inference procedures in more detail in the context of first-order logic (Section 9.4).

## 6.5 AN AGENT FOR THE WUMPUS WORLD

---

In this section, we show a snapshot of a propositional logic agent reasoning about the wumpus world. We assume that the agent has reached the point shown in Figure 6.4(a), repeated here as Figure 6.15, and show how the agent can conclude that the wumpus is in [1,3].

### The knowledge base

On each turn, the agent's percepts are converted into sentences and entered into the knowledge base, along with some valid sentences that are entailed by the percept sentences. Let us assume that the symbol<sup>9</sup>  $S_{1,2}$ , for example, means "There is a stench in [1,2]." Similarly,  $B_{2,1}$  means "There is a breeze in [2,1]." At this point, then, the knowledge base contains, among others, the percept sentences

$$\begin{array}{ll} \neg S_{1,1} & \neg B_{1,1} \\ \neg S_{2,1} & B_{2,1} \\ S_{1,2} & \neg B_{1,2} \end{array}$$

In addition, the agent must start out with some knowledge of the environment. For example, the agent knows that if a square has no smell, then neither the square nor any of its adjacent squares

<sup>8</sup> Also known as **Horn clauses**. The name honors the logician Alfred Horn.

<sup>9</sup> The subscripts make these symbols look like they have some kind of internal structure, but do not let that mislead you. We could have used  $Q$  or *StenchOneTwo* instead of  $S_{1,2}$ , but we wanted symbols that are both mnemonic and succinct.

	1,4	2,4	3,4	4,4	<b>A</b> = Agent <b>B</b> = Breeze <b>G</b> = Glitter, Gold <b>OK</b> = Safe square <b>P</b> = Pit <b>S</b> = Stench <b>V</b> = Visited <b>W</b> = Wumpus
1,3 W!	2,3	3,3	4,3		
1,2 S OK	2,2 OK	3,2	4,2		
1,1 V OK	2,1 B V OK	3,1 P!	4,1		

Figure 6.15 The agent's knowledge after the third move. The current percept is [Stench, None, None, None, None].

can house a wumpus. The agent needs to know this for each square in the world, but here we just show sentences for three relevant squares, labeling each sentence with a rule number:

$$R_1 : \neg S_{1,1} \Rightarrow \neg W_{1,1} \wedge \neg W_{1,2} \wedge \neg W_{2,1}$$

$$R_2 : \neg S_{2,1} \Rightarrow \neg W_{1,1} \wedge \neg W_{2,1} \wedge \neg W_{2,2} \wedge \neg W_{3,1}$$

$$R_3 : \neg S_{1,2} \Rightarrow \neg W_{1,1} \wedge \neg W_{1,2} \wedge \neg W_{2,2} \wedge \neg W_{1,3}$$

Another useful fact is that if there is a stench in [1,2], then there must be a wumpus in [1,2] or in one or more of the neighboring squares. This fact can be represented by the sentence

$$R_4 : S_{1,2} \Rightarrow W_{1,3} \vee W_{1,2} \vee W_{2,2} \vee W_{1,1}$$

## Finding the wumpus

Given these sentences, we will now show how an agent can mechanically conclude  $W_{1,3}$ . All the agent has to do is construct the truth table for  $KB \Rightarrow W_{1,3}$  to show that this sentence is valid. There are 12 propositional symbols,<sup>10</sup> so the truth table will have  $2^{12} = 4096$  rows, and every row in which the sentence  $KB$  is true also has  $W_{1,3}$  true. Rather than show all 4096 rows, we use inference rules instead, but it is important to recognize that we could have done it in one (long) step just by following the truth-table algorithm.

First, we will show that the wumpus is not in one of the other squares, and then conclude by elimination that it must be in [1,3]:

1. Applying Modus Ponens with  $\neg S_{1,1}$  and the sentence labelled  $R_1$ , we obtain

$$\neg W_{1,1} \wedge \neg W_{1,2} \wedge \neg W_{2,1}$$

2. Applying And-Elimination to this, we obtain the three separate sentences

$$\neg W_{1,1} \quad \neg W_{1,2} \quad \neg W_{2,1}$$

<sup>10</sup> The 12 symbols are  $S_{1,1}, S_{2,1}, S_{1,2}, W_{1,1}, W_{1,2}, W_{2,1}, W_{2,2}, W_{3,1}, W_{1,3}, B_{1,1}, B_{2,1}, B_{1,2}$ .

3. Applying Modus Ponens to  $\neg S_{2,1}$  and the sentence labelled  $R_2$ , and then applying And-Elimination to the result, we obtain the three sentences

$$\neg W_{2,2} \quad \neg W_{2,1} \quad \neg W_{3,1}$$

4. Applying Modus Ponens to  $S_{1,2}$  and the sentence labelled  $R_4$ , we obtain

$$W_{1,3} \vee W_{1,2} \vee W_{2,2} \vee W_{1,1}$$

5. Now we apply the unit resolution rule, where  $\alpha$  is  $W_{1,3} \vee W_{1,2} \vee W_{2,2}$  and  $\beta$  is  $W_{1,1}$ . (We derived  $\neg W_{1,1}$  in step 2.) Unit resolution yields

$$W_{1,3} \vee W_{1,2} \vee W_{2,2}$$

6. Applying unit resolution again with  $W_{1,3} \vee W_{1,2}$  as  $\alpha$  and  $W_{2,2}$  as  $\beta$  ( $\neg W_{2,2}$  was derived in step 3), we obtain

$$W_{1,3} \vee W_{1,2}$$

7. Finally, one more resolution with  $W_{1,3}$  as  $\alpha$  and  $W_{1,2}$  as  $\beta$  (we derived  $\neg W_{1,2}$  in step 2) gives us the answer we want, namely, that the wumpus is in [1,3]:

$$W_{1,3}$$

## Translating knowledge into action

We have shown how propositional logic can be used to infer knowledge such as the whereabouts of the wumpus. But the knowledge is only useful if it helps the agent take action. To do that, we will need additional rules that relate the current state of the world to the actions the agent should take. For example, if the wumpus is in the square straight ahead, then it is a bad idea to execute the action *Forward*. We can represent this with a series of rules, one for each location and orientation in which the agent might be. Here is the rule for the case where the agent is in [1,1] facing east:

$$A_{1,1} \wedge East_A \wedge W_{2,1} \Rightarrow \neg Forward$$

Once we have these rules, we need a way to ASK the knowledge base what action to take. Unfortunately, propositional logic is not powerful enough to represent or answer the question "what action should I take?", but it is able to answer a series of questions such as "should I go forward?" or "should I turn right?" That means that the algorithm for a knowledge-based agent using propositional logic would be as in Figure 6.16.

## Problems with the propositional agent

Propositional logic allows us to get across all the important points about what a logic is and how it can be used to perform inference that eventually results in action. But propositional logic is so weak that it really cannot handle even a domain as simple as the wumpus world.

The main problem is that there are just too many propositions to handle. The simple rule "don't go forward if the wumpus is in front of you" can only be stated in propositional logic by a set of 64 rules (16 squares x 4 orientations for the agent). Naturally, it just gets worse if the

```

function PROPOSITIONAL-KB-AGENT(percept) returns an action
  static: KB, a knowledge base
    t, a counter, initially 0, indicating time

  TELL(KB,MAKE-PERCEPT-SENTENCE(percept,t))
  for each action in the list of possible actions do
    if ASK(KB,MAKE-ACTION-QUERY(t,action)) then
      t  $\leftarrow$  t + 1
      return action
  end

```

**Figure 6.16** A knowledge-based agent using propositional logic.

world is larger than a  $4 \times 4$  grid. Given this multiplication of rules, it will take thousands of rules to define a competent agent. The problem is not just that it is taxing to write the rules down, but also that having so many of them slows down the inference procedure. Remember that the size of a truth table is  $2^n$ , where  $n$  is the number of propositional symbols in the knowledge base.

Another problem is dealing with change. We showed a snapshot of the agent reasoning at a particular point in time, and all the propositions in the knowledge base were true at that time. But in general the world changes over time. When the agent makes its first move, the proposition  $A_{1,1}$  becomes false and  $A_{2,1}$  becomes true. But it may be important for the agent to remember where it was in the past, so it cannot just forget  $A_{1,1}$ . To avoid confusion, we will need different propositional symbols for the agent's location at each time step. This causes difficulties in two ways. First, we do not know how long the game will go on, so we do not know how many of these time-dependent propositions we will need. Second, we will now have to go back and rewrite time-dependent versions of each rule. For example, we will need

$$\begin{aligned}
 A_{1,1}^0 \wedge East_A^0 \wedge W_{2,1}^0 &\Rightarrow \neg Forward^0 \\
 A_{1,1}^1 \wedge East_A^1 \wedge W_{2,1}^1 &\Rightarrow \neg Forward^1 \\
 A_{1,1}^2 \wedge East_A^2 \wedge W_{2,1}^2 &\Rightarrow \neg Forward^2 \\
 &\vdots \\
 A_{1,1}^0 \wedge North_A^0 \wedge W_{1,2}^0 &\Rightarrow \neg Forward^0 \\
 A_{1,1}^1 \wedge North_A^1 \wedge W_{1,2}^1 &\Rightarrow \neg Forward^1 \\
 A_{1,1}^2 \wedge North_A^2 \wedge W_{1,2}^2 &\Rightarrow \neg Forward^2 \\
 &\vdots
 \end{aligned}$$

where the superscripts indicate times. If we want the agent to run for 100 time steps, we will need 6400 of these rules, just to say one should not go forward when the wumpus is there.

In summary, the problem with propositional logic is that it only has one representational device: the proposition. In the next chapter, we will introduce first-order logic, which can represent *objects* and *relations* between objects in addition to propositions. In first-order logic, the 6400 propositional rules can be reduced to one.

## 6.6 SUMMARY

---

We have introduced the idea of a knowledge-based agent, and showed how we can define a logic with which the agent can reason about the world and be guaranteed to draw correct conclusions, given correct premises. We have also showed how an agent can turn this knowledge into action. The main points are as follows:

- Intelligent agents need knowledge about the world in order to reach good decisions.
- Knowledge is contained in agents in the form of **sentences in a knowledge representation language** that are stored in a **knowledge base**.
- A knowledge-based agent is composed of a knowledge base and an inference mechanism. It operates by storing sentences about the world in its knowledge base, using the inference mechanism to infer new sentences, and using them to decide what action to take.
- A representation language is defined by its **syntax** and **semantics**, which specify the structure of sentences and how they relate to facts in the world.
- **The interpretation** of a sentence is the fact to which it refers. If it refers to a fact that is part of the actual world, then it is **true**.
- Inference is the process of deriving new sentences from old ones. We try to design **sound** inference processes that derive true conclusions given true premises. An inference process is **complete** if it can derive *all* true conclusions from a set of premises.
- A sentence that is true in all worlds under all interpretations is called **valid**. If an implication sentence can be shown to be valid, then we can derive its consequent if we know its premise. The ability to show validity independent of meaning is essential.
- Different logics make different commitments about what the world is made of and what kinds of beliefs we can have regarding facts.
- Logics are useful for the commitments they *do not* make, because the lack of commitment gives the knowledge base writer more freedom.
- Propositional logic commits only to the existence of facts that may or may not be the case in the world being represented. It has a simple syntax and semantics, but suffices to illustrate the process of inference.
- Propositional logic can accommodate certain inferences needed by a logical agent, but quickly becomes impractical for even very small worlds.

---

## BIBLIOGRAPHICAL AND HISTORICAL NOTES

Logic had its origins in ancient Greek philosophy and mathematics. Logical principles—principles connecting the syntactic structure of sentences with their truth and falsity, their meaning, or the validity of arguments in which they figure—can be found in scattered locations in the works of Plato (428–348 B.C.). The first known systematic study of logic was carried

out by Aristotle, whose work was assembled by his students after his death in 322 B.C. as a treatise called the *Organon*, the first systematic treatise on logic. However, Aristotle's logic was very weak by modern standards; except in a few isolated instances, he did not take account of logical principles that depend essentially on embedding one entire syntactic structure within another structure of the same type, in the way that sentences are embedded within other sentences in modern propositional logic. Because of this limitation, there was a fixed limit on the amount of internal complexity within a sentence that could be analyzed using Aristotelian logic.

The closely related Megarian and Stoic schools (originating in the fifth century B.C. and continuing for several centuries thereafter) introduced the systematic study of implication and other basic constructs still used in modern propositional logic. The Stoics claimed that their logic was complete in the sense of capturing all valid inferences, but what remains is too fragmentary to tell. A good account of the history of Megarian and Stoic logic, as far as it is known, is given by Benson Mates (1953).

The ideas of creating an artificial formal language patterned on mathematical notation in order to clarify logical relationships, and of reducing logical inference to a purely formal and mechanical process, were due to Leibniz (1646-1716). Leibniz's own mathematical logic, however, was severely defective, and he is better remembered simply for introducing these ideas as goals to be attained than for his attempts at realizing them.

George Boole (1847) introduced the first reasonably comprehensive and approximately correct system of logic based on an artificial formal language with his book *The Mathematical Analysis of Logic*. Boole's logic was closely modeled on the ordinary algebra of real numbers. Boole's system subsumed the main parts of Aristotelian logic and also contained a close analogue to modern propositional logic. Although Boole's system still fell short of full propositional logic, it was close enough that other 19th-century writers following Boole could quickly fill in the gaps. The first comprehensive exposition of modern propositional logic (and first-order logic) is found in Gottlob Frege's (1879) *Begriffschrift* ("Concept Writing" or "Conceptual Notation").

Truth tables as a method of testing the validity or unsatisfiability of sentences in the language of propositional logic were independently introduced simultaneously by Ludwig Wittgenstein (1922) and by Emil Post (1921). (As a method of explaining the meanings of propositional connectives, truth tables go back to Philo of Megara.)

Quine (1982) describes "truth-value analysis," a proof method closely resembling truth tables but more efficient because, in effect, it can handle multiple lines of the truth table simultaneously. Wang (1960) takes a general proof method for first-order logic designed by Gentzen (1934) and selects a convenient and efficient subset of inference rules for use in a tree-based procedure for deciding validity in propositional logic.

John McCarthy's (1968) paper "Programs with Common Sense" promulgated the notion of agents that use logical reasoning to mediate between percepts and actions. This paper was the first to make this conception widely known, although it draws on much earlier work (McCarthy, 1958). The 1968 paper is also called the "Advice Taker" paper because it introduces a hypothetical program by that name which uses logic to enable its designers to communicate useful knowledge to it without having to write further directly executable computer code. Alien Newell's (1982) article "The Knowledge Level" focuses on the use of logic by agent designers to describe the knowledge that is, in effect, being used by the agents they are designing, whether or not the agents themselves use explicit logical formulas to represent this knowledge internally. This theme of Newell's was

hinted at in 1943 by the psychologist Kenneth Craik, who writes, "My hypothesis then is that thought models, or parallels, reality—that its essential feature is not 'the mind,' 'the self,' 'sense-data,' nor propositions but symbolism, and that this symbolism is largely of the same kind as that which is familiar to us in mechanical devices which aid thought and calculation . . ." (Craik, 1943). Further work along these lines has been done by Rosenschein and Kaelbling (Rosenschein, 1985; Kaelbling and Rosenschein, 1990). Rosenschein and Genesereth (1987) have researched the problem of cooperative action among agents using propositional logic internally to represent the world. Gabbay (1991) has explored extensions to standard logic to enhance guidance of reasoning and retrieval from large knowledge bases.

## EXERCISES

**6.1** We said that truth tables can be used to establish the validity of a complex sentence. Show how they can be used to decide if a given sentence is valid, satisfiable, or unsatisfiable.

**6.2** Use truth tables to show that the following sentences are valid, and thus that the equivalences hold. Some of these equivalence rules have standard names, which are given in the right column.

$P \wedge (Q \wedge R)$	$\Leftrightarrow$	$(P \wedge Q) \wedge R$	Associativity of conjunction
$P \vee (Q \vee R)$	$\Leftrightarrow$	$(P \vee Q) \vee R$	Associativity of disjunction
$P \wedge Q$	$\Leftrightarrow$	$Q \wedge P$	Commutativity of conjunction
$P \vee Q$	$\Leftrightarrow$	$Q \vee P$	Commutativity of disjunction
$P \wedge (Q \vee R)$	$\Leftrightarrow$	$(P \wedge Q) \vee (P \wedge R)$	Distributivity of A over V
$P \vee (Q \wedge R)$	$\Leftrightarrow$	$(P \vee Q) \wedge (P \vee R)$	Distributivity of V over A
$\neg(P \wedge Q)$	$\Leftrightarrow$	$\neg P \vee \neg Q$	de Morgan's Law
$\neg(P \vee Q)$	$\Leftrightarrow$	$\neg P \wedge \neg Q$	de Morgan's Law
$P \Rightarrow Q$	$\Leftrightarrow$	$\neg Q \Rightarrow \neg P$	Contraposition
$\neg\neg P$	$\Leftrightarrow$	$P$	Double Negation
$P \Rightarrow Q$	$\Leftrightarrow$	$\neg P \vee Q$	
$P \Leftrightarrow Q$	$\Leftrightarrow$	$(P \Rightarrow Q) \wedge (Q \Rightarrow P)$	
$\neg P \Leftrightarrow \neg Q$	$\Leftrightarrow$	$(P \wedge \neg Q) \vee (\neg P \wedge Q)$	
$P \wedge \neg P$	$\Leftrightarrow$	<i>False</i>	
$P \vee \neg P$	$\Leftrightarrow$	<i>True</i>	

**6.3** Look at the following sentences and decide for each if it is valid, unsatisfiable, or neither. Verify your decisions using truth tables, or by using the equivalence rules of Exercise 6.2. Were there any that initially confused you?

- $\text{Smoke} \Rightarrow \text{Smoke}$
- $\text{Smoke} \Rightarrow \text{Fire}$
- $(\text{Smoke} \Rightarrow \text{Fire}) \Rightarrow (\neg \text{Smoke} \Rightarrow \neg \text{Fire})$
- $\text{Smoke} \vee \text{Fire} \vee \neg \text{Fire}$

- e.  $((Smoke \wedge Heat) \Rightarrow Fire) \Leftrightarrow ((Smoke \Rightarrow Fire) \vee (Heat \Rightarrow Fire))$
- f.  $(Smoke \Rightarrow Fire) \Rightarrow ((Smoke \wedge Heat) \Rightarrow Fire)$
- g.  $Big \vee Dumb \vee (Big \Rightarrow Dumb)$
- h.  $(Big \wedge Dumb) \vee \neg Dumb$

**6.4** Is the sentence "Either  $2 + 2 = 4$  and it is raining, or  $2 + 2 = 4$  and it is not raining" making a claim about arithmetic, weather, or neither? Explain.

**6.5** (Adapted from (Barwise and Etchemendy, 1993).) Given the following, can you prove that the unicorn is mythical? How about magical? Horned?

- \* If the unicorn is mythical, then it is immortal, but if it is not mythical, then it is a mortal mammal. If the unicorn is either immortal or a mammal, then it is horned.  
The unicorn is magical if it is horned.

**6.6** What ontological and epistemological commitments are made by the language of real-number arithmetic?

**6.7** Consider a world in which there are only four propositions,  $A$ ,  $B$ ,  $C$ , and  $D$ . How many models are there for the following sentences?

- a.  $A \wedge B$
- b.  $A \vee B$
- c.  $A \wedge A \wedge B \wedge C$

**6.8** We have defined four different binary logical connectives.

- a. Are there any others that might be useful?
- b. How many binary connectives can there possibly be?
- c. Why are some of them not very useful?

**6.9** Some agents make inferences as soon as they are told a new sentence, while others wait until they are asked before they do any inferencing. What difference does this make at the knowledge level, the logical level, and the implementation level?

**6.10** We said it would take 64 propositional logic sentences to express the simple rule "don't go forward if the wumpus is in front of you." What if we represented this fact with the single rule

*WumpusAhead*  $\Rightarrow \neg Forward$

Is this feasible? What effects does it have on the rest of the knowledge base?

**6.11** Provide a formal syntax, semantics, and proof theory for algebraic equations including variables, numbers,  $+$ ,  $-$ ,  $\times$ , and  $\div$ . You should be able to provide inference steps for most standard equation manipulation techniques.

**6.12** (Adapted from (Davis, 1990).) Jones, Smith, and Clark hold the jobs of programmer, knowledge engineer, and manager (not necessarily in that order). Jones owes the programmer \$10. The manager's spouse prohibits borrowing money. Smith is not married. Your task is to figure out which person has which job.

Represent the facts in propositional logic. You should have nine propositional symbols to represent the possible person/job assignments. For example, you might use the symbol *SM* to indicate that Smith is the manager. You do not need to represent the relation between owing and borrowing, or being married and having a spouse; you can just use these to draw conclusions (e.g., from "Smith is not married" and "the manager's spouse" we know that Smith can't be the manager, which you can represent as  $\neg SM$ ). The conjunction of all the relevant facts forms a sentence which you can call *KB*. The possible answers to the problem are sentences like *JP A SK A CM*. There are six such combinations of person/job assignments. Solve the problem by showing that only one of them is implied by *KB*, and by saying what its interpretation is.

**6.13** What is the performance score that one could expect from the optimal wumpus world agent? Design an experiment wherein you look at all possible  $4 \times 4$  wumpus worlds (or a random sample of them if there are too many), and for each one determine the shortest safe path to pick up the gold and return to start, and thus the best score. This gives you an idea of the expected performance score for an ideal omniscient agent. How could you determine the expected score for an optimal non-omniscient agent?



**6.14** Implement a function VALIDITY that takes a sentence as input and returns either valid, satisfiable, or unsatisfiable. Use it to answer the questions in Exercise 6.12. You will need to define an implementation-level representation of sentences. The cleanest way to do this is to define an abstract data type for compound sentences. Begin by writing EVAL-TRUTH as a recursive function that takes a sentence and an assignment of truth values to proposition symbols, and returns true or false. Then call EVAL-TRUTH for all possible assignments of truth values to the proposition symbols.



**6.15** SAT is the abbreviation for the satisfiability problem: given a propositional sentence, determine if it is satisfiable, and if it is, show which propositions have to be true to make the sentence true. 3SAT is the problem of finding a satisfying truth assignment for a sentence in a special format called 3-CNF, which is defined as follows:

- **A literal** is a proposition symbol or its negation (e.g., *P* or  $\neg P$ ).
- **A clause** is a disjunction of literals; a 3-clause is a disjunction of exactly 3 literals (e.g., *P V Q V*  $\neg R$ ).
- A sentence in CNF or **conjunctive normal form** is a conjunction of clauses; a 3-CNF sentence is a conjunction of 3-clauses.

For example,

$$(P \vee Q \vee \neg S) \wedge (\neg P \vee \neg Q \vee R) \wedge (\neg P \vee \neg R \vee \neg S) \wedge (P \vee \neg S \vee T)$$

is a 3-CNF sentence with four clauses and five proposition symbols.

In this exercise, you will implement and test GSAT, an algorithm for solving SAT problems that has been used to investigate how hard 3SAT problems are. GSAT is a random-restart, hill-

climbing search algorithm. The initial state is a random assignment of *true* and *false* to the proposition symbols. For example, for the preceding 3-CNF sentence, we might start with *P* and *Q* false and *R*, *S*, and *T* true.

The evaluation function measures the number of satisfied clauses, that is, clauses with at least one *true* disjunct. Thus, the initial state gets an evaluation of 3, because the second, third, and fourth clauses are true. If there are  $n$  proposition symbols, then there are  $n$  operators, where each operator is to change the truth assignment for one of the symbols. As a hill-climbing search, we always use the operator that yields the best evaluation (randomly choosing one if there are several equally good operators). Our example 3-CNF sentence is solved in one step, because changing *S* from *true* to *false* yields a solution. As with a random-restart algorithm, unless we find a solution after a certain amount of hill-climbing, we give up and start over from a new random truth assignment. After a certain number of restarts, we give up entirely. The complete algorithm is shown in Figure 6.17.

```
function GSAT(sentence,max-restarts, max-climbs) returns a truth assignment or failure
    for i — 1 to max-restarts do
        A — A randomly generated truth assignment
        for j — 1 to max-climbs do
            if A satisfies sentence then return A
            A — a random choice of one of the best successors of A
        end
    end
    return failure
```

**Figure 6.17** The GSAT algorithm for satisfiability testing. The successors of an assignment *A* are truth assignment with one symbol flipped. A "best assignment" is one that makes the most clauses true.

Answer the following questions about the algorithm:

- a. Is the GSAT algorithm sound?
- b. Is it complete?
- c. Implement GSAT and use it to solve the problems in Exercise 6.3.
- d. Use GSAT to solve randomly generated 3SAT problems of different sizes. There are two key parameters:  $N$ , the number of propositional symbols, and  $C$ , the number of clauses. We will investigate the effects of the ratio  $C/N$  on the execution time of GSAT. With  $N$  fixed at 20, make a graph of the median execution time versus  $C/N$  for  $C/N$  from 1 to 10. (The median is a better statistic than the mean because one or two outliers can really throw off the mean.) Use  $N$  as the value of *max-restarts* and  $5N$  as the value of *max-climbs*.
- e. Repeat for other values of  $N$  as time permits.
- f. What can you conclude about the difficulty of 3SAT problems for different values of  $C$ ,  $N$ , and the ratio  $C/N$ ?

See Selman *et al.* (1992) for more on GSAT. They present an implementation of GSAT that solves even the hardest 3SAT problems with  $N = 70$  in under a second. GSAT can be used to solve a wide variety of problems by constructing a **reduction** from each class of problems to 3SAT. It is so efficient that it often outperforms special-purpose algorithms that are expertly designed for specific problems.

**6.16** Consider the problem of designing a logical agent for the wumpus world using a Boolean circuit—that is, a collection of logic gates connecting the inputs (percept values) to outputs (action values).

- a. Explain why you would need flip-flops.
- b. Give an order-of-magnitude estimate of how many gates and flip-flops would you need.

# 7

# FIRST-ORDER LOGIC

*In which we introduce a logic that is sufficient for building knowledge-based agents.*

In Chapter 6, we showed how a knowledge-based agent could represent the world in which it operates and use those representations to deduce what actions to take. We used propositional logic as our representation language because it is one of the simplest languages that demonstrates all the important points. Unfortunately, propositional logic has a very limited ontology, making only the commitment that the world consists of facts. This made it difficult to represent even something as simple as the wumpus world.

FIRST-ORDER LOGIC  
OBJECTS  
PROPERTIES  
RELATIONS  
FUNCTIONS

In this chapter, we examine **first-order logic**,<sup>1</sup> which makes a stronger set of ontological commitments. The main one is that the world consists of **objects**, that is, things with individual identities and **properties** that distinguish them from other objects.

Among these objects, various **relations** hold. Some of these relations are **functions**—relations in which there is only one "value" for a given "input." It is easy to start listing examples of objects, properties, relations, and functions:

- Objects: people, houses, numbers, theories, Ronald McDonald, colors, baseball games, wars, centuries ...
- Relations: brother of, bigger than, inside, part of, has color, occurred after, owns ...
- Properties: red, round, bogus, prime, multistoried ...
- Functions: father of, best friend, third inning of, one more than ...

Indeed, almost any fact can be thought of as referring to objects and properties or relations. Some examples follow:

- "One plus two equals three"

Objects: one, two, three, one plus two; Relation: equals; Function: plus. (One plus two is a name for the object that is obtained by applying the function plus to the objects one and two. Three is another name for this object.)

- "Squares neighboring the wumpus are smelly."

Objects: wumpus, square; Property: smelly; Relation: neighboring.

---

<sup>1</sup> Also called **first-order predicate calculus**, and sometimes abbreviated as **FOL** or **FOPC**.

- "Evil King John ruled England in 1200."

Objects: John, England, 1200; Relation: ruled; Properties: evil, king.

First-order logic has been so important to mathematics, philosophy, and artificial intelligence precisely because those fields—and indeed, much of everyday human existence—can be usefully thought of as dealing with objects and the relations between them. We are not claiming that the world really *is* made up of objects and relations, just that dividing up the world that way helps us reason about it. First-order logic can also express facts about all of the objects in the universe. This, together with the implication connective from propositional logic, enables one to represent general laws or rules, such as the statement "Squares neighboring the wumpus are smelly."

Although first-order logic commits to the existence of objects and relations, it does not make an ontological commitment to such things as categories, time, and events, which also seem to show up in most facts about the world. Strangely enough, this reluctance to tackle categories, time, and events has not hurt the popularity of first-order logic; in fact it has contributed to its success. Important as these things are, there are just too many different ways to deal with them, and a logic that committed to a single treatment would only have limited appeal. By remaining neutral, first-order logic gives its users the freedom to describe these things in a way that is appropriate for the domain. This freedom of choice is a general characteristic of first-order logic. In the previous example we listed *King* as a property of people, but we could just as well have made *King* a relation between people and countries, or a function from countries to people (in a world in which each country has only one king).

There are many different representation schemes in use in AI, some of which we will discuss in later chapters. Some are theoretically equivalent to first-order logic and some are not. But first-order logic is universal in the sense that it can express anything that can be programmed. We choose to study knowledge representation and reasoning using first-order logic because it is by far the most studied and best understood scheme yet devised. Generally speaking, other proposals involving additional capabilities are still hotly debated and only partially understood. Other proposals that are a subset of first-order logic are useful only in limited domains. Despite its limitations, first-order logic will be around for a long time.

## 7.1 SYNTAX AND SEMANTICS

TERMS

In propositional logic every expression is a **sentence**, which represents a fact. First-order logic has sentences, but it also has **terms**, which represent objects. Constant symbols, variables, and function symbols are used to build terms, and quantifiers and predicate symbols are used to build sentences. Figure 7.1 gives a complete grammar of first-order logic, using Backus-Naur form (see page 854 if you are not familiar with this notation). A more detailed explanation of each element, describing both syntax and semantics, follows:

CONSTANT SYMBOLS

- ◊ **Constant symbols:** *A, B, C, John ...*

An interpretation must specify which object in the world is referred to by each constant symbol. Each constant symbol names exactly one object, but not all objects need to have names, and some can have several names. Thus, the symbol *John*, in one particular

```

Sentence — AtomicSentence
|   Sentence Connective Sentence
|   Quantifier Variable, . . . Sentence
|    $\neg$  Sentence
|   ( Sentence )

```

*AtomicSentence* — *Predicate(Term, . . .)* | *Term* = *Term*

```

Term —  $\rightarrow$  Function(Term, . . .)
|   Constant
\   Variable

```

*Connective* —  $\Rightarrow$  | A V |  $\Leftrightarrow$

*Quantifier* — V | 3

*Constant* — A | X | John | . . .

*Variable* — a | x | s | . . .

*Predicate* — Before | HasColor | Raining | . . .

*Function* — Mother | LeftLegOf | . . .

**Figure 7.1** The syntax of first-order logic (with equality) in BNF (Backus-Naur Form).

interpretation, might refer to the evil King John, king of England from 1199 to 1216 and younger brother of Richard the Lionheart. The symbol *King* could refer to the same object/person in the same interpretation.

PREDICATE  
SYMBOLS

TUPLES

#### ◊ **Predicate symbols:** *Round*, *Brother*, . . .

An interpretation specifies that a predicate symbol refers to a particular relation in the model. For example, the *Brother* symbol might refer to the relation of brotherhood. *Brother* is a binary predicate symbol, and accordingly brotherhood is a relation that holds (or fails to hold) between pairs of objects. In any given model, the relation is defined by the set of **tuples** of objects that satisfy it. A tuple is a collection of objects arranged in a fixed order. They are written with angle brackets surrounding the objects. In a model containing three objects, King John, Robin Hood, and Richard the Lionheart, the relation of brotherhood is defined by the set of tuples

$$\{ (\text{King John}, \text{Richard the Lionheart}), \\ (\text{Richard the Lionheart}, \text{King John}) \}$$

Thus, formally speaking, *Brother* refers to this set of tuples under the interpretation we have chosen.

## FUNCTION SYMBOLS

0 Function symbols: *Cosine*, *FatherOf*, *LeftLegOf*...

Some relations *are functional*—that is, any given object is related to exactly one other object by the relation. For example, any angle has only one number that is its cosine; any person has only one person that is his or her father. In such cases, it is often more convenient to define a function symbol (e.g., *Cosine*) that refers to the appropriate relation between angles and numbers. In the model, the mapping is just a set of  $n + 1$ -tuples with a special property, namely, that the last element of each tuple is the value of the function for the first  $n$  elements, and each combination of the first  $n$  elements appears in exactly one tuple. A table of cosines is just such a set of tuples—for each possible angle of interest, it gives the cosine of the angle. Unlike predicate symbols, which are used to state that relations hold among certain objects, function symbols are used to refer to particular objects without using their names, as we will see in the next section.

The choice of constant, predicate, and function symbols is entirely up to the user. A mathematician might want to use  $+$  and *Cosine*, a composer *Crescendo* and *F-sharp*. The names do not matter from a formal point of view, but it enhances readability if the intended interpretation of the symbols is clear. We return to this point in Section 8.1.

## Terms

A **term** is a logical expression that refers to an object. Constant symbols are therefore terms. Sometimes, it is more convenient to use an expression to refer to an object. For example, in English we might use the expression "King John's left leg" rather than giving a name to his leg. This is what function symbols are for: instead of using a constant symbol, we use *LeftLegOfJohn*. In the general case, a complex term is formed by a function symbol followed by a parenthesized list of terms as arguments to the function symbol. It is important to remember that a complex term is just a complicated kind of name. It is not a "subroutine call" that "returns a value." There is no *LeftLegOf* subroutine that takes a person as input and returns a leg. We can reason about left legs (e.g., stating the general rule that everyone has one and then deducing that John must have one) without ever providing a definition of *LeftLegOf*. This is something that cannot be done with subroutines in programming languages.

The formal semantics of terms is straightforward. An interpretation specifies a functional relation referred to by the function symbol, and objects referred to by the terms that are its arguments. Thus, the whole term refers to the object that appears as the  $(n+1)$ -th entry in that tuple in the relation whose first  $n$  elements are the objects referred to by the arguments. Thus, the *LeftLegOf* function symbol might refer to the following functional relation:

$$\{ (\text{King John}, \text{King John's left leg}), \\ (\text{Richard the Lionheart}, \text{Richard's left leg}) \}$$

and if *King John* refers to King John, then *LeftLegOf(King John)* refers, according to the relation, to King John's left leg.

## Atomic sentences

Now that we have terms for referring to objects, and predicate symbols for referring to relations, we can put them together to make **atomic sentences** that state "facts". An atomic sentence is formed from a predicate symbol followed by a parenthesized list of terms. For example,

*Brother(Richard, John)*

states, under the interpretation given before, that Richard the Lionheart is the brother of King John.<sup>2</sup> Atomic sentences can have arguments that are complex terms:

*Married(FatherOf(Richard), MotherOf(John))*

states that Richard the Lionheart's father is married to King John's mother (again, under a suitable interpretation). *An atomic sentence is true if the relation referred to by the predicate symbol holds between the objects referred to by the arguments.* The relation holds just in case the tuple of objects is in the relation. The truth of a sentence therefore depends on both the interpretation and the world.

## Complex sentences

We can use **logical connectives** to construct more complex sentences, just as in propositional calculus. The semantics of sentences formed using logical connectives is identical to that in the propositional case. For example:

- *Brother(Richard, John) A Brother(John, Richard)* is true just when John is the brother of Richard and Richard is the brother of John.
- *Older(John, 30) V Younger(John, 30)* is true just when John is older than 30 or John is younger than 30.
- *Older(John, 30)  $\Rightarrow$   $\neg$ Younger(John, 30)* states that if John is older than 30, then he is not younger than 30.<sup>3</sup>
- $\neg$ *Brother(Robin, John)* is true just when Robin is not the brother of John.

## Quantifiers

QUANTIFIERS

Once we have a logic that allows objects, it is only natural to want to express properties of entire collections of objects, rather than having to enumerate the objects by name. **Quantifiers** let us do this. First-order logic contains two standard quantifiers, called *universal* and *existential*.

### Universal quantification (V)

Recall the difficulty we had in Chapter 6 with the problem of expressing general rules in propositional logic. Rules such as "All cats are mammals" are the bread and butter of first-order logic.

<sup>2</sup> We will usually follow the argument ordering convention that  $P(x, y)$  is interpreted as "x is a P of y."

<sup>3</sup> Although these last two sentences may seem like tautologies, they are not. There are interpretations of *Younger* and *Older* in which they are false.

To express this particular rule, we will use unary predicates *Cat* and *Mammal*, thus, "Spot is a cat" is represented by *Cat(Spot)* and "Spot is a mammal" by *Mammal(Spot)*. In English, what we want to say is that for any object  $x$ , if  $x$  is a cat then  $x$  is a mammal. First-order logic lets us do this as follows:

$$\forall x \ Cat(x) \Rightarrow Mammal(x)$$

**V** is usually pronounced "For all . . ." Remember that the upside-down A stands for "all." You can think of a sentence  $\forall x P$ , where  $P$  is any logical expression, as being equivalent to the conjunction (i.e., the A) of all the sentences obtained by substituting the name of an object for the **variable**  $x$  wherever it appears in  $P$ . The preceding sentence is therefore equivalent to

$$\begin{aligned} Cat(Spot) &\Rightarrow Mammal(Spot) \\ Cat(Rebecca) &\Rightarrow Mammal(Rebecca) \\ Cat(Felix) &\Rightarrow Mammal(Felix) \\ Cat(Richard) &\Rightarrow Mammal(Richard) \\ Cat(John) &\Rightarrow Mammal(John) \\ &\vdots \end{aligned}$$

Thus, it is true if and only if all these sentences are true, that is, if  $P$  is true for all objects  $x$  in the universe. Hence V is called a **universal** quantifier.

We use the convention that all variables start with a lowercase letter, and that all constant, predicate, and function symbols are capitalized. A variable is a term all by itself, and as such can also serve as the argument of a function, for example, *ChildOf(x)*. A term with no variables is called a **ground term**.

It is worth looking carefully at the conjunction of sentences given before. If Spot, Rebecca, and Felix are known to be cats, then the first three conjuncts allow us to conclude that they are mammals. But what about the next two conjuncts, which appear to make claims about King John and Richard the Lionheart? Is that part of the meaning of "all cats are mammals"? In fact, these conjuncts are true, but make no claim whatsoever about the mammalian qualifications of John and Richard. This is because *Cat(Richard)* and *Cat(John)* are (presumably) false. Looking at the truth table for  $\Rightarrow$  (Figure 6.9), we see that the whole sentence is true whenever the left-hand side of the implication is false—*regardless* of the truth of the right-hand side. Thus, by asserting the universally quantified sentence, which is equivalent to asserting a whole list of individual implication sentences, we end up asserting the right-hand side of the rule just for those individuals for whom the left-hand side is true, and saying nothing at all about those individuals for whom the left-hand side is false. Thus, the truth-table entries for  $\Rightarrow$  turn out to be perfect for writing general rules with universal quantifiers.

It is tempting to think that the presence of the condition *Cat(x)* on the left-hand side of the implication means that somehow the universal quantifier ranges only over cats. This is perhaps helpful but not technically correct. Again, the universal quantification makes a statement about *everything*, but it does not make any claim about whether non-cats are mammals or not. On the other hand, if we tried to express "all cats are mammals" using the sentence

$$\forall x \ Cat(x) \text{ A } Mammal(x)$$

this would be equivalent to

$$\begin{aligned} & \text{Cat(Spot) A Mammal(Spot)A} \\ & \text{Cat(Rebecca)A Mammal(Rebecca)A} \\ & \text{Cat(Felix) A Mammal(Felix)A} \\ & \text{Cat(Richard)A Mammal(Richard) A} \\ & \text{Cat(John)A Mammal(John)A} \\ & \vdots \end{aligned}$$

Obviously, this does not capture what we want, because it says that Richard the Lionheart is both a cat and a mammal.

### Existential quantification (3)

Universal quantification makes statements about every object. Similarly, we can make a statement about *some* object in the universe without naming it, by using an existential quantifier. To say, for example, that Spot has a sister who is a cat, we write

$$\exists x \ Sister(x, Spot) A Cat(x)$$

$\exists$  is pronounced "There exists ...". In general,  $\exists x P$  is true if  $P$  is true for *some* object in the universe. It therefore can be thought of as equivalent to the disjunction (i.e., the V) of all the sentences obtained by substituting the name of an object for the variable  $x$ . Doing the substitution for the above sentence, we would get

$$\begin{aligned} & (Sister(Spot, Spot) A Cat(Spot))V \\ & (Sister(Rebecca, Spot) A Cat(Rebecca))V \\ & (Sister(Felix, Spot) A Cat(Felix))V \\ & (Sister(Richard, Spot) A Cat(Richard))V \\ & (Sister(John, Spot) A Cat(John))V \\ & \vdots \end{aligned}$$

The existentially quantified sentence is true just in case at least one of these disjuncts is true. If Spot had two sisters who were cats, then two of the disjuncts would be true, making the whole disjunction true also. This is entirely consistent with the original sentence "Spot has a sister who is a cat."<sup>4</sup>

Just as  $\Rightarrow$  appears to be the natural connective to use with V, A is the natural connective to use with 3. Using A as the main connective with V led to an overly strong statement in the example in the previous section; using  $\Rightarrow$  with 3 usually leads to a very weak statement indeed. Consider the following representation:

$$\exists x \ Sister(x, Spot) \Rightarrow Cat(x)$$


---

<sup>4</sup> There is a variant of the existential quantifier, usually written  $\exists^1$  or  $\exists!$ , that means "There exists exactly one ...". The same meaning can be expressed using equality statements, as we show in Section 7.1.

On the surface, this might look like a reasonable rendition of our sentence. But expanded out into a disjunction, it becomes

$$\begin{aligned}
 & (\text{Sister}(\text{Spot}, \text{Spot}) \Rightarrow \text{Cat}(\text{Spot})) \vee \\
 & (\text{Sister}(\text{Rebecca}, \text{Spot}) \Rightarrow \text{Cat}(\text{Rebecca})) \vee \\
 & (\text{Sister}(\text{Felix}, \text{Spot}) \Rightarrow \text{Cat}(\text{Felix})) \vee \\
 & (\text{Sister}(\text{Richard}, \text{Spot}) \Rightarrow \text{Cat}(\text{Richard})) \vee \\
 & (\text{Sister}(\text{John}, \text{Spot}) \Rightarrow \text{Cat}(\text{John})) \vee \\
 & \vdots
 \end{aligned}$$

Now, this disjunction will be satisfied if any of its disjuncts are true. An implication is true if both premise and conclusion are true, or if its premise is false. So if Richard the Lionheart is not Spot's sister, then the implication  $\text{Sister}(\text{Spot}, \text{Richard}) \Rightarrow \text{Cat}(\text{Richard})$  is true and the entire disjunction is therefore true. So, an existentially quantified implication sentence is true in a universe containing any object for which the premise of the implication is false; hence such sentences really do not say much at all.

### Nested quantifiers

We will often want to express more complex sentences using multiple quantifiers. The simplest case is where the quantifiers are of the same type. For example, "For all  $x$  and all  $y$ , if  $x$  is the parent of  $y$  then  $y$  is the child of  $x$ " becomes

$$\forall x, y \text{ Parent}(x, y) \Rightarrow \text{Child}(y, x)$$

$\forall x, y$  is equivalent to  $\forall x \ \forall y$ . Similarly, the fact that a person's brother has that person as a sibling is expressed by

$$\forall x, y \text{ Brother}(x, y) \Rightarrow \text{Sibling}(y, x)$$

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

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

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

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

The order of quantification is therefore very important. It becomes clearer if we put in parentheses. In general,  $\forall x \ (\exists y \ P(x, y))$ , where  $P(x, y)$  is some arbitrary sentence involving  $x$  and  $y$ , says that *every* object in the universe has a particular property, namely, the property that it is related to some object by the relation  $P$ . On the other hand,  $\exists x \ (\forall y \ P(x, y))$  says that there is *some* object in the world that has a particular property, namely the property of being related by  $P$  to *every* object in the world.

A minor difficulty arises when two quantifiers are used with the same variable name. Consider the sentence

$$\forall x \ [\text{Cat}(x) \vee (\exists x \text{ Brother}(\text{Richard}, x))]$$

Here the  $x$  in  $\text{Brother}(\text{Richard}, x)$  is *existentially* quantified. The rule is that the variable belongs to the innermost quantifier that mentions it; then it will not be subject to any other quantification.

This is just like variable scoping in block-structured programming languages like Pascal and Lisp, where an occurrence of a variable name refers to the innermost block that declared the variable. Another way to think of it is this:  $\exists x \ Brother(Richard, x)$  is a sentence about Richard (that he has a brother), not about  $x$ ; so putting a  $\forall x$  outside it has no effect. It could equally well have been written  $\exists z \ Brother(Richard, z)$ . Because this can be a source of confusion, we will always use different variables.

WFF

Every variable must be introduced by a quantifier before it is used. A sentence like  $\forall x P(y)$ , in which  $y$  does not have a quantifier, is incorrect.<sup>5</sup> The term **well-formed formula** or **wff** is sometimes used for sentences that have all their variables properly introduced.

### Connections between V and 3

The two quantifiers are actually intimately connected with each other, through negation. When one says that everyone dislikes parsnips, one is also saying that there does not exist someone who likes them; and vice versa:

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

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

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

Because V is really a conjunction over the universe of objects and 3 is a disjunction, it should not be surprising that they obey De Morgan's rules. The De Morgan rules for quantified and unquantified sentences are as follows:

$$\begin{array}{ll} \forall x \ \neg P = \neg \exists x \ P & \neg P \wedge \neg Q = \neg(P \vee Q) \\ \neg \forall x \ P = \exists x \ \neg P & \neg(P \wedge Q) = \neg P \vee \neg Q \\ \forall x \ P = \neg \exists x \ \neg P & P \wedge Q = \neg(\neg P \vee \neg Q) \\ \exists x \ P = \neg \forall x \ \neg P & P \vee Q = \neg(\neg P \wedge \neg Q) \end{array}$$

Thus, we do not really need both V and 3, just as we do not really need both A and V. Some formal logicians adopt a principle of parsimony, throwing out any syntactic item that is not strictly necessary. For the purposes of AI, the content, and hence the readability, of the sentences are important. Therefore, we will keep both of the quantifiers.

### Equality

EQUALITY SYMBOL

First-order logic includes one more way to make atomic sentences, other than using a predicate and terms as described earlier. We can use the **equality symbol** to make statements to the effect that two terms refer to the same object. For example,

$$Father(John) = Henry$$

says that the object referred to by  $Father(John)$  and the object referred to by  $Henry$  are the same.

<sup>5</sup> Sometimes there is an assumption that all unquantified variables are introduced with an implicit V. This is the case in most logic programming languages.

## IDENTITY RELATION

Equality can be viewed as a predicate symbol with a predefined meaning, namely, that it is fixed to refer to the **identity relation**. The identity relation is the set of all pairs of objects in which both elements of each pair are the same object:

$$\{ \langle \text{Spot}, \text{Spot} \rangle, \\ \langle \text{Rebecca}, \text{Rebecca} \rangle, \\ \langle \text{Felix}, \text{Felix} \rangle, \\ \langle \text{Richard the Lionheart}, \text{Richard the Lionheart} \rangle, \\ \langle \text{King John}, \text{King John} \rangle, \\ \langle \text{Henry II}, \text{Henry II} \rangle, \\ \dots \}$$

Thus, to see if  $\text{Father}(\text{John}) = \text{Henry}$  is true in a particular interpretation, we first look in the functional relation for  $\text{Father}$  and find the entry

$$\{ \dots \\ \langle \text{King John}, \text{Henry II} \rangle, \\ \dots \}$$

Then, because  $\text{Henry}$  refers to Henry II, the equality statement is true because  $(\text{Henry II}, \text{Henry II})$  is in the equality relation.

The equality symbol can be used to describe the properties of a given function, as we did above for the  $\text{Father}$  symbol. It can also be used with negation to insist that two terms are not the same object. To say that Spot has at least two sisters, we would write

$$\exists x, y \text{ Sister}(\text{Spot}, x) \wedge \text{Sister}(\text{Spot}, y) \wedge \neg(x = y)$$

If we simply wrote

$$\exists x, y \text{ Sister}(\text{Spot}, x) \wedge \text{Sister}(\text{Spot}, y)$$

that would not assert the existence of two distinct sisters, because nothing says that  $x$  and  $y$  have to be different. Consider the expansion of the existential statement into a disjunction: it will include as a disjunct

$$\dots \vee (\text{Sister}(\text{Spot}, \text{Rebecca}) \wedge \text{Sister}(\text{Spot}, \text{Rebecca})) \vee \dots$$

which occurs when  $\text{Rebecca}$  is substituted for both  $x$  and  $y$ . If  $\text{Rebecca}$  is indeed Spot's sister, then the existential will be satisfied because this disjunct will be true. The addition of  $\neg(x = y)$  makes the disjunct false, because  $\text{Rebecca} = \text{Rebecca}$  is necessarily true. The notation  $x \neq y$  is sometimes used as an abbreviation for  $\neg(x = y)$ .

## 7.2 EXTENSIONS AND NOTATIONAL VARIATIONS

In this section we look at three types of alternatives to first-order logic. First, we look at an extension called higher-order logic. Second, we consider some abbreviations that add no new power to first-order logic but do make the resulting sentences more concise. Third, we look at variations on our notation for first-order logic.

## Higher-order logic

First-order logic gets its name from the fact that one can quantify over *objects* (the first-order entities that actually exist in the world) but not over relations or functions on those objects.

**Higher-order logic** allows us to quantify over relations and functions as well as over objects. For example, in higher-order logic we can say that two objects are equal if and only if all properties applied to them are equivalent:

$$\forall x, y (x = y) \Leftrightarrow (\forall p p(x) \Leftrightarrow p(y))$$

Or we could say that two functions are equal if and only if they have the same value for all arguments:

$$\forall f, g (f = g) \Leftrightarrow (\forall x f(x) = g(x))$$

Higher-order logics have strictly more expressive power than first-order logic. As yet, however, logicians have little understanding of how to reason effectively with sentences in higher-order logic, and the general problem is known to be undecidable. In this book, we will stick to first-order logic, which is much better understood and still very expressive.

## Functional and predicate expressions using the A operator

It is often useful to be able to construct complex predicates and functions from simpler components, just as we can construct complex sentences from simpler components (e.g.,  $P \wedge Q$ ), or complex terms from simpler ones (e.g.,  $x^2 + y^3$ ). To turn the term  $x^2 - y^2$  into a function, we need to say what its arguments are: is it the function where you square the first argument and subtract the square of the second argument, or vice versa? The operator A (the Greek letter lambda) is traditionally used for this purpose. The function that takes the difference of the squares of its first and second argument is written as

$$\lambda x, y x^2 - y^2$$

This **A-expression**<sup>6</sup> can then be applied to arguments to yield a logical term in the same way that an ordinary, named function can:

$$(\lambda x, y x^2 - y^2)(25, 24) = 25^2 - 24^2 = 49$$

We will also find it useful (in Chapter 22) to generate A-expressions for predicates. For example, the two-place predicate "are of differing gender and of the same address" can be written

$$\lambda x, y \text{Gender}(x) \neq \text{Gender}(y) \wedge \text{Address}(x) = \text{Address}(y)$$

As one would expect, the application of a predicate A-expression to an appropriate number of arguments yields a logical sentence. Notice that the use of A in this way does *not* increase the formal expressive power of first-order logic, because any sentence that includes a A-expression can be rewritten by "plugging in" its arguments to yield a standard term or sentence.

---

<sup>6</sup> The same terminology is used in Lisp, where lambda plays exactly the same role as the A operator.

## The uniqueness quantifier $\exists!$

We have seen how to use  $\exists$  to say that *some* objects exist, but there is no concise way to say that a *unique* object satisfying some predicate exists. Some authors use the notation

$$\exists! x \ King(x)$$

to mean "there exists a unique object  $x$  satisfying  $King(x)$ " or more informally, "there's exactly one King." You should think of this not as adding a new quantifier,  $\exists!$ , but rather as being a convenient abbreviation for the longer sentence

$$\exists x \ King(x) \wedge \forall y \ King(y) \Rightarrow x = y$$

Of course, if we knew from the start there was only one King, we probably would have used the constant *King* rather than the predicate  $King(x)$ . A more complex example is "Every country has exactly one ruler":

$$\forall c \ Country(c) \Rightarrow \exists! r \ Ruler(r, c)$$

## The uniqueness operator $\iota$

It is convenient to use  $\exists!$  to state uniqueness, but sometimes it is even more convenient to have a term representing the unique object directly. The notation  $\iota / x P(x)$  is commonly used for this. (The symbol  $\iota$  is the Greek letter iota.) To say that "the unique ruler of Freedonia is dead" or equivalently "the  $r$  that is the ruler of Freedonia is dead," we would write:

$$Dead(\iota r Ruler(r, Freedonia))$$

This is just an abbreviation for the following sentence:

$$\exists! r \ Ruler(r, Freedonia) \wedge Dead(r)$$

## Notational variations

The first-order logic notation used in this book is the *de facto* standard for artificial intelligence; one can safely use the notation in a journal article without defining it, because it is recognizable to most readers. Several other notations have been developed, both within AI and especially in other fields that use logic, including mathematics, computer science, and philosophy. Here are some of the variations:

Syntax item	This book	Others
Negation (not)	$\neg P$	$\sim P \quad P$
Conjunction (and)	$P \wedge Q$	$P \& Q \quad P \quad Q \quad PQ \quad P, Q$
Disjunction (or)	$P \vee Q$	$P \mid Q \quad P; Q \quad P + Q$
Implication (if)	$P \Rightarrow Q$	$P \rightarrow Q \quad P \supset Q$
Equivalence (iff)	$P \Leftrightarrow Q$	$P \equiv Q \quad P \leftarrow Q$
Universal (all)	$\forall x \ P(x)$	$(\forall x)P(x) \quad \bigwedge x P(x) \quad P(x)$
Existential (exists)	$\exists x \ P(x)$	$(\exists x)P(x) \quad \bigvee x P(x) \quad P(Skolem_i)$
Relation	$R(x, y)$	$(R x y) \quad Rxy \quad xRy$

Two other common notations derive from implementations of logic in computer systems. The logic programming language Prolog (which we discuss further in Chapter 10) has two main differences. It uses uppercase letters for variables and lowercase for constants, whereas most other notations do the reverse. Prolog also reverses the order of implications, writing  $Q :- P$  instead of  $P \Rightarrow Q$ . A comma is used both to separate arguments and for conjunction, and a period marks the end of a sentence:

```
cat(X) :- furry(X), meows(X), has(X, claws).
```

In reasoning systems implemented in Lisp, a consistent prefix notation is common. Each sentence and nonconstant term is surrounded by parentheses, and the connectives come first, just like the predicate and function symbols. Because Lisp does not distinguish between uppercase and lowercase symbols, variables are usually distinguished by an initial ? or \$ character, as in this example:

```
(forall ?x  
  (implies (and (furry ?x) (meows ?x) (has ?x claws))  
            (cat ?x)))
```

## 7.3 USING FIRST-ORDER LOGIC

### DOMAIN

In knowledge representation, a **domain** is a section of the world about which we wish to express some knowledge. In this chapter, we start with some simple domains, using first-order logic to represent family relationships and mathematical sets. We then move on to show the knowledge that an agent in the wumpus world would need. Chapter 8 goes into more depth on the use of first-order logic for knowledge representation.

### The kinship domain

The first example we consider is the domain of family relationships, or kinship. This domain includes facts such as "Elizabeth is the mother of Charles" and "Charles is the father of William," and rules such as "If x is the mother of y and y is a parent of z, then x is a grandmother of z."

Clearly, the objects in our domain are people. The properties they have include gender, and they are related by relations such as parenthood, brotherhood, marriage, and so on. Thus, we will have two unary predicates, *Male* and *Female*. Most of the kinship relations will be binary predicates: *Parent*, *Sibling*, *Brother*, *Sister*, *Child*, *Daughter*, *Son*, *Spouse*, *Wife*, *Husband*, *Grandparent*, *Grandchild*, *Cousin*, *Aunt*, *Uncle*. We will use functions for *Mother* and *Father*, because every person has exactly one of each of these (at least according to nature's design).

We can go through each function and predicate, writing down what we know in terms of the other symbols. For example, one's mother is one's female parent:

$$\forall m, c \text{ } Mother(c) = m \Leftrightarrow Female(m) \wedge Parent(m, c)$$

One's husband is one's male spouse:

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

Male and female are disjoint categories:

$$\forall x \ Male(x) \Leftrightarrow \neg Female(x)$$

Parent and child are inverse relations:

$$\forall p, c \ Parent(p, c) \Leftrightarrow Child(c, p)$$

A grandparent is a parent of one's parent:

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

A sibling is another child of one's parents:

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

We could go on for several more pages like this, and in Exercise 7.6 we ask you to do just that.

## Axioms, definitions, and theorems

AXIOMS  
THEOREMS

Mathematicians write **axioms** to capture the basic facts about a domain, define other concepts in terms of those basic facts, and then use the axioms and definitions to prove **theorems**. In AI, we rarely use the term "theorem," but the sentences that are in the knowledge base initially are sometimes called "axioms," and it is common to talk of "definitions." This brings up an important question: how do we know when we have written down enough axioms to fully specify a domain? One way to approach this is to decide on a set of basic predicates in terms of which all the other predicates can be defined. In the kinship domain, for example, *Child*, *Spouse*, *Male*, and *Female* would be reasonable candidates for basic predicates. In other domains, as we will see, there is no clearly identifiable basic set.<sup>7</sup>

The converse problem also arises: do we have too many sentences? For example, do we need the following sentence, specifying that siblinghood is a symmetric relation?

$$\forall x, y \ Sibling(x, y) \Leftrightarrow Sibling(y, x)$$

In this case, we do not. From *Sibling(John, Richard)*, we can infer that

$$\exists p \ Parent(p, John) \wedge Parent(p, Richard),$$

INDEPENDENT  
AXIOM

and from that we can infer *Sibling(Richard, John)*. In mathematics, an **independent axiom** is one that cannot be derived from all the other axioms. Mathematicians strive to produce a minimal set of axioms that are all independent. In AI it is common to include redundant axioms, not because they can have any effect on what can be proved, but because they can make the process of finding a proof more efficient.

DEFINITION

An axiom of the form  $\forall x, y \ P(x, y) = \dots$  is often called a **definition** of *P*, because it serves to define exactly for what objects *P* does and does not hold. It is possible to have several definitions for the same predicate; for example, a triangle could be defined both as a polygon

<sup>7</sup> In *all* cases, the set of sentences will have models other than the intended model; this follows from a theorem of Löwenheim's stating that *all* consistent axiom sets have a model whose domain is the integers.

with three sides and as a polygon with three angles. Many predicates will have no complete definition of this kind, because we do not know enough to fully characterize them. For example, how would you complete the sentence:

$$\forall x \text{ Person}(x) \Leftrightarrow \dots$$

Fortunately, first-order logic allows us to make use of the *Person* predicate without completely defining it. Instead we can write partial specifications of properties that every person has and properties that make something a person:

$$\forall x \text{ Person}(x) \Rightarrow \dots$$

$$\forall x \dots \Rightarrow \text{Person}(x)$$

## The domain of sets

The domain of mathematical sets is somewhat more abstract than cats or kings, but nevertheless forms a coherent body of knowledge that we would like to be able to represent. We want to be able to represent individual sets, including the empty set. We need a way to build up sets by adding an element to a set or taking the union or intersection of two sets. We will want to know if an element is a member of a set, and to be able to distinguish sets from objects that are not sets.

We will use the normal vocabulary of set theory: *EmptySet* is a constant; *Member* and *Subset* are predicates; and *Intersection*, *Union*, and *Adjoin* are functions. (*Adjoin* is the function that adds one element to a set.) *Set* is a predicate that is true only of sets. The following eight axioms provide this:

1. The only sets are the empty set and those made by adjoining something to a set.

$$\forall s \text{ Set}(s) \Leftrightarrow (s = \text{EmptySet}) \vee (\exists x, s_2 \text{ Set}(s_2) \wedge s = \text{Adjoin}(x, s_2))$$

2. The empty set has no elements adjoined into it. (In other words, there is no way to decompose *EmptySet* into a smaller set and an element.)

$$\neg \exists x, s \text{ Adjoin}(x, s) - \text{EmptySet}$$

3. Adjoining an element already in the set has no effect:

$$\forall x, s \text{ Member}(x, s) \Leftrightarrow s = \text{Adjoin}(x, s)$$

4. The only members of a set are the elements that were adjoined into it. We express this recursively, saying that  $x$  is a member of  $s$  if and only if  $s$  is equal to some set  $s_2$  adjoined with some element  $y$ , where either  $y$  is the same as  $x$  or  $x$  is a member of  $s_2$ .

$$\begin{aligned} \forall x, s \text{ Member}(x, s) \Leftrightarrow \\ \exists y, s_2 (s = \text{Adjoin}(y, s_2) \wedge (x = y \vee \text{Member}(x, s_2))) \end{aligned}$$

5. A set is a subset of another if and only if all of the first set's members are members of the second set.

$$\forall s_1, s_2 \text{ Subset}(s_1, s_2) \Leftrightarrow (\forall x \text{ Member}(x, s_1) \Rightarrow \text{Member}(x, s_2))$$

6. Two sets are equal if and only if each is a subset of the other.

$$\forall s_1, s_2 (s_1 = s_2) \Leftrightarrow (\text{Subset}(s_1, s_2) \wedge \text{Subset}(s_2, s_1))$$

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

$$\forall x, s_1, s_2 \text{ } Member(x, Intersection(s_1, s_2)) \Leftrightarrow \\ Member(x, s_1) \wedge Member(x, s_2)$$

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

$$\forall x, s_1, s_2 \text{ } Member(x, Union(s_1, s_2)) \Leftrightarrow \\ Member(x, s_1) \vee Member(x, s_2)$$

The domain of lists is very similar to the domain of sets. The difference is that lists are ordered, and the same element can appear more than once in a list. We can use the vocabulary of Lisp for lists: *Nil* is the constant list with no elements; *Cons*, *Append*, *First*, and *Rest* are functions; and *Find* is the predicate that does for lists what *Member* does for sets. *List?* is a predicate that is true only of lists. Exercise 7.8 asks you to write axioms for the list domain.

## Special notations for sets, lists and arithmetic

SYNTACTIC SUGAR

Because sets and lists are so common, and because there is a well-defined mathematical notation for them, it is tempting to extend the syntax of first-order logic to include this mathematical notation. The important thing to remember is that when this is done, it is only a change to the syntax of the language; it does not change the semantics. The notation is just an abbreviation for the normal first-order logic notation. Such notational extensions are often called **syntactic sugar**. We will use standard mathematical notation for arithmetic and set theory from here on, and will also use the nonstandard notation  $\{as\}$  as an abbreviation for  $Adjoin(as)$ . There is less consensus among mathematicians about the notation for lists; we have chosen the notation used by Prolog:

$0 = EmptySet$	$[] = Nil$
$\{x\} = Adjoin(x, EmptySet)$	$[x] = Cons(x, Nil)$
$\{x, y\} = Adjoin(x, Adjoin(y, EmptySet))$	$[x, y] = Cons(x, Cons(y, Nil))$
$\{x, y, s\} = Adjoin(x, Adjoin(y, s))$	$[x, y   l] = Cons(x, Cons(y, l))$
$r \cup s = Union(r, s)$	
$r \cap s = Intersection(r, s)$	
$x \in s = Member(x, s)$	
$r \subset s = Subset(r, s)$	

We will also use standard arithmetic notation in logical sentences, saying, for example,  $x > 0$  instead of  $>(x, Zero)$  and  $1 + 2$  instead of  $+(1, 2)$ . It should be understood that each use of an infix mathematical operator is just an abbreviation for a standard prefix predicate or function. The integers are used as constant symbols, with their usual interpretation.

## Asking questions and getting answers

Although we will delay discussion of the internals of TELL and ASK until chapters 9 and 10, it is important to understand how they are used in first-order logic. If we want to add the kinship

sentences to a knowledge base  $KB$ , we would call

$\text{TELL}(KB, (\forall m, c \ Mother(c) = m \Leftrightarrow \text{Female}(m) \wedge \text{Parent}(m, c)))$

and so on. Now if we tell it

$\text{TELL}(KB, (\text{Female}(\text{Maxi}) \wedge \text{Parent}(\text{Maxi}, \text{Spot}) \wedge \text{Parent}(\text{Spot}, \text{Boots})))$

then we can

$\text{ASK}(KB, \text{Grandparent}(\text{Maxi}, \text{Boots}))$

ASSERTIONS  
QUERIES  
GOAL

and receive an affirmative answer. The sentences added using TELL are often called **assertions**, and the questions asked using ASK are called **queries** or **goal** (not to be confused with goals as used to describe an agent's desired states).

Certain people think it is funny to answer questions such as "Can you tell me the time?" with "Yes." A knowledge base should be more cooperative. When we ask a question that is existentially quantified, such as

$\text{ASK}(KB, \exists x \ \text{Child}(x, \text{Spot}))$

SUBSTITUTION  
BINDING LIST

we should receive an answer indicating that Boots is a child of Spot. Thus, a query with existential variables is asking "Is there an  $x$  such that ...," and we solve it by providing such an  $x$ . The standard form for an answer of this sort is a **substitution or binding list**, which is a set of variable/term pairs. In this particular case, the answer would be  $\{x/\text{Boots}\}$ . (If there is more than one possible answer, a list of substitutions can be returned. Different implementations of ASK do different things.)

## 7.4 LOGICAL AGENTS FOR THE WUMPUS WORLD

MODEL-BASED  
AGENTS  
GOAL-BASED  
AGENTS

In Chapter 6 we showed the outline of a knowledge-based agent, repeated here in slightly modified form as Figure 7.2. We also hinted at how a knowledge-based agent could be constructed for the wumpus world, but the limitations of propositional logic kept us from completing the agent. With first-order logic, we have all the representational power we need, and we can turn to the more interesting question of how an agent should organize what it knows in order to take the right actions. We will consider three agent architectures: **reflex** agents<sup>8</sup> that merely classify their percepts and act accordingly; **model-based agents**<sup>9</sup> that construct an internal representation of the world and use it to act; and **goal-based agents** that form goals and try to achieve them. (Goal-based agents are usually also model-based agents.)

The first step in constructing an agent for the wumpus world (or for any world) is to define the interface between the environment and the agent. The percept sentence must include both the percept and the time at which it occurred; otherwise the agent will get confused about when it saw what. We will use integers for time steps. A typical percept sentence would be

$\text{Percept}([\text{Stench}, \text{Breeze}, \text{Glitter}, \text{None}, \text{None}], 5)$

<sup>8</sup> Reflex agents are also known as **tropic** agents. The biological term *tropism* means having a tendency to react in a definite manner to stimuli. A heliotropic plant, for example, is one that turns toward the sun.

<sup>9</sup> Note that this usage of "model" is related but not identical to its meaning as a world referred to by a sentence.

```

function KB-AGENT(percept) returns an action
  static: KB, a knowledge base
    t, a counter, initially 0, indicating time

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  action — ASK(KB, MAKE-ACTION-QUERY(t))
  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t  $\leftarrow$  t + 1
  return action

```

**Figure 7.2** A generic knowledge-based agent.

The agent's action must be one of the following:

*Turn(*Right*), Turn(*Left*), Forward, Shoot, Grab, Release, Climb*

To determine which one is best, the function MAKE-ACTION-QUERY creates a query such as

$\exists a \text{ Action}(a, 5)$

with the intention that ASK returns a binding list such as  $\{a/\text{Grab}\}$  and *Grab* is assigned as the value of the variable *action*. The agent program then calls TELL once again to record which action was taken. Because the agent does not perceive its own actions, this is the only way the *KB* could know what the agent has done.

## 7.5 A SIMPLE REFLEX AGENT

The simplest possible kind of agent has rules directly connecting percepts to actions. These rules resemble reflexes or instincts. For example, if the agent sees a glitter, it should do a grab in order to pick up the gold:

$\forall s, b, u, c, t \text{ Percept}([s, b, \text{Glitter}, u, c], t) \Rightarrow \text{Action}(\text{Grab}, t)$

The connection between percept and action can be mediated by rules for perception, which abstract the immediate perceptual input into more useful forms:

$$\begin{aligned} \forall b, g, u, c, t \text{ Percept}([s, b, \text{Stench}, g, u, c], t) &\Rightarrow \text{Stench}(t) \\ \forall s, g, u, c, t \text{ Percept}([s, b, \text{Breeze}, g, u, c], t) &\Rightarrow \text{Breeze}(t) \\ \forall s, b, u, c, t \text{ Percept}([s, b, \text{Glitter}, u, c], t) &\Rightarrow \text{AtGold}(t) \\ \dots \end{aligned}$$

Then a connection can be made from these predicates to action choices:

$\forall t \text{ AtGold}(t) \Rightarrow \text{Action}(\text{Grab}, t)$

This rule is more flexible than the direct connection, because it could be used with other means for deducing *AtGold*—for example, if one stubs one's toe on the gold.

In a more complex environment, the percept might be an entire array of gray-scale or color values (a camera image), and the perceptual rules would have to infer such things as "There's a large truck right behind me flashing its lights." Chapter 24 covers the topic of perception in more detail. Of course, computer vision is a very difficult task, whereas these rules are trivial, but the idea is the same.

## Limitations of simple reflex agents

Simple reflex agents will have a hard time in the wumpus world. The easiest way to see this is to consider the *Climb* action. The optimal agent should either retrieve the gold or determine that it is too dangerous to get the gold, and then return to the start square and climb out of the cave. A pure reflex agent cannot know for sure when to *Climb*, because neither having the gold nor being in the start square is part of the percept; they are things the agent knows by forming a representation of the world.

Reflex agents are also unable to avoid infinite loops. Suppose that such an agent has picked up the gold and is headed for home. It is likely that the agent will enter one of the squares it was in before, and receive the same percept. In that case, it *must*, by definition, do exactly what it did before. Like the dung beetle of Chapter 2, it does not know whether it is carrying the gold, and thus cannot make its actions conditional on that. Randomization provides some relief, but only at the expense of risking many fruitless actions.

## 7.6 REPRESENTING CHANGE IN THE WORLD

In our agent design, all percepts are added into the knowledge base, and in principle the percept history is all there is to know about the world. If we allow rules that refer to past percepts as well as the current percept, then we can, in principle, extend the capabilities of an agent to the point where the agent is acting optimally.

INTERNAL MODEL

Writing such rules, however, is remarkably tedious unless we adopt certain patterns of reasoning that correspond to maintaining an **internal model** of the world, or at least of the relevant aspects thereof. Consider an example from everyday life: finding one's keys. An agent that has taken the time to build a representation of the fact "my keys are in my pocket" need only recall that fact to find the keys. In contrast, an agent that just stored the complete percept sequence would have to do a lot of reasoning to discover where the keys are. It would be theoretically *possible* for the agent to in effect rewind the video tape of its life and replay it in fast-forward, using inference rules to keep track of where the keys are, but it certainly would not be *convenient*.



DIACHRONIC

It can be shown that *any system that makes decisions on the basis of past percepts can be rewritten to use instead a set of sentences about the current world state*, provided that these sentences are updated as each new percept arrives and as each action is done.

Rules describing the way in which the world changes (or does not change) are called **diachronic** rules, from the Greek for "across time." Representing change is one of the most important areas in knowledge representation. The real world, as well as the wumpus world, is

characterized by change rather than static truth. Richard has no brother until John is born; Spot is a kitten for a while and then becomes a full-grown tomcat; the agent moves on.

The easiest way to deal with change is simply to change the knowledge base; to erase the sentence that says the agent is at [1,1], and replace it with one that says it is at [1,2]. This approach can mitigate some of the difficulties we saw with the propositional agent. If we only want the knowledge base to answer questions about the latest situation, this will work fine. But it means that all knowledge about the past is lost, and it prohibits speculation about different possible futures.

A second possibility was hinted at in Chapters 3 and 4: an agent can search through a space of past and possible future states, where each state is represented by a different knowledge base. The agent can explore hypothetical situations—what would happen if I went two squares forward? This approach allows an agent to switch its attention between several knowledge bases, but it does not allow the agent to reason about more than one situation simultaneously.

To answer questions like "was there a stench in both [1,2] and [2,3]?" or "did Richard go to Palestine before he became king?" requires representing different situations and actions in the same knowledge base. In principle, *representing situations and actions is no different from representing more concrete objects such as cats and kings, or concrete relations such as brotherhood*. We need to decide on the appropriate objects and relations, and then write axioms about them. In this chapter, we will use the simplest, and oldest, solution to the problem. In Chapter 8, we will explore more complex approaches.



## SITUATION CALCULUS

**Situation calculus** is the name for a particular way of describing change in first-order logic. It conceives of the world as consisting of a sequence of **situations**, each of which is a "snapshot" of the state of the world. Situations are generated from previous situations by actions, as shown in Figure 7.3.

Every relation or property that can change over time is handled by giving an extra situation argument to the corresponding predicate. We use the convention that the situation argument is always the last, and situation constants are of the form  $S_i$ . Thus, instead of  $\text{At}(\text{Agent}, \text{location})$ , we might have

$$\neg\text{At}(\text{Agent}, [1, 1], S_0) \wedge \text{At}(\text{Agent}, [1, 2], S_1)$$

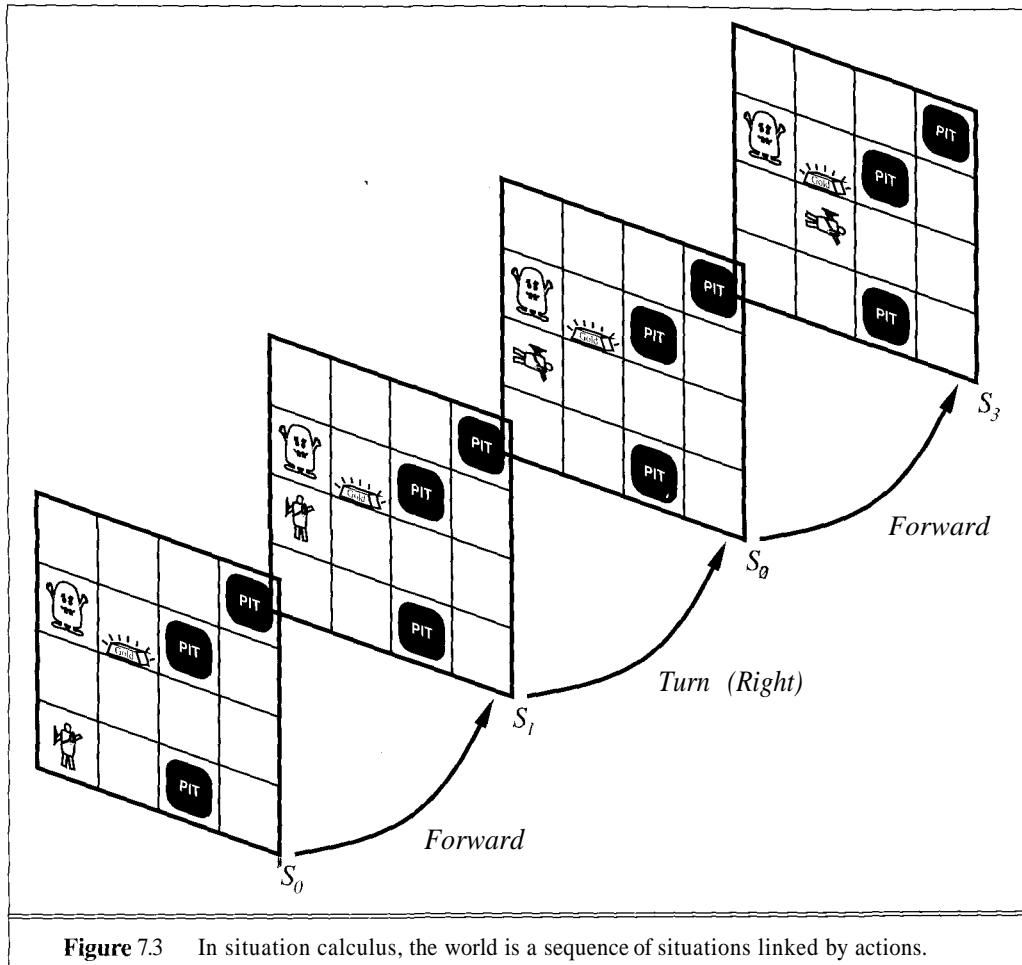
to describe the location of the agent in the first two situations in Figure 7.3. Relations or properties that are not subject to change do not need the extra situation argument. For example, we can just say *Even(8)* instead of *Even(8, S<sub>0</sub>)*. In the wumpus world, where walls cannot move, we can just use *Wall([0, 1])* to say that there is a wall at [0,1].

The next step is to represent how the world changes from one situation to the next. Situation calculus uses the function *Result(action, situation)* to denote the situation that results from performing an action in some initial situation. Hence, in the sequence shown in Figure 7.3, We have the following:

$$\text{Result}(\text{Forward}, S_0) = S_1$$

$$\text{Result}(\text{Turn}(Right), S_0) = S_2$$

$$\text{Result}(\text{Forward}, S_2) = S_3$$



**Figure 7.3** In situation calculus, the world is a sequence of situations linked by actions.

Actions are described by stating their effects. That is, we specify the properties of the situation that results from doing the action. Suppose, for example, that the agent wants to keep track of whether it is holding the gold. The description should state that, in any situation, if gold is present and the agent does a grab, then it will be holding the gold in the resulting situation. We write this in such a way as to apply to any portable object:

*Portable(Gold)*

$\forall s \text{ AtGold}(s) \Rightarrow \text{Present}(Gold, s)$

$\forall x, s \text{ Present}(x, s) \wedge \text{Portable}(x) \Rightarrow \text{Holding}(x, \text{Result}(\text{Grab}, s))$

A similar axiom says that the agent is not holding anything after a *Release* action:

$\forall x, s \neg \text{Holding}(\text{Result}(\text{Release}, s))$

These axioms are called **effect axioms**. Unfortunately, they are not sufficient to keep track of whether the agent is holding the gold. We also need to say that if the agent is holding something

and does *not* release it, it will be holding it in the next state. Similarly, if the agent is not holding something and does not or cannot grab it, it will not be holding it in the next state:

$$\begin{aligned} \forall a, x, s \ Holding(x, s) \wedge (a \neq \text{Release}) &\Rightarrow Holding(x, \text{Result}(a, s)) \\ \forall a, x, s \ \neg Holding(x, s) \wedge (a \neq \text{Grab} \vee \neg(\text{Present}(x, s) \wedge \text{Portable}(x))) \\ &\Rightarrow \neg Holding(x, \text{Result}(a, s)) \end{aligned}$$

FRAME AXIOMS

Axioms like these that describe how the world stays the same (as opposed to how it changes) are called **frame axioms**.<sup>10</sup> Together, effect axioms and frame axioms provide a complete description of how the world evolves in response to the agent's actions.

We can obtain a more elegant representation by combining the effect axioms and frame axioms into a single axiom that describes how to compute the *Holding* predicate for the next time step, given its value for the current time step. The axiom has the following structure:

$$\begin{aligned} \text{true afterwards} &\Leftrightarrow [\text{an action made it true} \\ &\quad \vee \text{true already and no action made it false}] \end{aligned}$$

Notice the use of “ $\Leftrightarrow$ ” here. It says that the predicate will be true after the action if it is made true or if it stays true; *and* that it will be false afterwards in other cases. In the case of *Holding*, the axiom is the following:

$$\begin{aligned} \forall a, x, s \ Holding(x, \text{Result}(a, s)) &\Leftrightarrow [(a = \text{Grab} \wedge \text{Present}(x, s) \wedge \text{Portable}(x)) \\ &\quad \vee (Holding(x, s) \wedge a \neq \text{Release})] \end{aligned}$$

SUCCESSOR-STATE AXIOM

This axiom is called a successor-state **axiom**. One such axiom is needed for each predicate that may change its value over time. A successor-state axiom must list all the ways in which the predicate can become true, and all the ways in which it can become false.

## Keeping track of location

In the wumpus world, location is probably the most important thing to worry about; it cannot be perceived directly, but the agent needs to remember where it has been and what it saw there in order to deduce where pits and wumpuses are and to make sure it does a complete exploration for the gold. We have already seen that the initial location can be described by the sentence *At(Agent, [1, 1], So)*. The agent also needs to know the following:

- What direction it is facing (an angle in degrees, where 0 degrees is out along the X-axis, 90 degrees is up along the Y-axis, and so on):

$$\text{Orientation}(\text{Agent}, \text{So}) = 0$$

- How locations are arranged (a simple “map”). The map consists of values of the function *LocationToward*, which takes a location and a direction and gives the location that is one

<sup>10</sup> The name derives from film animation, where the background image usually remains constant as the characters move around from frame to frame.

## THE FRAME PROBLEM AND ITS RELATIVES

Many AI texts refer to something called the **frame problem**. The problem was noticed soon after situation calculus was applied to reasoning about actions. Originally, it centered on the apparently unavoidable need for a large number of frame axioms that made for a very inelegant and inefficient description of actions. Many researchers considered the problem insoluble within first-order logic. There are at least three entire volumes of collected papers on the problem, and at least one author (Crockett, 1994) cites the problem as one symptom of the inevitable failure of the AI enterprise.

The proliferation of frame axioms is now called the **representational frame problem**, and as we have shown, is easily solved using successor-state axioms. The **inferential frame problem** concerns the way we make inferences about the world. When reasoning about the result of a long sequence of actions in situation calculus, one has to carry each property through all intervening situations one step at a time, *even if the property remains unchanged throughout*. This is true whether one uses frame axioms or successor-state axioms. One would like to be able to make only the changes required by the action descriptions, and have the rest available without further effort. This seems like the natural and efficient thing to do because we do not expect more than a small fraction of the world to change at any given moment. Of course, we cannot expect a *general-purpose* representation language such as first-order logic (and associated general-purpose reasoning systems) to have this bias. It is possible, however, to build *special-purpose* reasoning systems that do work efficiently for reasoning about actions that change only a small fraction of the world. These are called planning systems, and are the subject of Part IV.

Other problems besides the frame problem have surfaced in the study of reasoning about actions. The **qualification problem** arises because it is difficult, in the real world, to define the circumstances under which a given action is *guaranteed* to work. In the real world, grabbing a gold brick may not work if the brick is wet and slippery, or if it is electrified or screwed to the table, or if your back gives out when you bend over, or if the guard shoots you, and so on. If some of these conditions are left out of the successor-state axiom, then the agent is in danger of generating false beliefs.

Finally, the **ramification problem** concerns the proliferation of *implicit* consequences of actions. For example, if a gold brick is covered in dust, then when the agent picks up the brick it also picks up each particle of dust that adheres to the brick. One prefers to avoid describing the motion of the dust (if any) as part of the description of picking up things in general. It is better to state separately that the dust is stuck to the brick, and to *infer* the new location of the dust as necessary. Doing this efficiently requires a special-purpose reasoning system, just as with the frame problem.

step forward in the given direction.

$$\begin{aligned}\forall x, y \text{ } LocationToward([x, y], 0) &= [x + 1, y] \\ \forall x, y \text{ } LocationToward([x, y], 90) &= [x, y + 1] \\ \forall x, y \text{ } LocationToward([x, y], 180) &= [x - 1, y] \\ \forall x, y \text{ } LocationToward([x, y], 270) &= [x, y - 1]\end{aligned}$$

From the map, it is possible to tell which square is directly ahead of the agent, or indeed of any agent  $p$  at any location  $/$ :

$$\begin{aligned}\forall p, l, s \text{ } At(p, l, s) \Rightarrow \\ LocationAhead(p, s) = LocationToward(l, Orientation(p, s))\end{aligned}$$

It is also useful to define adjacency:

$$\forall l_1, l_2 \text{ } Adjacent(l_1, l_2) \Leftrightarrow \exists d \text{ } l_1 = LocationToward(l_2, d)$$

- Whatever is known about the contents of the locations (geographical details on the map). In what follows we assume that the locations surrounding the  $4 \times 4$  cave contain walls, and other locations do not. Sometimes the agent knows this kind of information to start, and sometimes it does not.

$$\forall x, y \text{ } Wall([x, y]) \Leftrightarrow (x = 0 \vee x = 5 \vee y = 0 \vee y = 5)$$

- What the actions do to location. Only going forward changes location, and then only if there is no wall ahead. The successor-state axiom for location is

$$\begin{aligned}\forall a, d, p, s \text{ } At(p, l, Result(a, s)) \Leftrightarrow \\ [ (a = Forward \wedge l = LocationAhead(p, s) \wedge \neg Wall(l)) \\ \vee (At(p, l, s) \wedge a \neq Forward)]\end{aligned}$$

- What the actions do to orientation. Turning is the only action that changes orientation. The successor-state axiom for orientation is

$$\begin{aligned}\forall a, d, p, s \text{ } Orientation(p, Result(a, s)) = d \Leftrightarrow \\ [ (a = Turn(Right) \wedge d = Mod(Orientation(p, s) - 90, 360)) \\ \vee (a = Turn(Left) \wedge d = Mod(Orientation(p, s) + 90, 360)) \\ \vee (Orientation(p, s) = d \wedge \neg(a = Turn(Right) \wedge a = Turn(Left)))]\end{aligned}$$

In addition to keeping track of location and the gold, the agent should also keep track of whether the wumpus is alive or dead. We leave the problem of describing *Shoot* as an exercise.

## 7.7 DEDUCING HIDDEN PROPERTIES OF THE WORLD

Once the agent knows where it is, it can associate qualities with the *places* rather than just the situations, so, for example, one might say that if the agent is at a place and perceives a breeze, then that place is breezy, and if the agent perceives a stench, then that place is smelly:

$$\begin{aligned}\forall /, s \text{ } At(Agent, l, s) \wedge Breeze(s) \Rightarrow Breezy(l) \\ \forall /, s \text{ } At(Agent, /, s) \wedge Stench(s) \Rightarrow Smelly(l)\end{aligned}$$

It is useful to know if *a place* is breezy or smelly because we know that the wumpus and the pits cannot move about. Notice that neither *Breezy* nor *Smelly* needs a situation argument.

Having discovered which places are breezy or smelly (and, very importantly, *not* smelly or *not* breezy), the agent can deduce where the pits are, and where the wumpus is. Furthermore, it can deduce which squares are safe to move to (we use the predicate *OK* to represent this), and can use this information to hunt for the gold.

SYNCHRONIC

The axioms we will write to capture the necessary information for these deductions are called **synchronic** ("same time") rules, because they relate properties of a world state to other properties of the same world state. There are two main kinds of synchronic rules:

CAUSAL RULES

◊ **Causal rules:**

Causal rules reflect the assumed direction of causality in the world: some hidden property of the world causes certain percepts to be generated. For example, we might have rules stating that squares adjacent to wumpuses are smelly and squares adjacent to pits are breezy:

$$\begin{aligned} \forall l_1, l_2, s \ At(Wumpus, l_1, s) \wedge \text{Adjacent}(l_1, l_2) &\Rightarrow \text{Smelly}(l_2) \\ \forall l_1, l_2, s \ At(Pit, l_1, s) \wedge \text{Adjacent}(l_1, l_2) &\Rightarrow \text{Breezy}(l_2) \end{aligned}$$

MODEL-BASED REASONING

DIAGNOSTIC RULES

Systems that reason with causal rules are called **model-based reasoning** systems.

◊ **Diagnostic rules:**

Diagnostic rules infer the presence of hidden properties directly from the percept-derived information. We have already seen two diagnostic rules:

$$\begin{aligned} \forall l, s \ At(\text{Agent}, l, s) \wedge \text{Breeze}(s) &\Rightarrow \text{Breezy}(l) \\ \forall l, s \ At(\text{Agent}, l, s) \wedge \text{Stench}(s) &\Rightarrow \text{Smelly}(l) \end{aligned}$$

For deducing the presence of wumpuses, a diagnostic rule can only draw a weak conclusion, namely, that if a location is smelly, then the wumpus must either be in that location or in an adjacent location:

$$\begin{aligned} \forall l_1, s \ Smelly(l_1) &\Rightarrow \\ (\exists l_2 \ At(Wumpus, l_2, s) \wedge (l_2 = l \vee \text{Adjacent}(l_1, l_2))) \end{aligned}$$

Although diagnostic rules seem to provide the desired information more directly, it is very tricky to ensure that they derive the strongest possible conclusions from the available information. For example, the absence of stench or breeze implies that adjacent squares are OK:

$$\begin{aligned} \forall x, y, g, u, c, s \ Percept([None, None, g, u, c]t) \wedge \\ At(\text{Agent}, x, s) \wedge \text{Adjacent}(x, y) &\Rightarrow \text{OK}(y) \end{aligned}$$

But sometimes a square can be OK even when smells and breezes abound. The model-based rule

$$\forall x, t \ (\neg At(Wumpus, x, t) \wedge \neg Pit(x)) \Leftrightarrow \text{OK}(x)$$

is probably the best way to represent safety.

The distinction between model-based and diagnostic reasoning is important in many areas of AI. Medical diagnosis in particular has been an active area of research, where approaches based on direct associations between symptoms and diseases (a diagnostic approach) have gradually been replaced by approaches using an explicit model of the disease process and how it manifests itself in symptoms. The issues come up again in Chapter 14.



The important thing to remember is that *if the axioms correctly and completely describe the way the world works and the way that percepts are produced, then the inference procedure will correctly infer the strongest possible description of the world state given the available percepts.* A complete specification of the wumpus world axioms is left as an exercise.

## 7.8 PREFERENCES AMONG ACTIONS



So far, the only way we have to decide on actions is to write rules recommending them on the basis of certain conditions in the world. This can get very tedious. For example, generally it is a good idea to explore by moving to OK squares, but not when there is a glitter afoot. Hence our rules for exploring would also have to mention glitter. This seems arbitrary and means the rules are not **modular**: *changes in the agent's beliefs about some aspects of the world would, require changes in rules dealing with other aspects also.* It is more modular to separate facts about actions from facts about goals, which means our agent can be reprogrammed simply by asking it to achieve something different. Goals describe the desirability of outcome states, regardless of how achieved. We discuss goals further in Section 7.9.

A first step is to describe the desirability of actions themselves, and leave the inference engine to choose whichever is the action that has the highest desirability. We will use a simple scale: actions can be *Great*, *Good*, *Medium*, *Risky*, or *Deadly*. The agent should always do a great action if it can find one; otherwise, a good one; otherwise, an OK action; and a risky one if all else fails.

$$\begin{aligned} \forall a, s \ Great(a, s) &\Rightarrow Action(a, s) \\ \forall a, s \ Good(a, s) \wedge (\neg \exists b \ Great(b, s)) &\Rightarrow Action(a, s) \\ \forall a, s \ Medium(a, s) \wedge (\neg \exists b \ Great(b, s) \vee Good(b, s)) &\Rightarrow Action(a, s) \\ \forall a, s \ Risky(a, s) \wedge (\neg \exists b \ Great(b, s) \vee Good(b, s) \vee OK(b, s)) &\Rightarrow Action(a, s) \end{aligned}$$

ACTION-VALUE

A system containing rules of this type is called an **action-value** system. Notice that the rules do not refer to what the actions actually *do*, just how desirable they are.

Up to the point where it finds the gold, the basic strategy for our agent will be as follows:

- Great actions include picking up the gold when found and climbing out of the cave with the gold.
- Good actions include moving to a square that's OK and has not yet been visited.
- Medium actions include moving to a square that's OK and has been visited already.
- Risky actions include moving to a square that's not known to be deadly, but is not known to be OK either.
- Deadly actions are moving into a square that is known to contain a pit or a live wumpus.

Again, we leave the specification of the action values as an exercise.

## 7.9 TOWARD A GOAL-BASED AGENT

The preceding set of action value statements is sufficient to prescribe a reasonably intelligent exploration policy. It can be shown (Exercise 7.15) that the agent using these axioms will always succeed in finding the gold safely whenever there is a safe sequence of actions that does so. This is about as much as we can ask from a logical agent.

Once the gold is found, the policies need to change radically. The aim now is to return to the start square as quickly as possible. What we would like to do is infer that the agent now has the **goal** of being at location [1,1]:

$$\text{V.5 } \text{Holding}(\text{Gold}, s) \Rightarrow \text{GoalLocation}([1, 1], s)$$

The presence of an explicit goal allows the agent to work out a sequence of actions that will achieve the goal. There are at least three ways to find such a sequence:

- 0 **Inference:** It is not hard to write axioms that will allow us to ASK the *KB* for a sequence of actions that is guaranteed to achieve the goal safely. For the  $4 \times 4$  wumpus world, this is feasible, but for larger worlds, the computational demands are too high. In any case, we have the problem of distinguishing good solutions from wasteful solutions (e.g., ones that make a long series of wandering moves before getting on the right track).
- 0 **Search:** We can use a best-first search procedure (see Chapter 4) to find a path to the goal. This requires the agent to translate its knowledge into a set of operators, and accompanying state representation, so that the search algorithm can be applied.
- 0 **Planning:** This involves the use of special-purpose reasoning systems designed to reason about actions. Chapter 11 describes these systems in detail, explaining their advantages over search algorithms.

## 7.10 SUMMARY

This chapter has shown how first-order logic can be used as the representation language for a knowledge-based agent. The important points are as follows:

- First-order logic is a general-purpose representation language that is based on an ontological commitment to the existence of objects and relations in the world.
- **Constant symbols** and **predicate symbols** name objects and relations, respectively. **Complex terms** name objects using **function symbols**. The interpretation specifies what the symbols refer to.
- **An atomic sentence** consists of a predicate applied to one or more terms; it is true just when the relation named by the predicate holds between the objects named by the terms. **Complex sentences** use connectives just like propositional logic, and **quantified sentences** allow the expression of general rules.
- It is possible to define an agent that reasons using first-order logic. Such an agent needs to

1. react to what it perceives;
  2. extract abstract descriptions of the current state from percepts;
  3. maintain an internal model of relevant aspects of the world that are not directly available from percepts;
  4. express and use information about the desirability of actions in various circumstances;
  5. use goals in conjunction with knowledge about actions to construct plans.
- Knowledge about actions and their effects can be represented using the conventions of **situation calculus**. This knowledge enables the agent to keep track of the world and to deduce the effects of plans of action.
  - We have a choice of writing **diagnostic rules** that reason from percepts to propositions about the world or **causal rules** that describe how conditions in the world cause percepts to come about. Causal rules are often more flexible and entail a wider range of consequences, but can be more expensive to use in inference.

---

## BIBLIOGRAPHICAL AND HISTORICAL NOTES

Although even Aristotle's logic deals with generalizations over objects, true first-order logic dates from the introduction of quantifiers in Gottlob Frege's (1879) *Begriffsschrift* ("Concept Writing" or "Conceptual Notation"). Frege's ability to nest quantifiers was a big step forward, but he used an awkward notation. (An example appears on the front cover of this book.) The present notation for first-order logic is substantially due to Giuseppe Peano (1889), but the semantics is virtually identical to Frege's.

A major barrier to the development of first-order logic had been the concentration on one-place predicates to the exclusion of many-place relational predicates. This fixation on one-place predicates had been nearly universal in logical systems from Aristotle up to and including Boole. The first systematic treatment of the logic of relations was given by Augustus De Morgan (1864). De Morgan cited the following example to show the sorts of inferences that Aristotle's logic could not handle: "All horses are animals; therefore, the head of a horse is the head of an animal." This inference is inaccessible to Aristotle because any valid rule that can support this inference must first analyze the sentence using the two-place predicate " $x$  is the head of  $y$ ." The logic of relations was studied in depth by Charles Sanders Peirce (1870), who also developed first-order logic independently of Frege, although slightly later (Peirce, 1883).

Leopold Löwenheim (1915) gave a systematic treatment of model theory in 1915. This paper also treated the equality symbol as an integral part of logic. Löwenheim's results were further extended by Thoralf Skolem (1920). Tarski (1935) gave an explicit definition of truth and model-theoretic satisfaction in first-order logic, using set theory. (An English translation of this German article is given in (Tarski, 1956).)

McCarthy (1958) was primarily responsible for the introduction of first-order logic as a tool for building AI systems, and later (1963) proposed the use of states of the world, or situations, as objects to be reasoned about using first-order logic. The first AI system to make substantial use of general-purpose reasoning about actions in first-order logic was QA3 (Green, 1969b).

Kowalski (1979b) further advanced situation calculus by introducing propositions as objects. The frame problem was pointed out as a major problem for the use of logic in AI by McCarthy and Hayes (1969). The axiomatization we use in the chapter to avoid the representational frame problem was proposed by Charles Elkan (1992) and independently by Ray Reiter (1991). The qualification problem was also pointed out by McCarthy (1977). The inferential frame problem is discussed at length by Shoham and McDermott (1988).

There are a number of good modern introductory texts on first-order logic. Quine (1982) is one of the most readable. Enderton (1972) gives a more mathematically oriented perspective. A highly formal treatment of first-order logic, along with many more advanced topics in logic, is provided by Bell and Machover (1977). Manna and Waldinger (1985) give a readable introduction to logic from a computer science perspective. Gallier (1986) provides an extremely rigorous mathematical exposition of first-order logic, along with a great deal of material on its use in automated reasoning. *Logical Foundations of Artificial Intelligence* (Genesereth and Nilsson, 1987) provides both a solid introduction to logic and the first systematic treatment of logical agents with percepts and actions. Barwise and Etchemendy (1993) give a modern overview of logic that includes an interactive graphical logic game called *Tarski's World*.

---

## EXERCISES

7.1 A logical knowledge base represents the world using a set of sentences with no explicit structure. An **analogical** representation, on the other hand, is one in which the representation has structure that corresponds directly to the structure of the thing represented. Consider a road map of your country as an analogical representation of facts about the country. The two-dimensional structure of the map corresponds to the two-dimensional surface of the area.

- a. Give five examples of *symbols* in the map language.
- b. An *explicit* sentence is one that the creator of the representation actually writes down. An *implicit* sentence is one that results from explicit sentences because of properties of the analogical representation. Give three examples each of *implicit* and *explicit* sentences in the map language.
- c. Give three examples of facts about the physical structure of your country that cannot be represented in the map language.
- d. Give two examples of facts that are much easier to express in the map language than in first-order logic.
- e. Give two other examples of useful analogical representations. What are the advantages and disadvantages of each of these languages?

7.2 Represent the following sentences in first-order logic, using a consistent vocabulary (which you must define):

- a. Not all students take both History and Biology.
- b. Only one student failed History.

- c. Only one student failed both History and Biology.
- d. The best score in History was better than the best score in Biology.
- e. Every person who dislikes all vegetarians is smart.
- f. No person likes a smart vegetarian.
- g. There is a woman who likes all men who are not vegetarians.
- h. There is a barber who shaves all men in town who do not shave themselves.
- i. No person likes a professor unless the professor is smart.
- j. Politicians can fool some of the people all of the time, and they can fool all of the people some of the time, but they can't fool all of the people all of the time.

7.3 We noted that there is often confusion because the  $\Rightarrow$  connective does not correspond directly to the English "if ... then" construction. The following English sentences use "and," "or," and "if" in ways that are quite different from first-order logic. For each sentence, give both a translation into first-order logic that preserves the intended meaning in English, and a straightforward translation (as if the logical connectives had their regular first-order logic meaning). Show an unintuitive consequence of the latter translation, and say whether each translation is valid, satisfiable or invalid.

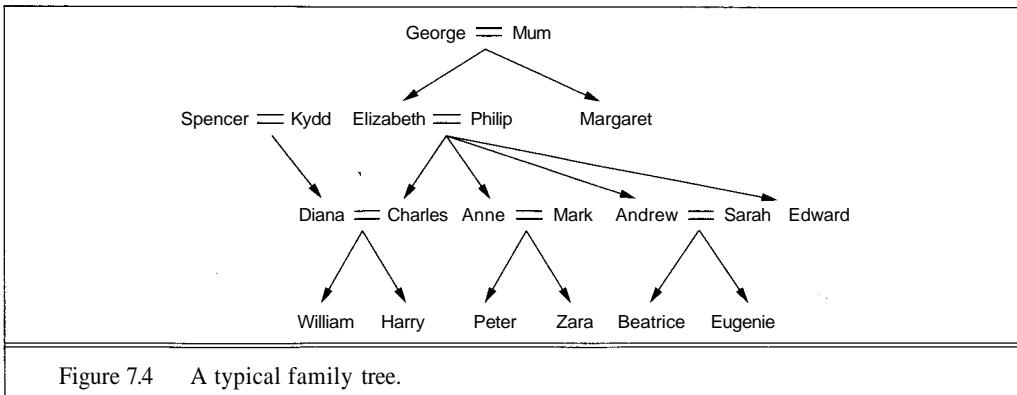
- a. One more outburst like that and you'll be in contempt of court.
- b. *Annie Hall* is on TV tonight, if you're interested.
- c. Either the Red Sox win or I'm out ten dollars.
- d. The special this morning is ham and eggs.
- e. Maybe I'll come to the party and maybe I won't.
- f. Well, I like Sandy and I don't like Sandy.
- g. I don't jump off the Empire State Building implies if I jump off the Empire State Building then I float safely to the ground.
- h. It is not the case that if you attempt this exercise you will get an F. Therefore, you will attempt this exercise.
- i. If you lived here you would be home now. If you were home now, you would not be here. Therefore, if you lived here you would not be here.

7.4 Give a predicate calculus sentence such that every world in which it is true contains exactly one object.

7.5 Represent the sentence "All Germans speak the same languages" in predicate calculus. Use  $Speaks(x, l)$ , meaning that person  $x$  speaks language  $l$ .

7.6 Write axioms describing the predicates *Grandchild*, *GreatGrandparent*, *Brother*, *Sister*, *Daughter*, *Son*, *Aunt*, *Uncle*, *BrotherInLaw*, *SisterInLaw*, and *FirstCousin*. Find out the proper definition of  $m$ th cousin  $n$  times removed, and write it in first-order logic.

Write down the basic facts depicted in the family tree in Figure 7.4. Using the logical reasoning system in the code repository, TELL it all the sentences you have written down, and ASK it who are Elizabeth's grandchildren, Diana's brothers-in-law, and Zara's great-grandparents.



7.7 Explain what is wrong with the following proposed definition of the set membership predicate  $\in$ :

$$\begin{aligned} \forall x, s \quad x \in \{x|s\} \\ \forall x, s \quad x \in s \Rightarrow \forall y \quad x \in \{y|s\} \end{aligned}$$

7.8 Using the set axioms as examples, write axioms for the list domain, including all the constants, functions, and predicates mentioned in the chapter.

7.9 This exercise can be done without the computer, although you may find it useful to use a backward chainer to check your proof for the last part. The idea is to formalize the blocks world domain using the situation calculus. The objects in this domain are blocks, tables, and situations. The predicates are

$$On(x, y, s) \quad ClearTop(x, s) \quad Block(x) \quad Table(x)$$

The only action is  $PutOn(x, y)$ , where  $x$  must be a block whose top is clear of any other blocks, and  $y$  can be either the table or a different block with a clear top. The initial situation  $S_0$  has A on B on C on the table.

- Write an axiom or axioms describing  $PutOn$ .
- Describe the initial state,  $S_0$ , in which there is a stack of three blocks, A on B on C, where C is on the table,  $T$ .
- Give the appropriate query that a theorem prover can solve to generate a plan to build a stack where C is on top of B and B is on top of A. Write down the solution that the theorem prover should return. (Hint: The solution will be a situation described as the result of doing some actions to  $S_0$ .)
- Show formally that the solution fact follows from your description of the situation and the axioms for  $PutOn$ .

7.10 Write sentences to define the effects of the *Shoot* action in the wumpus world. As well as describing its effects on the wumpus, remember that shooting uses the agent's arrow.

7.11 In this exercise, we will consider the problem of planning a route from one city to another. The basic action taken by the robot is  $Go(x, y)$ , which takes it from city  $x$  to city  $y$  provided there

is a direct route.  $\text{DirectRoute}(x,y)$  is true if and only if there is a direct route from  $x$  to  $y$ ; you can assume that all such facts are already in the KB (see the map on page 62). The robot begins in Arad and must reach Bucharest.

- a. Write a suitable logical description of the initial situation of the robot.
- b. Write a suitable logical query whose solutions will provide possible paths to the goal.
- c. Write a sentence describing the *Go* action.
- d. Now suppose that following the direct route between two cities consumes an amount of fuel equal to the distance between the cities. The robot starts with fuel at full capacity. Augment your representation to include these considerations. Your action description should be such that the query you specified earlier will still result in feasible plans.
- e. Describe the initial situation, and write a new rule or rules describing the *Go* action.
- f. Now suppose some of the vertices are also gas stations, at which the robot can fill its tank using the *Fillup* action. Extend your representation to include gas stations and write all the rules needed to completely describe the *Fillup* action.

**7.12** In this exercise, you will extend situation calculus to allow for actions that take place simultaneously. You will use a function called *Simultaneously*, which takes two actions as arguments and denotes the combined action. Consider a grid world containing two agents. Write axioms describing the effects of simultaneous *Forward* actions:

- a. When two agents move at once, unless they are both trying to move to the same location, the result is the same as if one had moved and then the other had moved.
- b. If the agents are trying to move to the same location, they remain in place.



**7.13** Using the wumpus world simulator and the logical reasoning system in the code repository, implement a working agent for the wumpus world. You will need all of the wumpus-related axioms in the chapter, and perhaps more besides. Evaluate the performance of your agent.

**7.14** How hard would it be to build a successful wumpus world agent by writing a program in your favorite programming language? Compare this to the logical reasoning agent.

**7.15** Sketch an argument to the effect that a logical agent using the axioms and action preferences given in the chapter will always succeed in finding the gold safely whenever there is a safe sequence of actions that does so.

**7.16** A reflex agent is one whose action is always a function of its percepts in the current time step. That is, the agent's action cannot be based on anything it learned in the past, and it cannot carry over any internal state information from one time step to the next. In the wumpus world, there are 32 different possible percepts and 6 different actions.

- a. How many different reflex agents can there be in the wumpus world?
- b. How many different  $4 \times 4$  wumpus world are there? How many  $10 \times 10$  worlds?
- c. What do you think the chances are that a reflex agent can be successful in a majority of wumpus worlds? Why?

# 8

# BUILDING A KNOWLEDGE BASE

*In which we develop a methodology for building knowledge bases for particular domains, sketch a representation for the world in general, and go shopping.*

The previous chapter showed that first-order logic is a powerful tool for knowledge representation and reasoning. However, a logic by itself consists of only the syntax, semantics, and proof theory. A logic does not offer any guidance as to what facts should be expressed, nor what vocabulary should be used to express them.

KNOWLEDGE  
ENGINEERING

The process of building a knowledge base is called **knowledge engineering**. A knowledge engineer is someone who investigates a particular domain, determines what concepts are important in that domain, and creates a formal representation of the objects and relations in the domain. Often, the knowledge engineer is trained in representation but is not an expert in the domain at hand, be it circuit design, space station mission scheduling, or whatever. The knowledge engineer will usually interview the real experts to become educated about the domain and to elicit the required knowledge, in a process called **knowledge acquisition**. This occurs prior to, or interleaved with, the process of creating formal representations. In this chapter, we will use domains that should already be fairly familiar, so that we can concentrate on the representational issues involved.

KNOWLEDGE  
ACQUISITION

One does not become a proficient knowledge engineer just by studying the syntax and semantics of a representation language. It takes practice and exposure to lots of examples before one can develop a good style in any language, be it a language for programming, reasoning, or communicating. Sections 8.1 and 8.2 discuss the principles and pitfalls of knowledge engineering. We then show how to represent knowledge in the fairly narrow domain of electronic circuits in Section 8.3. A number of narrow domains can be tackled by similar techniques, but domains such as shopping in a supermarket seem to require much more general representations. In Section 8.4, we discuss ways to represent time, change, objects, substances, events, actions, money, measures, and so on. These are important because they show up in one form or another in every domain. Representing these very general concepts is sometimes called **ontological engineering**. Section 8.5 describes in detail a simplified shopping environment, and uses the general ontology to develop representations capable of sustaining rational action in the domain.

ONTOLOGICAL  
ENGINEERING

## 8.1 PROPERTIES OF GOOD AND BAD KNOWLEDGE BASES

In Chapter 6, we said that a good knowledge representation language should be expressive, concise, unambiguous, context-insensitive, and effective. A knowledge base should, in addition, be *clear* and *correct*. The relations that matter should be defined, and the irrelevant details should be suppressed. Of course, there will be trade-offs between properties: we can make simplifications that sacrifice some correctness to gain clarity and brevity.

The question of efficiency is a little more difficult to deal with. Ideally, the separation between the knowledge base and the inference procedure should be maintained. This allows the creator of the knowledge base to worry only about the *content* of the knowledge, and not about how it will be used by the inference procedure. The same answers should be obtainable by the inference procedure, no matter how the knowledge is encoded. As far as possible, ensuring efficient inference is the task of the designer of the inference procedure and should not distort the representation.

In practice, some considerations of efficiency are unavoidable. Automatic methods exist that can eliminate the most obvious sources of inefficiency in a given encoding, in much the same way that optimizing compilers can speed up the execution of a program, but at present these methods are too weak to overcome the determined efforts of a profligate knowledge engineer who has no concern for efficiency. Even in the best case, then, the knowledge engineer should have some understanding of how inference is done, so that the representation can be designed for maximum efficiency. In the worst case, the representation language is used primarily as a way of "programming" the inference procedure.



As we will see throughout this chapter, *you cannot do, or understand, knowledge engineering by just talking about it*. To explain the general principles of good design, we need to have an example. We will start by doing the example incorrectly, and then fix it.

Every knowledge base has two potential consumers: human readers and inference procedures. A common mistake is to choose predicate names that are meaningful to the human reader, and then be lulled into assuming that the name is somehow meaningful to the inference procedure as well. The sentence *BearOfVerySmallBrain(Pooh)* might be appropriate in certain domains,<sup>1</sup> but from this sentence alone, the inference procedure will not be able to infer either that Pooh is a bear or that he has a very small brain; that he has a brain at all; that very small brains are smaller than small brains; or that this fact implies something about Pooh's behavior. The hard part is for the human reader to resist the temptation to make the inferences that seem to be implied by long predicate names. A knowledge engineer will often notice this kind of mistake when the inference procedure fails to conclude, for example, *Silly(Pooh)*. It is compounding the mistake to write

$$\forall b \text{ } BearOfVerySmallBrain(b) \Rightarrow \text{ } Silly(b)$$

because this expresses the relevant knowledge at too *specific* a level. Although such *VeryLongNames* can be made to work for simple examples covering a small, sparse portion of a larger domain, they do not scale up well. Adding *AnotherVeryLongName* takes just as much work as

<sup>1</sup> Winnie the Pooh is a toy bear belonging to Christopher Robin in the well-known series of children's' books (Milne, 1926). The style of our introductory sentence in each chapter is borrowed from these works.

## KNOWLEDGE ENGINEERING vs. PROGRAMMING

A useful analogy can be made between knowledge engineering and programming. Both activities can be seen as consisting of four steps:

<i>Knowledge Engineering</i>	<i>Programming</i>
(1) Choosing a logic	Choosing a programming language
(2) Building a knowledge base	Writing a program
(3) Implementing the proof theory	Choosing or writing a compiler
(4) Inferring new facts	Running a program

In both activities, one writes down a description of a problem or state of affairs, and then uses the definition of the language to derive new consequences. In the case of a program, the output is derived from the input and the program; in the case of a knowledge base, answers are derived from descriptions of problems and the knowledge base.

Given these similarities, what is the point of doing "knowledge engineering" at all? Why not just admit that the final result will be a program, and set about to write that program from the start, using a traditional programming language?

The main advantage of knowledge engineering is that it requires less commitment, and thus less work. A knowledge engineer only has to decide what objects and relations are worth representing, and which relations hold among which objects. A programmer has to do all that, and in addition must decide how to compute the relations between objects, given some initial input. The knowledge engineer specifies *what* is true, and the inference procedure figures out *how* to turn the facts into a solution to the problem. Furthermore, because a fact is true regardless of what task one is trying to solve, knowledge bases can, in principle, be reused for a variety of different tasks without modification. Finally, debugging a knowledge base is made easier by the fact that any given sentence is true or false *by itself*, whereas the correctness of a program statement depends very strongly on its context.

The advantages of this **declarative approach** to system building have not been lost on other subfields of computer science. Database systems derive most of their usefulness from the fact that they provide a method to store and retrieve information in a way that is independent of the particular application. Database systems have also started to add the capability to do logical inference, thereby moving more of the functionality from the application program into the database system (Stonebraker, 1992). The field of **agent-based software engineering** (Genesereth and Ketchpel, 1994) attempts to make all sorts of systems and resources interoperable by providing a declarative interface based on first-order logic.

adding the first one. For example, to derive *Silly(Piglet)* from *ShyBabyPigOfSmallBrain(Piglet)*, we would have to write

$$\forall b \text{ ShyBabyPigOfSmallBrain}(b) \Rightarrow \text{Silly}(b)$$

This is a sign that something is wrong. The first fact about silliness is of no help in a similar situation. In a properly designed knowledge base, facts that were entered for one situation should end up being used in new situations as well. As you go along, you should need fewer new facts, and fewer new predicates. This will only happen if one writes rules at the most *general* level at which the knowledge is applicable. In a good knowledge base, *BearOfVerySmallBrain(Pooh)* would be replaced by something like the following:

1. Pooh is a bear; bears are animals; animals are physical things.

$$\begin{aligned} &\text{Bear(Pooh)} \\ &\forall b \text{ Bear}(b) \Rightarrow \text{Animal}(b) \\ &\forall a \text{ Animal}(a) \Rightarrow \text{PhysicalThing}(a) \end{aligned}$$

These sentences help to tie knowledge about Pooh into a broader context. They also enable knowledge to be expressed at an appropriate level of generality, depending on whether the information is applicable to bears, animals, or all physical objects.

2. Pooh has a very small brain.

$$\text{RelativeSize(BrainOf(Pooh), BrainOf(TypicalBear))} = \text{VerySmall}$$

This provides a precise sense of "very small," which would otherwise be highly ambiguous. Is Pooh's brain very small compared to a molecule or a moon?

3. All animals (and only animals) have a brain, which is a part of the animal.

$$\begin{aligned} \forall a \text{ Animal}(a) &\Leftrightarrow \text{Brain(BrainOf}(a)\text{)} \\ \forall a \text{ PartOf(BrainOf}(a)\text{, }a\text{)} \end{aligned}$$

This allows us to connect Pooh's brain to Pooh himself, and introduces some useful, general vocabulary.

4. If something is part of a physical thing, then it is also a physical thing:

$$\forall x, y \text{ PartOf}(x, y) \text{ A PhysicalThing}(y) \Rightarrow \text{PhysicalThing}(x)$$

This is a very general and important fact that is seldom seen in physics textbooks!

5. Animals with brains that are small (or below) relative to the normal brain size for their species are silly.<sup>2</sup>

$$\begin{aligned} \forall a \text{ RelativeSize(BrainOf}(a)\text{, BrainOf(TypicalMember(SpeciesOf}(a)\text{)} \leq \text{Small} \\ \Rightarrow \text{Silly}(a) \\ \forall b \text{ Bear}(b) \Leftrightarrow \text{SpeciesOf}(b) = \text{Ursidae} \\ \text{TypicalBear} = \text{TypicalMember(Ursidae)} \end{aligned}$$

<sup>2</sup> It is important to remember that the goal for this knowledge base was to be consistent and useful within a world of talking stuffed animals, not to be a model of the real world. Although biologists are in agreement that brain size is not a good predictor for silliness, the rules given here are the right ones for this world.

6. Every physical thing has a size. Sizes are arranged on a scale from Tiny to Huge. A relative size is a ratio of two sizes.

$$\forall x \text{PhysicalThing}(x) \Rightarrow \exists s \text{Size}(x) = s$$

$$\text{Tiny} < \text{Small} < \text{Medium} < \text{Large} < \text{Huge}$$

$$\forall a, b \text{RelativeSize}(a, b) = \text{Size}(a)/\text{Size}(b)$$

7. The function *Very* maps a point on a scale to a more extreme value. *Medium* is the neutral value for a scale.

$$\text{Medium} = 1$$

$$\forall x x > \text{Medium} \Rightarrow \text{Very}(x) > x$$

$$\forall x x < \text{Medium} \Rightarrow \text{Very}(x) < x$$

This is more work than writing a single rule for *BearOfVerySmallBrain*, but it achieves far more. It has articulated some of the basic properties of physical things and animals, properties that will be used many times, but need be stated only once. It has begun to sketch out a hierarchy of objects (bears, animals, physical things). It has also made a representational choice for values on scales, which will come in handy later in the chapter.

Every time one writes down a sentence, one should ask oneself the following:

- Why is this true? Could I write down the facts that make it true instead?
- How generally is it applicable? Can I state it for a more general class of objects?
- Do I need a new predicate to denote this class of objects? How does the class relate to other classes? Is it part of a larger class? Does that class have other subclasses? What are other properties of the objects in this class?

We cannot provide a foolproof recipe for successful knowledge engineering, but we hope this example has provided some pointers.

## 8.2 KNOWLEDGE ENGINEERING

The knowledge engineer must understand enough about the domain in question to represent the important objects and relationships. He or she must also understand enough about the representation language to correctly encode these facts. Moreover, the knowledge engineer must also understand enough about the implementation of the inference procedure to assure that queries can be answered in a reasonable amount of time. To help focus the development of a knowledge base and to integrate the engineer's thinking at the three levels, the following five-step methodology can be used:

- *Decide what to talk about.* Understand the domain well enough to know which objects and facts need to be talked about, and which can be ignored. For the early examples in this chapter, this step is easy. In some cases, however, it can be the hardest step. Many knowledge engineering projects have failed because the knowledge engineers started to formalize the domain before understanding it. Donald Michie (1982) gives the example of a cheese factory that had a single cheese tester who decided if the Camembert was

ripe by sticking his finger into a sample and deciding if it "felt right." When the cheese tester approached retirement age, the factory invested much time and money developing a complex system with steel probes that would test for just the right surface tension, but the system was next to useless. Eventually, it turned out that feel had nothing to do with it; pushing the finger in just served to break the crust and let the aroma out, and that was what the cheese tester was subconsciously relying on.

- *Decide on a vocabulary of predicates, functions, and constants.* That is, translate the important domain-level concepts into logic-level names. This involves many choices, some arbitrary and some important. Should *Size* be a function or a predicate? Would *Bigness* be a better name than *Size*? Should *Small* be a constant or a predicate? Is *Small* a measure of relative size or absolute size? Once the choices have been made, the result is a vocabulary that is known as the **ontology** of the domain. The word ontology means a particular theory of the nature of being or existence. Together, this step and the previous step are known as ontological engineering. They determine what kinds of things exist, but do not determine their specific properties and interrelationships.
- *Encode general knowledge about the domain.* The ontology is an informal list of the concepts in a domain. By writing logical sentences or **axioms** about the terms in the ontology, we accomplish two goals: first, we make the terms more precise so that humans will agree on their interpretation. Without the axioms, we would not know, for example, whether *Bear* refers to real bears, stuffed bears, or both. Second, we make it possible to run inference procedures to automatically derive consequences from the knowledge base. Once the axioms are in place, we can say that a knowledge base has been produced.

Of course, nobody expects a knowledge base to be correct and complete on the first try. There will be a considerable debugging process. The main difference between debugging a knowledge base and debugging a program is that it is easier to look at a single logic sentence and tell if it is correct. For example, a typical error in a knowledge base looks like this:

$$\forall x \text{Animal}(x) \Rightarrow \exists b \text{BrainOf}(x)=b$$

This says that there is some object that is the value of the *BrainOf* function applied to an animal. Of course, a function has a value for *any* input, although the value may be an undefined object for inputs that are outside the expected range. So this sentence makes a vacuous claim. We can "correct" it by adding the conjunct *Brain(b)*. Then again, if we are potentially dealing with single-celled animals, we could correct it again, replacing *Animal* by, say, *Vertebrate*.

In contrast, a typical error in a program looks like this:

```
offset := position + 1
```

It is impossible to tell if this statement is correct without looking at the rest of the program to see if, for example, *of set* is used elsewhere in the program to refer to the position, or to one beyond the position; or whether the statement was accidentally included twice in different places.

Programming language statements therefore tend to depend on a lot of context, whereas logic sentences tend to be more self-contained. In that respect, a sentence in a knowledge base is more like an entire procedure in a program, not like an individual statement.

- *Encode a description of the specific problem instance.* If the ontology is well thought out, this step will be easy. It will mostly involve writing simple atomic sentences about instances of concepts that are already part of the ontology.
- *Pose queries to the inference procedure and get answers.* This is where the reward is: we can let the inference procedure operate on the axioms and problem-specific facts to derive the facts we are interested in knowing.

To understand this five-step process better, we turn to some examples of its use. We first consider the domain of Boolean electronic circuits.

## 8.3 THE ELECTRONIC CIRCUITS DOMAIN

Within the domain of discrete digital electronic circuits, we would like to analyze the circuit shown in Figure 8.1. The circuit purports to be a one-bit full adder, where the first two inputs are the two bits to be added, and the third input is a carry bit. The first output is the sum, and the second output is a carry bit for the next adder. The goal is to provide an analysis that determines if the circuit is in fact an adder, and that can answer questions about the value of current flow at various points in the circuit.<sup>3</sup> We follow the five-step process for knowledge engineering.

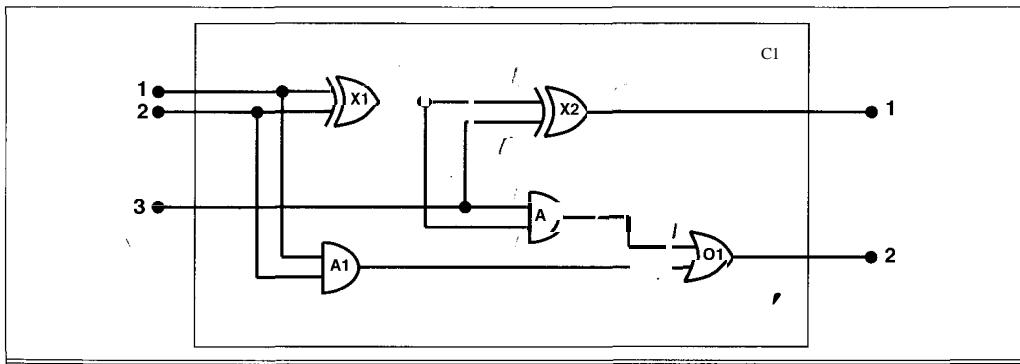


Figure 8.1 A digital circuit C1, with three inputs and two outputs, containing two XOR gates, two AND gates and one OR gate. The inputs are bit values to be added, and the outputs are the sum bit and the carry bit.

### Decide what to talk about

Digital circuits are composed of wires and gates. Signals flow along wires to the input terminals of gates, and each gate produces a signal on the output terminal that flows along another wire.

<sup>3</sup> If you are intimidated by the electronics, try to get a feel for how the knowledge base was constructed without worrying about the details.

There are four types of gates: AND, OR, and XOR gates have exactly two input terminals, and NOT gates have one. All gates have exactly one output terminal. Circuits, which are composed of gates, also have input and output terminals.

Our main purpose is to analyze the design of circuits to see if they match their specification. Thus, we need to talk about *circuits*, their *terminals*, and the *signals* at the terminals. To determine what these signals will be, we need to know about individual *gates*, and *gate types*: AND, OR, XOR, and NOT.

Not everything that is in the domain needs to show up in the ontology. We do not need to talk about the wires themselves, or the paths the wires take, or the junctions where two wires come together. All that matters is the connectivity of terminals—we can say that one output terminal is connected to another input terminal without having to mention the wire that actually connects them. There are many other factors of the domain that are irrelevant to our analysis, such as the size, shape, color, or cost of the various components.

A special-purpose ontology such as this depends not only on the domain, but also on the task to be solved. If our purpose were something other than verifying designs at the gate level, the ontology would be different. For example, if we were interested in debugging faulty circuits, then it would probably be a good idea to include the wires in the ontology, because a faulty wire can corrupt the signal flowing along it. For resolving timing faults, we would need to include gate delays. If we were interested in designing a product that would be profitable, then the cost of the circuit and its speed relative to other products on the market would be important.

## Decide on a vocabulary

We now know that we want to talk about circuits, terminals, signals, gates, and gate types. The next step is to choose functions, predicates, and constants to name them. We will start from individual gates and move up to circuits.

First, we need to be able to distinguish a gate from other gates. This is handled by naming gates with constants:  $X_1, X_2$ , and so on. Next, we need to know the type of a gate.<sup>4</sup> A function is appropriate for this:  $Type(X) = XOR$ . This introduces the constant *XOR* for a particular gate type; the other constants will be called *OR*, *AND*, and *NOT*. The *Type* function is not the only way to encode the ontological distinction. We could have used a type predicate:  $Type(X_1, XOR)$  or several predicates, such as  $XOR(X_1)$ . Either of these choices would work fine, but by choosing the function *Type*, we avoid the need for an axiom that says that each individual gate can have only one type. The semantics of functions already guarantees this.

Next we consider terminals. A gate or circuit can have one or more input terminals and one or more output terminals. We could simply name each one with a constant, just as we named gates. Thus, gate  $X_1$  could have terminals named  $X_1In_1, X_1In_2$ , and  $X_1Out_1$ . Names as long and structured as these, however, are as bad as *BearOfVerySmallBrain*. They should be replaced with a notation that makes it clear that  $X_1Out_1$  is a terminal for gate  $X_1$ , and that it is the first output terminal. A function is appropriate for this; the function  $Out(1, X_1)$  denotes the first (and only) output terminal for gate  $X_1$ . A similar function *In* is used for input terminals.

<sup>4</sup> Note that we have used names beginning with appropriate letters— $A_1, X_1$ , and so on—purely to make the example easier to read. The knowledge base must still contain type information for the gates.

The connectivity between gates can be represented by the predicate *Connected*, which takes two terminals as arguments, as in *Connected(Out(1, X<sub>1</sub>), In(1, X<sub>2</sub>))*.

Finally, we need to know if a signal is on or off. One possibility is to use a unary predicate, *On*, which is true when the signal at a terminal is on. This makes it a little difficult, however, to pose questions such as "What are all the possible values of the signals at the following terminals ... ?" We will therefore introduce as objects two "signal values" *On* or *Off* and a function *Signal* which takes a terminal as argument and denotes a signal value.

## Encode general rules

One sign that we have a good ontology is that there are very few general rules that need to be specified. A sign that we have a good vocabulary is that each rule can be stated clearly and concisely. With our example, we need only seven simple rules:

1. If two terminals are connected, then they have the same signal:

$$\forall t_1, t_2 \text{ } \textit{Connected}(t_1, t_2) \Rightarrow \textit{Signal}(t_1) = \textit{Signal}(t_2)$$

2. The signal at every terminal is either on or off (but not both):

$$\begin{aligned} \forall t \text{ } \textit{Signal}(t) = \textit{On} \vee \textit{Signal}(t) = \textit{Off} \\ \textit{On} \neq \textit{Off} \end{aligned}$$

3. Connected is a commutative predicate:

$$\forall t_1, t_2 \text{ } \textit{Connected}(t_1, t_2) \Leftrightarrow \textit{Connected}(t_2, t_1)$$

4. An OR gate's output is on if and only if any of its inputs are on:

$$\begin{aligned} \forall g \text{ } \textit{Type}(g) = \textit{OR} \Rightarrow \\ \textit{Signal}(\textit{Out}(1, g)) = \textit{On} \Leftrightarrow \exists n \text{ } \textit{Signal}(\textit{In}(n, g)) = \textit{On} \end{aligned}$$

5. An AND gate's output is off if and only if any of its inputs are off:

$$\begin{aligned} \forall g \text{ } \textit{Type}(g) = \textit{AND} \Rightarrow \\ \textit{Signal}(\textit{Out}(1, g)) = \textit{Off} \Leftrightarrow \exists n \text{ } \textit{Signal}(\textit{In}(n, g)) = \textit{Off} \end{aligned}$$

6. An XOR gate's output is on if and only if its inputs are different:

$$\begin{aligned} \forall g \text{ } \textit{Type}(g) = \textit{XOR} \Rightarrow \\ \textit{Signal}(\textit{Out}(1, g)) = \textit{On} \Leftrightarrow \textit{Signal}(\textit{In}(1, g)) \neq \textit{Signal}(\textit{In}(2, g)) \end{aligned}$$

7. A NOT gate's output is different from its input:

$$\forall g \text{ } (\textit{Type}(g) = \textit{NOT}) \Rightarrow \textit{Signal}(\textit{Out}(1, g)) \neq \textit{Signal}(\textit{In}(1, g))$$

## Encode the specific instance

The circuit shown in Figure 8.1 is encoded as circuit *C<sub>1</sub>* with the following description. First, we categorize the gates:

$$\begin{array}{ll} \textit{Type}(X_1) = \textit{XOR} & \textit{Type}(X_2) = \textit{XOR} \\ \textit{Type}(A_1) = \textit{AND} & \textit{Type}(A_2) = \textit{AND} \\ \textit{Type}(O_1) = \textit{OR} & \end{array}$$

Then, the connections between them:

$Connected(Out(1, X_1), In(1, X_2))$	$Connected(In(1, C\backslash), In(1, X_1))$
$Connected(Out(1, X_1), In(2, A_2))$	$Connected(In(1, C_1), In(1, A_1))$
$Connected(Out(1, A_2), In(1, O_1))$	$Connected(In(2, C_1), In(2, X_1))$
$Connected(Out(1, A_1), In(2, O_1))$	$Connected(In(2, CO), In(2, A_1))$
$Connected(Out(1, X_2), Out(1, CO))$	$Connected(In(3, C_1), In(2, X_2))$
$Connected(Out(1, OO), Out(2, CO))$	$Connected(In(3, C\backslash), In(1, A_2))$

### Pose queries to the inference procedure

What combinations of inputs would cause the first output of  $C_1$  (the sum bit) to be off and the second output of  $C\backslash$  (the carry bit) to be on?

$$\exists i_1, i_2, i_3 \ Signal(In(1, C\backslash)) = i_1 \wedge Signal(In(2, C_1)) = i_2 \wedge signal(In(3, CO) = i_3) \\ A Signal(Out(1, CO)) = Off \wedge A Signal(Out(2, CO)) = On$$

The answer is

$$(i_1 = On \wedge i_2 = On \wedge i_3 = Off) \vee \\ (i_1 = On \wedge i_2 = Off \wedge i_3 = On) \vee \\ (i_1 = Off \wedge i_2 = On \wedge i_3 = On)$$

What are the possible sets of values of all the terminals for the adder circuit?

$$\exists i_1, i_2, i_3, o_1, o_2 \ Signal(In(1, C_1)) = i_1 \wedge Signal(In(2, CO)) = i_2 \\ A Signal(In(3, CO)) = i_3 \wedge Signal(Out(1, CO)) = 01 \wedge Signal(Out(2, CO)) = 02$$

This final query will return a complete input/output table for the device, which can be used to check that it does in fact add its inputs correctly. This is a simple example of **circuit verification**. We can also use the definition of the circuit to build larger digital systems, for which the same kind of verification procedure can be carried out (see Exercises 8.1 and 8.3). Many domains are amenable to the same kind of structured knowledge-base development, in which more complex concepts are defined on top of simpler concepts.

## 8.4 GENERAL ONTOLOGY

This section is about a general ontology that incorporates decisions about how to represent a broad selection of objects and relations. It is encoded within first-order logic, but makes many ontological commitments that first-order logic does not make. A general ontology is rather more demanding to construct, but once done has many advantages over special-purpose ontologies.

Consider again the ontology for circuits in the previous section. It makes a large number of simplifying assumptions. For example, time is omitted completely. Signals are fixed, and there is no propagation of signals. The structure of the circuit remains constant. Now we could take a step toward generality by considering signals at particular times, and including the wire lengths and propagation delays in wires and devices. This would allow us to simulate the timing properties

of the circuit, and indeed such simulations are often carried out by circuit designers. We could also introduce more interesting classes of gates, for example by describing the technology (TTL, MOS, CMOS, and so on) as well as the input/output specification. If we wanted to discuss reliability or diagnosis, we would include the possibility that the structure of the circuit, or the properties of the gates, might change spontaneously. To account for stray capacitances, we would need to move from a purely topological representation of connectivity to a more realistic description of geometric properties.

If we look at the wumpus world, similar considerations apply. Although we do include time, it has a very simple structure. Nothing happens except when the agent acts, and all changes can be considered instantaneous. A more general ontology, better suited for the real world, would allow for simultaneous changes extended over time. We also used the constant symbol *Pit* to say that there was a pit in a particular square, because all pits were identical. We could have allowed for different kinds of pits by having several individuals belonging to the class of pits but having different properties. Similarly, we might want to allow for several different kinds of animals, not just wumpuses. It might not be possible to pin down the exact species from the available percepts, so we would need to build up a wumpus-world biological taxonomy to help the agent predict behavior from scanty clues.

For any area of a special-purpose ontology, it is possible to make changes like these to move toward greater generality. An obvious question then arises: do all these ontologies converge on a general-purpose ontology? The answer is, "Possibly." In this section, we will present one version, representing a synthesis of ideas from many knowledge representation efforts in AI and philosophy. There are two major characteristics of general-purpose ontologies that distinguish them from collections of special-purpose ontologies:

- A general-purpose ontology should be applicable in more or less any special-purpose domain (with the addition of domain-specific axioms). This means that as far as possible, no representational issue can be finessed or brushed under the carpet. For example, a general ontology cannot use situation calculus, which finesse the issues of duration and simultaneity, because domains such as circuit timing analysis require those issues to be handled properly.
- In any sufficiently demanding domain, different areas of knowledge must be *unified* because reasoning and problem solving may involve several areas simultaneously. A robot circuit-repair system, for instance, needs to reason about circuits in terms of electrical connectivity and physical layout, and about time both for circuit timing analysis and estimating labor costs. The sentences describing time therefore must be capable of being combined with those describing spatial layout, and must work equally well for nanoseconds and minutes, and for angstroms and meters.

After we present the general ontology, we will apply it to write sentences describing the domain of grocery shopping. A brief reverie on the subject of shopping brings to mind a vast array of topics in need of representation: locations, movement, physical objects, shapes, sizes, grasping, releasing, colors, categories of objects, anchovies, amounts of stuff, nutrition, cooking, nonstick frying pans, taste, time, money, direct debit cards, arithmetic, economics, and so on. The domain is more than adequate to exercise our ontology, and leaves plenty of scope for the reader to do some creative knowledge representation of his or her own.

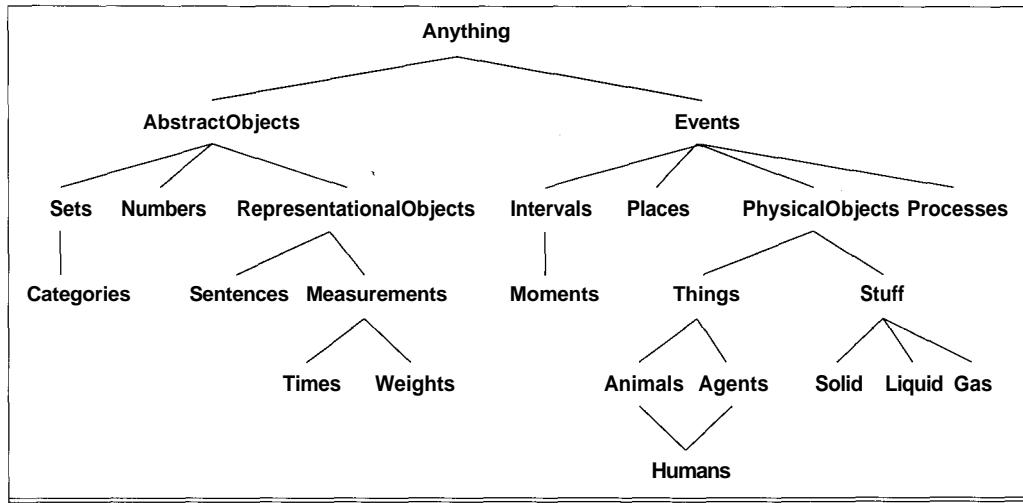
Our discussion of the general-purpose ontology is organized under the following headings, each of which is really worth a chapter by itself:

- 0 **Categories:** Rather than being an entirely random collection of objects, the world exhibits a good deal of regularity. For example, there are many cases in which several objects have a number of properties in common. It is usual to define **categories**<sup>5</sup> that include as members all objects having certain properties. For example, we might wish to have categories for tomatoes or ten-dollar bills, for peaches or pound notes, for fruits or monetary instruments. We describe how categories can be objects in their own right, and how they are linked into a unified **taxonomic hierarchy**.
- ◊ **Measures:** Many useful properties such as mass, age, and price relate objects to quantities of particular types, which we call measures. We explain how measures are represented in logic, and how they relate to units of measure.
- ◊ **Composite objects:** It is very common for objects to belong to categories by virtue of their constituent structure. For example, cars have wheels, an engine, and so on, arranged in particular ways; typical baseball games have nine innings in which each team alternates pitching and batting. We show how such structures can be represented.
- 0 **Time, Space, and Change:** In order to allow for actions and events that have different durations and can occur simultaneously, we enlarge our ontology of time. The basic picture is of a universe that is continuous in both temporal and spatial dimensions. Times, places, and objects will be parts of this universe.
- ◊ **Events and Processes:** Events such as the purchase of a tomato will also become individuals in our ontology. Like tomatoes, they are usually grouped into categories. Individual events take place at particular times and places. Processes are events that are continuous and homogeneous in nature, such as raining or cooking tomatoes.
- 0 **Physical Objects:** We are already familiar with the representation of ordinary objects such as AND-gates and wumpuses. As things that are extended in both time and space, physical objects have much in common with events.
- 0 **Substances:** Whereas objects such as tomatoes are relatively easy to pin down, substances such as tomato juice are a little slippery. Natural language usage seems to provide conflicting intuitions. Is there an object called *TomatoJuice*? Is it a category, or a real physical object? What is the connection between it and the liter of tomato juice I bought yesterday? Between it and constituent substances such as *Water*? We will see that these questions can be resolved by careful use of the ontology of space, time, and physical objects.
- 0 **Mental Objects and Beliefs:** An agent will often need to reason about its own beliefs, for example, when trying to decide why it thought that anchovies were on sale. It will also need to reason about the beliefs of others, for example, in order to decide whom to ask about the right aisle to find the tomatoes. In our ontology, sentences are explicitly represented, and are believed by agents.

In this section, we will try to cover the highest levels of the ontology. The top levels of the hierarchy of categories are shown in Figure 8.2. While the scope of the effort might seem

---

<sup>5</sup> Categories are also called **classes, collections, kinds, types, and concepts** by other authors. They have little or nothing to do with the mathematical topic of **category theory**.



**Figure 8.2** The top-level ontology of the world, showing the topics to be covered later in the chapter. Arcs indicate subset relations.

daunting at first, representing knowledge of the commonsense world can be highly illuminating. One is constantly amazed by how much one knows but never took the time to think about. With a good ontology, writing it down becomes more of a pleasure than a problem. Connections between seemingly disparate areas become obvious, and one is awed by the scope of human thought.

The following subsections fill in the details of each topic. We should first state one important caveat. We have chosen to discuss the content and organization of knowledge using first-order logic. Certain aspects of the real world are hard to capture in this language. The principal feature we must omit is the fact that almost all generalizations have exceptions, or have the status of a default in the absence of more exact information, or only hold to a degree. For example, although "tomatoes are red" is a useful rule, some tomatoes are green, yellow, or orange. Similar exceptions can be found to almost all the general statements in this section. The ability to handle exceptions and uncertain rules is extremely important, but is orthogonal to the task of understanding the general ontology. For this reason, we will delay the discussion of exceptions and defaults until Chapter 10, and the more general topic of uncertain information until Chapter 14.

## Representing Categories



The organization of objects into **categories** is a vital part of knowledge representation. Although interaction with the world takes place at the level of individual objects, *much of reasoning takes place at the level of categories*. For example, a shopper might have the goal of buying a *cantaloupe*, rather than a particular cantaloupe such as *Cantaloupe*<sup>37</sup>.<sup>6</sup> Categories also serve to

<sup>6</sup> We often use subscripts as a reminder that a constant refers to an individual rather than a collection.

make predictions about objects once they are classified. One infers the presence of certain objects from perceptual input, infers category membership from the perceived properties of the objects, and then uses category information to make predictions about the objects. For example, from its green, mottled skin, large size, and ovoid shape, one can infer that an object is a watermelon; from this, one infers that it would be useful for fruit salad.

There are two main choices for representing categories in first-order logic. The first we have already seen: categories are represented by unary predicates. The predicate symbol *Tomato*, for example, represents the unary relation that is true only for objects that are tomatoes, and *Tomato(x)* means that *x* is a tomato.

## REIFICATION

The second choice is to **reify** the category. **Reification** is the process of turning a predicate or function into an object in the language.<sup>7</sup> We will see several examples of reification in this chapter. In this case, we use *Tomatoes* as a constant symbol referring to the object that is the *set of all tomatoes*. We use *x G Tomatoes* to say that *x* is a tomato. Reified categories allow us to make assertions about the category itself, rather than about members of the category. For example, we can say *Population(Humans) = 5,000,000,000*, even though there is no individual human with a population of five billion.

## INHERITANCE

Categories perform one more important role: they serve to organize and simplify the knowledge base through **inheritance**. If we say that all instances of the category *Food* are edible, and if we assert that *Fruit* is a subclass of *Food* and *Apples* is a subclass of *Fruit*, then we know that every apple is edible. We say that the individual apples **inherit** the property of edibility, in this case from their membership in the *Food* category.

## TAXONOMY

Subclass relations organize categories into a **taxonomy** or **taxonomic hierarchy**. Taxonomies have been used explicitly for centuries in technical fields. For example, systematic biology aims to provide a taxonomy of all living and extinct species; library science has developed a taxonomy of all fields of knowledge, encoded as the Dewey Decimal system; tax authorities and other government departments have developed extensive taxonomies of occupations and commercial products. Taxonomies are also an important aspect of general commonsense knowledge, as we will see in our investigations that follow.

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

- An object is a member of a category. For example:  
*Tomato*<sub>12</sub>*G Tomatoes*
- A category is a subclass of another category. For example:  
*Tomatoes C Fruit*
- All members of a category have some properties. For example:  
 $\forall x \ x \text{ G Tomatoes} \Rightarrow \text{Red}(x) \wedge \text{Round}(x)$
- Members of a category can be recognized by some properties. For example:  
 $\forall x \ \text{Red}(\text{Interior}(x)) \wedge \text{Green}(\text{Exterior}(x)) \wedge x \in \text{Melons} \Rightarrow x \in \text{Watermelons}$
- A category as a whole has some properties. For example:  
*Tomatoes G DomesticatedSpecies*

<sup>7</sup> The term "reification" comes from the Latin word *res*, or thing. John McCarthy proposed the term "thingification," but it never caught on.

Notice that because *Tomatoes* is a category, and is a member of *DomesticatedSpecies*, then *DomesticatedSpecies* must be a category of categories. One can even have categories of categories of categories, but they are not much use.

Although subclass and instance relations are the most important ones for categories, we also want to be able to state relations between categories that are not subclasses of each other. For example, if we just say that *Males* and *Females* are subclasses of *Animals*, then we have not said that a male cannot be a female. We say that two or more categories are **disjoint** if they have no members in common. And even if we know that males and females are disjoint, we will not know that an animal that is not a male must be a female unless we say that males and females constitute an **exhaustive decomposition** of the animals. A disjoint exhaustive decomposition is known as a **partition**. The following examples illustrate these three concepts:

*Disjoint*( $\{Animals, Vegetables\}$ )  
*ExhaustiveDecomposition*( $\{Americans, Canadians, Mexicans\}$ , *NorthAmericans*)  
*Partition*( $\{Males, Females\}$ , *Animals*)

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

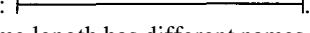
$$\begin{aligned} \forall s \ Disjoint(s) &\Leftrightarrow (\forall c_1, c_2 \in s \ A \ c_1 \neq c_2 \Rightarrow \text{Intersection}(c_1, c_2) = \text{EmptySet}) \\ \forall s, c \ ExhaustiveDecomposition(s, c) &\Leftrightarrow (\forall i \ i \in s \Rightarrow \exists c_1 \ c_1 \in s \ A \ i \in c_1) \\ \forall s, c \ Partition(s, c) &\Leftrightarrow Disjoint(s) \ A \ ExhaustiveDecomposition(s, c) \end{aligned}$$

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

$$\forall v \ Bachelor(v) \Leftrightarrow Male(v) \wedge Adult(v) \ A \ Unmarried(v)$$

As we discuss in the sidebar on natural kinds, strict logical definitions for categories are not always possible, nor always necessary.

## Measures

In both scientific and commonsense theories of the world, objects have height, mass, cost, and so on. The values that we assign for these properties are called **measures**. Ordinary, quantitative measures are quite easy to represent. We imagine that the universe includes abstract "measure objects," such as the *length* that is the length of this line segment:  We can call this length 1.5 inches, or 3.81 centimeters. Thus, the same length has different names in our language. Logically, this can be done by combining a **units function** with a number. If  $L_1$  is the name of the line segment, then we can write

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

Conversion between units is done with sentences such as

$$\begin{aligned} \forall l \ Centimeters(2.54 \times l) &= Inches(l) \\ \forall t \ Centigrade(t) &= Fahrenheit(32 + 1.8 \times t) \end{aligned}$$

## NATURAL KINDS

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

This poses a problem for a logical agent. The agent cannot be sure that an object it has perceived is a tomato, and even if it was sure, it could not be certain which of the properties of typical tomatoes this one has. This problem is an inevitable consequence of operating in inaccessible environments. The following mechanism provides a way to deal with natural kinds within a logical system.

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

$$\forall c \text{ } \textit{Typical}(c) \subseteq c$$

Most of the knowledge about natural kinds will actually be about the typical instances:

$$\forall x \text{ } x \in \textit{Typical}(\textit{Tomatoes}) \Rightarrow \textit{Red}(x) \wedge \textit{Spherical}(x)$$

In this way, we can still write down useful facts about categories without providing exact definitions.

The difficulty of providing exact definitions for most natural categories was explained in depth by Wittgenstein (1953), in his book *Philosophical Investigations*. He used the example of *games* to show that members of a category shared "family resemblances" rather than necessary and sufficient characteristics. The *Investigations* also revolutionized our understanding of language, as we discuss further in Chapter 22.

The utility of the notion of strict definition was also challenged by Quine (1953). He pointed out that even the definition of "bachelor" given before is suspect; one might, for example, question a statement such as "the Pope is a bachelor." The category "bachelor" still plays a useful role in natural language and in formal knowledge representation, because it simplifies many sentences and inferences.

Similar axioms can be written for pounds and kilograms; seconds and days; dollars and cents. (Exercise 8.9 asks you to represent exchange rates between currencies, where those exchange rates can vary over time.)

Measures can be used to describe objects as follows:

$$\text{Mass}(\text{Tomato}_{12}) = \text{Kilograms}(0.16)$$

$$\text{Price}(\text{Tomato}_{12}) = \$0.32$$

$$\forall d \in \text{Days} \Rightarrow \text{Duration}(d) = \text{Hours}(24)$$

It is very important to be able to distinguish between monetary amounts and monetary instruments:

$$\forall b \in \text{DollarBills} \Rightarrow \text{CashValue}(b) = \$1.00$$

This will be useful when it comes to paying for things later in the chapter.

Simple, quantitative measures are easy to represent. There are other measures that present more of a problem, because they have no agreed scale of values. Exercises have difficulty, desserts have deliciousness, and poems have beauty, yet numbers cannot be assigned to these qualities. One might, in a moment of pure accountancy, dismiss such properties as useless for the purpose of logical reasoning; or, still worse, attempt to impose a numerical scale on beauty. This would be a grave mistake, because it is unnecessary. The most important aspect of measures is not the particular numerical values, but the fact that measures can be *ordered*.

Although measures are not numbers, we can still compare them using an ordering symbol such as  $>$ . For example, we might well believe that Norvig's exercises are tougher than Russell's, and that one scores less on tougher exercises:

$$\begin{aligned} \forall e_1, e_2 \in \text{Exercises} \ A e_2 \in \text{Exercises} \ A \text{Wrote}(\text{Norvig}, e_1) \ A \text{Wrote}(\text{Russell}, e_2) \Rightarrow \\ \text{Difficulty}(e_1) > \text{Difficulty}(e_2) \end{aligned}$$

$$\begin{aligned} \forall e_1, e_2 \in \text{Exercises} \ A e_2 \in \text{Exercises} \ A \text{Difficulty}(e_1) > \text{Difficulty}(e_2) \Rightarrow \\ \text{ExpectedScore}(e_1) < \text{ExpectedScore}(e_2) \end{aligned}$$

This is enough to allow one to decide which exercises to do, even though no numerical values for difficulty were ever used. (One does, however, have to determine who wrote which exercises.) These sorts of monotonic relationships among measures form the basis for the field of **qualitative physics**, a subfield of AI that investigates how to reason about physical systems without plunging into detailed equations and numerical simulations. Qualitative physics is discussed in the historical notes section.

## Composite objects

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

$$\text{PartOf}(\text{Bucharest}, \text{Romania})$$

$$\text{PartOf}(\text{Romania}, \text{EasternEurope})$$

$$\text{PartOf}(\text{EasternEurope}, \text{Europe})$$

From these, given the transitivity of *PartOf*, we can infer that *PartOf(Bucharest, Europe)*.

COMPOSITE OBJECT  
STRUCTURE

Any object that has parts is called a **composite object**. Categories of composite objects are often characterized by the **structure** of those objects, that is, the parts and how the parts are related. For example, a biped has exactly two legs that are attached to its body:

$$\begin{aligned} \forall a \ Biped(a) \Rightarrow \\ \exists l_1, l_2, b \ Leg(l_1) \wedge Leg(l_2) \wedge Body(b) \wedge \\ PartOf(l_1, a) \wedge PartOf(l_2, a) \wedge PartOf(b, a) \wedge \\ Attached(l_1, b) \wedge Attached(l_2, b) \wedge \\ l_1 \neq l_2 \wedge \forall l_3 \ Leg(l_3) \wedge PartOf(l_3, a) \Rightarrow (l_3 = l_1 \vee l_3 = l_2) \end{aligned}$$

SCHEMA  
SCRIPT

This general form of sentence can be used to define the structure of any composite object, including events: for example, for all baseball games, there exist nine innings such that each is a part of the game, and so on. A generic event description of this kind is often called a **schema** or **script**, particularly in the area of natural language understanding. Some approaches to text understanding rely mainly on the ability to recognize instances of schematic events from descriptions of their parts, so that the text can be organized into coherent events, and questions can be answered about parts not explicitly mentioned. We discuss these issues further in Chapter 22.

BUNCH

We can define a *PartPartition* relation analogous to the *Partition* relation for categories (see Exercise 8.4). An object is composed of the parts in its *PartPartition*, and can be viewed as deriving some properties from those parts. For example, the mass of a composite object is the sum of the masses of the parts. Notice that this is not the case with categories: categories have no mass, even though their elements might.

It is also useful to define composite objects with definite parts but no particular structure. For example, we might want to say, "The apples in this bag weigh three pounds." Rather than commit the error of assigning the weight to the *category* of apples-in-the-bag, we can make a **bunch** out of the apples. For example, if the apples are *Apple*<sub>1</sub>, *Apple*<sub>2</sub>, and *Apple*<sub>3</sub>, then

$$BunchOf\{\textit{Apple}_1, \textit{Apple}_2, \textit{Apple}_3\})$$

denotes the composite object with the three apples as parts. We can then use the bunch as a normal, albeit unstructured, object. Notice that *BunchOfApples*) is the composite object consisting of all apples—not to be confused with *Apples*, the category.

## Representing change with events

Section 7.6 showed how situation calculus could be used to represent change. Situation calculus is perfect for the vacuum world, the wumpus world, or any world in which a single agent takes discrete actions. Unfortunately, situation calculus has two problems that limit its applicability. First, situations are instantaneous points in time, which are not very useful for describing the gradual growth of a kitten into a cat, the flow of electrons along a wire, or any other process where change occurs continuously over time. Second, situation calculus works best when only one action happens at a time. When there are multiple agents in the world, or when the world can change spontaneously, situation calculus begins to break down. It is possible to prop it back up for a while by defining composite actions, as in Exercise 7.12. If there are actions that have different durations, or whose effects depend on duration, then situation calculus in its intended form cannot be used at all.

EVENT CALCULUS

EVENT

SUBEVENTS

INTERVAL

Because of these limitations, we now turn to a different approach toward representing change, which we call the **event calculus**, although the name is not standard. Event calculus is rather like a continuous version of the situation-calculus "movie" shown in Figure 7.3. We think of a particular universe as having both a "spatial" and a temporal dimension. The "spatial" dimension ranges over all of the objects in an instantaneous "snapshot" or "cross-section" of the universe.<sup>8</sup> The temporal dimension ranges over time. An **event** is, informally, just a "chunk" of this universe with both temporal and spatial extent. Figure 8.3 gives the general idea.

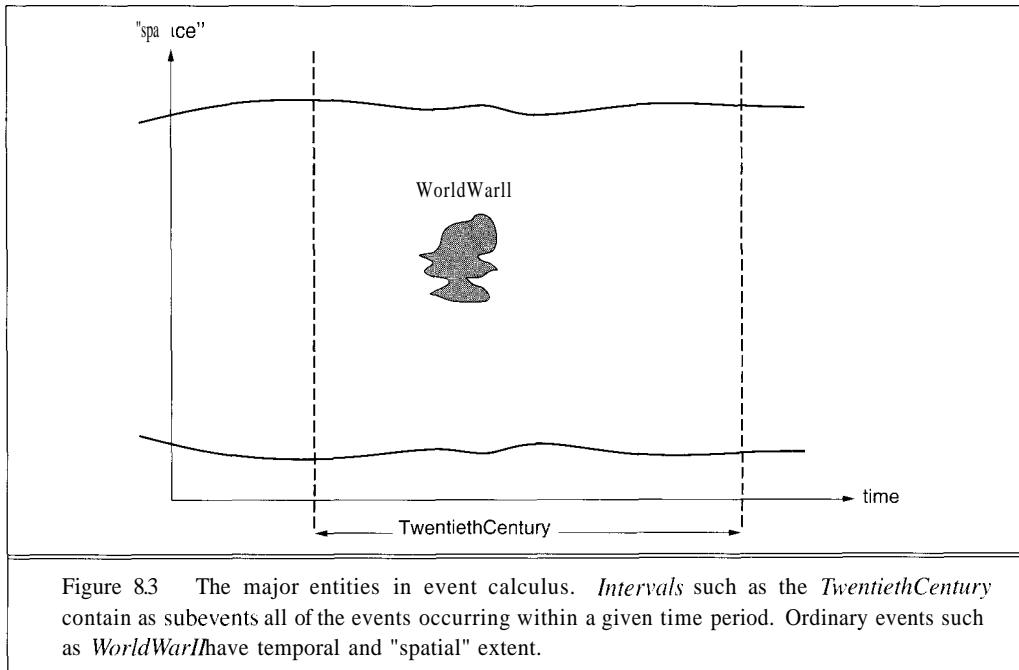


Figure 8.3 The major entities in event calculus. *Intervals* such as the *TwentiethCentury* contain as subevents all of the events occurring within a given time period. Ordinary events such as *WorldWarII* have temporal and "spatial" extent.

Let us look at an example: World War II, referred to by the symbol *WorldWarII*. World War II has parts that we refer to as **subevents**:<sup>9</sup>

*SubEvent(BattleOfBritain, WorldWarII)*

Similarly, World War II is a subevent of the twentieth century:

*SubEvent(WorldWarII, TwentiethCentury)*

The twentieth century is a special kind of event called an **interval**. An interval is an event that includes as subevents all events occurring in a given time period. Intervals are therefore entire temporal sections of the universe, as the figure illustrates. In situation calculus, a given fact is true in a particular situation. In event calculus, a given event occurs during a particular interval. The previous *SubEvent* sentences are examples of this kind of statement.

<sup>8</sup> We put "spatial" in quotes because it is possible that the set of objects being considered does not include places at all; nonetheless, it is a helpful metaphor. From now on, we will leave off the quotes.

<sup>9</sup> Note that *SubEvent* is a special case of the *PartOf* relation, and is also transitive and reflexive.

Like any other sort of object, events can be grouped into categories. For example, *WorldWarII* belongs to the category *Wars*. To say that a war occurred in the Middle East in 1967, we would say

$$\exists w \ w \in \text{Wars} \ A \text{SubEvent}(w, \text{AD } 1967) \ A \text{PartOf}(\text{Location}(w) \text{MiddleEast})$$

To say that Shankar travelled from New York to New Delhi yesterday, we might use the category *Journeys*, as follows:

$$\begin{aligned} \exists j \ j \in \text{Journeys} \ A \text{Origin}(\text{NewYork}, j) \ A \text{Destination}(\text{NewDelhi}, j) \\ A \text{Traveller}(\text{Shankar}, j) \ A \text{SubEvent}(j, \text{Yesterday}) \end{aligned}$$

This notation can get a little tedious, particularly because we are often interested more in the event's properties than in the event itself. We can simplify the descriptions by using complex terms to name event categories. For example, *Go(Shankar, NewYork, NewDelhi)* names the category of events in which Shankar travels from New York to New Delhi. The function symbol *Go* can be defined by the following sentence:

$$\begin{aligned} \forall e, x, o, d \ e \in \text{Go}(x, o, d) \Leftrightarrow \\ e \in \text{Journeys} \ A \text{Traveller}(x, e) \ A \text{Origin}(o, e) \ A \text{Destination}(d, e) \end{aligned}$$

Finally, we use the notation *E(c, i)* to say that an event of category *c* is a subevent of the event (or interval) *i*:

$$\forall c, i \ E(c, i) \Leftrightarrow \exists e \ e \in c \ A \text{SubEvent}(e, i)$$

Thus, we have

$$E(\text{Go}(\text{Shankar}, \text{NewYork}, \text{NewDelhi}), \text{Yesterday})$$

which means "there was an event that was a going by Shankar from New York to New Delhi that took place sometime yesterday."

## Places

PLACES

**Places**, like intervals, are special kinds of space-time chunks. A place can be thought of as a constant piece of space, extended through time.<sup>10</sup> New York and the Middle East are places, at least as far as recent history is concerned. We use the predicate *In* to denote the special kind of subevent relation that holds between places; for example:

$$In(\text{NewYork}, \text{USA})$$

Places come in different varieties; for example, *NewYork* is an *Area*, whereas the *SolarSystem* is a *Volume*. The *Location* function, which we used earlier, maps an object to the smallest place that contains it:

$$\begin{aligned} \forall x, / \ Location(x) = l \Leftrightarrow \\ At(x, l) \wedge \forall l_2 \ At(x, l_2) \Rightarrow In(l, l_2) \end{aligned}$$

MINIMIZATION

This last sentence is an example of a standard logical construction called **minimization**.

<sup>10</sup> We will not worry about local inertial coordinate frames and other things that physicists need to pin down exactly what this means. Places that *change* over time are dealt with in a later section.

## Processes

DISCRETE EVENTS

The events we have seen so far have been what we call **discrete events**—they have a definite structure. Shankar's trip has a beginning, middle, and end. If interrupted halfway, the event would be different—it would not be a trip from New York to New Delhi, but instead a trip from New York to somewhere in the Eastern Mediterranean. On the other hand, the category of events denoted by *Flying(Shankar)* has a different quality. If we take a small interval of Shankar's flight, say, the third twenty-minute segment (while he waits anxiously for a second bag of honey-roasted peanuts), that event is still a member of *Flying(Shankar)*. In fact, this is true for any subinterval.

PROCESS

LIQUID EVENT

Categories of events with this property are called **process** categories or **liquid event** categories. Any subinterval of a process is also a member of the same process category. We can use the same notation used for discrete events to say that, for example, Shankar was flying at some time yesterday:

$$E(Flying(Shankar), \text{Yesterday})$$

We often want to say that some process was going on *throughout* some interval, rather than just in some subinterval of it. To do this, we use the predicate *T*:

$$T(Working(Stuart), \text{TodayLunchHour})$$

STATES

*T(c, i)* means that some event of type *c* occurred over exactly the interval *i*—that is, the event begins and ends at the same time as the interval. Exercise 8.6 asks you to define *T* formally.

As well as describing processes of continuous change, liquid events can describe processes of continuous non-change. These are often called **states**. For example, "Mary being in the local supermarket" is a category of states that we might denote by *In(Mary, Supermarket<sub>1</sub>)*. To say she was in the supermarket all this afternoon, we would write

$$T(In(Mary, Supermarket_1), \text{ThisAfternoon})$$

An interval can also be a discontinuous sequence of times; we can represent the fact that the supermarket is closed every Sunday with

$$T(Closed(Supermarket_1), \text{BunchOf}(Sundays))$$

## Special notation for combining propositions

It is tempting to write something like

$$T((At(Agent, Loc_1) \wedge At(Tomato_1, Loc_1)), I_3)$$

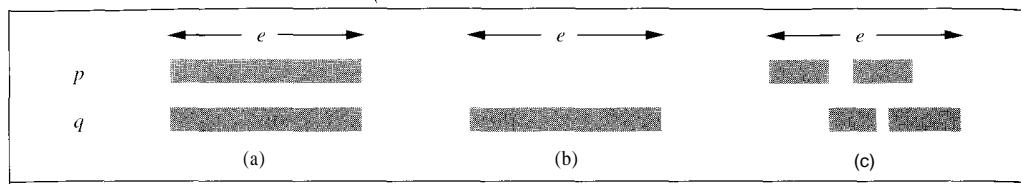
but technically this is nonsense, because a sentence appears as the first argument of the predicate *T*, and all arguments to predicates must be terms, not sentences. This is easily fixed by introducing a function called *And* that takes two event categories as arguments and returns a category of composite events of the appropriate kind:

$$T(And(At(AgentLoc_1), At(Tomato_1, Loc_1)), E)$$

We can define the function *And* with the axiom

$$\forall p, q, e \ T(And(p, q), e) \Leftrightarrow \ T(p, e) \wedge T(q, e)$$

Thus,  $And(p, q)$  is the category of composite “ $p$ - $q$ -events,” where a  $p$ - $q$ -event is an event in which both a  $p$  and a  $q$  occur. If you think of a  $p$ -event as a piece of “videotape” of a  $p$  happening, and a  $q$ -event as a “videotape” of a  $q$  happening, then the  $p$ - $q$ -event is like having the two pieces of tape spliced together in parallel (see Figure 8.4(a)).



**Figure 8.4** A depiction of complex events. (a)  $T(p \wedge q, e)$  (b)  $T(p \vee q, e)$  (c)  $T(p \veebar q, e)$

Once a method for conjoining event categories is defined, it is convenient to extend the syntax to allow regular infix connective symbols to be used in place of the function name:

$$T(p \wedge q, e) \Leftrightarrow T(And(p, q), e) \Leftrightarrow T(p, e) \text{ A } T(q, e)$$

This is fine as long as you remember that in  $T(p \wedge q, s)$ , the expression  $p \wedge q$  is a term denoting a category of events, not a sentence.

One might think that we can just go ahead and define similar functions for disjunctive and negated events. In fact, because the  $T$  predicate is essentially a *conjunction* (over all the subintervals of the interval in question), it can interact in two different ways with disjunction and negation. For example, consider the English sentence “One of the two shops was open all day on Sunday.” This could mean, “Either the first shop was open all day on Sunday, or the second shop was open all day on Sunday” (Figure 8.4(b)), or it could mean, “At any given time on Sunday, at least one of the two shops was open” (Figure 8.4(c)). Both of these meanings are useful sorts of things to say, so we will need a concise representation for each. There are no accepted notations in this case, so we will make some up. Let  $\vee$  be used for the first kind of disjunctive event, and  $\veebar$  be used for the second. For the first, we have the following definition:

$$T(p \veebar q, e) \Leftrightarrow T(p, e) \vee T(q, e)$$

We leave the second as an exercise, along with the definitions for negated events.

## Times, intervals, and actions

In this section, we flesh out the vocabulary of time intervals. Because it is a limited domain, we can be more complete in deciding on a vocabulary and encoding general rules. Time intervals are partitioned into moments and extended intervals. The distinction is that only moments have zero duration:

$$\begin{aligned} & \text{Partition}(\{\text{Moments}, \text{ExtendedIntervals}\}, \text{Intervals}) \\ & \forall i : i \in \text{Intervals} \Rightarrow (\exists G \text{ Moments} \Leftrightarrow \text{Duration}(i)=0) \end{aligned}$$

Now we invent a time scale and associate points on that scale with moments, giving us absolute times. The time scale is arbitrary; we will measure it in seconds and say that the moment at

midnight (GMT) on January 1, 1900, has time 0. The functions *Start* and *End* pick out the earliest and latest moments in an interval, and the function *Time* delivers the point on the time scale for a moment. The function *Duration* gives the difference between the end time and the start time.

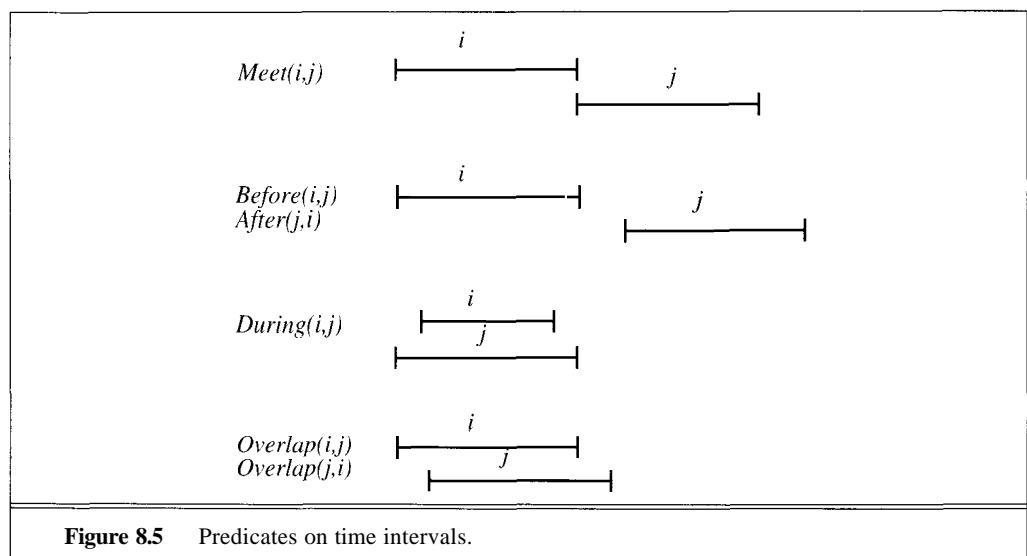
$$\begin{aligned} \forall i \text{ Interval}(i) &\Rightarrow Duration(i) = (Time(End(i)) - Time(Start(i))) \\ Time(Start(AD1900)) &= Seconds(0) \\ Time(Start(AD1991)) &= Seconds(2871694800) \\ Time(End(AD1991)) &= Seconds(2903230800) \\ Duration(AD1991) &= Seconds(31536000) \end{aligned}$$

To make these numbers easier to read, we also introduce a function *Date*, which takes six arguments (hours, minutes, seconds, month, day, and year) and returns a point on the time scale:

$$\begin{aligned} Time(Start(AD1991)) &= SecondsDate(00, 00, 00, Jan, 1, 1991) \\ Date(12, 34, 56, Feb, 14, 1993) &= 2938682096 \end{aligned}$$

The simplest relation between intervals is *Meet*. Two intervals *Meet* if the end time of the first equals the start time of the second. It is possible to define predicates such as *Before*, *After*, *During*, and *Overlap* solely in terms of *Meet*, but it is more intuitive to define them in terms of points on the time scale. (See Figure 8.5 for a graphical representation.)

$$\begin{aligned} \forall i, j \text{ Meet}(i, j) &\Leftrightarrow Time(End(i)) = Time(Start(j)) \\ \forall i, j \text{ Before}(i, j) &\Leftrightarrow Time(End(i)) < Time(Start(j)) \\ \forall i, j \text{ After}(j, i) &\Leftrightarrow Before(i, j) \\ \forall i, j \text{ During}(i, j) &\Leftrightarrow Time(Start(j)) < Time(Start(i)) \wedge Time(End(i)) < Time(End(j)) \\ \forall i, j \text{ Overlap}(i, j) &\Leftrightarrow \exists k \text{ During}(k, i) \wedge During(k, j) \end{aligned}$$



For example, to say that the reign of Elizabeth II followed that of George VI, and the reign of Elvis overlapped with the 1950s, we can write the following:

$$\text{After}(\text{ReignOf}(\text{ElizabethII}), \text{ReignOf}(\text{GeorgeVI}))$$

$$\text{Overlap}(\text{Fifties}, \text{ReignOf}(\text{Elvis}))$$

$$\text{Start}(\text{Fifties}) = \text{Start}(\text{AD1950})$$

$$\text{End}(\text{Fifties}) = \text{End}(\text{AD1959})$$

Temporal relations among intervals are used principally in describing actions. This is done in much the same way in event calculus as it is in situation calculus. The difference is that instead of defining a resulting situation and describing it, one defines a resulting interval, in which a certain state occurs. The following examples illustrate the general idea:

1. If two people are engaged, then in some future interval, they will either marry or break the engagement.

$$\begin{aligned} \forall x, y, i_0 \quad T(\text{Engaged}(x, y), i_0) \Rightarrow \\ \exists i_1 \quad (\text{Meet}(i_0, i_1) \vee \text{After}(i_1, i_0)) \wedge \\ T(\text{Marry}(x, y) \vee \text{BreakEngagement}(x, y), i_1) \end{aligned}$$

2. When two people marry, they are spouses for some interval starting at the end of the marrying event.

$$\forall x, y, i_0 \quad T(\text{Marry}(x, y), i_0) \Rightarrow \exists i_1 \quad T(\text{Spouse}(x, y), i_1) \wedge \text{Meet}(i_0, i_1)$$

3. The result of going from one place to another is to be at that other place.

$$\forall x, a, b, i_0, \exists i_1 \quad T(\text{Go}(x, a, b), i_0) \Rightarrow T(\text{In}(x, b), i_1) \wedge \text{Meet}(i_0, i_1)$$

We shall have more to say on the subject of actions and intervals in Part IV, which covers planning with these sorts of action descriptions.

## Objects revisited

One purpose of situation calculus was to allow objects to have different properties at different times. Event calculus achieves the same goal. For example, we can say that Poland's area in 1426 was 233,000 square miles, whereas in 1950 it was 117,000 square miles:

$$T(\text{Area}(\text{Poland}, \text{SqMiles}(233000)), \text{AD1426})$$

$$T(\text{Area}(\text{Poland}, \text{SqMiles}(117000)), \text{AD 1950})$$

In fact, as well as growing and shrinking, Poland has moved about somewhat on the map. We could plot its land area over time, as shown in Figure 8.6. We see that Poland has a temporal as well as spatial extent. *It turns out to be perfectly consistent to view Poland as an event.* We can then use temporal subevents such as *19thCenturyPoland*, and spatial subevents such as *CentralPoland*.

The USA has also changed various aspects over time. One aspect that changes every four or eight years, barring mishaps, is its president. In event calculus, *President(USA)* denotes an object that consists of different people at different times. *President(USA)* is the object that is George Washington from 1789 to 1796, John Adams from 1796 to 1800, and so on (Figure 8.7). To say that the president of the USA in 1994 is a Democrat, we would use

$$T(\text{Democrat}(\text{President}( \text{USA})), \text{AD 1994})$$

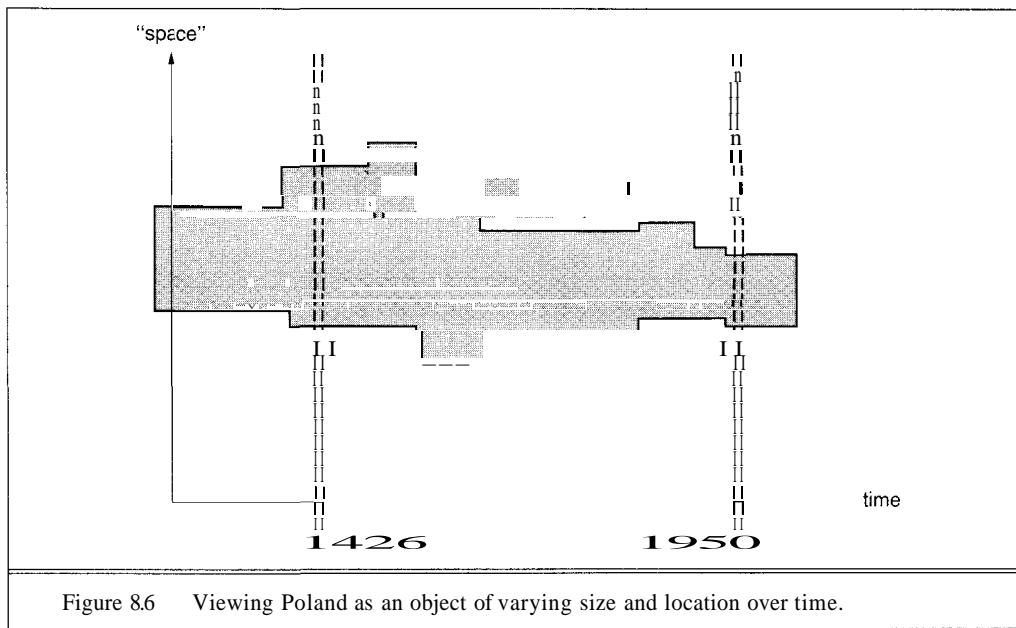



Figure 8.6 Viewing Poland as an object of varying size and location over time.

## FLUENTS

Objects such as *Poland* and *President(USA)* are called **fluents**. The dictionary says that a fluent is something that is capable of flowing, like a liquid. For our purposes, a fluent is something that flows or changes across situations.

It may seem odd to reify objects that are as transient as *President(USA)*, yet fluents allow us to say some things that would otherwise be cumbersome to express. For example, "The president of the USA was male throughout the 19th century" can be expressed by

$T(\text{Male}(\text{President}(\text{USA})), \text{19thCentury})$

even though there were 24 different presidents of the USA in the nineteenth century.

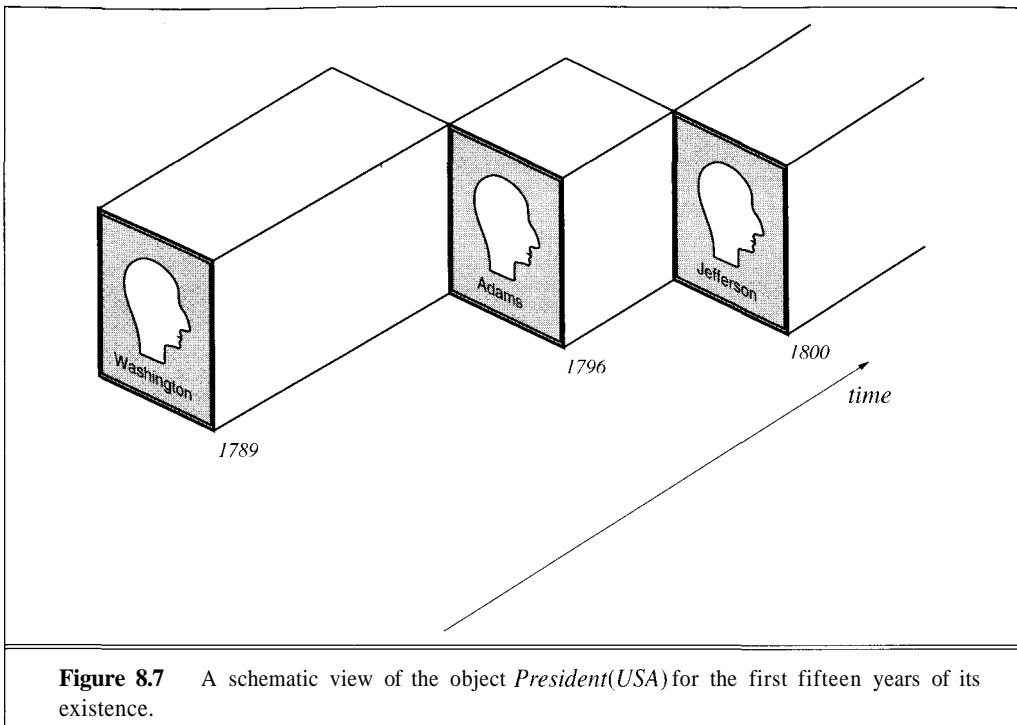
There are some things that can be easily expressed with fluents but not with situation calculus. For example, *Location(x)* denotes the place in which  $x$  is located, even if that place varies over time. We can then use sentences about the location of an object to express the fact that, for example, the location of the Empire State Building is fixed:

$\text{Fixed}(\text{Location}(\text{EmpireStateBuilding}))$

Without fluents, we could find a way to talk about *objects* being widespread or immobile, but we could not talk directly about the object's location over time. We leave it as an exercise to define the *Fixed* predicate (Exercise 8.7).

## Substances and objects

The real world perhaps can be seen as consisting of primitive objects (particles) and composite objects built from them. By reasoning at the level of large objects such as apples and cars, we can overcome the complexity involved in dealing with vast numbers of primitive objects



**Figure 8.7** A schematic view of the object *President(USA)* for the first fifteen years of its existence.

INDIVIDUATION  
STUFF

TEMPORAL  
SUBSTANCES  
SPATIAL  
SUBSTANCES

COUNT NOUNS  
MASS NOUNS

individually. There is, however, a significant portion of reality that seems to defy any obvious **individuation**—division into distinct objects. We give this portion the generic name **stuff**. For example, suppose I have some butter and an aardvark in front of me. I can say there is one aardvark, but there is no obvious number of "butter-objects," because any part of a butter-object is also a butter-object, at least until we get to very small parts indeed. This is the major distinction between stuff and things. If we cut an aardvark in half, we do not get two aardvarks, unfortunately. The distinction is exactly analogous to the difference between liquid and nonliquid events. In fact, some have called liquid event types **temporal substances**, whereas things like butter are **spatial substances** (Lenat and Guha, 1990).

Notice that English enforces the distinction between stuff and things. We say "an aardvark," but, except in pretentious California restaurants, one cannot say "a butter." Linguists distinguish between **count nouns**, such as aardvarks, holes, and theorems, and **mass nouns**, such as butter, water, and energy. Should we also enforce this distinction in our representation by treating butter and aardvarks differently, or can we treat them using a uniform mechanism?

To represent stuff properly, we begin with the obvious. We will need to have as objects in our ontology at least the gross "lumps" of stuff that we interact with. For example, we might recognize the butter as the same butter that was left on the table the night before; we might pick it up, weigh it, sell it, or whatever. In these senses, it is an object just like the aardvark. Let us call it *Butter<sub>3</sub>*. We will also define the category *Butter*. Informally, its elements will be all those things of which one might say "It's butter," including *Butter<sub>3</sub>*.

The next thing to state was mentioned earlier: with some caveats about very small parts that we will omit for now, any part of a butter-object is also a butter-object:

$$\forall x, y \ x \text{ G } Butter \ A \ PartOf(yx) \Rightarrow y \in Butter$$

Individual aardvarks derive properties such as approximate shape, size, weight, and diet from membership in the category of aardvarks. What sorts of properties does an object derive from being a member of the *Butter* category? Butter melts at around 30 degrees centigrade:

$$\forall x \ Butter(x) \Rightarrow MeltingPoint(x, Centigrade(30))$$

Butter is yellow, less dense than water, soft at room temperature, has a high fat content, and so on. On the other hand, butter has no particular size, shape, or weight. We can define more specialized categories of butter such as *UnsaltedButter*, which is also a kind of stuff because any part of an unsalted-butter-object is also an unsalted-butter-object. On the other hand, if we define a category *PoundOfButter*, which includes as members all butter-objects weighing one pound, we no longer have a substance! If we cut a pound of butter in half, we do not get two pounds of butter—another of those annoying things about the world we live in.

INTRINSIC

EXTRINSIC

What is actually going on is this: there are some properties that are **intrinsic**: they belong to the very substance of the object, rather than to the object as a whole. When you cut something in half, the two pieces retain the same set of intrinsic properties—things like density, boiling point, flavor, color, ownership, and so on. On the other hand, **extrinsic** properties are the opposite: properties such as weight, length, shape, function, and so on are not retained under subdivision.

A class of objects that includes in its definition only *intrinsic* properties is then a substance, or mass noun; a class that includes *any* extrinsic properties in its definition is a count noun. The category *Stuff* is the most general substance category, specifying no intrinsic properties. The category *Thing* is the most general discrete object category, specifying no extrinsic properties.

It follows that an object belongs to both mass and count classes. For example, *LakeMichigan* is an element of both *Water* and *Lakes*. *Lake* specifies such extrinsic properties as (approximate) size, topological shape, and the fact that it is surrounded by land. Note that we can also handle the fact that the water in Lake Michigan changes over time, simply by viewing the lake as an event whose constituent objects change over time, in much the same way as *President(USA)*.

This approach to the representation of stuff is not the only possibility. The major competing approach considers *Butter* to be what we would call *BunchOfButter*), namely, the object composed of all butter in the world. All individual butter-objects are thus *PartOf* butter, rather than instances of butter. Like any consistent knowledge representation scheme, it cannot be *proved incorrect*, but it does seem to be awkward for representing specialized kinds of substances such as *UnsaltedButter* and its relation to *Butter*.

## Mental events and mental objects

The agents we have constructed so far have beliefs and can deduce new beliefs. Yet none of them has any knowledge *about* beliefs or deduction. For single-agent domains, knowledge about one's own knowledge and reasoning processes is useful for controlling inference. For example, if one knows that one does not know anything about Romanian geography, then one need not expend enormous computational effort trying to calculate the shortest path from Arad to Bucharest. In

'multiagent domains, it becomes important for an agent to reason about the mental processes of the other agents. Suppose a shopper in a supermarket has the goal of buying some anchovies. The agent deduces that a good plan is to go where the anchovies are, pick some up, and bring them to the checkout stand. A key step is for the shopper to realize that it cannot execute this plan until it knows where the anchovies are, and that it can come to know where they are by asking someone. The shopper should also deduce that it is better to ask a store employee than another customer, because the employee is more likely to know the answer. To do this kind of deduction, an agent needs to have a model of what other agents know, as well as some knowledge of its own knowledge, lack of knowledge, and inference procedures.

In effect, we want to have a model of the mental objects that are in someone's head (or something's knowledge base) and of the mental processes that manipulate those mental objects. The model should be faithful, but it does not have to be detailed. We do not have to be able to predict how many milliseconds it will take for a particular agent to make a deduction, nor do we have to predict what neurons will fire when an animal is faced with a particular visual stimulus. But we do want an abstract model that says that if a logical agent believes  $P \vee Q$  and it learns  $\neg P$ , then it should come to believe  $Q$ .

The first step is to ask how mental objects are represented. That is, if we have a relation *Believes*(Agent,  $x$ ), what kind of thing is  $x$ ? First of all, it's clear that  $x$  cannot be a logical sentence. If *Flies*(*Superman*) is a logical sentence, we can't say *Believes*(Agent, *Flies*(*Superman*)) because only terms (not sentences) can be arguments of relations. But if *Flies*(*Superman*) is reified as a fluent, then it is a candidate for being a mental object, and *Believes* can be a relation that takes an agent and a propositional fluent that the agent believes in. We could define other relations such as *Knows* and *Wants* to express other relationships between agents and propositions. Relations of this kind are called **propositional attitudes**.

This appears to give us what we want: the ability for an agent to reason about the beliefs of agents. Unfortunately, there is a problem with this approach. If Clark and Superman are one and the same (i.e.,  $Clark = Superman$ ) then Clark flying and Superman flying are one and the same event. Thus, if the object of propositional attitudes are reified events, we must conclude that if Lois believes that Superman can fly, she also believes that Clark can fly, even if she doesn't believe that Clark is Superman. That is,

$$(Superman = Clark) \models \\ (Believes(Lois, Flies(Superman)) \Leftrightarrow Belives(Lois, Flies(Clark)))$$

There is a sense in which this is right: Lois does believe of a certain person, who happens to be called Clark sometimes, that that person can fly. But there is another sense in which this is wrong: if you asked Lois "Can Clark fly?" she would certainly say no. Reified objects and events work fine for the first sense of *Believes*, but for the second sense we need to reify *descriptions* of those objects and events, so that Clark and Superman can be different descriptions (even though they refer to the same object).

Technically, the property of being able to freely substitute a term for an equal term is called **referential transparency**. In first-order logic, every relation is referentially transparent. We would like to define *Belives* (and the other propositional attitudes) as relations whose second argument is referentially **opaque**—that is, one cannot substitute an equal term for the second argument without changing the meaning.

SYNTACTIC THEORY  
STRINGS

We will concentrate on what is called a **syntactic theory** of mental objects." In this approach, we represent mental objects with **strings** written in a representation language. (We will use first-order logic itself as the representation language, but we are not required to.) A string is just a list of symbols, so the event *Flies(Clark)* can be represented by the string of characters  $[F, l, i, e, s, (, C, l, a, r, k, )]$ , which we will abbreviate as "*Flies(Clark)*". In this formulation, "*Clark*"  $\neq$  "*Superman*" because they are two different strings consisting of different symbols. The idea is that a knowledge-based agent has a knowledge base consisting of strings that were added either via TLLL or through inference. The syntactic theory models the knowledge base and the strings that are in it.

Now all we have to do is provide a syntax, semantics, and proof theory for the string representation language, just as we did in Chapter 6. The difference is that we have to define them all in first-order logic. We start by defining *Den* as the function that maps a string to the object that it denotes, and *Name* as a function that maps an object to a string that is the name of a constant that denotes the object. For example, the denotation of both "*Clark*" and "*Superman*" is the object referred to by the constant symbol *ManOfSteel*, and the name of that object could be either "*Superman*", "*Clark*", or some other constant, such as " $K_{11}$ ".

$$\begin{aligned} \text{Den}(\text{"Clark"}) - \text{ManOfSteel} & \quad \text{A } \text{Den}(\text{"Superman"}) = \text{ManOfSteel} \\ \text{Name}(\text{ManOfSteel}) &= "K_{11}" \end{aligned}$$

The next step is to define inference rules for logical agents. For example, we might want to say that a logical agent can do Modus Ponens: if it believes  $p$  and believes  $p \Rightarrow q$  then it will also believe  $q$ . The first attempt at writing this axiom is

$$\forall a, p, q \text{ LogicalAgent}(a) \text{A Believes}(a, p) \text{A Believes}(a, "p \Rightarrow q") \Rightarrow \text{Believes}(a, q)$$

But this is not right because the string " $p \Rightarrow q$ " contains the letters 'p' and 'q' but has nothing to do with the strings that are the values of the variables  $p$  and  $q$ . In fact, " $p \Rightarrow q$ " is not even a syntactically correct sentence, because only variables can be lower-case letters. The correct formulation is:

$$\begin{aligned} \forall a, p, q \text{ LogicalAgent}(a) \text{A Believes}(a, p) \text{A Believes}(a, \text{Concat}(p, \Rightarrow, q)) \\ \Rightarrow \text{Believes}(a, q) \end{aligned}$$

where *Concat* is a function on strings that concatenates their elements together. We will abbreviate *Concat*( $p, [\Rightarrow], q$ ) as " $p \Rightarrow q$ ". That is, an occurrence of  $x$  within a string means to substitute in the value of the variable  $x$ . Lisp programmers will recognize this as the backquote operator.

Once we add in the other inference rules besides Modus Ponens, we will be able to answer questions of the form "given that a logical agent knows these premises, can it draw that conclusion?" Besides the normal inference rules, we need some rules that are specific to belief. For example, the following rule says that if a logical agent believes something, then it believes that it believes it.

$$\forall a, p \text{ LogicalAgent}(a) \wedge \text{Believes}(a, p) \Rightarrow \text{Believes}(a, \text{"Believes"}(\text{Name}(a), p))$$

Note that it would not do to have just  $a$  as part of the string, because  $a$  is an agent, not a description of an agent. We use *Name(a)* to get a string that names the agent.

---

<sup>11</sup> An alternative based on **modal logic** is covered in the historical notes section.

There are at least three directions we could go from here. One is to recognize that it is unrealistic to expect that there will be any real logical agents. Such an agent can, according to our axioms, deduce any valid conclusion instantaneously. This is called **logical omniscience**. It would be more realistic to define limited rational agents, which can make a limited number of deductions in a limited time. But it is very hard to axiomatize such an agent. Pretending that all agents are logically omniscient is like pretending that all problems with polynomial time bounds are tractable—it is clearly false, but if we are careful, it does not get us into too much trouble.

A second direction is to define axioms for other propositional attitudes. The relation between believing and knowing has been studied for centuries by philosophers of the mind. It is commonly said that knowledge is justified true belief. That is, if you believe something, and if it is actually true, and if you have a proof that it is true, then you know it. The proof is necessary to prevent you from saying "I know this coin flip will come up heads" and then taking credit for being right when the coin does end up heads, when actually you just made a lucky guess. If you accept this definition of knowledge, then it can be defined in terms of belief and truth:

$$\forall a, p \ Knows(a, p) \Leftrightarrow Believes(a, p) \wedge T(Den(p)) \wedge T(Den(KB(a))) \Rightarrow Den(p))$$

This version of *Knows* can be read as "knows that." It is also possible to define other kinds of knowing. For example, here is a definition of "knowing whether":

$$\forall a, p \ KnowsWhether(a, p) \Leftrightarrow Knows(a, p) \vee Knows(a, \neg p)$$

Continuing our example, Lois knows whether Clark can fly if she either knows that Clark can fly or knows that he cannot.

The concept of "knowing what" is more complicated. One is tempted to say that an agent knows what Bob's phone number is if there is some  $x$  for which the agent knows  $x = Phone\#(Bob)$ . But that is not right, because the agent might know that Alice and Bob have the same number, but not know what it is (i.e.,  $Phone\#(Alice) = Phone\#(Bob)$ ), or the agent might know that there is some Skolem constant that is Bob's number without knowing anything at all about it (i.e.,  $K_{23} = Phone\#(Bob)$ ). A better definition of "knowing what" says that the agent has to know of some  $x$  that is a string of digits and that is Bob's number:

$$\begin{aligned} \forall a, b \ KnowsWhat(a, "Phone\#(b)") &\Leftrightarrow \\ 3x \ Knows(a, "x = Phone\#(b)") \wedge DigitString(x) \end{aligned}$$

Of course, for other questions we have different criteria for what is an acceptable answer. For the question "what is the capital of New York," an acceptable answer is a proper name, "Albany," not something like "the city where the state house is." To handle this, we will make *KnowsWhat* a three place relation: it takes an agent, a term, and a predicate that must be true of the answer. For example:

$$\begin{aligned} KnowsWhat(Agent, Capital(NewYork), ProperName) \\ KnowsWhat(Agent, Phone\#(Bob), DigitString) \end{aligned}$$

A third direction is to recognize that propositional attitudes change over time. When we recognized that processes occur over a limited interval of time, we introduced the relation  $T(process, interval)$ . Similarly, we can use  $Believe(agent, string, interval)$  to mean that an agent believes in a proposition over a given interval. For example, to say that Lois believed yesterday that Superman can fly, we write

$$Believes(Lois, Flies(Superman), Yesterday)$$

Actually, it would be more consistent to have *Believes* be an event fluent just as *Flies* is. Then we could say that it will be true tomorrow that Lois knew that Superman could fly yesterday:

$T(\text{Believes}(Lois, \text{Flies}(\text{Superman}), \text{Yesterday}), \text{Tomorrow})$

We can even say that it is true now that Jimmy knows today that Lois believes that Superman could fly yesterday:

$T(\text{Knows}(\text{Jimmy}, \text{Believes}(Lois, \text{Flies}(\text{Superman}), \text{Yesterday}), \text{Today}), \text{Now})$

## Knowledge and action

We have been so busy trying to represent knowledge that there is a danger of losing track of what knowledge is/or. Recall that we are interested in building agents that perform well. That means that the only way knowledge can help is if it allows the agent to do some action it could not have done before, or if it allows the agent to choose a better action than it would otherwise have chosen. For example, if the agent has the goal of speaking to Bob, then knowing Bob's phone number can be a great help. It enables the agent to perform a dialing action and have a much better chance of reaching Bob than if the agent did not know the number and dialed randomly.

One way of looking at this is to say that actions have **knowledge preconditions** and **knowledge effects**. For example, the action of dialing a person's number has the precondition of knowing the number, and the action of calling directory assistance sometimes has the effect of knowing the number.

Note that each action has its own requirements on the form of the knowledge, just as each question to *KnowsWhat* had its own requirements. Suppose I am in China, and the telephone there has Chinese numerals on the buttons.<sup>12</sup> Then knowing what Bob's number is as a digit string is not enough—I need to know it as a string of Chinese digits. Similarly, the question of whether I know where Bob lives has a different answer depending on how I want to use the information. If I'm planning to go there by taxi, all I need is an address; if I'm driving myself, I need directions; if I'm parachuting in, I need exact longitude and latitude.

KNOWLEDGE  
PRECONDITIONS  
KNOWLEDGE  
EFFECTS

## 8.5 THE GROCERY SHOPPING WORLD

---

In this section, all our hard work in defining a general ontology pays off: we will be able to define the knowledge that an agent needs to shop for a meal in a market. To demonstrate that the knowledge is sufficient, we will run a knowledge-based agent in our environment simulator. That means providing a simulated shopping world, which will by necessity be simpler than the real world. But much of the knowledge shown here is the same for simulated or real worlds. The big differences are in the complexity of vision, motion, and tactile manipulation. (These topics will be covered in Chapters 24 and 25.)

---

<sup>12</sup> Actually, Chinese phones have Arabic numerals, but bear with the example.

## Complete description of the shopping simulation

We start by giving a PAGE (percepts, actions, goals, and environment) description of the shopping simulation. First the **percepts**:

1. The agent receives three percepts at each time step: feel, sound, and vision.
2. The feel percept is just a bump or no bump, as in the vacuum world. The agent perceives a bump only when on the previous time step it executed a *Forward* action and there is not enough room in the location it tried to move to.
3. The sound percept is a list of spoken words. The agent perceives words spoken by agents within two squares of it.
4. If the agent's camera is zoomed in, it perceives detailed visual images of each object in the square it is zoomed at.
5. If the agent's camera is not zoomed in, it perceives coarse visual images of each object in the three squares directly and diagonally ahead.
6. A visual percept consists of a relative location, approximate size, color, shape, and possibly some other features. It will be explained in detail later.

Now for the **actions**:

1. An agent can speak a string of words.
2. An agent can go one square forward.
3. An agent can turn 90° to the right or left.
4. An agent can zoom its camera in at its current square, or at any of the three squares directly or diagonally ahead.
5. An agent can also zoom its camera out.
6. An agent can grab an object that is within one square of it. To do so, it needs to specify the relative coordinates of the object, and it needs to be empty-handed.
7. An agent can release an object that it has grabbed. To do so, it needs to specify the relative coordinates of the point where it wants to release the object.

The agent's **goal** initially will be to buy all the items on a shopping list. This goal can be modified if some items are unavailable or too expensive. The agent should also try to do the shopping quickly, and avoid bumping into things. A more ambitious problem is to give the agent the goal of making dinner, and let it compose the shopping list.

The **environment** is the interior of a store, along with all the objects and people in it. As in the vacuum and wumpus worlds, the store is represented by a grid of squares, with aisles separating rows of display cases. At one end of the store are the checkout stands and their attendant clerks. Other customers and store employees may be anywhere in the store. The agent begins at the entrance, and must leave the store from the same square. There is an EXIT sign there in case the agent forgets. There are also signs marking the aisles, and smaller signs (readable only when the camera zooms in) marking some (but not necessarily all) of the items for sale.

A real agent would have to decipher the video signals from the camera (or some digitization of them). We assume that this work has already been done. Still, the vision component of a percept is a complex list of descriptions. The first component of each description is its relative

position with respect to the agent's position and orientation. For example, the relative position [-2,1] is the square two squares to the agent's left and one square ahead. The second component is the size of the object, given as the average diameter of the object in meters. Next is the color of the object, given as a symbol (red, green, yellow, orange, ...), followed by the object's shape (flat, round, square, ...). Finally, we assume that an optical character recognition routine has run over the video image; if there are any letters in the visual field, they are given as a list of words. Figure 8.8 shows an overview of a supermarket, with the agent at [4,5] still dressed for the wumpus world. The agent is facing left. Figure 8.9(a) shows what the agent perceives with the camera zoomed out, and Figure 8.9(b) shows the agent's visual percept with the camera zoomed in at the square [3,6].

## Organizing knowledge

The grocery shopping domain is too big to handle all at once. Instead, we will break it down into smaller clusters of knowledge, work on each cluster separately, and then see how they fit together. One good way of decomposing the domain into clusters is to consider the tasks facing the agent. This is called a functional decomposition. We divide the domain into five clusters:

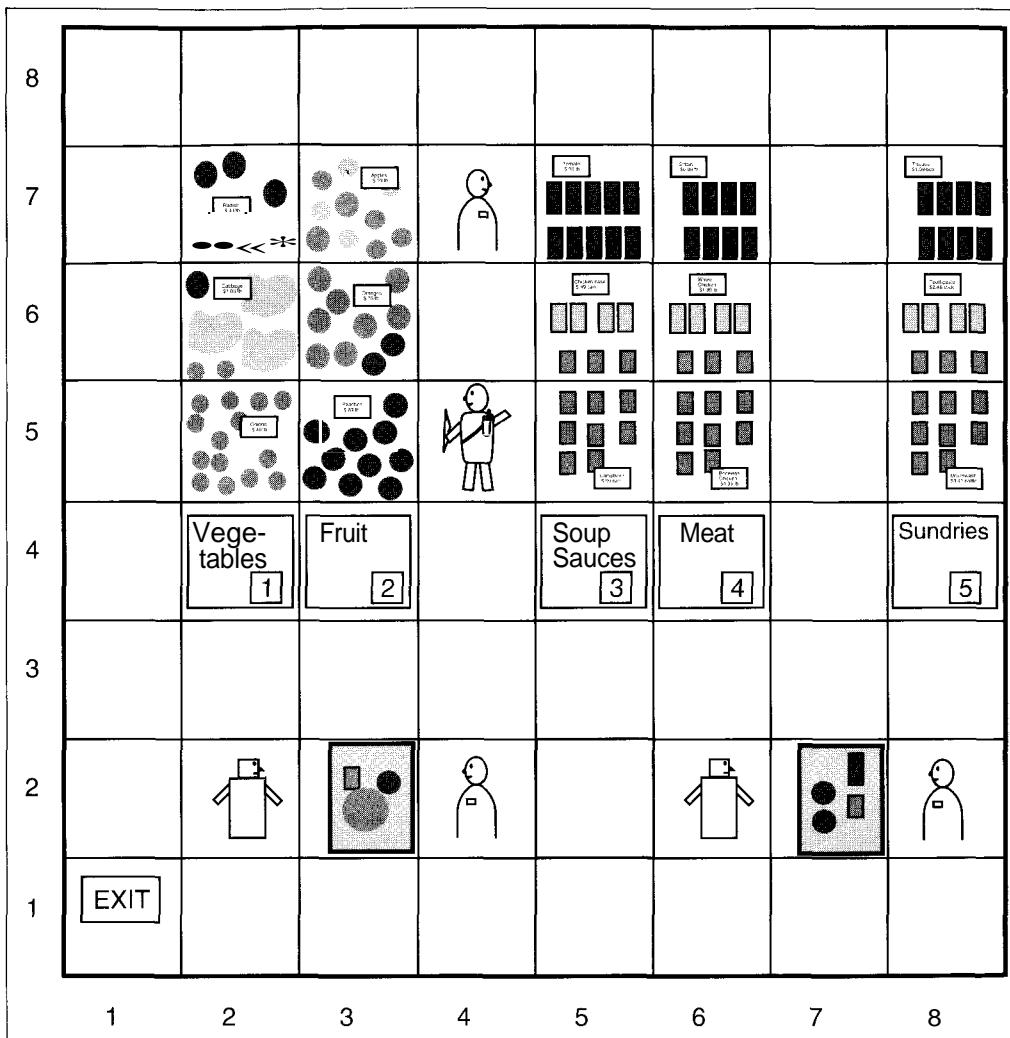
- ◊ **Menu Planning:** The agent will need to know how to modify the shopping list when the store is out of stock of an item.
- 0 **Navigating:** As in the wumpus world, the agent will need to understand the effect of movement actions and create an internal map of the world.
- 0 **Gathering:** The agent must be able to find and gather the items it wants. Part of this involves inducing objects from percepts: the agent will need recognition rules to infer that a red roughly spherical object about three inches in diameter could be a tomato.
- 0 **Communicating:** The agent should be able to ask questions when there is something it cannot find out on its own.
- 0 **Paying:** Even a shy agent that prefers not to ask questions will need enough interagent skills to be able to pay the checkout clerk. The agent will need to know that \$5.00 is too much for a single tomato, and that if the total price is \$17.35, then it should receive \$2.65 in change from a \$20 bill.

An advantage of functional decomposition is that we can pose a problem completely within a cluster and see if the knowledge can solve it. Other kinds of decomposition often require the whole knowledge base to be fleshed out before the first question can be posed.

### Menu-Planning

A good cook can walk into a market, pick out the best bargain from the various fish, fowl, or other main course ingredients that look fresh that day, select the perfect accompanying dishes, and simultaneously figure out how to make tomorrow's meal from whatever will be left over. A competent errand runner can take a shopping list and find all the items on it. Our agent will be closer to the second of these, but we will give it some ability to make intelligent choices.

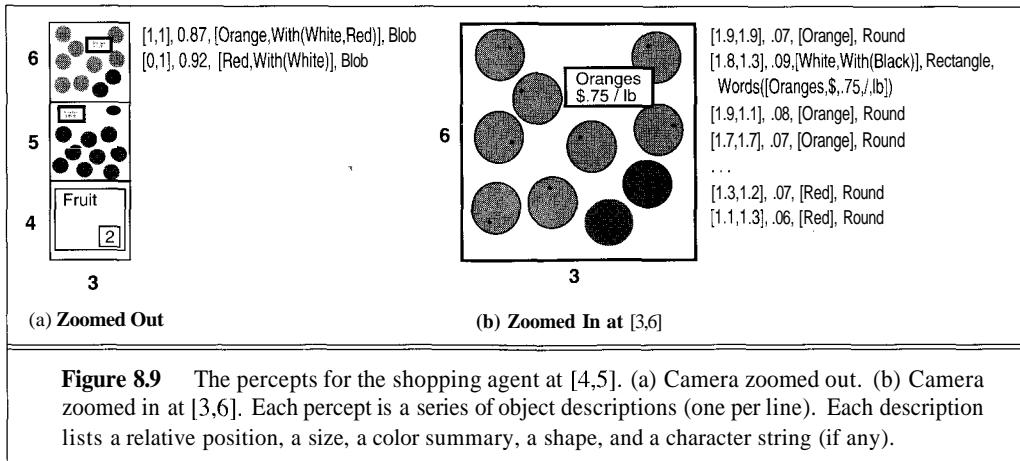
Suppose the store is out of tomatoes one day. An agent with the shopping list "tomatoes, lettuce, cucumber, olive oil, vinegar" should recognize that the ingredients form a salad, and that



**Figure 8.8** An overview of a supermarket. Note the agent at [4,5], the other shoppers at [2,2] and [6,2], the checkout clerks at [4,2] and [8,2], the signs in the fourth row, and the groceries spread throughout the world.

a red pepper would be a good substitute, as it would add color and flavor to the salad. An agent with the list "tomatoes, yellow onions, celery, a carrot, ground beef, milk, white wine, tagliatelle" should infer a Bolognese sauce (Hazan, 1973), and therefore that it is appropriate to substitute canned tomatoes.

To make these inferences, an agent needs to understand that the items on a shopping list fit together to form one or more composite objects known as **dishes**, that the dishes go together to



**Figure 8.9** The percepts for the shopping agent at [4,5]. (a) Camera zoomed out. (b) Camera zoomed in at [3,6]. Each percept is a series of object descriptions (one per line). Each description lists a relative position, a size, a color summary, a shape, and a character string (if any).

form composite objects known as **meals**, and that an object can be recognized by its components. Our agent will be called upon to make two classes of inference. First, from a list of parts it should induce the composite object that these parts make up. This is made difficult because the parts (the items on the shopping list) may make up several composite objects, and because not all the parts will be listed (some are already at home in the cupboard). Second, the agent should be able to decide how to replace an unavailable part to complete the intended composite object. This can be done at two levels: replacing one ingredient with another to complete a dish, and if that is not possible, replacing the whole dish with another to complete the meal. Some of the necessary knowledge will involve individual dishes, and some of it will be at a general level that will also be useful for, say, replacing a faulty muffler in a car.

The first step is to convert a shopping list—a list of words—into a parts list—a list of categories. A dictionary is used to associate words with their referents:

*Referent("tomatoes", Tomatoes)*

*Referent("onions", Onions)*

⋮

The next step is to describe objects in terms of their required and optional parts. If we wanted to actually prepare a dish, we would have to know more about the relations between the parts. But to do shopping, all we need is a list of the parts. We define *RequiredParts* so that *RequiredParts*{Lettuce,Dressing}, GreenSalads means that every object that is an element of *GreenSalads* has one *RequiredPart* that is an element of *Lettuce*, and another that is an element of *Dressing*. For lettuce to be a required part of green salads means that every element of green salads has an element of lettuce as one of its parts. Similar reasoning holds for *OptionalParts*, except that only some elements of a category have to manifest the optional parts.

$$\forall r, w \text{ } \textit{RequiredParts}(r, w) \Rightarrow \forall p \in r \Rightarrow \textit{RequiredPart}(p, w)$$

$$\forall o, w \text{ } \textit{OptionalParts}(o, w) \Rightarrow \forall p \in p \notin o \Rightarrow \textit{OptionalPart}(p, w)$$

$$\forall r, w \text{ } \textit{RequiredPart}(r, w) \wedge \forall c \in r \in w \Rightarrow \exists i \in r \text{ } \textit{A PartOf}(i, c)$$

$$\forall o, w \text{ } \textit{OptionalPart}(o, w) \wedge \forall c \in o \in w \Rightarrow \exists i \in o \text{ } \textit{A PartOf}(o, c)$$

The next step is to describe meals and dishes in terms of their parts:

```

RequiredParts(MainCourses, Meals)
OptionalParts(FirstCourses, SideDishes, Salads, Desserts, ..., Meals)
RequiredParts(Lettuce, Dressing), GreenSalads)
OptionalParts(Tomatoes, Cucumbers, Peppers, Carrots, ..., GreenSalads)
RequiredParts(Pasta, BologneseSauce), PastaBolognese)
OptionalParts(GratedCheese), PastaBolognese)
RequiredParts(Onions, OliveOil, Butter, Celery, Carrots, GroundBeef, Salt,
WhiteWines, Milk, TomatoStuff), BologneseSauce)

```

Then we need taxonomic information for dishes and foods:

```

GreenSalads C Salads
Salads C Dishes
PastaBolognese C FirstCourses
FirstCourses C Dishes
Tomatoes C TomatoStuff
CannedTomatoes C TomatoStuff
Tagliatelle C Pasta

```

Now we want to be able to determine what dishes can be made from the shopping list "tomatoes, yellow onions, celery, a carrot, ground beef, milk, white wine, tagliatelle." As mentioned earlier, this is complicated by the fact that the salt, butter, and olive oil that are required for the Bolognese dish are not on the shopping list. We define the predicate *CanMake* to hold between a shopping list and a dish if the categories on the list, when combined with typical staples, cover all the required parts of the dish.

3.3

$$\forall l, d \text{ } CanMake(l, d) \Leftrightarrow d \in Dishes \wedge \text{RequiredPart}(s(p, d)) \wedge p \in \text{Union}(l, Staples) \\ \{Salt, Butter, OliveOil\} \subset Staples$$

With what we have so far, this would allow us to infer that Pasta Bolognese is the only dish that can be made from the shopping list. The next question is what to do if fresh tomatoes are not available. It turns out that all we have to do is replace the shopping list (or the part of the shopping list that makes up this dish) with the list of part categories for this dish. In this case, that means that *Tomatoes* would be replaced by *TomatoStuff* which could be satisfied by gathering an instance of *CannedTomatoes*.

## Navigating

An agent that wants to find a book in a library could traverse the entire library, looking at each book until the desired one is found. But it would be more efficient to find the call number for the book, find a map describing where that number can be found, and go directly there. It is the same way with a supermarket, although the catalog system is not as good. A shopping agent should know that supermarkets are arranged into aisles, that aisles have signs describing their contents (in rough terms), and that objects that are near each other in the taxonomic hierarchy are likely to be near each other in physical space. For example, British immigrants to the United States learn that to find a package of tea, they should look for the aisle marked "Coffee."

Most of the navigation problem is the same as in the vacuum or wumpus world. The agent needs to remember where it started, and can compute its current location from the movements it has made where it is now. The supermarket is not as hostile as the wumpus world so it is safer to explore, but an agent can find better routes if it knows that supermarkets are generally laid out in aisles. Supermarkets also provide aisle numbers, unlike the wumpus world, so that the agent does not need to rely on dead reckoning. On the second trip to a store, it can save a lot of wandering by remembering where things are. It is not helpful, however, to remember the exact location of each individual item, because the tomato that is at location  $[x, y]$  today will probably be gone tomorrow. Because much of the logic is the same as for the wumpus world, it will not be repeated here.

A typical navigation problem is to locate the tomatoes. The following strategy is usually believed to work:

1. If the agent knows the location of the tomatoes from a previous visit, calculate a path to that spot from the current location.
2. Otherwise, if the location of the vegetable aisle is known, plan a path there.
3. Otherwise, move along the front of the store until a sign for the vegetable aisle is spotted.
4. If none of these work, wander about and find someone to ask where the tomatoes are. (This is covered in the "Communicating" section.)
5. Once the vegetable aisle is found, move down the aisle with the camera zoomed out, looking for something red. When spotted, zoom in to see if they are in fact tomatoes. (This is covered in the "Gathering" section.)

## Gathering

Once in the right aisle, the agent still needs to find the items on its list. This is done by matching the visual percepts against the expected percepts from each category of objects. In the wumpus world, this kind of perception is trivial—a breeze signals a pit and a stench signals a wumpus. But in the grocery shopping world, there are thousands of different kinds of objects and many of them (such as tomatoes and apples) present similar percepts. The agent never can be sure that it has classified an object correctly based on its percepts, but it can know when it has made a good guess. The shopping agent can use the following classification rules:

1. If only one known category matches a percept, assume the object is a member of that category. (This may lead to an error when an unknown object is sighted.)
2. If a percept matches several categories, but there is a sign nearby that identifies one of them, assume the object is a member of that category.
3. If there is an aisle sign that identifies one category (or one supercategory), assume the object is of that category. For example, we could categorize a round, red percept as a tomato rather than an apple if we were in an aisle marked "Vegetables" and not "Fruit." At a cricket match, it would be something else altogether.

To implement this, we start with a set of *causal* rules for percepts:

$$\begin{aligned}
 \forall x \ x \in Tomatoes &\Rightarrow SurfaceColor(x, Red) \\
 \forall x \ x \in Oranges &\Rightarrow SurfaceColor(x, Orange) \\
 \forall x \ x \in Apples &\Rightarrow SurfaceColor(x, Red) \vee SurfaceColor(x, Green) \\
 \forall x \ x \in Tomatoes &\Rightarrow Shape(x, Round) \\
 \forall x \ x \in Oranges &\Rightarrow Shape(x, Round) \\
 &\vdots \\
 \forall x \ SurfaceColor(x, c) \wedge Visible(x) &\Rightarrow CausesColorPercept(x, c) \\
 \forall x \ Shape(x, s) \wedge Visible(x) &\Rightarrow CausesShapePercept(x, s)
 \end{aligned}$$

DOMAIN CLOSURE

These rules, and many more like them, give a flavor of a causal theory of how percepts are formed by objects in the world. Notice how simplistic it is. For example, it does not mention lighting at all. (Fortunately, the lights are always on in our supermarket.) From these rules, the agent will be able to deduce a set of possible objects that might explain its percepts. Knowledge about what sorts of objects appear where will usually eliminate all but one category. Of course, the fact that the agent only *knows about* one sort of object that might produce a given percept does not mean that the percept must be produced by that sort of object. Logically, there might be other sorts of objects (e.g., plastic tomatoes) that produce the same percept as the known category. This can be handled either by a **domain closure** axiom, stating that the known categories are all the ones there are, or by a default assumption, as described in Chapter 15.

The other part of the gathering problem is manipulation: being able to pick up objects and carry them. In our simulation, we just assume a primitive action to grasp an object, and that the agent can carry all the items it will need. In the real world, the actions required to pick up a bunch of bananas without bruising them and a gallon jug of milk without dropping it pose serious problems. Chapter 25 considers them in more detail.

## Communicating

The successful shopper knows when to ask questions (Where are the anchovies? Are these tomatillas?). Unfortunately, being able to carry on a conversation is a difficult task, so we will delay covering it until Chapter 22. Instead, we will cover a simpler form of one-way communication: reading signs. If a word appears on an aisle's sign, then members of the category that the word refers to will be located in that aisle.

$$\begin{aligned}
 \forall a \ (a \in Aisles \wedge \exists s, w \ SignOf(sa) \wedge w \in G \ Words(s)) \Rightarrow \\
 \exists x, c \ Referent(w, c) \wedge x \in G \ c \wedge At(x, a)
 \end{aligned}$$

If a word appears on a small sign, then items of that category will be located nearby.

$$\begin{aligned}
 \forall s, w, l \ (s \in Signs \wedge Size(s) < Meters(3) \wedge w \in G \ Words(s) \wedge At(s, l)) \Rightarrow \\
 \exists x, c \ Referent(w, c) \wedge x \in G \ c \wedge At(x, AreaAround(l))
 \end{aligned}$$

## Paying

The shopping agent also has to know enough so that it will not overpay for an item. First, it needs to know typical fair prices for items, for example:

$$\forall g \ g \in \text{Typical}(\text{GroundBeef}) \ A \ \text{Weight}(g) = \text{Pounds}(1) \Rightarrow \$1 < \text{FairPrice}(g) < \$2$$

The agent should know that total price is roughly proportional to quantity, but that often discounts are given for buying larger sizes. The following rule says that this discount can be up to 50%:

$$\begin{aligned} \forall q, c, w, p \ q \in G \ c \in A \ \text{Weight}(q) = w \ A \ \text{Price}(q) = p \Rightarrow \\ \forall m, q_2 \ m > 1 \ A \ q_2 \in c \ A \ \text{Weight}(q_2) = m \times w \Rightarrow \\ (1 + \frac{m-1}{2}) \times p < \text{FairPrice}(q_2) < m \times p \end{aligned}$$

Most importantly, the agent should know that it is a bad deal to pay more than the fair price for an item, and that buying anything that is a bad deal is a bad action:

$$\begin{aligned} \forall i \ \text{Price}(i) > \text{FairPrice}(i) \Rightarrow \text{BadDeal}(i) \\ \forall i \ \text{BadDeal}(i) \Rightarrow \forall a \ \text{Bad}(\text{Buy}(a, i)) \end{aligned}$$

Buying events belong to the category  $\text{Buy}(b, x, s, p)$ —buyer  $b$  buying object  $x$  from seller  $s$  for price  $p$ . The complete description of buying is quite complex, but follows the general pattern laid down earlier in the chapter for marriage. The preconditions include the fact that  $p$  is the price of the object  $x$ ; that  $b$  has at least that much money in the form of one or more monetary instruments; and that  $s$  owns  $x$ . The event includes a monetary exchange that results in a net gain of  $p$  for  $s$ , and finally  $b$  owns  $x$ . Exercise 8.10 asks you to complete this description.

One final thing an agent needs to know about shopping: it is bad form to exit a shop while carrying something that the shop owns.

$$\begin{aligned} \forall a, x, s, i \ s \in \text{Shops} \ A \ T(\text{Carrying}(ax) \ A \ \text{At}(x, s) \ A \ \text{Owns}(s, x)) \Rightarrow \\ T(\text{Bad}(\text{Exit}(a)), i) \end{aligned}$$

An agent with the goal of exiting will use this goal to set up the subgoal of owning all the objects it is carrying. So all we need now is a description of the parts of a buying event, so that the agent can execute the buying. In a supermarket, a buying event consists of going to a checkout stand, placing all the items on the stand, waiting for the cashier to ring them up, placing a sum of money equal to the total price on the stand, and picking up the items again. Note that if the total is \$4, it will not do to place the same dollar bill onto the checkout stand four times.

$$\begin{aligned} \forall b, m, s, p, e \ e \in \text{SupermarketBuy}(b, m, s, p) \Rightarrow \\ \exists e_1, e_2, e_3, e_4, e_5 \ e = \text{Go}(b, c) \ A \ \text{CheckoutStand}(c) \ A \\ e_2 = \text{Put}(b, m, c) \ A \ e_3 = \text{TotalUpPrice}(sm) \ A \\ e_4 = \text{Put}(b, p, c) \ A \ e_5 = \text{Grab}(b, m) \ A \\ \text{Before}(e_1, e_2) \ A \ \text{Before}(e_2, e_3) \ A \ \text{Before}(e_3, e_4) \ A \ \text{Before}(e_4, e_5) \ A \\ \text{PartOf}(e_1, e) \ A \ \text{PartOf}(e_2, e) \ A \ \text{PartOf}(e_3, e) \ A \ \text{PartOf}(e_4, e) \ A \ \text{PartOf}(e_5, e) \end{aligned}$$

Now we have touched on all the major areas of knowledge necessary for an agent to cope with the grocery shopping world. A complete specification would make this chapter too long, but we have outlined the approach one would take to complete this specification. Although it is hard work, building actual knowledge bases of this kind is an invaluable experience.

## 8.6 SUMMARY

---

This has been the most detailed chapter of the book so far. As we said earlier in the chapter, one *cannot* understand knowledge representation without doing it, or at least seeing it. The following are some of the major points of the chapter:

- The process of representing knowledge of a domain goes through several stages. The first, informal stage involves deciding what kinds of objects and relations need to be represented (the ontology). Then a vocabulary is selected, and used to encode general knowledge of the domain. After encoding specific problem instances, automated inference procedures can be used to solve them.
- Good representations eliminate irrelevant detail, capture relevant distinctions, and express knowledge at the most general level possible.
- Constructing knowledge-based systems has advantages over programming: the knowledge engineer has to concentrate only on what's true about the domain, rather than on solving the problems and encoding the solution process; the same knowledge can often be used in several ways; debugging knowledge is often simpler than debugging programs.
- Special-purpose ontologies, such as the one constructed for the circuits domain, can be effective within the domain but often need to be generalized to broaden their coverage.
- A general-purpose ontology needs to cover a wide variety of knowledge, and should be capable in principle of handling any domain.
- We presented a general ontology based around categories and the event calculus. We covered structured objects, time and space, change, processes, substances, and beliefs.
- We presented a detailed analysis of the shopping domain, exercising the general ontology and showing how the domain knowledge can be used by a shopping agent.

Finally, it is worth recalling that the nature of an appropriate representation depends on the world being represented and the intended range of uses of the representation. The representation choices in this chapter are specific to the world of human experience, but this is unavoidable.

---

## BIBLIOGRAPHICAL AND HISTORICAL NOTES

There are plausible claims (Briggs, 1985) that formal knowledge representation research began with classical Indian theorizing about the grammar of Shastric Sanskrit, which dates back to the first millennium B.C. Shastric Sanskrit grammatical theory proposed not only a formal syntax and vocabulary for a general-purpose language, but also provided an analysis of its semantics. Shastric Sanskrit grammatical theory therefore can be regarded as the earliest instance of systematic representation of knowledge in a specific area in order to facilitate inference. In the West, the use of definitions of terms in ancient Greek mathematics can be regarded as the earliest instance. Indeed, the development of technical terminology or artificial languages in any field can be regarded as a form of knowledge representation research. The connection between knowledge

representation in this sense, and knowledge representation in AI, is closer than it may seem; twentieth century AI research draws widely upon the formalisms of other fields, especially logic and philosophy. Aristotle (384-322 B.C.) developed a comprehensive system of what we would now call ontology and knowledge representation in connection with his work in logic, natural science, and philosophical metaphysics.

Besides the logicist tradition started by McCarthy, which we discussed in Chapter 6, there have been many other threads in the history of representation in AI. Early discussions in the field tended to focus on “*problem*representation” rather than “*knowledge*representation.” The emphasis was on formulating the problem to be solved, rather than formulating the resources available to the program. A conscious focus on knowledge representation had to await the discovery that high performance in AI problem solving required the accumulation and use of large amounts of problem-specific knowledge. The realization that AI systems needed such knowledge was largely driven by two types of research. The first was the attempt to match human performance in the everyday world, particularly in understanding natural human languages and in rapid, content-based retrieval from a general-purpose memory. The second was the design of “expert systems”—also, significantly, called “knowledge-based systems”—that could match (or, in some cases, exceed) the performance of human experts on narrowly defined tasks. The need for problem-specific knowledge was stressed forcefully by the designers of DENDRAL, the first expert system, which interpreted the output of a mass spectrometer, a type of instrument used for analysis of the structure of organic chemical compounds. An early statement of the DENDRAL philosophical perspective can be found in Feigenbaum, Buchanan, and Lederberg (1971); Lindsay *et al.* (1980) provide a book-length description of the DENDRAL project, along with a complete bibliography from 1964 to 1979. Although the success of DENDRAL was instrumental in bringing the AI research community as a whole to realize the importance of knowledge representation, the representational formalisms used in DENDRAL are highly specific to the domain of chemistry. As expert systems continued to succeed and proliferate, expert system researchers became interested in standardized knowledge representation formalisms and ontologies that could reduce the difficulty of creating a new expert system in yet another previously unexplored field. In so doing, they ventured into territory previously explored by philosophers of science and of language. The discipline imposed in AI by the need for one’s theories to “work” has led to more rapid and deeper progress than was the case when these problems were the exclusive domain of philosophy (although it has at times also led to the repeated reinvention of the wheel).

Research in memory and natural language processing (NLP) also had to deal with the need for general-purpose knowledge representation languages from the very start. Indeed, Ross Quillian’s (1961) work on “semantic networks” predates DENDRAL. Because of the need to get heterogeneous bodies of knowledge to interact fruitfully, memory and NLP research was the original spark for semantic networks, frames, and other very general formalisms. Such “knowledge representation languages” are covered in greater detail in Chapter 10. The present chapter focuses instead on the content of the knowledge itself and on the representational concepts that are common to a number of distinct formalisms.

The creation of comprehensive taxonomies or classifications dates back to ancient times. Aristotle strongly emphasized classification and categorization schemes. His *Organon*, a collection of works on logic assembled by his students after his death, included a treatise called *Categories* in which he attempted a comprehensive high-level classification and also introduced

the use of genus and species for lower-level classification, although these terms did not have the precise and specifically biological sense which is now attached to them. Our present system of biological classification, including the use of "binomial nomenclature" (classification via genus and species in the technical sense), was invented by the Swedish biologist Carolus Linnaeus, or Carl von Linne (1707–1778). Lakoff (1987) presents a model of classification based on prototypes rather than strict categorical boundaries.

Within modern AI specifically, comprehensive taxonomies have usually been developed as part of large projects that also included research in other areas of knowledge representation. These include the "commonsense summer" project led by Jerry Hobbs (1985) and the knowledge representation portion of the ensuing TACITUS natural language interpretation project (Hobbs, 1986; Hobbs *et al.*, 1990), as well as the massive CYC project (Lenat and Guha, 1990). The taxonomy used in this chapter was developed by the authors, based in part on their experience in the CYC project and in part on work by Hwang and Schubert (1993) and Davis (1990). An inspirational discussion of the general project of commonsense knowledge representation appears in Hayes's (1978; 1985b) "The Naive Physics Manifesto."

The philosophical study of the part-whole relation was initiated by the Polish logician i Lesniewski (1916), who was a hardcore "nominalist" or skeptic about abstract entities such as sets and numbers. He intended his "mereology" (the name is derived from the Greek word for "part") as a substitute for mathematical set theory. Although mereology showed promise as a way of analyzing the distinction between mass nouns and count nouns, Leśniewski's publications are extremely difficult to follow, because they are written in a very idiosyncratic formal notation with (in some cases) almost no natural-language commentary. A more readable exposition and i axiomatization of mereology was provided in 1940 by the philosophers Nelson Goodman (another hardcore nominalist) and Henry Leonard under the name of "the calculus of individuals" (Leonard and Goodman, 1940). Goodman's *The Structure of Appearance* (1977) applies the calculus of individuals to what in AI would be called knowledge representation. Quine (1960) also supports the nominalist view of substances. Harry Bunt (1985) has provided an extensive analysis of its j use in knowledge representation.

Few if any AI researchers have any problems with abstract entities; the pragmatic "Ontological Promiscuity" endorsed by Hobbs (1985) in the article of that title is more typical. The position adopted in this chapter, in which substances are categories of objects, was championed by Richard Montague (1973). It has also been adopted in the CYC project. Copeland (1993) mounts a serious but not invincible attack.

Several different approaches have been taken in the study of time and events. The oldest approach is **temporal logic**, which is a form of modal logic in which modal operators are used specifically to refer to the times at which facts are true. Typically, in temporal logic, " $\Box p$ " means " $p$  will be true at all times in the future," and " $\Diamond p$ " means " $p$  will be true at some time in the future." The study of temporal logic was initiated by Aristotle and the Megarian and Stoic schools in ancient Greece.

In modern times, Findlay (1941) was the first to conceive the idea of a "formal calculus" for reasoning about time; Findlay also sketched a few proposed laws of temporal logic. The further development of modern temporal logic was carried out by a number of researchers, including Arthur Prior (1967). The modern development was actually strongly influenced by historical studies of Megarian and Stoic temporal logic. Burstall (1974) introduced the idea of using

modal logic to reason about computer programs. Soon thereafter, Vaughan Pratt (1976) designed **dynamic logic**, in which modal operators indicate the effects of programs or other actions. For instance, in dynamic logic, if  $a$  is the name of a program, then " $[\alpha]p$ " might mean " $p$  would be true in all world states resulting from executing program  $a$  in the current world state", and " $(a\rangle p$ " might mean " $p$  would be true in at least one world state resulting from executing program  $a$  in the current world state." Dynamic logic was applied to the actual analysis of programs by Fischer and Ladner (1977). Pnueli (1977) introduced the idea of using classical temporal logic to reason about programs. Shoham (1988) discusses the use of temporal logic in AI.

Despite the long history of temporal logic, the considerable mathematical theory built up around it, and its extensive use in other branches of computer science, AI research on temporal reasoning has more often taken a different approach. A temporal logic is usually conceptualized around an underlying model involving events, world states, or temporal intervals. The tendency in AI has been to refer to these events, states, and intervals directly, using terms that denote them, rather than indirectly through the interpretation of the sentence operators of temporal logic. The language used is typically either first-order logic or, in some cases, a restricted algebraic formalism geared toward efficient computation (but still capable of being embedded within first-order logic). This approach may allow for greater clarity and flexibility in some cases. Also, temporal knowledge expressed in first-order logic can be more easily integrated with other knowledge that has been accumulated in that notation.

One of the earliest formalisms of this kind was McCarthy's situation calculus, mentioned in Chapter 7. McCarthy (1963) introduced situational fluents and made extensive use of them in later papers. Recent work by Raymond Reiter (1991) and others in the "cognitive robotics" project at the University of Toronto (Scherl and Levesque, 1993) has re-emphasized the use of situation calculus for knowledge representation. The relationship between temporal logic and situation calculus was analyzed by McCarthy and Hayes (1969). They also brought to the attention of AI researchers the work of philosophers such as Donald Davidson on events. Davidson's research, collected in (Davidson, 1980), had a heavy emphasis on the analysis of natural language, particularly of the adverb. It has strongly influenced later AI research, particularly in natural language understanding. Other philosophical and linguistic approaches to events that are of significance for AI research are those of Zeno Vendler (1967; 1968), Alexander Mourelatos (1978), and Emmon Bach (1986).

James Alien's introduction of time intervals, and a small, fixed set of relationships between them, as the primitives for reasoning about time (Alien, 1983; Alien, 1984) marked a major advance over situation calculus and other systems based on time points or instantaneous events. Preliminary versions of Alien's work were available as technical reports as early as 1981. Peter Ladkin (1986a; 1986b) introduced "concave" time intervals (intervals with gaps; essentially, unions of ordinary "convex" time intervals) and applied the techniques of mathematical abstract algebra to time representation. Alien (1991) systematically investigates the wide variety of techniques currently available for time representation. Shoham (1987) describes the reification of events and sets forth a novel scheme of his own for the purpose. The term "event calculus" is also used by Kowalski and Sergot (1986), who show how to reason about events in a logic programming system.

The syntactic theory of mental objects was first studied in depth by Kaplan and Montague (1960), who showed that it led to paradoxes if not handled carefully. Because it has a

natural model in terms of beliefs as physical configurations of a computer or a brain, it has been popular in AI in recent years. Konolige (1982) and Haas (1986) used it to describe inference engines of limited power, and Morgenstern (1987) showed how it could be used to describe knowledge preconditions in planning. The methods for planning observation actions in Chapter 13 are based on the syntactic theory.

## MODAL LOGIC

## POSSIBLE WORLDS

**Modal logic** is the classical method for reasoning about knowledge in philosophy. Modal logic augments first-order logic with modal operators, such as *B* (believes) and *K* (knows), that take *sentences* as arguments rather than terms. The proof theory for modal logic restricts substitution within modal contexts, thereby achieving referential opacity. The modal logic of knowledge was invented by Jaakko Hintikka (1962). Saul Kripke (1963) defined the semantics of the modal logic of knowledge in terms of **possible worlds**. Roughly speaking, a world is possible for an agent if it is consistent with everything the agent knows. From this, one can derive rules of inference involving the *K* operator. Robert C. Moore relates the modal logic of knowledge to a style of reasoning about knowledge which refers directly to possible worlds in first-order logic (Moore, 1980; Moore, 1985a). Modal logic can be an intimidatingly arcane field, but has also found significant applications in reasoning about information in distributed computer systems (Halpern, 1987). For an excellent comparison of the syntactic and modal theories of knowledge, see (Davis, 1990).

For obvious reasons, this chapter does not cover *every* area of knowledge representation in depth. The three principal topics omitted are the following:

## QUALITATIVE PHYSICS

- ◊ **Qualitative physics:** A subfield of knowledge representation concerned specifically with constructing a logical, nonnumeric theory of physical objects and processes. The term was coined by Johan de Kleer (1975), although the enterprise could be said to have started in Fahlman's (1974) BUILD. BUILD was a sophisticated planner for constructing complex towers of blocks. Fahlman discovered in the process of designing it that most of the effort (80%, by his estimate) went into modelling the physics of the blocks world to determine the stability of various subassemblies of blocks, rather than into planning per se. He sketches a hypothetical naive-physics like process to explain why young children can solve BUILD-like problems without access to the high-speed floating-point arithmetic used in BUILD'S physical modelling. Hayes (1985a) uses "histories," four-dimensional slices of space-time similar to Davidson's events, to construct a fairly complex naive physics of liquids. Hayes was the first to prove that a bath with the plug in will eventually overflow if the tap keeps running; and that a person who falls into a lake will get wet all over. De Kleer and Brown (1985) and Ken Forbus (1985) attempted to construct something like a general-purpose theory of the physical world, based on qualitative abstractions of physical equations. In recent years, qualitative physics has developed to the point where it is possible to analyze an impressive variety of complex physical systems (Sacks and Joskowicz, 1993; Yip, 1991). Qualitative techniques have been also used to construct novel designs for clocks, windscreen wipers, and six-legged walkers (Subramanian, 1993; Subramanian and Wang, 1994). The collection *Readings in Qualitative Reasoning about Physical Systems* (Weld and de Kleer, 1990) provides a good introduction to the field.
- ◊ **Spatial reasoning:** The reasoning necessary to navigate in the wumpus world and supermarket world is trivial in comparison to the rich spatial structure of the real world. The

## SPATIAL REASONING

most complete attempt to capture commonsense reasoning about space appears in the work of Ernest Davis (1986; 1990). As with qualitative physics, it appears that an agent can go a long way, so to speak, without resorting to a full metric representation. When such a representation is necessary, techniques developed in robotics (Chapter 25) can be used.

- 0 Psychological reasoning:** The development of a working *psychology* for artificial agents to use in reasoning about themselves and other agents. This is often based on so-called "folk psychology," the theory that humans in general are believed to use in reasoning about themselves and other humans. When AI researchers provide their artificial agents with psychological theories for reasoning about other agents, the theories are frequently based on the researchers' description of the logical agents' own design. Also, this type of psychological theorizing frequently takes place within the context of natural language understanding, where divining the speaker's intentions is of paramount importance. For this reason, the historical and bibliographical background for this topic has been relegated to other chapters, especially Chapter 22.

The proceedings of the international conferences on *Principles of Knowledge Representation and Reasoning* provide the most up-to-date sources for work in this area. *Readings in Knowledge Representation* (Brachman and Levesque, 1985) and *Formal Theories of the Commonsense World* (Hobbs and Moore, 1985) are excellent anthologies on knowledge representation; the former focuses more on historically important papers in representation languages and formalisms, the latter on the accumulation of the knowledge itself. *Representations of Commonsense Knowledge* (Davis, 1990) is a good recent textbook devoted specifically to knowledge representation rather than AI in general. Hughes and Cresswell (1968; 1984) and Chellas (1980) are introductory texts on modal logic; *A Manual of Intensional Logic* (van Benthem, 1985) provides a useful survey of the field. The proceedings of the annual conference *Theoretical Aspects of Reasoning About Knowledge* (TARK) contain many interesting papers from AI, distributed systems, and game theory. Rescher and Urquhart (1971) and van Benthem (1983) cover temporal logic specifically. David Harel (1984) provides an introduction to dynamic logic.

---

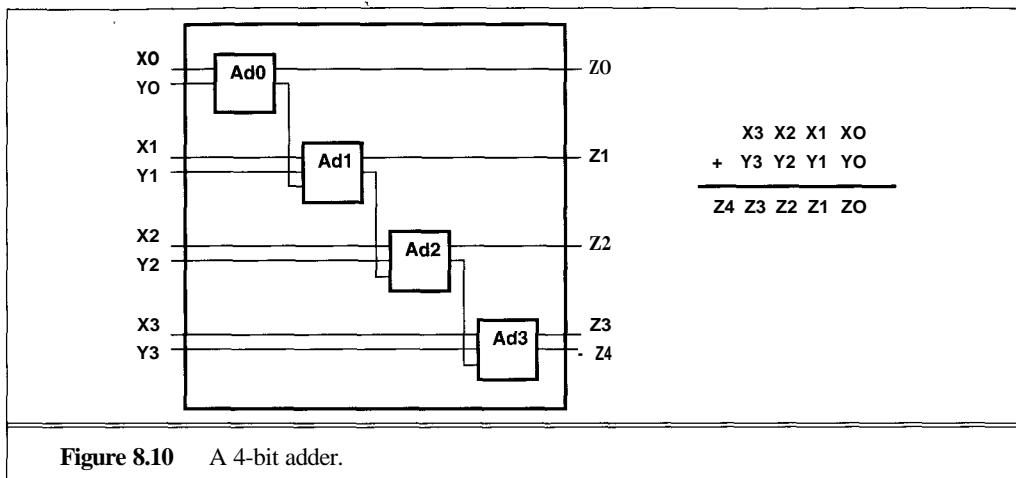
## EXERCISES

- 8.1 Extend the vocabulary from Section 8.3 to define addition and an adder circuit.
- 8.2 Represent the following six sentences using the representations developed in the chapter.
  - a. Water is a liquid between 0 and 100 degrees.
  - b. Water boils at 100 degrees.
  - c. The water in John's water bottle is frozen.
  - d. Perrier is a kind of water.
  - e. John has Perrier in his water bottle.
  - f. All liquids have a freezing point. (Don't use HasFreezingPoint!)

Now repeat the exercise using a representation based on the mereological approach, in which, for example, *Water* is an object containing as parts all the water in the world.



8.3 Encode the description of the 4-bit adder in Figure 8.10 and pose queries to verify that it is in fact correct.



**Figure 8.10** A 4-bit adder.

8.4 Write definitions for the following:

- ExhaustivePartDecomposition*
- PartPartition*
- PartwiseDisjoint*

These should be analogous to those for *ExhaustiveDecomposition*, *Partition*, and *Disjoint*.

8.5 Write a set of sentences that allows one to calculate the price of an individual tomato (or other object), given the price per pound. Extend the theory to allow the price of a bag of tomatoes to be calculated.

8.6 This exercise concerns the relationships between event categories and the time intervals in which they occur.

- Define the predicate  $T(c, i)$  in terms of *SubEvent* and  $\in$ .
- Explain precisely why we do not need two different notations ( $A$  and  $A$ ) to describe conjunctive event categories.
- Give a formal definition for  $T(p \vee q, i)$ .
- Give formal definitions for  $T(\neg p, i)$  and  $T(\neg p, j)$ , analogous to those using  $V$  and  $V$ .

8.7 Define the predicate *Fixed*, where  $\text{Fixed}(\text{Location}(x))$  means that the location of object  $x$  is fixed over time.

8.8 Define the predicates *Before*, *After*, *During*, and *Overlap* using the predicate *Meet* and the functions *Start* and *End*, but not the function *Time* or the predicate  $<$ .

8.9 Construct a representation for exchange rates between currencies that allows fluctuations on a daily basis.

8.10 In this chapter, we sketched some of the properties of buying events. Provide a formal logical description of buying using event calculus.

8.11 Describe the event of trading something for something else. Describe buying as a kind of trading where one of the objects is a sum of money.



8.12 The exercises on buying and trading used a fairly primitive notion of ownership. For example, the buyer starts by *owning* the dollar bills. This picture begins to break down when, for example, one's money is in the bank, because there is no longer any specific collection of dollar bills that one owns. The picture is complicated still further by borrowing, leasing, renting, and bailment. Investigate the various commonsense and legal concepts of ownership, and propose a scheme by which they can be represented formally.

8.13 You are to create a system for advising computer science undergraduates on what courses to take over an extended period in order to satisfy the program requirements. (Use whatever requirements are appropriate for your institution.) First, decide on a vocabulary for representing all the information, then represent it; then use an appropriate query to the system, that will return a legal program of study as a solution. You should allow for some tailoring to individual students, in that your system should ask the student what courses or equivalents he has already taken, and not generate programs that repeat those courses.

Suggest ways in which your system could be improved, for example to take into account knowledge about student preferences, workload, good and bad instructors, and so on. For each kind of knowledge, explain how it could be expressed logically. Could your system easily incorporate this information to find the *best* program of study for a student?

8.14 Figure 8.2 shows the top levels of a hierarchy for everything. Extend it to include as many real categories as possible. A good way to do this is to cover all the things in your everyday life. This includes objects and events. Start with waking up, proceed in an orderly fashion noting everything that you see, touch, do, and think about. For example, a random sampling produces music, news, milk, walking, driving, gas, Soda Hall, carpet, talking, Professor Fateman, chicken curry, tongue, \$4, sun, the daily newspaper, and so on.

You should produce both a single hierarchy chart (large sheet of paper) and a listing of objects and categories with one or more relations satisfied by members of each category. Every object should be in a category, and every category should be incorporated in the hierarchy.

8.15 (Adapted from an example by Doug Lenat.) Your mission is to capture, in logical form, enough knowledge to answer a series of questions about the following simple sentence:

Yesterday John went to the North Berkeley Safeway and bought two pounds of tomatoes and a pound of ground beef.

Start by trying to represent its content as a series of assertions. You should write sentences that have straightforward logical structure (e.g., statements that objects have certain properties; that objects are related in certain ways; that all objects satisfying one property satisfy another). The following may help you get started:

- Which classes, individuals, relations, and so on, would you need? What are their parents, siblings and so on? (You will need events and temporal ordering, among other things.)
- Where would they fit in a more general hierarchy?
- What are the constraints and interrelationships among them?
- How detailed must you be about each of the various concepts?

The knowledge base you construct must be capable of answering a list of questions that we will give shortly. Some of the questions deal with the material stated explicitly in the story, but most of them require one to know other background knowledge—to read between the lines. You'll have to deal with what kind of things are at a supermarket, what is involved with purchasing the things one selects, what will purchases be used for, and so on. Try to make your representation as general as possible. To give a trivial example: don't say "People buy food from Safeway," because that won't help you with those who shop at another supermarket. Don't say "Joe made spaghetti with the tomatoes and ground beef," because that won't help you with anything else at all. Also, don't turn the questions into answers; for example, question (c) asks "Did John buy any meat?"—not "Did John buy a pound of ground beef?"

Sketch the chains of reasoning that would answer the questions. In the process of doing so, you will no doubt need to create additional concepts, make additional assertions, and so on. If possible, use a logical reasoning system to demonstrate the sufficiency of your knowledge base. Many of the things you write may only be approximately correct in reality, but don't worry too much; the idea is to extract the common sense that lets you answer these questions at all.

- a. Is John a child or an adult? [Adult]
- b. Does John now have at least 2 tomatoes? [Yes]
- c. Did John buy any meat? [Yes]
- d. If Mary was buying tomatoes at the same time as John, did he see her? [Yes]
- e. Are the tomatoes made in the supermarket? [No]
- f. What is John going to do with the tomatoes? [Eat them]
- g. Does Safeway sell deodorant? [Yes]
- h. Did John bring any money to the supermarket? [Yes]
- i. Does John have less money after going to the supermarket? [Yes]

**8.16** Make the necessary additions/changes to your knowledge base from the previous exercise so that the following questions can be answered. Show that they can indeed be answered by the KB, and include in your report a discussion of the fixes, explaining why they were needed, whether they were minor or major, and so on.

- a. Are there other people in Safeway while John is there? [Yes—staff!]
- b. Did Mary see John? [Yes]
- c. Is John a vegetarian? [No]
- d. Who owns the deodorant in Safeway? [Safeway Corporation]
- e. Did John have an ounce of ground beef? [Yes]
- f. Does the Shell station next door have any gas? [Yes]
- g. Do the tomatoes fit in John's car trunk? [Yes]

# 9

# INFERENCE IN FIRST-ORDER LOGIC

*In which we define inference mechanisms that can efficiently answer questions posed in first-order logic.*

Chapter 6 defined the notion of **inference**, and showed how sound and complete inference can be achieved for propositional logic. In this chapter, we extend these results to first-order logic. In Section 9.1 we provide some additional basic inference rules to deal with quantifiers. In Section 9.2, we show how these inference rules, along with those for propositional logic, can be chained together to make proofs. By examining these proofs, we can come up with more powerful inference rules that make proofs much shorter and more intuitive. This makes it possible to design inference procedures that are sufficiently powerful to be of use to a knowledge-based agent. These rules and procedures are discussed in Sections 9.3 and 9.4. Section 9.5 describes the problem of completeness for first-order inference, and discusses the remarkable result, obtained by Kurt Gödel, that if we extend first-order logic with additional constructs to handle mathematical induction, then there is no complete inference procedure; even though the needle is in the metaphorical haystack, no procedure can guarantee to find it. Section 9.6 describes an inference procedure called **resolution** that is complete for any set of sentences in first-order logic, and Section 9.7 proves that it is complete.

## 9.1 INFERENCE RULES INVOLVING QUANTIFIERS

In Section 6.4, we saw the inference rules for propositional logic: Modus Ponens, And-Elimination, And-Introduction, Or-Introduction, and Resolution. These rules hold for first-order logic as well. But we will need additional inference rules to handle first-order logic sentences with quantifiers. The three additional rules we introduce here are more complex than previous ones, because we have to talk about substituting particular individuals for the variables. We will use the notation  $\text{SUBST}(\theta, a)$  to denote the result of applying the substitution (or binding list)  $\theta$  to the sentence  $a$ . For example:

$$\text{SUBST}(\{x/\text{Sam}, y/\text{Pam}\}, \text{Likes}(x, y)) = \text{Likes}(\text{Sam}, \text{Pam})$$

UNIVERSAL  
ELIMINATION

The three new inference rules are as follows:

**O Universal Elimination:** For any sentence  $a$ , variable  $v$ , and ground term<sup>1</sup>  $g$ :

$$\frac{V v \ a}{\text{SUBST}(\{v/g\}, \alpha)}$$

For example, from  $\forall x \ Likes(x, \text{IceCream})$ , we can use the substitution  $\{x/Ben\}$  and infer  $Likes(Ben, \text{IceCream})$ .

EXISTENTIAL  
ELIMINATION

**O Existential Elimination:** For any sentence  $\alpha$ , variable  $v$ , and constant symbol  $k$  that does not appear elsewhere in the knowledge base:

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

For example, from  $\exists x \ Kill(x, \text{Victim})$ , we can infer  $Kill(\text{Murderer}, \text{Victim})$ , as long as *Murderer* does not appear elsewhere in the knowledge base.

EXISTENTIAL  
INTRODUCTION

**◊ Existential Introduction:** For any sentence  $\alpha$ , variable  $v$  that does not occur in  $a$ , and ground term  $g$  that does occur in  $\alpha$ :

$$\frac{\alpha}{\exists v \ \text{SUBST}(\{g/v\}, \alpha)}$$

For example, from  $Likes(\text{Jerry}, \text{IceCream})$  we can infer  $\exists x \ Likes(x, \text{IceCream})$ .

You can check these rules using the definition of a universal sentence as the conjunction of all its possible instantiations (so Universal Elimination is like And-Elimination) and the definition of an existential sentence as the disjunction of all its possible instantiations.

It is very important that the constant used to replace the variable in the Existential Elimination rule is new. If we disregard this requirement, it is easy to produce consequences that do not follow logically. For example, suppose we have the sentence  $\exists x \ Father(x, \text{John})$  ("John has a father"). If we replace the variable  $x$  by the constant *John*, we get *Father(John, John)*, which is certainly not a logical consequence of the original sentence. Basically, the existential sentence says there is some object satisfying a condition, and the elimination process is just giving a name to that object. Naturally, that name must not already belong to another object. Mathematics provides a nice example: suppose we discover that there is a number that is a little bigger than 2.71828 and that satisfies the equation  $d(x^y)/dy = x$  for  $x$ . We can give this number a name, such as  $e$ , but it would be a mistake to give it the name of an existing object, like  $\pi$ .

## 9.2 AN EXAMPLE PROOF

Having defined some inference rules, we now illustrate how to use them to do a proof. From here, it is only a short step to an actual proof procedure, because the application of inference rules is simply a question of matching their premise patterns to the sentences in the KB and then adding

<sup>1</sup> Recall from Chapter 7 that a ground term is a term that contains no variables—that is, either a constant symbol or a function symbol applied to some ground terms.

their (suitably instantiated) conclusion patterns. We will begin with the situation as it might be described in English:

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

What we wish to prove is that West is a criminal. We first represent these facts in first-order logic, and then show the proof as a sequence of applications of the inference rules.<sup>2</sup>

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

$$\begin{aligned} \forall x, y, z \ A\ American(x) \wedge A\ Weapon(y) \wedge A\ Nation(z) \wedge A\ Hostile(z) \\ A\ Sells(x, z, y) \Rightarrow A\ Criminal(x) \end{aligned} \quad (9.1)$$

"Nono ... has some missiles":

$$\exists x \ A\ Owns(Nono, x) \wedge A\ Missile(x) \quad (9.2)$$

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

$$\forall x \ A\ Owns(Nono, x) \wedge A\ Missile(x) \Rightarrow A\ Sells(West, Nono, x) \quad (9.3)$$

We will also need to know that missiles are weapons:

$$\forall x \ A\ Missile(x) \Rightarrow A\ Weapon(x) \quad (9.4)$$

and that an enemy of America counts as "hostile":

$$\forall x \ A\ Enemy(x, America) \Rightarrow A\ Hostile(x) \quad (9.5)$$

"West, who is American ...":

$$A\ American(West) \quad (9.6)$$

"The country Nono ...":

$$A\ Nation(Nono) \quad (9.7)$$

"Nono, an enemy of America ...":

$$A\ Enemy(Nono, America) \quad (9.8)$$

$$A\ Nation(America) \quad (9.9)$$

The proof consists of a series of applications of the inference rules:

From (9.2) and Existential Elimination:

$$A\ Owns(Nono, M1) \wedge A\ Missile(M1) \quad (9.10)$$

From (9.10) and And-Elimination:

$$A\ Owns(Nono, M1) \quad (9.11)$$

$$A\ Missile(M1) \quad (9.12)$$

From (9.4) and Universal Elimination:

$$A\ Missile(M1) \Rightarrow A\ Weapon(M1) \quad (9.13)$$

<sup>2</sup> Our representation of the facts will not be ideal according to the standards of Chapter 8, because this is mainly an exercise in proof rather than knowledge representation.

From (9.12), (9.13), and Modus Ponens:

$$\text{Weapon}(M1) \quad (9.14)$$

From (9.3) and Universal Elimination:

$$\text{Owns}(\text{Nono}, M1) \wedge \text{Missile}(M1) \Rightarrow \text{Sells}(\text{West}, \text{Nono}, M1) \quad (9.15)$$

From (9.15), (9.10), and Modus Ponens:

$$\text{Sells}(\text{West}, \text{Nono}, M1) \quad (9.16)$$

From (9.1) and Universal Elimination (three times):

$$\begin{aligned} \text{American}(\text{West}) \wedge \text{Weapon}(M1) \wedge \text{Nation}(\text{Nono}) \wedge \text{Hostile}(\text{Nono}) \\ \wedge \text{Sells}(\text{West}, \text{Nono}, M1) \Rightarrow \text{Criminal}(\text{West}) \end{aligned} \quad (9.17)$$

From (9.5) and Universal Elimination:

$$\text{Enemy}(\text{Nono}, \text{America}) \Rightarrow \text{Hostile}(\text{Nono}) \quad (9.18)$$

From (9.8), (9.18), and Modus Ponens:

$$\text{Hostile}(\text{Nono}) \quad (9.19)$$

From (9.6), (9.7), (9.14), (9.16), (9.19), and And-Introduction:

$$\begin{aligned} \text{American}(\text{West}) \wedge \text{Weapon}(M1) \wedge \text{Nation}(\text{Nono}) \\ \wedge \text{Hostile}(\text{Nono}) \wedge \text{Sells}(\text{West}, \text{Nono}, M1) \end{aligned} \quad (9.20)$$

From (9.17), (9.20), and Modus Ponens:

$$\text{Criminal}(\text{West}) \quad (9.21)$$

If we formulate the process of finding a proof as a search process, then obviously this proof is the solution to the search problem, and equally obviously it would have to be a pretty smart program to find the proof without following any wrong paths. As a search problem, we would have

Initial state = KB (sentences 9.1-9.9)

Operators = applicable inference rules

Goal test = KB containing *Criminal(West)*

This example illustrates some important characteristics:

- The proof is 14 steps long.
- The branching factor increases as the knowledge base grows; this is because some of the inference rules combine existing facts.
- Universal Elimination can have an enormous branching factor on its own, because we can replace the variable by any ground term.
- We spent a lot of time combining atomic sentences into conjunctions, instantiating universal rules to match, and then applying Modus Ponens.

Thus, we have a serious difficulty, in the form of a collection of operators that give long proofs and a large branching factor, and hence a potentially explosive search problem. We also have an opportunity, in the form of an identifiable pattern of application of the operators (combining atomic sentences, instantiating universal rules, and then applying Modus Ponens). If we can define a better search space in which a single operator takes these three steps, then we can find proofs more efficiently.

## 9.3 GENERALIZED MODUS PONENS

In this section, we introduce a generalization of the Modus Ponens inference rule that does in a single blow what required an And-Introduction, Universal Elimination, and Modus Ponens in the earlier proof. The idea is to be able to take a knowledge base containing, for example:

*Missile(M1)*

*Owns(Nono, M1)*

$\forall x \text{Missile}(x) \wedge \text{Owns}(\text{Nono}, x) \Rightarrow \text{Sells}(\text{West}, \text{Nono}, x)$

and infer in one step the new sentence

*Sells(West, Nono, M1)*

Intuitively, this inference seems quite obvious. The key is to find some  $x$  in the knowledge base such that  $x$  is a missile and Nono owns  $x$ , and then infer that West sells this missile to Nono. More generally, if there is some substitution involving  $x$  that makes the premise of the implication identical to sentences already in the knowledge base, then we can assert the conclusion of the implication, after applying the substitution. In the preceding case, the substitution  $\{x/M1\}$  achieves this.

We can actually make Modus Ponens do even more work. Suppose that instead of knowing *Owns(Nono, M1)*, we knew that everyone owns M1 (a communal missile, as it were):

$\forall y \text{ Owns}(y, M1)$

Then we would still like to be able to conclude that *Sells(West, Nono, M1)*. This inference could be carried out if we first applied Universal Elimination with the substitution  $\{y/\text{Nono}\}$  to get *Owns(Nono, M1)*. The generalized version of Modus Ponens can do it in one step by finding a substitution for both the variables in the implication sentence and the variables in the sentences to be matched. In this case, applying the substitution  $\{x/M1, y/\text{Nono}\}$  to the premise *Owns(Nono, x)* and the sentence *Owns(y, M1)* will make them identical. If this can be done for all the premises of the implication, then we can infer the conclusion of the implication. The rule is as follows:

GENERALIZED  
MODUS PONENS

**0 Generalized Modus Ponens:** For atomic sentences  $p_i, p'_i$ , and  $q$ , where there is a substitution  $\theta$  such that  $\text{SUBST}(\theta, p'_i) = \text{SUBST}(\theta, p_i)$ , for all  $i$ :

$$\frac{p_1', p_2', \dots, p_n', (p_1 \wedge p_2 \wedge \dots \wedge p_n \Rightarrow q)}{\text{SUBST}(\theta, q)}$$

There are  $n + 1$  premises to this rule: the  $n$  atomic sentences  $p'_i$  and the one implication. There is one conclusion: the result of applying the substitution to the consequent  $q$ . For the example with West and the missile:

$p_1$  is *Missile(M1)*

$p_1$  is *Missile(x)*

$p_2$  is *Owns(y, M1)*

$p_2$  is *Owns(Nono, x)*

$0$  is  $\{x/M1, y/\text{Nono}\}$

$q$  is *Sells(West, Nono, x)*

$\text{SUBST}(\theta, q)$  is *Sells(West, Nono, M1)*

*Generalized Modus Ponens is an efficient inference rule for three reasons:*

1. It takes bigger steps, combining several small inferences into one.



UNIFICATION

CANONICAL FORM

HORN SENTENCES

2. It takes sensible steps—it uses substitutions that are guaranteed to help rather than randomly trying Universal Eliminations. The **unification** algorithm takes two sentences and returns a substitution that makes them look the same if such a substitution exists.
3. It makes use of a precompilation step that converts all the sentences in the knowledge base into a **canonical form**. Doing this once and for all at the start means we need not waste time trying conversions during the course of the proof.

We will deal with canonical form and unification in turn.

## Canonical Form

We are attempting to build an inferencing mechanism with one inference rule—the generalized version of Modus Ponens. That means that all sentences in the knowledge base should be in a form that matches one of the premises of the Modus Ponens rule—otherwise, they could never be used. In other words, the canonical form for Modus Ponens mandates that each sentence in the knowledge base be either an atomic sentence or an implication with a conjunction of atomic sentences on the left hand side and a single atom on the right. As we saw on page 174, sentences of this form are called **Horn sentences**, and a knowledge base consisting of only Horn sentences is said to be in Horn Normal Form.

We convert sentences into Horn sentences when they are first entered into the knowledge base, using Existential Elimination and And-Elimination.<sup>3</sup> For example,  $\exists x \text{ } \text{Owns}(\text{Nono}, x) \wedge \text{Missile}(x)$  is converted into the two atomic Horn sentences  $\text{Owns}(\text{Nono}, M1)$  and  $\text{Missile}(M1)$ . Once the existential quantifiers are all eliminated, it is traditional to drop the universal quantifiers, so that  $\forall y \text{ } \text{Owns}(y, M1)$  would be written as  $\text{Owns}(y, M1)$ . This is just an abbreviation—the meaning of  $y$  is still a universally quantified variable, but it is simpler to write and read sentences without the quantifiers. We return to the issue of canonical form on page 278.

## Unification

The job of the unification routine, UNIFY, is to take two atomic sentences  $p$  and  $q$  and return a substitution that would make  $p$  and  $q$  look the same. (If there is no such substitution, then UNIFY should return *fail*.) Formally,

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

UNIFIER

$\theta$  is called the **unifier** of the two sentences. We will illustrate unification in the context of an example, delaying discussion of detailed algorithms until Chapter 10. Suppose we have a rule

$$\text{Knows}(\text{John}, x) \Rightarrow \text{Hates}(\text{John}, x)$$

("John hates everyone he knows") and we want to use this with the Modus Ponens inference rule to find out whom he hates. In other words, we need to find those sentences in the knowledge base

<sup>3</sup> We will see in Section 9.5 that not all sentences can be converted into Horn form. Fortunately, the sentences in our example (and in many other problems) can be.

that unify with  $\text{Knows}(\text{John}, x)$ , and then apply the unifier to  $\text{Hates}(\text{John}, x)$ . Let our knowledge base contain the following sentences:

$\text{Knows}(\text{John}, \text{Jane})$

$\text{Knows}(y, \text{Leonid})$

$\text{Knows}(y, \text{Mother}(y))$

$\text{Knows}(x, \text{Elizabeth})$

(Remember that  $x$  and  $y$  are implicitly universally quantified.) Unifying the antecedent of the rule against each of the sentences in the knowledge base in turn gives us:

$\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(\text{John}, \text{Jane})) = \{x/\text{Jane}\}$

$\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(y, \text{Leonid})) = \{x/\text{Leonid}, y/\text{John}\}$

$\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(y, \text{Mother}(y))) = \{y/\text{John}, x/\text{Mother}(\text{John})\}$

$\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(x, \text{Elizabeth})) = \text{fail}$

The last unification fails because  $x$  cannot take on the value *John* and the value *Elizabeth* at the same time. But intuitively, from the facts that John hates everyone he knows and that everyone knows Elizabeth, we should be able to infer that John hates Elizabeth. It should not matter if the sentence in the knowledge base is  $\text{Knows}(x, \text{Elizabeth})$  or  $\text{Knows}(y, \text{Elizabeth})$ .

STANDARDIZE APART

One way to handle this problem is to **standardize apart** the two sentences being unified, which means renaming the variables of one (or both) to avoid name clashes. After standardizing apart, we would have

$\text{UNIFY}(\text{Knows}(\text{John}, x_1), \text{Knows}(x_2, \text{Elizabeth})) = \{x_1/\text{Elizabeth}, x_2/\text{John}\}$

The renaming is valid because  $\forall x \text{ Knows}(x, \text{Elizabeth})$  and  $\forall x_2 \text{ Knows}(x_2, \text{Elizabeth})$  have the same meaning. (See also Exercise 9.2.)

There is one more complication: we said that UNIFY should return a substitution that makes the two arguments look the same. But if there is one such substitution, then there are an infinite number:

$\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(y, z)) = \{y/\text{John}, x/z\}$   
 or  $\{y/\text{John}, x/z, w/\text{Freda}\}$   
 or  $\{y/\text{John}, x/\text{John}, z/\text{John}\}$   
 or ...

MOST GENERAL UNIFIER

Thus, we insist that UNIFY returns the **most general unifier** (or MGU), which is the substitution that makes the least commitment about the bindings of the variables. In this case it is  $\{y/\text{John}, x/z\}$ .

## Sample proof revisited

Let us solve our crime problem using Generalized Modus Ponens. To do this, we first need to put the original knowledge base into Horn form. Sentences (9.1) through (9.9) become

$\text{American}(x) \wedge \text{Weapon}(y) \wedge \text{Nation}(z) \wedge \text{Hostile}(z)$

$\text{A Sells}(x, z, y) \Rightarrow \text{Criminal}(x) \quad (9.22)$

$\text{Owns}(\text{Nono}, M) \quad (9.23)$

$$\text{Missile}(M1) \quad (9.24)$$

$$\text{Owns}(\text{Nono}, x) \wedge \text{Missile}(x) \Rightarrow \text{Sells}(\text{West}, \text{Nono}, x) \quad (9.25)$$

$$\text{Missile}(x) \Rightarrow \text{Weapon}(x) \quad (9.26)$$

$$\text{Enemy}(x, \text{America}) \Rightarrow \text{Hostile}(x) \quad (9.27)$$

$$\text{American}(\text{West}) \quad (9.28)$$

$$\text{Nation}(\text{Nono}) \quad (9.29)$$

$$\text{Enemy}(\text{Nono}, \text{America}) \quad (9.30)$$

$$\text{Nation}(\text{America}) \quad (9.31)$$

The proof involves just four steps. From (9.24) and (9.26) using Modus Ponens:

$$\text{Weapon}(M1) \quad (9.32)$$

From (9.30) and (9.27) using Modus Ponens:

$$\text{Hostile}(\text{Nono}) \quad (9.33)$$

From (9.23), (9.24), and (9.25) using Modus Ponens:

$$\text{Sells}(\text{West}, \text{Nono}, M1) \quad (9.34)$$

From (9.28), (9.32), (9.29), (9.33), (9.34) and (9.22), using Modus Ponens:

$$\text{Criminal}(\text{West}) \quad (9.35)$$

This proof shows how natural reasoning with Generalized Modus Ponens can be. (In fact, one might venture to say that it is not unlike the way in which a human might reason about the problem.) In the next section, we describe systematic reasoning algorithms using Modus Ponens. These algorithms form the basis for many large-scale applications of AI, which we describe in Chapter 10.

## 9.4 FORWARD AND BACKWARD CHAINING

Now that we have a reasonable language for representing knowledge, and a reasonable inference rule (Generalized Modus Ponens) for using that knowledge, we will study how a reasoning program is constructed.

The Generalized Modus Ponens rule can be used in two ways. We can start with the sentences in the knowledge base and generate new conclusions that in turn can allow more inferences to be made. This is called **forward chaining**. Forward chaining is usually used when a new fact is added to the database and we want to generate its consequences. Alternatively, we can start with something we want to prove, find implication sentences that would allow us to conclude it, and then attempt to establish their premises in turn. This is called **backward chaining**, because it uses Modus Ponens backwards. Backward chaining is normally used when there is a goal to be proved.

FORWARD CHAINING

BACKWARD CHAINING

RENAMEING

COMPOSITION

## Forward-chaining algorithm

Forward chaining is normally triggered by the addition of a new fact  $p$  to the knowledge base. It can be incorporated as part of the TELL process, for example. The idea is to find all implications that have  $p$  as a premise; then if the other premises are already known to hold, we can add the consequent of the implication to the knowledge base, triggering further inference (see Figure 9.1).

The FORWARD-CHAIN procedure makes use of the idea of a **renaming**. One sentence is a renaming of another if they are identical except for the names of the variables. For example,  $Likes(x, IceCream)$  and  $Likes(y, IceCream)$  are renamings of each other because they only differ in the choice of  $x$  or  $y$ , but  $Likes(x, x)$  and  $Likes(x, y)$  are not renamings of each other.

We also need the idea of a **composition** of substitutions.  $\text{COMPOSE}(\theta_1, \theta_2)$  is the substitution whose effect is identical to the effect of applying each substitution in turn. That is,

$$\text{SUBST}(\text{COMPOSE}(\theta_1, \theta_2), p) = \text{SUBST}(\theta_2, \text{SUBST}(\theta_1, p))$$

We will use our crime problem again to illustrate how FORWARD-CHAIN works. We will begin with the knowledge base containing only the implications in Horn form:

*American(x) A Weapon(y)A Nation(z)A Hostile(z)*

$$\text{A } Sells(x, z, y) \Rightarrow \text{Criminal}(x) \quad (9.36)$$

$$\text{Owns}(Nono, x) \text{ A } \text{Missile}(x) \Rightarrow \text{Sells}(West, Nono, x) \quad (9.37)$$

$$\text{Missile}(x) \Rightarrow \text{Weapon}(x) \quad (9.38)$$

$$\text{Enemy}(x, America) \Rightarrow \text{Hostile}(x) \quad (9.39)$$

**procedure** FORWARD-CHAIN( $KB, p$ )

if there is a sentence in  $KB$  that is a renaming of  $p$  then return

Add  $p$  to  $KB$

for each  $(p_1 \text{ A } \dots \text{ A } p_n \Rightarrow q)$  in  $KB$  such that for some  $i$ ,  $\text{UNIFY}(p_i, p) = \theta$  succeeds do

FIND-AND-INFER( $KB, [p_1, \dots, p_{i-1}, p_{i+1}, \dots, p_n], q, \theta$ )

end

**procedure** FIND-AND-INFER( $KB, premises, conclusion, \theta$ )

if  $premises = \emptyset$  then

FORWARD-CHAIN( $KB, \text{SUBST}(\theta, conclusion)$ )

else for each  $p'$  in  $KB$  such that  $\text{UNIFY}(p', \text{SUBST}(\theta, \text{FIRST}(premises))) = \theta_2$  do

FIND-AND-INFER( $KB, \text{REST}(premises), conclusion, \text{COMPOSE}(\theta, \theta_2)$ )

end

**Figure 9.1** The forward-chaining inference algorithm. It adds to  $KB$  all the sentences that can be inferred from the sentence  $p$ . If  $p$  is already in  $KB$ , it does nothing. If  $p$  is new, consider each implication that has a premise that matches  $p$ . For each such implication, if all the remaining premises are in  $KB$ , then infer the conclusion. If the premises can be matched several ways, then infer each corresponding conclusion. The substitution  $\theta$  keeps track of the way things match.

Now we add the atomic sentences to the knowledge base one by one, forward chaining each time and showing any additional facts that are added:

FORWARD-CHAIN( $KB, American(West)$ )

Add to the KB. It unifies with a premise of (9.36), but the other premises of (9.36) are not known, so FORWARD-CHAIN returns without making any new inferences.

FORWARD-CHAIN( $KB, Nation(Nono)$ )

Add to the KB. It unifies with a premise of (9.36), but there are still missing premises, so FORWARD-CHAIN returns.

FORWARD-CHAIN( $KB, Enemy(Nono, America)$ )

Add to the KB. It unifies with the premise of (9.39), with unifier  $\{x/Nono\}$ . Call

FORWARD-CHAIN( $KB, Hostile(Nono)$ )

Add to the KB. It unifies with a premise of (9.36). Only two other premises are known, so processing terminates.

FORWARD-CHAIN( $KB, Owns(Nono, M1)$ )

Add to the KB. It unifies with a premise of (9.37), with unifier  $\{x/M1\}$ . The other premise, now  $Missile(M1)$  is not known, so processing terminates.

FORWARD-CHAIN( $KB, Missile(M1)$ )

Add to the KB. It unifies with a premise of (9.37) and (9.38). We will handle them in that order.

- $Missile(M1)$  unifies with a premise of (9.37) with unifier  $\{x/M1\}$ . The other premise, now  $Owns(Nono, M1)$ , is known, so call

FORWARD-CHAIN( $KB, Sells(West, Nono, M1)$ )

Add to the KB. It unifies with a premise of (9.36), with unifier  $\{x/West, y/M1, z/Nono\}$ . The premise  $Weapon(M1)$  is unknown, so processing terminates.

- $Missile(M1)$  unifies with a premise of (9.38) with unifier  $\{x/M1\}$ . Call

FORWARD-CHAIN( $KB, Weapon(M1)$ )

Add to the KB. It unifies with a premise of (9.36), with unifier  $\{y/M1\}$ . The other premises are all known, with accumulated unifier  $\{x/West, y/M1, z/Nono\}$ . Call

FORWARD-CHAIN( $KB, Criminal(West)$ )

Add to the KB. Processing terminates.

As can be seen from this example, forward chaining builds up a picture of the situation gradually as new data comes in. Its inference processes are not directed toward solving any particular problem; for this reason it is called a **data-driven or data-directed** procedure. In this example, there were no rules capable of drawing irrelevant conclusions, so the lack of directedness was not a problem. In other cases (for example, if we have several rules describing the eating habits of Americans and the price of missiles), FORWARD-CHAIN will generate many irrelevant conclusions. In such cases, it is often better to use backward chaining, which directs all its effort toward the question at hand.

## Backward-chaining algorithm

Backward chaining is designed to find all answers to a question posed to the knowledge base. Backward chaining therefore exhibits the functionality required for the ASK procedure. The backward-chaining algorithm BACK-CHAIN works by first checking to see if answers can be provided directly from sentences in the knowledge base. It then finds all implications whose conclusion unifies with the query, and tries to establish the premises of those implications, also by backward chaining. If the premise is a conjunction, then BACK-CHAIN processes the conjunction conjunct by conjunct, building up the unifier for the whole premise as it goes. The algorithm is shown in Figure 9.2.

Figure 9.3 is the proof tree for deriving *Criminal(West)* from sentences (9.22) through (9.30). As a diagram of the backward chaining algorithm, the tree should be read depth-first, left to right. To prove *Criminal(x)* we have to prove the five conjuncts below it. Some of these are in the knowledge base, and others require further backward chaining. Each leaf node has the substitution used to obtain it written below. Note that once one branch of a conjunction succeeds, its substitution is applied to subsequent branches. Thus, by the time BACK-CHAIN gets to *Sells(x, z, y)*, all the variables are instantiated. Figure 9.3 can also be seen as a diagram of forward chaining. In this interpretation, the premises are added at the bottom, and conclusions are added once all their premises are in the KB. Figure 9.4 shows what can happen if an incorrect choice is made in the search—in this case, choosing America as the nation in question. There is no way to prove that America is a hostile nation, so the proof fails to go through, and we have to back up and consider another branch in the search space.

```

function BACK-CHAIN(KB, q) returns a set of substitutions
    BACK-CHAIN-LIST(KB, [q], {})

function BACK-CHAIN-LIST(KB, qlist, θ) returns a set of substitutions
    inputs: KB, a knowledge base
    qlist, a list of conjuncts forming a query (θ already applied)
    θ, the current substitution
    static: answers, a set of substitutions, initially empty

    if qlist is empty then return {θ}
    q  $\leftarrow$  FIRST(qlist)
        for each q'_i in KB such that  $\theta_i \leftarrow \text{UNIFY}(q, q'_i)$  succeeds do
            Add COMPOSE(θ, θi) to answers
        end
        for each sentence (p1 A ... A pn  $\Rightarrow$  q'_i) in KB such that  $\theta_i \leftarrow \text{UNIFY}(q, q'_i)$  succeeds do
            answers  $\leftarrow$  BACK-CHAIN-LIST(KB, SUBST(θi, [p1 ... pn]), COMPOSE(θ, θi)) U answers
        end
    return the union of BACK-CHAIN-LIST(KB, REST(qlist), θ) foreach θi  $\in$  answers

```

**Figure 9.2** The backward-chaining algorithm.

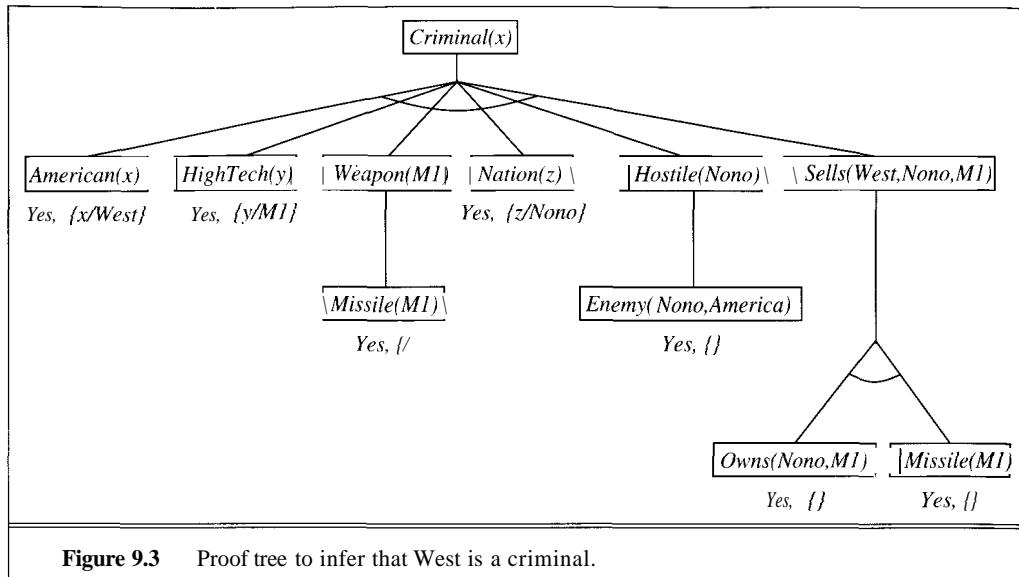


Figure 9.3 Proof tree to infer that West is a criminal.

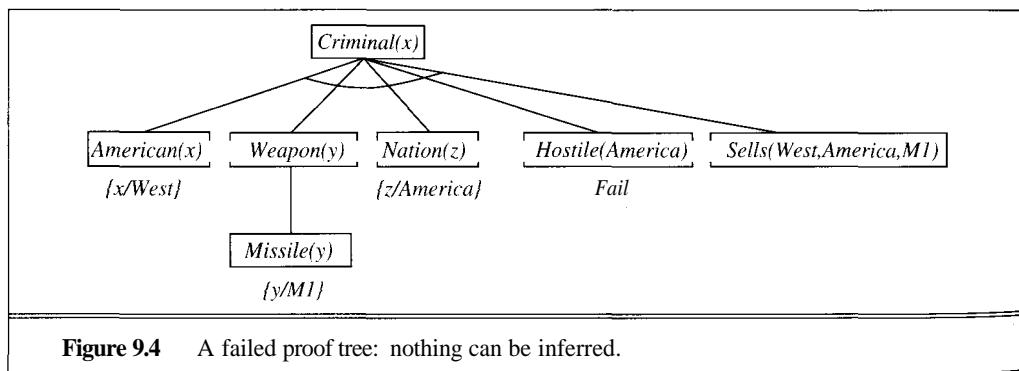


Figure 9.4 A failed proof tree: nothing can be inferred.

## 9.5 COMPLETENESS

Suppose we have the following knowledge base:

$$\begin{aligned}
 & \forall x \ P(x) \Rightarrow Q(x) \\
 & \forall x \ \neg P(x) \Rightarrow R(x) \\
 & \forall x \ Q(x) \Rightarrow S(x) \\
 & \forall x \ R(x) \Rightarrow S(x)
 \end{aligned} \tag{9.40}$$

Then we certainly want to be able to conclude  $S(A)$ ;  $S(A)$  is true if  $Q(A)$  or  $R(A)$  is true, and one of those must be true because either  $P(A)$  is true or  $\neg P(A)$  is true.

INCOMPLETE

Unfortunately, chaining with Modus Ponens cannot derive  $S(A)$  for us. The problem is that  $\forall x \neg P(x) \Rightarrow R(x)$  cannot be converted to Horn form, and thus cannot be used by Modus Ponens. That means that a proof procedure using Modus Ponens is **incomplete**: there are sentences entailed by the knowledge base that the procedure cannot infer.

The question of the existence of complete proof procedures is of direct concern to mathematicians. If a complete proof procedure can be found for mathematical statements, two things follow: first, all conjectures can be established mechanically; second, all of mathematics can be established as the logical consequence of a set of fundamental axioms. A complete proof procedure for first-order logic would also be of great value in AI: barring practical issues of computational complexity, it would enable a machine to solve any problem that can be stated in the language.

COMPLETENESS THEOREM

The question of completeness has therefore generated some of the most important mathematical work of the twentieth century. This work culminated in the results proved by the German mathematician Kurt Gödel in 1930 and 1931. Gödel has some good news for us; his **completeness theorem** showed that, for first-order logic, any sentence that is entailed by another set of sentences can be proved from that set. That is, we can find inference rules that allow a **complete** proof procedure  $R$ :

$$\text{if } KB \models \alpha \text{ then } KB \vdash_R \alpha$$

RESOLUTION ALGORITHM

The completeness theorem is like saying that a procedure for finding a needle in a haystack does exist. This is not a trivial claim, because universally quantified sentences and arbitrarily nested function symbols add up to haystacks of infinite size. Gödel showed that a proof procedure exists, but he did not demonstrate one; it was not until 1965 that Robinson published his **resolution algorithm**, which we discuss in the next section.

SEMIDEcidable

There is one problem with the completeness theorem that is a real nuisance. Note that we said that *if* a sentence follows, then it can be proved. Normally, we do not know until the proof is done that the sentence *does* follow; what happens when the sentence doesn't follow? Can we tell? Well, for first-order logic, it turns out that we cannot; our proof procedure can go on and on, but we will not know if it is stuck in a hopeless loop or if the proof is just about to pop out. (This is like the halting problem for Turing machines.) Entailment in first-order logic is thus **semidecidable**, that is, we can show that sentences follow from premises, if they do, but we cannot always show it if they do not. As a corollary, consistency of sets of sentences (the question of whether there is a way to make all the sentences true) is also semidecidable.

## 9.6 RESOLUTION: A COMPLETE INFERENCE PROCEDURE

Recall from Chapter 6 that the simple version of the resolution inference rule for propositional logic has the following form:

$$\frac{\alpha \vee \beta, \neg \beta \vee \gamma}{\alpha \vee \gamma} \quad \text{or equivalently} \quad \frac{\neg \alpha \Rightarrow \beta, \beta \Rightarrow \gamma}{\neg \alpha \Rightarrow \gamma}$$

The rule can be understood in two ways. First, we can see it as reasoning by cases. If  $\beta$  is false, then from the first disjunction,  $a$  must be true; but if  $\beta$  is true, then from the second disjunction  $\neg a$  must be true. Hence, either  $a$  or  $\neg a$  must be true. The second way to understand it is as transitivity of implication: from two implications, we derive a third that links the premise of the first to the conclusion of the second. Notice that Modus Ponens does not allow us to derive new implications; it only derives atomic conclusions. Thus, the resolution rule is more powerful than Modus Ponens. In this section, we will see that a generalization of the simple resolution rule can serve as the sole inference rule in a complete inference procedure for first-order logic.

## The resolution inference rule

In the simple form of the resolution rule, the premises have exactly two disjuncts. We can extend that to get a more general rule that says that for two disjunctions of any length, if one of the disjuncts in one clause ( $p_j$ ) unifies with the negation of a disjunct in the other ( $q_k$ ), then infer the disjunction of all the disjuncts except for those two:

GENERALIZED  
RESOLUTION  
(DISJUNCTIONS)

**0 Generalized Resolution (disjunctions):** For literals  $p_i$  and  $q_i$ ,  
where  $\text{UNIFY}(p_j, \neg q_k) = 0$ :

$$\frac{\begin{array}{c} p_1 \vee \dots \vee p_j \dots \vee p_m, \\ q_1 \vee \dots \vee q_k \dots \vee q_n \end{array}}{\text{SUBST}(\theta, (p_1 \vee \dots \vee p_{j-1} \vee p_{j+1} \dots \vee p_m \vee q_1 \dots \vee q_{k-1} \vee q_{k+1} \dots \vee q_n))}$$

Equivalently, we can rewrite this in terms of implications:

GENERALIZED  
RESOLUTION  
(IMPLICATIONS)

**◊ Generalized Resolution (implications):** For atoms  $p_i, q_i, r_i, s_i$   
where  $\text{UNIFY}(p_j, q_k) = 0$ :

$$\frac{\begin{array}{c} p_1 \wedge \dots \wedge p_j \dots \wedge p_{n_1} \Rightarrow r_1 \vee \dots \vee r_{n_2} \\ s_1 \wedge \dots \wedge s_{n_3} \Rightarrow q_1 \vee \dots \vee q_{n_4} \end{array}}{\text{SUBST}(\theta, (p_1 \wedge \dots \wedge p_{j-1} \wedge p_{j+1} \wedge p_{n_1} \wedge s_1 \wedge \dots \wedge s_{n_3} \Rightarrow r_1 \vee \dots \vee r_{n_2} \vee q_1 \vee \dots \vee q_{k-1} \vee q_{k+1} \wedge \dots \wedge q_{n_4}))}$$

## Canonical forms for resolution

CONJUNCTIVE  
NORMAL FORM

In the first version of the resolution rule, every sentence is a disjunction of literals. All the disjunctions in the KB are assumed to be joined in one big, implicit conjunction (as in a normal KB), so this form is called **conjunctive normal form** (or CNF), even though each individual sentence is a disjunction (confusing, isn't it?).

IMPPLICATIVE  
NORMAL FORM

In the second version of the resolution rule, each sentence is an implication with a conjunction of atoms on the left and a disjunction of atoms on the right. We call this **implicative normal form** (or INF), although the name is not standard. We can transform the sentences in (9.40) into either of the two forms, as we now show. (Notice that we have standardized apart the variable names in these sentences.)

## Conjunctive Normal Form

$$\neg P(w) \vee Q(w)$$

$$P(x) \vee R(x)$$

$$\neg Q(y) \vee S(y)$$

$$\neg R(z) \vee S(z)$$

## Implicative Normal Form

$$P(w) \Rightarrow Q(w)$$

$$\text{True} \Rightarrow P(x) \vee R(x)$$

$$Q(y) \Rightarrow S(y)$$

$$R(z) \Rightarrow S(z)$$

(9.41)

The two forms are notational variants of each other, and as we will see on page 281, any set of sentences can be translated to either form. Historically, conjunctive normal form is more common, but we will use implicative normal form, because we find it more natural.

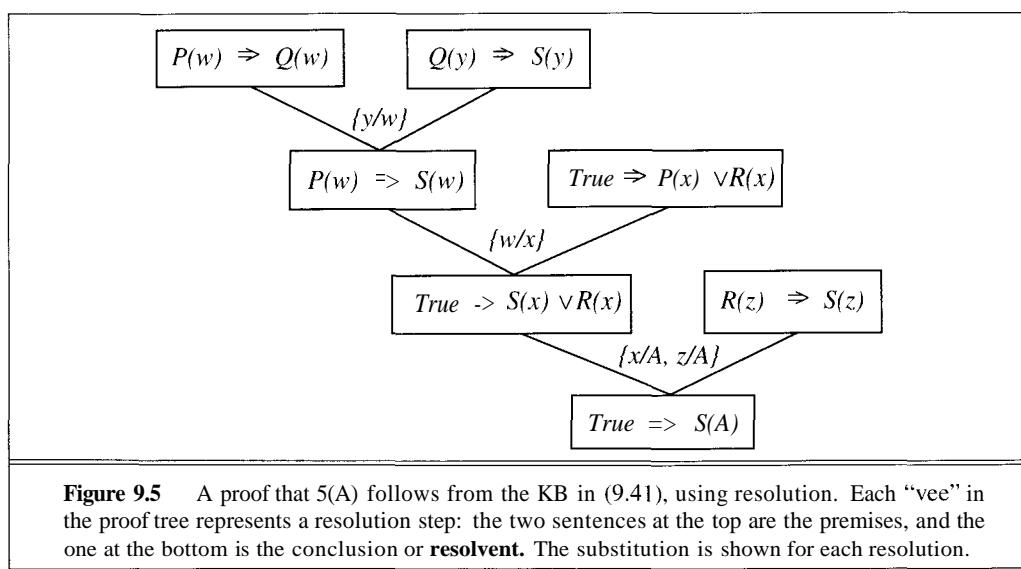
 It is important to recognize that *resolution is a generalization of modus ponens*. Clearly, the implicative normal form is more general than Horn form, because the right-hand side can be a disjunction, not just a single atom. But at first glance it seems that Modus Ponens has the ability to combine atoms with an implication to infer a conclusion in a way that resolution cannot do. This is just an illusion—once we realize that an atomic sentence  $\alpha$  in implicative normal form is written as  $\text{True} \Rightarrow \alpha$ , we can see that modus ponens is just a special case of resolution:

$$\frac{\alpha, \quad a \Rightarrow \beta}{\text{ft}} \quad \text{is equivalent to} \quad \frac{\text{True} \Rightarrow a, \quad a \Rightarrow \beta}{\text{True} \Rightarrow \beta}$$

Even though  $\text{True} \Rightarrow \alpha$  is the "correct" way to write an atomic sentence in implicative normal form, we will sometimes write  $\alpha$  as an abbreviation.

## Resolution proofs

One could use resolution in a forward- or backward-chaining algorithm, just as Modus Ponens is used. Figure 9.5 shows a three-step resolution proof of  $S(A)$  from the KB in (9.41).



FACTORING

Technically, the final resolvent should be  $\text{True} \Rightarrow S(A) \vee S(A)$ , but we have taken the liberty of removing the redundant disjunct. In some systems, there is a separate inference rule called **factoring** to do this, but it is simpler to just make it be part of the resolution rule.

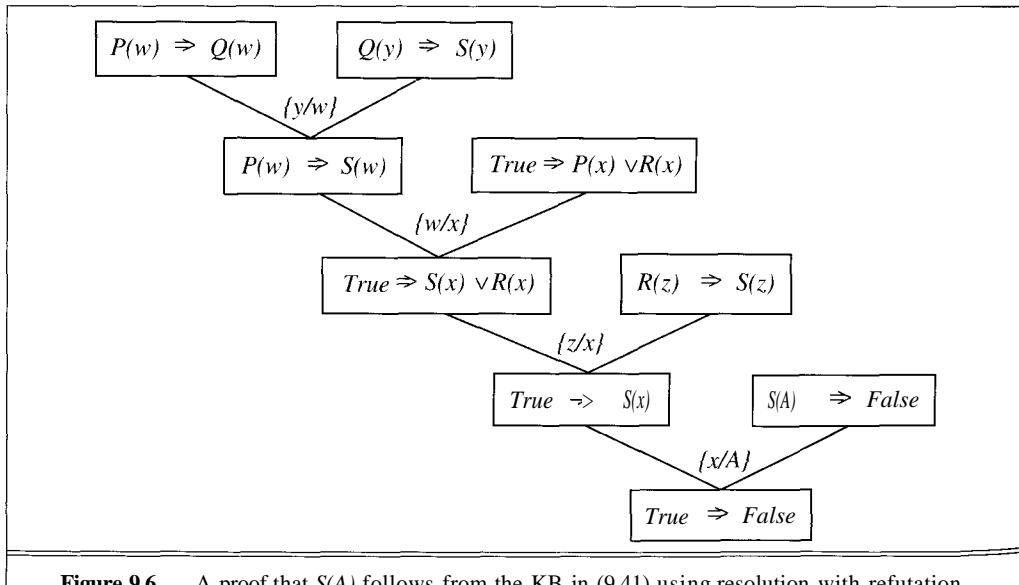
REFUTATION

Chaining with resolution is more powerful than chaining with Modus Ponens, but it is still not complete. To see that, consider trying to prove  $P \vee \neg P$  from the empty KB. The sentence is valid, but with nothing in the KB, there is nothing for resolution to apply to, and we are unable to prove anything.

One complete inference procedure using resolution is **refutation**, also known as **proof by contradiction** and **reductio ad absurdum**. The idea is that to prove  $P$ , we assume  $P$  is false (i.e., add  $\neg P$  to the knowledge base) and prove a contradiction. If we can do this, then it must be that the knowledge base implies  $P$ . In other words:

$$(KB \ A \ \neg P \Rightarrow \text{False}) \Leftrightarrow (KB \Rightarrow P)$$

Proof by contradiction is a powerful tool throughout mathematics, and resolution gives us a simple, sound, complete way to apply it. Figure 9.6 gives an example of the method. We start with the knowledge base of (9.41) and are attempting to prove  $S(A)$ . We negate this to get  $\neg S(A)$ , which in implicative normal form is  $S(A) \Rightarrow \text{False}$ , and add it to the knowledge base. Then we apply resolution until we arrive at a contradiction, which in implicative normal form is  $\text{True} \Rightarrow \text{False}$ . It takes one more step than in Figure 9.5, but that is a small price to pay for the security of a complete proof method.



**Figure 9.6** A proof that  $S(A)$  follows from the KB in (9.41) using resolution with refutation.

## Conversion to Normal Form

So far, we have claimed that resolution is complete, but we have not shown it. In this section we show that *any* first-order logic sentence can be put into implicative (or conjunctive) normal form, and in the following section we will show that from a set of sentences in normal form we can prove that a given sentence follows from the set.

We present the procedure for converting to normal form, step by step, showing that each step does not change the meaning; it should be fairly clear that all possible sentences are dealt with properly. You should understand why each of the steps in this procedure is valid, but few people actually manage to remember them all.

- ◊ **Eliminate implications:** Recall that  $p \Rightarrow q$  is the same as  $\neg p \vee q$ . So replace all implications by the corresponding disjunctions.
- ◊ **Move  $\neg$  inwards:** Negations are allowed only on atoms in conjunctive normal form, and not at all in implicative normal form. We eliminate negations with wide scope using de Morgan's laws (see Exercise 6.2), the quantifier equivalences and double negation:

$$\begin{aligned}\neg(p \vee q) &\text{ becomes } \neg p \wedge \neg q \\ \neg(p \wedge q) &\text{ becomes } \neg p \vee \neg q \\ \neg\forall x, p &\text{ becomes } \exists x \neg p \\ \neg\exists x, p &\text{ becomes } \forall x \neg p \\ \neg\neg p &\text{ becomes } p\end{aligned}$$

- 0 **Standardize variables:** For sentences like  $(\forall x P(x)) \vee (\exists x Q(x))$  that use the same variable name twice, change the name of one of the variables. This avoids confusion later when we drop the quantifiers.
- 0 **Move quantifiers left:** The sentence is now in a form in which all the quantifiers can be moved to the left, in the order in which they appear, without changing the meaning of the sentence. It is tedious to prove this properly; it involves equivalences such as

$$p \vee \forall x q \text{ becomes } \forall x p \vee q$$

which is true because  $p$  here is guaranteed not to contain an  $x$ .

- 0 **Skolemize:** **Skolemization** is the process of removing existential quantifiers by elimination. In the simple case, it is just like the Existential Elimination rule of Section 9.1—translate  $\exists x P(x)$  into  $P(A)$ , where  $A$  is a constant that does not appear elsewhere in the KB. But there is the added complication that some of the existential quantifiers, even though moved left, may still be nested inside a universal quantifier. Consider "Everyone has a heart":

$$\forall x \text{ Person}(x) \Rightarrow \exists y \text{ Heart}(y) \wedge \text{Has}(x, y)$$

If we just replaced  $y$  with a constant,  $H$ , we would get

$$\forall x \text{ Person}(x) \Rightarrow \text{Heart}(H) \wedge \text{Has}(x, H)$$

which says that everyone has the same heart  $H$ . We need to say that the heart they have is not necessarily shared, that is, it can be found by applying to each person a function that maps from person to heart:

$$\forall x \text{ Person}(x) \Rightarrow \text{Heart}(F(x)) \wedge \text{Has}(x, F(x))$$

## SKOLEM FUNCTION

where  $F$  is a function name that does not appear elsewhere in the KB.  $F$  is called a **Skolem function**. In general, the existentially quantified variable is replaced by a term that consists of a Skolem function applied to all the variables universally quantified *outside* the existential quantifier in question. Skolemization eliminates all existentially quantified variables, so we are now free to drop the universal quantifiers, because any variable must be universally quantified.

- ◊ **Distribute A over  $\vee$ :**  $(a \text{ A } b) \vee c$  becomes  $(a \vee c) \text{ A } (b \vee c)$ .
- ◊ **Flatten nested conjunctions and disjunctions:**  $(a \vee b) \vee c$  becomes  $(a \vee b \vee c)$ , and  $(a \text{ A } b) \text{ A } c$  becomes  $(a \text{ A } b \text{ A } c)$ .

At this point, the sentence is in conjunctive normal form (CNF): it is a conjunction where every conjunct is a disjunction of literals. This form is sufficient for resolution, but it may be difficult for us humans to understand.

- 0 **Convert disjunctions to implications:** Optionally, you can take one more step to convert to implicative normal form. For each conjunct, gather up the negative literals into one list, the positive literals into another, and build an implication from them:  
 $(\neg a \vee \neg b \vee c \vee d) \text{ becomes } (a \text{ A } b \Rightarrow c \vee d)$

## Example proof

We will now show how to apply the conversion procedure and the resolution refutation procedure on a more complicated example, which is stated in English as:

- Jack owns a dog.
- Every dog owner is an animal lover.
- No animal lover kills an animal.
- Either Jack or Curiosity killed the cat, who is named Tuna.
- Did Curiosity kill the cat?

First, we express the original sentences (and some background knowledge) in first-order logic:

- A.  $\exists x \text{ Dog}(x) \text{ A } \text{Owns}(Jack, x)$
- B.  $\forall x (\exists y \text{ Dog}(y) \text{ A } \text{Owns}(x, y)) \Rightarrow \text{AnimalLover}(x)$
- C.  $\forall x \text{ AnimalLover}(x) \Rightarrow \forall y \text{ Animal}(y) \Rightarrow \neg \text{Kills}(x, y)$
- D.  $\text{Kills}(Jack, \text{Tuna}) \vee \text{Kills}(\text{Curiosity}, \text{Tuna})$
- E.  $\text{Cat}(\text{Tuna})$
- F.  $\forall x \text{ Cat}(x) \Rightarrow \text{Animal}(x)$

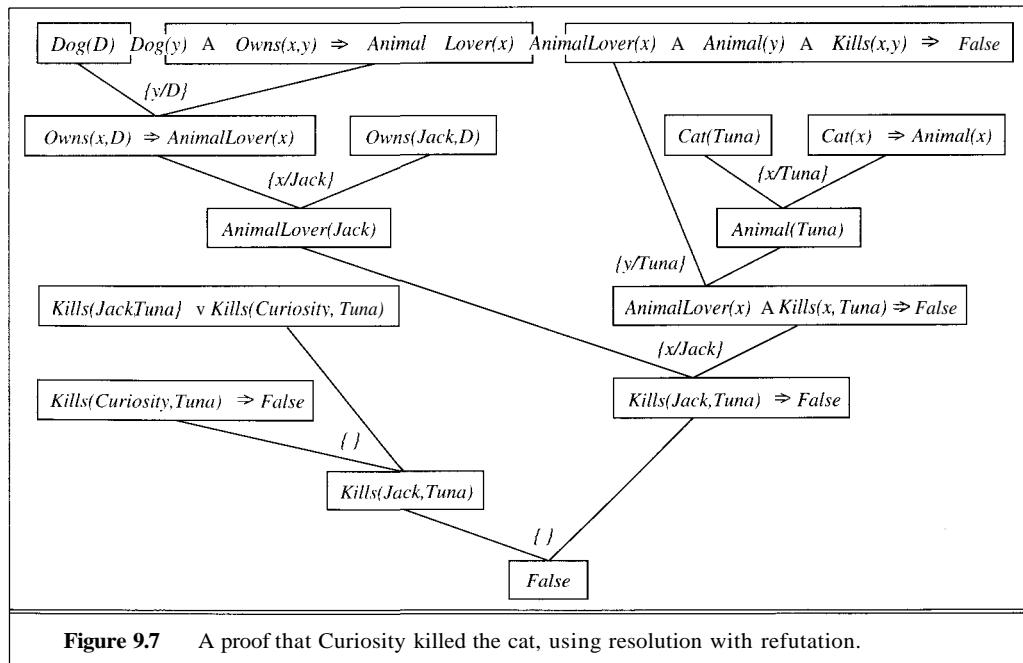
Now we have to apply the conversion procedure to convert each sentence to implicative normal form. We will use the shortcut of writing  $P$  instead of  $\text{True} \Rightarrow P$ :

- A1.  $\text{Dog}(D)$
- A2.  $\text{Owns}(Jack, D)$
- B.  $\text{Dog}(y) \text{ A } \text{Owns}(x, y) \Rightarrow \text{AnimalLover}(x)$
- C.  $\text{AnimalLover}(x) \text{ A } \text{Animal}(y) \text{ A } \text{Kills}(x, y) \Rightarrow \text{False}$
- D.  $\text{Kills}(Jack, \text{Tuna}) \vee \text{Kills}(\text{Curiosity}, \text{Tuna})$
- E.  $\text{Cat}(\text{Tuna})$
- F.  $\text{Cat}(x) \Rightarrow \text{Animal}(x)$

The problem is now to show that  $Kills(Curiosity, Tuna)$  is true. We do that by assuming the negation,  $Kills(Curiosity, Tuna) \Rightarrow False$ , and applying the resolution inference rule seven times, as shown in Figure 9.7. We eventually derive a contradiction,  $False$ , which means that the assumption must be false, and  $Kills(Curiosity, Tuna)$  is true after all. In English, the proof could be paraphrased as follows:

Suppose Curiosity did *not* kill Tuna. We know that either Jack or Curiosity did, thus Jack must have. But Jack owns D, and D is a dog, so Jack is an animal lover. Furthermore, Tuna is a cat, and cats are animals, so Tuna is an animal. Animal lovers don't kill animals, so Jack couldn't have killed Tuna. But this is a contradiction, because we already concluded that Jack must have killed Tuna. Hence, the original supposition (that Curiosity did not kill Tuna) must be wrong, and we have proved that Curiosity *did* kill Tuna.

The proof answers the question "Did Curiosity kill the cat?" but often we want to pose more general questions, like "Who killed the cat?" Resolution can do this, but it takes a little more work to obtain the answer. The query can be expressed as  $\exists w Kills(w, Tuna)$ . If you repeat the proof tree in Figure 9.7, substituting the negation of this query,  $Kills(w, Tuna) \Rightarrow False$  for the old query, you end up with a similar proof tree, but with the substitution  $\{w/Curiosity\}$  in one of the steps. So finding an answer to "Who killed the cat" is just a matter of looking in the proof tree to find the binding of  $w$ . It is straightforward to maintain a composed unifier so that a solution is available as soon as a contradiction is found.



**Figure 9.7** A proof that Curiosity killed the cat, using resolution with refutation.

## Dealing with equality

There is one problem that we have not dealt with so far, namely, finding appropriate inference rules for sentences containing the equality symbol. Unification does a good job of matching variables with other terms:  $P(x)$  unifies with  $P(A)$ . But  $P(A)$  and  $P(B)$  fail to unify, even if the sentence  $A = B$  is in the knowledge base. The problem is that unification only does a syntactic test based on the appearance of the argument terms, not a true semantic test based on the objects they represent. Of course, no semantic test is available because the inference system has no access to the objects themselves, but it should still be able to take advantage of what knowledge it has concerning the identities and differences among objects.

One way to deal with this is to axiomatize equality, by writing down its properties. We need to say that equality is reflexive, symmetric, and transitive, and we also have to say that we can substitute equals for equals in any predicate or function. So we need three basic axioms, and then one for each predicate and function:

$$\begin{aligned} \forall x \quad & x=x \\ \forall x, y \quad & x=y \Rightarrow y=x \\ \forall x, y, z \quad & x=y \wedge y=z \Rightarrow x=z \\ \forall x, y \quad & x=y \Rightarrow (P_1(x) \Leftrightarrow P_1(y)) \\ \forall x, y \quad & x=y \Rightarrow (P_2(x) \Leftrightarrow P_2(y)) \\ & \vdots \\ \forall w, x, y, z \quad & w=y \wedge x=z \Rightarrow (F_1(w, x) = F_1(y, z)) \\ \forall w, x, y, z \quad & w=y \wedge x=z \Rightarrow (F_2(w, x) = F_2(y, z)) \\ & \vdots \end{aligned}$$

The other way to deal with equality is with a special inference rule. The **demodulation** rule takes an equality statement  $x=y$  and any sentence with a nested term that unifies with  $x$  and derives the same sentence with  $y$  substituted for the nested term. More formally, we can define the inference rule as follows:

DEMODULATION

**0 Demodulation:** For any terms  $x, y$ , and  $z$ , where  $\text{UNIFY}(x, z) = 6$ :

$$\frac{x=y, (\dots z \dots)}{(\dots \text{SUBST}(\theta, >) \dots)}$$

PARAMODULATION

If we write all our equalities so that the simpler term is on the right (e.g.,  $(x + 0) = 0$ ), then demodulation will do simplification, because it always replaces an expression on the left with one on the right. A more powerful rule called **paramodulation** deals with the case where we do not know  $x = y$ , but we do know, say,  $x = y \vee P(x)$ .

## Resolution strategies

We know that repeated applications of the resolution inference rule will find a proof if one exists, but we have no guarantee of the efficiency of this process. In this section we look at four of the strategies that have been used to guide the search toward a proof.

UNIT CLAUSE

## Unit preference

This strategy prefers to do resolutions where one of the sentences is a single literal (also known as a **unit clause**). The idea behind the strategy is that we are trying to produce a very short sentence,  $\text{True} \Rightarrow \text{False}$ , and therefore it might be a good idea to prefer inferences that produce shorter sentences. Resolving a unit sentence (such as  $P$ ) with any other sentence (such as  $P \wedge Q \Rightarrow R$ ) always yields a sentence (in this case,  $Q \Rightarrow R$ ) that is shorter than the other sentence. When the unit preference strategy was first tried for propositional inference in 1964, it led to a dramatic speedup, making it feasible to prove theorems that could not be handled without the preference. Unit preference by itself does not, however, reduce the branching factor in medium-sized problems enough to make them solvable by resolution. It is, nonetheless, a useful heuristic that can be combined with other strategies.

SET OF SUPPORT

## Set of support

Preferences that try certain resolutions first are helpful, but in general it is more effective to try to eliminate some potential resolutions altogether. The set of support strategy does just that. It starts by identifying a subset of the sentences called the **set of support**. Every resolution combines a sentence from the set of support with another sentence, and adds the resolvent into the set of support. If the set of support is small relative to the whole knowledge base, this will cut the search space dramatically.

We have to be careful with this approach, because a bad choice for the set of support will make the algorithm incomplete. However, if we choose the set of support  $S$  so that the remainder of the sentences are jointly satisfiable, then set-of-support resolution will be complete. A common approach is to use the negated query as the set of support, on the assumption that the original knowledge base is consistent. (After all, if it is not consistent, then the fact that the query follows from it is vacuous.) The set-of-support strategy has the additional advantage of generating proof trees that are often easy for humans to understand, because they are goal-directed.

INPUT RESOLUTION

## Input resolution

In the **input resolution** strategy, every resolution combines one of the input sentences (from the KB or the query) with some other sentence. The proofs in Figure 9.5 and Figure 9.6 use only input resolutions; they have the characteristic shape of a diagonal "spine" with single sentences combining onto the spine. Clearly, the space of proof trees of this shape is smaller than the space of all proof graphs. In Horn knowledge bases, Modus Ponens is a kind of input resolution strategy, because it combines an implication from the original KB with some other sentences. Thus, it should not be surprising that input resolution is complete for knowledge bases that are in Horn form, but incomplete in the general case.

LINEAR RESOLUTION

The **linear resolution** strategy is a slight generalization that allows  $P$  and  $Q$  to be resolved together if either  $P$  is in the original KB or if  $P$  is an ancestor of  $Q$  in the proof tree. Linear resolution is complete.

## SUBSUMPTION

**Subsumption**

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

**9.7 COMPLETENESS OF RESOLUTION**

## REFUTATION-COMPLETE

This section proves that resolution is complete. It can be safely skipped by those who are willing to take it on faith.

Before we show that resolution is complete, we need to be more precise about the particular flavor of completeness that we will establish. Resolution is **refutation-complete**, which means that if a set of sentences is unsatisfiable, then resolution will derive a contradiction. Resolution cannot be used to generate all logical consequences of a set of sentences, but it can be used to establish that a given sentence is entailed by the set. Hence, it can be used to find all answers to a given question, using the negated-goal method described before.



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

Our proof sketch follows the original proof due to Robinson, with some simplifications from Genesereth and Nilsson (1987). The basic structure of the proof is shown in Figure 9.8, and is as follows:

1. We begin by observing that if  $S$  is unsatisfiable, then there exists a particular set of *ground instances* of the sentences of  $S$ , such that this set is also unsatisfiable (Herbrand's theorem).
2. We then show that resolution is complete for ground sentences (i.e., propositional logic). This is easy because the set of consequences of a propositional theory is always finite.
3. We then use a **lifting lemma** to show that, for any resolution proof using the set of ground sentences, there is a corresponding proof using the first-order sentences from which the ground sentences were obtained.

## HERBRAND UNIVERSE

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

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

- a. The function symbols in  $S$ , if any.
- b. The constant symbols in  $S$ , if any; if none, then the constant symbol A.

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

$$\{A, B, F(A, A), F(A, B), F(B, A), F(B, B), F(A, F(A, A)), F(A, F(A, B)), \dots\}$$

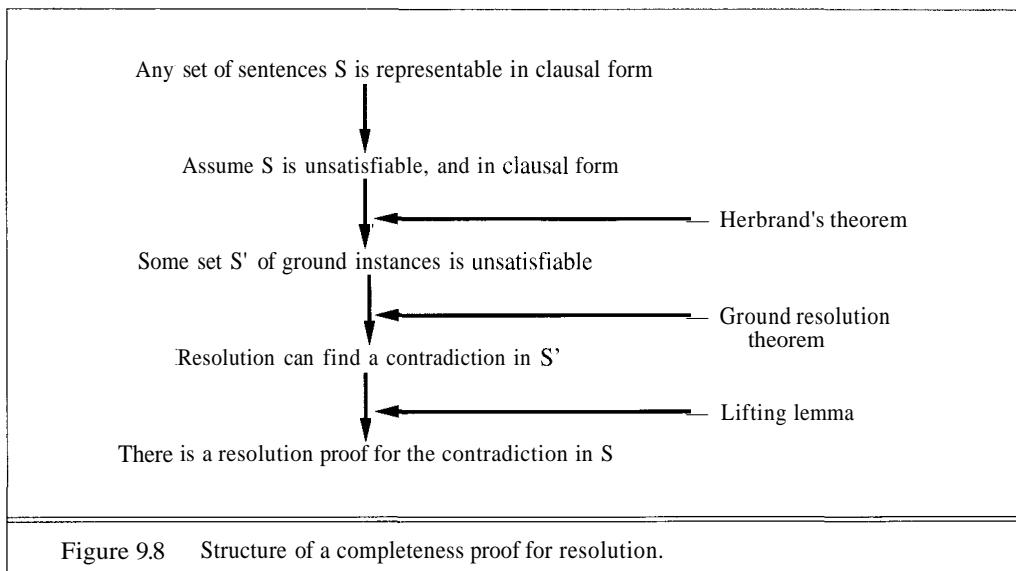


Figure 9.8 Structure of a completeness proof for resolution.

SATURATION

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

HERBRAND BASE

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

$$\begin{aligned}
 H_S(S) = & \{ P(A, F(A, A)) \wedge Q(A, A) \Rightarrow R(A, B), \\
 & P(B, F(B, A)) \wedge Q(B, A) \Rightarrow R(B, B), \\
 & P(F(A, A), F(F(A, A), A)) \wedge Q(F(A, A), A) \Rightarrow R(F(A, A), B), \\
 & P(F(A, B), F(F(A, B), A)) \wedge Q(F(A, B), A) \Rightarrow R(F(A, B), B), \\
 & \dots \}
 \end{aligned}$$

Notice that both the Herbrand universe and the Herbrand base can be infinite even if the original set of sentences  $S$  is finite.

HERBRAND'S THEOREM

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

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

RESOLUTION CLOSURE

Let  $S'$  be this finite subset. Now it will be useful to introduce the **resolution closure** of  $S'$ , which is the set of all clauses derivable by repeated application of the resolution inference step to clauses in  $S'$  or their derivatives. (To construct this closure, we could run a breadth-first search to completion using the resolution inference rule as the successor generator.) Let  $T$  be the resolution closure of  $S'$ , and let  $A_{S'} = \{A_1, A_2, \dots, A_k\}$  be the set of atomic sentences occurring in  $S'$ . Notice that because  $S'$  is finite,  $A_{S'}$  must also be finite. And because the clauses in  $T$  are constructed

## GÖDEL'S INCOMPLETENESS THEOREM

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

The proof of the incompleteness theorem is somewhat beyond the scope of this book, occupying, as it does, at least 30 pages, but we can give a hint here. We begin with the logical theory of numbers. In this theory, there is a single constant, 0, and a single function,  $S$  (the successor function). In the intended model,  $S(0)$  denotes 1,  $S(S(0))$  denotes 2, and so on; the language therefore has names for all the natural numbers. The vocabulary also includes the function symbols  $+$ ,  $\times$ , and  $Expt$  (exponentiation), and the usual set of logical connectives and quantifiers. The first step is to notice that the set of sentences that we can write in this language can be enumerated. (Imagine defining an alphabetical order on the symbols and then arranging in alphabetical order each of the sets of sentences of length 1, 2, and so on.) We can then number each sentence  $a$  with a unique natural number  $\#a$  (the **Gödel number**). This is crucial: number theory contains a name for each of its own sentences. Similarly, we can number each possible proof  $P$  with a Gödel number  $G(P)$ , because a proof is simply a finite sequence of sentences.

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

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

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

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

So, we have shown (barring 29 and a half pages) that for any set of true sentences of number theory, and in particular any set of basic axioms, there are other true sentences that *cannot* be proved from those axioms. This establishes, among other things, that we can never prove all the theorems of mathematics *within any given system of axioms*. Clearly, this was an important discovery for mathematics. Its significance for AI has been widely debated, beginning with speculations by Gödel himself. We take up the debate in Chapter 26.

entirely from members of  $A_{S'}$ ,  $T$  must be finite because only a finite number of distinct clauses can be constructed from a finite vocabulary of atomic sentences. To illustrate these definitions, we will use a slimmed-down example:

$$\begin{aligned} S' &= \{ P(A), P(A) \Rightarrow Q(A), Q(A) \Rightarrow \text{False} \} \\ A_{S'} &= \{ P(A), Q(A), \text{False} \} \\ T &= \{ P(A), P(A) \Rightarrow Q(A), Q(A) \Rightarrow \text{False}, Q(A), P(A) \Rightarrow \text{False}, \text{False} \} \end{aligned}$$

GROUND  
RESOLUTION  
THEOREM

Now we can state a completeness theorem for resolution on ground clauses. This is called the **ground resolution theorem**:

If a set of ground clauses is unsatisfiable, then the resolution closure of those clauses contains the clause *False*.

We prove this theorem by showing its contrapositive: if the closure  $T$  does *not* contain *False*, then  $S'$  is satisfiable; in fact, we can construct a satisfying assignment for the atomic sentences in  $S'$ . The construction procedure is as follows:

Pick an assignment (*True* or *False*) for each atomic sentence in  $Ay$  in some fixed order  $A_1, \dots, A_k$ :

- If there is a clause in  $T$  containing the literal  $\neg A_i$ , such that all its other literals are false under the assignment chosen for  $A_1, \dots, A_{i-1}$ , then assign  $A_i$  to be *False*.
- Otherwise, assign  $A_i$  to be *True*.

It is easy to show that the assignment so constructed will satisfy  $S'$ , provided  $T$  is closed under resolution and does not contain the clause *False* (Exercise 9.10).

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

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

LIFTING LEMMA

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

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

We see that indeed  $C'$  is a ground instance of  $C$ . In general, for  $C'_1$  and  $C'_2$  to have any resolvents, they must be constructed by first applying to  $C_1$  and  $C_2$  the most general unifier of a pair of

complementary literals in  $C_1$  and  $C_2$ . From the lifting lemma, it is easy to derive a similar statement about any sequence of applications of the resolution rule:

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

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

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

## 9.8 SUMMARY

---

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

- A simple extension of the propositional inference rules allows the construction of proofs for first-order logic. Unfortunately, the branching factor for the quantifier is huge.
- The use of **unification** to identify appropriate substitutions for variables eliminates the instantiation step in first-order proofs, making the process much more efficient.
- A generalized version of **Modus Ponens** uses unification to provide a natural and powerful inference rule, which can be used in a backward-chaining or forward-chaining algorithm.
- The canonical form for Modus Ponens is **Horn form**:

$$p \setminus A \dots A p_n \Rightarrow q, \text{ where } p_i \text{ and } q \text{ are atoms.}$$

This form cannot represent all sentences, and Modus Ponens is not a complete proof system.

- The generalized **resolution** inference rule provides a complete system for proof by refutation. It requires a normal form, but *any* sentence can be put into the form.
- Resolution can work with either **conjunctive normal form**—each sentence is a disjunction of literals—or **implicative normal form**—each sentence is of the form

$$p \setminus A \dots A p_n \Rightarrow q \setminus V \dots V q_m, \text{ where } p_i \text{ and } q_i \text{ are atoms.}$$

- Several strategies exist for reducing the search space of a resolution system without compromising completeness.

---

## BIBLIOGRAPHICAL AND HISTORICAL NOTES

**SYLLOGISM**

Logical inference was studied extensively in Greek mathematics. The type of inference most carefully studied by Aristotle was the **syllogism**. The syllogism is divided into "figures" and "moods," depending on the order of the terms (which we would call predicates) in the sentences, the degree of generality (which we would today interpret through quantifiers) applied to each term, and whether each term is negated. The most fundamental syllogism is that of the first mood of the first figure:

All  $S$  are  $M$ .  
All  $M$  are  $P$ .  
Therefore, all  $S$  are  $P$ .

Aristotle tried to prove the validity of other syllogisms by "reducing" them to those of the first figure. He was much less precise in describing what this "reduction" should involve than he was in characterizing the syllogistic figures and moods themselves.

Rigorous and explicit analysis of inference rules was one of the strong points of Megarian and Stoic propositional logic. The Stoics took five basic inference rules as valid without proof and then rigorously derived all others from these five. The first and most important of the five rules was the one known today as Modus Ponens. The Stoics were much more precise about what was meant by derivation than Aristotle had been about what was meant by reduction of one syllogism to another. They also used the "Principle of Conditionalization," which states that if  $q$  can be inferred validly from  $p$ , then the conditional " $p \Rightarrow q$ " is logically true. Both these principles figure prominently in contemporary logic.

Inference rules, other than logically valid schemas, were not a major focus either for Boole or for Frege. Boole's logic was closely modeled on the algebra of numbers. It relied mainly on the **equality substitution** inference rule, which allows one to conclude  $P(t)$  given  $P(s)$  and  $s = t$ . Logically valid schemas were used to obtain equations to which equality substitution could be applied. Whereas Frege's logic was much more general than Boole's, it too relied on an abundance of logically valid schemas plus a single inference rule that had premises—in Frege's case, this rule was Modus Ponens. Frege took advantage of the fact that the effect of an inference rule of the form "From  $p$  infer  $q$ " can be simulated by applying Modus Ponens to  $p$  along with a logically valid schema  $p \Rightarrow q$ . This "axiomatic" style of exposition, using Modus Ponens plus a number of logically valid schemas, was employed by a number of logicians after Frege; most notably, it was used in *Principia Mathematica* (Whitehead and Russell, 1910).

One of the earliest types of systems (after Frege) to focus prominently on inference rules was **natural deduction**, introduced by Gerhard Gentzen (1934) and by Stanisław Jaskowski (1934). Natural deduction is called "natural" because it does not require sentences to be subjected to extensive preprocessing before it can be applied to them (as many other proof procedures do) and because its inference rules are thought to be more intuitive than, say, the resolution rule. Natural deduction makes frequent use of the Principle of Conditionalization. The objects manipulated by Gentzen's inference rules are called *sequents*. A sequent can be regarded either as a logical argument (a pair of a set of premises and a set of alternative conclusions, intended to be an instance of some valid rule of inference) or as a sentence in implicative normal form. Prawitz (1965)

offers a book-length treatment of natural deduction. Gallier (1986) uses Gentzen sequents to expound the theoretical underpinnings of automated deduction.

Conjunctive normal form and disjunctive normal form for propositional formulas were known to Schröder (1877), although the principles underlying the construction of these forms go back at least to Boole. The use of clausal form (conjunctive normal form for first-order logic) depends upon certain techniques for manipulating quantifiers, in particular skolemization. Whitehead and Russell (1910) expounded the so-called *rules of passage* (the actual term is from Herbrand (1930)) that are used to move quantifiers to the front of formulas. (Moving all the quantifiers to the front of a formula is called *prenexing* the formula, and a formula with all the quantifiers in front is said to be in **prenex form**.) Horn form was introduced by Alfred Horn (1951). What we have called "implicative normal form" was used (with a right-to-left implication symbol) in Robert Kowalski's (1979b) *Logic for Problem Solving*, and this way of writing clauses is sometimes called "Kowalski form."

Skolem constants and Skolem functions were introduced, appropriately enough, by Thoralf Skolem (1920). The general procedure for skolemization is given in (Skolem, 1928), along with the important notion of the Herbrand universe.

Herbrand's theorem, named after the French logician Jacques Herbrand (1930), has played a vital role in the development of automated reasoning methods, both before and after Robinson's introduction of resolution. This is reflected in our reference to the "Herbrand universe" rather than the "Skolem universe," even though Skolem really invented this concept (and indeed Herbrand does not explicitly use Skolem functions and constants, but a less elegant although roughly equivalent device). Herbrand can also be regarded as the inventor of unification, because a variant of the unification algorithm occurs in (Herbrand, 1930).

First-order logic was shown to have complete proof procedures by Gödel (1930), using methods similar to those of Skolem and Herbrand. Alan Turing (1936) and Alonzo Church (1936) simultaneously showed, using very different proofs, that validity in first-order logic was not decidable. The excellent text by Enderton (1972) explains all of these results in a rigorous yet moderately understandable fashion.

The first mechanical device to carry out logical inferences was constructed by the third Earl of Stanhope (1753-1816). The Stanhope Demonstrator could handle syllogisms and certain inferences in the theory of probability. The first mechanical device to carry out inferences in *mathematical logic* was William Stanley Jevons's "logical piano," constructed in 1869. Jevons was one of the nineteenth-century logicians who expanded and improved Boole's work; the logical piano carried out reasoning in Boolean logic. An entertaining and instructive history of these and other early mechanical devices for reasoning is given by Martin Gardner (1968).

The first published results from research on automated deduction using electronic computers were those of Newell, Shaw, and Simon (1957) on the Logic Theorist. This program was based on an attempt to model human thought processes. Martin Davis (1957) had actually designed a program that came up with a proof in 1954, but the Logic Theorist's results were published slightly earlier. Both Davis's 1954 program and the Logic Theorist were based on somewhat ad hoc methods that did not strongly influence later automated deduction.

It was Abraham Robinson who suggested attempting to use Herbrand's Theorem to generate proofs mechanically. Gilmore (1960) wrote a program that uses Robinson's suggestion in a way influenced by the "Beth tableaux" method of proof (Beth, 1955). Davis and Putnam (1960)

introduced clausal form, and produced a program that attempted to find refutations by substituting members of the Herbrand universe for variables to produce ground clauses and then looking for propositional inconsistencies among the ground clauses. Prawitz (1960) developed the key idea of letting the quest for propositional inconsistency drive the search process, and generating terms from the Herbrand universe only when it was necessary to do so in order to establish propositional inconsistency. After further development by other researchers, this idea led J. A. Robinson (no relation) to develop the resolution method (Robinson, 1965), which used unification in its modern form to allow the demonstration of propositional inconsistency without necessarily making explicit use of terms from the Herbrand universe. The so-called "inverse method" developed at about the same time by the Soviet researcher S. Maslov (1964; 1967) is based on principles somewhat different from Robinson's resolution method but offers similar computational advantages in avoiding the unnecessary generation of terms in the Herbrand universe. The relations between resolution and the inverse method are explored by Maslov (1971) and by Kuehner (1971).

The demodulation rule described in the chapter was intended to eliminate equality axioms by combining equality substitution with resolution, and was introduced by Wos (1967). Term rewriting systems such as the Knuth-Bendix algorithm (Knuth and Bendix, 1970) are based on demodulation, and have found wide application in programming languages. The paramodulation inference rule (Wos and Robinson, 1968) is a more general version of demodulation that provides a complete proof procedure for first-order logic with equality.

In addition to demodulation and paramodulation for equality reasoning, other special-purpose inference rules have been introduced to aid reasoning of other kinds. Boyer and Moore (1979) provide powerful methods for the use of mathematical induction in automated reasoning, although their logic unfortunately lacks quantifiers and does not quite have the full power of first-order logic. Stickel's (1985) "theory resolution" and Manna and Waldinger's (1986) method of "special relations" provide general ways of incorporating special-purpose inference rules into a resolution-style framework.

A number of control strategies have been proposed for resolution, beginning with the unit preference strategy (Wos *et al.*, 1964). The set of support strategy was proposed by Wos *et al.* (1965), to provide a degree of goal-directedness in resolution. *Linear resolution* first appeared in (Loveland, 1968). Wolfgang Bibel (1981) developed the **connection method** which allows complex deductions to be recognized efficiently. Developments in resolution control strategies, and the accompanying growth in the understanding of the relationship between completeness and syntactic restrictions on clauses, contributed significantly to the development of **logic programming** (see Chapter 10). Genesereth and Nilsson (1987, Chapter 5) provide a short but thorough analysis of a wide variety of control strategies.

There are a number of good general-purpose introductions to automated deduction and to the theory of proof and inference; some were mentioned in Chapter 7. Additional textbooks on matters related to completeness and undecidability include *Computability and Logic* (Boolos and Jeffrey, 1989), *Metalogic* (Hunter, 1971), and (for an entertaining and unconventional, yet highly rigorous approach) *A Course in Mathematical Logic* (Manin, 1977). Many of the most important papers from the turn-of-the-century development of mathematical logic are to be found in *From Frege to Gödel: A Source Book in Mathematical Logic* (van Heijenoort, 1967). The journal of record for the field of pure mathematical logic (as opposed to automated deduction) is *The Journal of Symbolic Logic*.

Textbooks geared toward automated deduction include (in addition to those mentioned in Chapter 7) the classic *Symbolic Logic and Mechanical Theorem Proving* (Chang and Lee, 1973) and, more recently, *Automated Reasoning: Introduction and Applications* (Wos *et al.*, 1992). The two-volume anthology *Automation of Reasoning* (Siekmann and Wrightson, 1983) includes many important papers on automated deduction from 1957 to 1970. The historical summaries prefacing each volume provide a concise yet thorough overview of the history of the field. Further important historical information is available from Loveland's "Automated Theorem Proving: A Quarter-Century Review" (1984) and from the bibliography of (Wos *et al.*, 1992).

The principal journal for the field of theorem proving is the *Journal of Automated Reasoning*; important results are also frequently reported in the proceedings of the annual Conferences on Automated Deduction (CADE). Research in theorem proving is also strongly related to the use of logic in analyzing programs and programming languages, for which the principal conference is Logic in Computer Science.

---

## EXERCISES

- 9.1** For each of the following pairs of atomic sentences, give the most general unifier, if it exists.
- $P(A, B, B), P(x, y, z).$
  - $Q(y, G(A, B)), Q(G(x, x), y).$
  - $Older(Father(y), y), Older(Father(x), John).$
  - $Knows(Father(y), y).Knows(x, x).$
- 9.2** One might suppose that we can avoid the problem of variable conflict in unification by standardizing apart all of the sentences in the knowledge base once and for all. Show that for some sentences, this approach cannot work. (*Hint:* Consider a sentence, one part of which unifies with another.)
- 9.3** Show that the final state of the knowledge base after a series of calls to FORWARD-CHAIN is independent of the order of the calls. Does the number of inference steps required depend on the order in which sentences are added? Suggest a useful heuristic for choosing an order.
- 9.4** Write down logical representations for the following sentences, suitable for use with Generalized Modus Ponens:
- Horses, cows, and pigs are mammals.
  - An offspring of a horse is a horse.
  - Bluebeard is a horse.
  - Bluebeard is Charlie's parent.
  - Offspring and parent are inverse relations.
  - Every mammal has a parent.

9.5 In this question we will use the sentences you wrote in Exercise 9.4 to answer a question using a backward-chaining algorithm.

- a. Draw the proof tree generated by an exhaustive backward-chaining algorithm for the query  $\exists h \ Horse(h)$ .
- b. What do you notice about this domain?
- c. How many solutions for  $h$  actually follow from your sentences?
- d. Can you think of a way to find all of them? (*Hint:* You might want to consult (Smith *et al.*, 1986).)

9.6 A popular children's riddle is "Brothers and sisters have I none, but that man's father is my father's son." Use the rules of the family domain (Chapter 7) to show who that man is. You may use any of the inference methods described in this chapter.

9.7 How can resolution be used to show that a sentence is

- a. Valid?
- b. Unsatisfiable?

9.8 From "Horses are animals," it follows that "The head of a horse is the head of an animal." Demonstrate that this inference is valid by carrying out the following steps:

- a. Translate the premise and the conclusion into the language of first-order logic. Use three predicates:  $HeadOf(h, x)$ ,  $Horse(x)$ , and  $Animal(x)$ .
- b. Negate the conclusion, and convert the premise and the negated conclusion into conjunctive normal form.
- c. Use resolution to show that the conclusion follows from the premise.

9.9 Here are two sentences in the language of first-order logic:

$$(A): \forall x \ \exists y \ (x \geq y)$$

$$(B): \exists y \ \forall x \ (x \geq y)$$

- a. Assume that the variables range over all the natural numbers  $0, 1, 2, \dots, \infty$ , and that the " $>$ " predicate means "greater than or equal to." Under this interpretation, translate these sentences into English.
- b. Is (A) true under this interpretation?
- c. Is (B) true under this interpretation?
- d. Does (A) logically entail (B)?
- e. Does (B) logically entail (A)?
- f. Try to prove that (A) follows from (B) using resolution. Do this even if you think that (B) does not logically entail (A); continue until the proof breaks down and you cannot proceed (if it does break down). Show the unifying substitution for each resolution step. If the proof fails, explain exactly where, how, and why it breaks down.
- g. Now try to prove that (B) follows from (A).

**9.10** In this exercise, you will complete the proof of the ground resolution theorem given in the chapter. The proof rests on the claim that if  $T$  is the resolution closure of a set of ground clauses  $S'$ , and  $T$  does not contain the clause *False*, then a satisfying assignment can be constructed for  $S'$  using the construction given in the chapter. Show that the assignment does indeed satisfy  $S'$ , as claimed.

# 10

# LOGICAL REASONING SYSTEMS

*In which we show how to build efficient programs that reason with logic.*

## 10.1 INTRODUCTION

We have explained that it is a good idea to build agents as reasoning systems—systems that explicitly represent and reason with knowledge. The main advantage of such systems is a high degree of modularity. The control structure can be isolated from the knowledge, and each piece of knowledge can be largely independent of the others. This makes it easier to experiment with the system and modify it, makes it easier for the system to explain its workings to another agent, and, as we will see in Part VI, makes it easier for the system to learn on its own.

In this chapter the rubber hits the road, so to speak, and we discuss ways in which these advantages can be realized in an actual, efficient system. Automated reasoning systems come in several flavors, each designed to address different kinds of problems. We group them into four main categories:

THEOREM PROVERS  
LOGIC  
PROGRAMMING  
LANGUAGES

PRODUCTION  
SYSTEMS

- ◊ **Theorem provers and logic programming languages:** Theorem provers use resolution (or some other complete inference procedure) to prove sentences in full first-order logic, often for mathematical and scientific reasoning tasks. They can also be used to answer questions: the proof of a sentence containing variables serves as an answer to a question because it instantiates the variables. Logic programming languages typically restrict the logic, disallowing full treatment of negation, disjunction, and/or equality. They usually use backward chaining, and may include some nonlogical features of programming languages (such as input and output). Examples of theorem provers: SAM, AURA, OTTER. Examples of logic programming languages: Prolog, MRS, LIFE.
- ◊ **Production systems:** Like logic programming languages, these use implications as their primary representation. The consequent of each implication is interpreted as an action recommendation, rather than simply a logical conclusion. Actions include insertions and

deletions from the knowledge base as well as input and output. Production systems operate with a forward-chaining control structure. Some have a conflict resolution mechanism to decide which action to take when more than one is recommended. Examples: OPS-5, CLIPS, SOAR.

FRAME SYSTEMS  
SEMANTIC NETWORKS

DESCRIPTION LOGIC SYSTEMS

TERMINOLOGICAL LOGICS

- ◊ **Frame systems and semantic networks:** These systems use the metaphor that objects are nodes in a graph, that these nodes are organized in a taxonomic structure, and that links between nodes represent binary relations. In frame systems the binary relations are thought of as slots in one frame that are filled by another, whereas in semantic networks, they are thought of as arrows between nodes. The choice between the frame metaphor and the semantic network metaphor determines whether you draw the resulting networks as nested boxes or as graphs, but the meaning and implementation of the two types of systems can be identical. In this chapter we will say "semantic network" to mean "semantic network or frame system." Examples of frame systems: OWL, FRAIL, KODIAK. Examples of semantic networks: SNEPS, NETL, Conceptual Graphs.
- 0 **Description logic systems:** These systems evolved from semantic networks due to pressure to formalize what the networks mean while retaining the emphasis on taxonomic structure as an organizing principle. The idea is to express and reason with complex definitions of, and relations among, objects and classes. Description logics are sometimes called **terminological logics**, because they concentrate on defining terms. Recent work has concentrated on the trade-off between expressivity in the language and the computational complexity of certain operations. Examples: KL-ONE, CLASSIC, LOOM.

In this chapter, we will see how each of the four types of systems can be implemented, and how each of the following five tasks is addressed:

1. Add a new fact to the knowledge base. This could be a percept, or a fact derived via inference. We call this function TELL.
2. Given a knowledge base and a new fact, derive some of the facts implied by the conjunction of the knowledge base and the new fact. In a forward-chaining system, this is part of TELL.
3. Decide if a query is entailed by the knowledge base. We call this function ASK. Different versions of ASK do different things, ranging from just confirming that the query is entailed to returning a set of all the possible substitutions that make the query true.
4. Decide if a query is explicitly stored in the knowledge base—a restricted version of ASK.
5. Remove a sentence from the knowledge base. It is important to distinguish between correcting a sentence that proves to be false, forgetting a sentence that is no longer useful (perhaps because it was only relevant in the past), and updating the knowledge base to reflect a change in the world, while still remembering how the world was in the past. (Some systems can make this distinction; others rely on the knowledge engineer to keep things straight.)

All knowledge-based systems rely on the fundamental operation of retrieving sentences satisfying certain conditions—for example, finding an atomic sentence that unifies with a query, or finding an implication that has a given atomic sentence as one of its premises. We will therefore begin with techniques for maintaining a knowledge base in a form that supports efficient retrieval.

## 10.2 INDEXING, RETRIEVAL, AND UNIFICATION

The functions TELL and ASK can in general do complicated reasoning using forward and backward chaining or resolution. In this section, we consider two simpler functions that implement the part of TELL and ASK that deals directly with the physical implementation of the knowledge base. We call these functions STORE and FETCH. We also describe the implementation of a unification algorithm, another basic component of knowledge-based systems.

### **Implementing sentences and terms**

The first step in building a reasoning system is to define the data types for sentences and terms. This involves defining both the syntax of sentences—the format for interacting with the user at the logic level—and the internal representation in which the system will store and manipulate sentences at the implementation level. There may be several internal representations for different aspects of sentences. For example, there may be one form that allows the system to print sentences and another to represent sentences that have been converted to clausal form.

Our basic data type will represent the application of an operator (which could be a predicate, a function symbol or a logical connective) to a list of arguments (which could be terms or sentences). We will call this general data type a COMPOUND. It has fields for the operator (OP) and arguments (ARGS). For example, let  $c$  be the compound  $P(x) \wedge Q(x)$ ; then  $OP[c] = \wedge$  and  $ARGS[c] = [P(x), Q(x)]$ .

### **Store and fetch**

Now that we have a data type for sentences and terms, we need to be able to maintain a set of sentences in a knowledge base, which means storing them in such a way that they can be fetched efficiently. Typically, FETCH is responsible for finding sentences in the knowledge base that unify with the query, or at least have the same syntactic structure. ASK is responsible for the inference strategy, which results in a series of calls to FETCH. *The computational cost of inference is dominated by two aspects: the search strategy used by ASK and the data structures used to implement FETCH.*

The call  $STORE(KB, S)$  adds each conjunct of the sentence  $S$  to the knowledge base  $KB$ . The simplest approach is to implement the knowledge base as an array or linked list of conjuncts. For example, after

```
TELL(KB, A ∧ ¬B)
TELL(KB, ¬C ∨ D)
```

the  $KB$  will contain a list with the elements

```
[A, ¬B, ¬C, D]
```

The call  $FETCH(KB, Q)$  must then go through the elements of the knowledge base one at a time until it either finds a conjunct that matches  $Q$  or reaches the end. With this approach FETCH takes

$O(n)$  time on an  $n$ -element  $KB$ . STORE takes  $O(1)$  time to add a conjunct to the  $KB$ , but if we want to ensure that no duplicates are added, then STORE is also  $O(n)$ . This is impractical if one wants to do serious inference.

### Table-based indexing

A better approach is to implement the knowledge base as a hash table.<sup>1</sup> If we only had to deal with ground literal sentences<sup>2</sup> we could implement STORE so that when given  $P$  it stores the value *true* in the hash table under the key  $P$ , and when given  $\neg P$ , it stores *false* under the key  $P$ . Then FETCH could do a simple lookup in the hash table, and both FETCH and STORE would be  $O(1)$ .

There are two problems with this approach: it does not deal with complex sentences other than negated sentences, and it does not deal with variables within sentences. So FETCH would not be able to find "an implication with  $P$  as consequent," such as might be required by a backward-chaining algorithm; nor could it find  $Brother(Richard, John)$  when given a query  $\exists x \ Brother(Richard, x)$ .

The solution is to make STORE maintain a more complex table. We assume the sentences are all converted to a normal form. (We use implicative normal form.) The keys to the table will be predicate symbols, and the value stored under each key will have four components:

- A list of positive literals for that predicate symbol.
- A list of negative literals.
- A list of sentences in which the predicate is in the conclusion.
- A list of sentences in which the predicate is in the premise.

So, given the knowledge base:

*Brother(Richard, John)*  
*Brother(Ted, Jack) A Brother(Jack, Bobbie)*  
*\neg Brother(Ann, Sam)*  
*Brother(x, y) \Rightarrow Male(x)*  
*Brother(x, y) A Male(y) \Rightarrow Brother(y, x)*  
*Male(Jack)A Male(Ted)A ... A \neg Male(Ann)A ...*

the table for the knowledge base would be as shown in Figure 10.1.

Now suppose that we ask the query

*ASK(KB, Brother(Jack, Ted))*

and that ASK uses backward chaining (see Section 9.4). It would first call FETCH to find a positive literal matching the query. Because that fails, FETCH is called to find an implication with *Brother* as the consequent. The query matches the consequent, the antecedent becomes the new goal after the appropriate substitution is applied, and the procedure begins again.

<sup>1</sup> A hash table is a data structure for storing and retrieving information indexed by fixed keys. For practical purposes, a hash table can be considered to have constant storage and retrieval times, even when the table contains a very large number of items.

<sup>2</sup> Remember that a **ground literal** contains no variables. It is either an atomic sentence such as *Brother(Richard, John)* or a negated atomic sentence such as *\neg Brother(Ann, Victoria)*.

Key	Positive	Negative	Conclusion	Premise
<i>Brother</i>	<i>Brother(Richard, John)</i> <i>Brother(Ted, Jack)</i> <i>Brother(Jack, Bobbie)</i>	$\neg\text{Brother(Ann, Sam)}$	$\text{Brother}(x, y) \wedge \text{Male}(y) \Rightarrow \text{Brother}(y, x)$	$\text{Brother}(x, y) \wedge \text{Male}(y) \Rightarrow \text{Brother}(y, x)$ $\text{Brother}(x, y) \Rightarrow \text{Male}(x)$
<i>Male</i>	<i>Male(Jack)</i> <i>Male(Ted)</i> ...	$\neg\text{Male(Ann)}$ ...	$\text{Brother}(x, y) \Rightarrow \text{Male}(x)$	$\text{Brother}(x, y) \wedge \text{Male}(y) \Rightarrow \text{Brother}(y, x)$
Figure 10.1 Table-based indexing for a collection of logical sentences.				

Because in first-order logic the predicate is always fixed in a query, the simple device of dividing up the knowledge base by predicate reduces the cost of fetching considerably. But why stop with just the predicate?

## Tree-based indexing

Table-based indexing is ideal when there are many predicate symbols and a few clauses for each symbol. But in some applications, there are many clauses for a given predicate symbol. For example, in a knowledge base for the U.S. Census Bureau that uses social security numbers to represent people, the query *Brother(012-34-5678, x)* would require a search through millions of *Brother* literals.

To make this search efficient, we need to index on the arguments as well as on the predicate symbols themselves. One way to do this is to change the table entry for *Brother* so that each entry is itself a table, indexed by the first argument, rather than just a list. So to answer the query *Brother(012-34-5678, x)*, we first look in the predicate table under *Brother*, and then look in that entry under 012-34-5678. Both table lookups take a small constant amount of time, so the complete FETCH is efficient. We can view the process of searching for a matching literal as a walk down a tree, where the branch at each point is dictated by the symbol at the appropriate point in the query sentence (Figure 10.2).

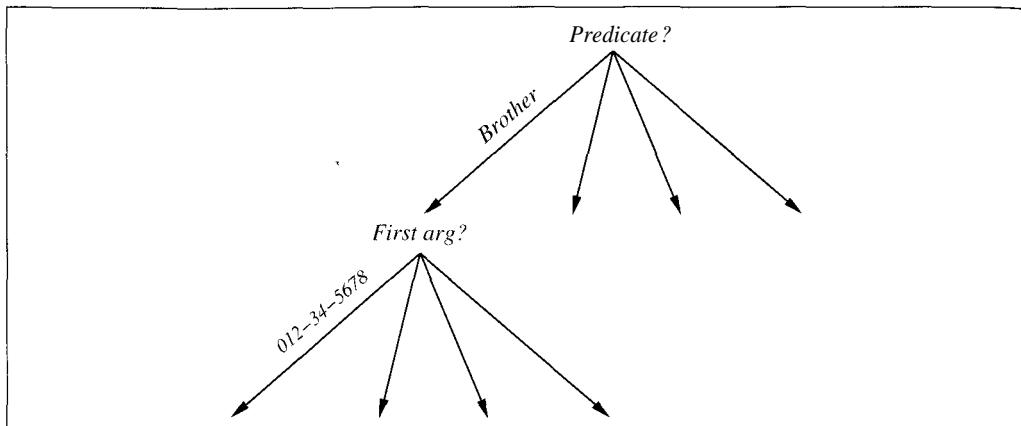
**Tree-based indexing** is one form of **combined indexing**, in that it essentially makes a combined key out of the sequence of predicate and argument symbols in the query. Unfortunately, it provides little help when one of the symbols in the sequence is a variable, because every branch has to be followed in that case. Suppose our census knowledge base has a predicate *Taxpayer* with four arguments: a person, a zip code, a net income to the nearest thousand, and the number of dependents, for example:

*Taxpayer(012-34-5678, 02138, 32000, 10)*

Suppose we were interested in finding all the people in zip code 02138 with exactly 10dependents:

*FETCH(Taxpayer(p, 02138, i, 10))*

There are tens of thousands of people with that zip code, and hundreds of thousands of people in the country with 10dependents, but there are probably only a few people that match both criteria. To find those people without undue effort, we need a combined index based on both the second and fourth argument. If we are to deal with every possible set of variable positions, we will need



**Figure 10.2** Tree-based indexing organizes a knowledge base into a nested series of hash tables. Each node in the tree is a hash table indexed by the value at a particular sentence position.

2<sup>n</sup> combined indices, where  $n$  is the number of symbol positions in the sentences being stored. When sentences include complex terms,  $n$  can easily grow quite large. At some point, the extra storage needed for the indices and the extra work that STORE must do to maintain them outweigh the benefits. We can respond by adopting a fixed policy, such as maintaining indices only on keys composed of predicate plus each argument; or by using an adaptive policy that creates indices to meet the demands of the kinds of queries being asked.

#### CROSS-INDEXING

The **cross-indexing** strategy indexes entries in several places, and when faced with a query chooses the most promising place for retrieval. Suppose we have the query

`FETCH(Taxpayer(p, 02138, 20000, 3))`

and the four available indices key on *Taxpayer* plus each of the four argument positions separately. A sentence matching the query will be indexed under *Taxpayer*(*\_*, 02138, *\_*, *\_*), *Taxpayer*(*\_*, *\_*, 20000, *\_*), and *Taxpayer*(*\_*, *\_*, *\_*, 3). The best strategy is usually to search through whichever of these collections of sentences is smallest.

## The unification algorithm

In Section 9.3 we showed how two statements such as

$$\begin{aligned} \text{Knows(John, } x) &\Rightarrow \text{Hates(John, } x) \\ \text{Knows(John, } \text{Jane)} \end{aligned}$$

can be combined to infer *Hates(John, Jane)*. The key to this Modus Ponens inference is to unify *Knows(John, x)* and *Knows(John, Jane)*. This unification yields as a result the substitution {*x/Jane*}, which can then be applied to *Hates(John, x)* to give the solution *Hates(John, Jane)*.

We have seen that by clever indexing, we can reduce the number of calls to the unification algorithm, but this number still can be quite large. Thus, the unification algorithm should be efficient. The algorithm shown in Figure 10.3 is reasonably simple. It recursively explores the

two expressions simultaneously, building up a unifier as it goes along but failing if it ever finds two corresponding points in the structures that do not match. Unfortunately, it has one expensive step. The **occur-check** takes time linear in the size of the expression being checked, and is done for each variable found. It makes the time complexity of the algorithm  $O(n^2)$  in the size of the expressions being unified. Later we will see how to make this algorithm more efficient by eliminating the need for explicit representations of substitutions. On page 308, we see how unification can be extended to handle more information besides equality.

```

function UNIFY(x, y) returns a substitution to make x and y identical, if possible
    UNIFY-INTERNAL(x, y, { })

function UNIFY-INTERNAL(x, y, θ) returns a substitution to make x and y identical (given θ)
    inputs: x, a variable, constant, list, or compound
            y, a variable, constant, list, or compound
            θ, the substitution built up so far

    if θ = failure then return failure
    else if x = y then return θ
    else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
    else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
    else if COMPOUND?(x) and COMPOUND?(y) then
        return UNIFY-INTERNAL(ARGS[x], ARGS[y], UNIFY-INTERNAL(OP[x], OP[y], θ))
    else if LIST?(x) and LIST?(y) then
        return UNIFY-INTERNAL(REST[x], REST[y], UNIFY-INTERNAL(FIRST[x], FIRST[y], θ))
    else return failure

function UNIFY-VAR(var, x, θ) returns a substitution
    inputs: var, a variable
            x, any expression
            θ, the substitution built up so far

    if {var/val} ∈ θ
        then return UNIFY-INTERNAL(val, x, θ)
    else if {x/val} ∈ θ
        then return UNIFY-INTERNAL(var, val, θ)
    else if var occurs anywhere in x /* occur-check */
        then return failure
    else return add {x/var} to θ
```

**Figure 10.3** The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution *θ* that is the argument to UNIFY-INTERNAL is built up along the way, and used to make sure that later comparisons are consistent with bindings that were established earlier.

## 10.3 LOGIC PROGRAMMING SYSTEMS

We now turn from the details of implementing a knowledge base to a comparison of ways in which a knowledge base can be constructed and used. We start with logic programming.



We have seen that the declarative approach has many advantages for building intelligent systems. Logic programming tries to extend these advantages to all programming tasks. *Any computation can be viewed as a process of making explicit the consequences of choosing a particular program for a particular machine and providing particular inputs.* Logic programming views the program and inputs as logical statements about the world, and the process of making consequences explicit as a process of inference. The relation between logic and algorithms is summed up in Robert Kowalski's equation

$$\text{Algorithm} = \text{Logic} + \text{Control}$$

PROLOG

A logic programming language makes it possible to write algorithms by augmenting logical sentences with information to control the inference process. **Prolog** is by far the most widely used logic programming language. Its users number in the hundreds of thousands. It is used primarily as a rapid-prototyping language and for symbol-manipulation tasks such as writing compilers (Van Roy, 1990) and parsing natural language (Pereira and Warren, 1980). It has also been used to develop expert system applications in legal, medical, financial, and other domains.

### The Prolog language

We have already explained the notational conventions of Prolog in Chapter 7. Viewed as a logical knowledge base, a Prolog program has the following characteristics:

NEGATION AS FAILURE

- A program consists of a sequence of sentences, implicitly conjoined. All variables have implicit universal quantification, and variables in different sentences are considered distinct.
- Only Horn clause sentences are acceptable. This means that each sentence is either an atomic sentence or an implication with no negated antecedents and an atomic consequent.
- Terms can be constant symbols, variables, or functional terms.
- Queries can include conjunctions, disjunctions, variables, and functional terms.
- Instead of negated antecedents in implications, Prolog uses a **negation as failure** operator: a goal  $\text{not } P$  is considered proved if the system fails to prove  $P$ .
- All syntactically distinct terms are assumed to refer to distinct objects. That is, you cannot assert  $A = B$  or  $A = F(x)$ , where  $A$  is a constant. You can assert  $x = B$  or  $x = F(y)$ , where  $x$  is a variable.
- There is a large set of built-in predicates for arithmetic, input/output, and various system and knowledge base functions. Literals using these predicates are "proved" by executing code rather than doing further inference. In Prolog notation (where capitalized names are variables), the goal  $X$  is  $4 + 3$  succeeds with  $X$  bound to 7. However, the goal  $5$  is  $X + Y$  cannot be proved, because the built-in functions do not do arbitrary equation solving.<sup>3</sup>

<sup>3</sup> Note that if proper axioms are provided for addition, such goals can be solved by inference within a Prolog program.

As an example, here is a Prolog program for the *Member* relation, given both in normal first-order logic notation and in the format actually used by Prolog:

$\forall x, l \text{ Member}(x, [x l])$	$\text{member}(X, [X L]).$
$\forall x, y, l \text{ Member}(x, l) \Rightarrow$	$\text{member}(X, [Y L]) :-$
$\text{Member}(x, [y l])$	$\text{member}(X, L).$

HEAD  
BODY

As we mentioned in Chapter 7, the Prolog representation has the consequent, or **head**, on the left-hand side, and the antecedents, or **body**, on the right. A Prolog clause is often read as "To prove *(the head)*, prove *(the body)*." To preserve this intuitive reading along with our logical notation, we will compromise and write Prolog clauses using a leftward implication. For example, the second clause of the *Member* definition becomes

$$\text{Member}(x, [y|l]) \Leftarrow \text{Member}(x, l)$$

The definition of *Member* can be used to answer several kinds of queries. It can be used to confirm that *Member*(2, [1, 2, 3]) is true. It can also enumerate the three values of *x* that make *Member*(*x*, [1, 2, 3]) true. It can be used to find the value of *x* such that *Member*(2, [1, *x*, 3]) is true. It even can be used to enumerate the lists for which *Member*(1, *list*) is true.

## Implementation

The designers of Prolog made a number of implementation decisions designed to provide a simple, fast execution model:

- All inferences are done by backward chaining, with depth-first search. That means that whenever there is a dead end in the attempt to prove a sentence, Prolog backs up to the most recent step that has alternatives.
- The order of search through the conjuncts of an antecedent is strictly left to right, and clauses in the knowledge base are applied in first-to-last order.
- The occur-check is omitted from the unification routine.

The omission of the occur-check makes Prolog inference unsound, but actual errors happen very seldom in practice. The use of depth-first search makes Prolog incomplete, because of infinite paths created by circular sentences (but see page 311 for another approach). Programmers must therefore keep termination in mind when writing recursive sentences. Despite these caveats, one good point of Prolog is that *the execution model is simple enough that a trained programmer can add control information to yield an efficient program*.



Like our BACK-CHAIN algorithm (page 275), Prolog enumerates all the solutions to a query, but it does not gather them into a set. Instead, it is up to the user's program to do what it will with each solution as it is enumerated. The most common thing to do is to print the answers. In fact, Prolog's top level does this automatically. A query such as

```
member(loc(X, X), [loc(1, 1), loc(2, 1), loc(2, 2)])?
```

results in the user seeing two pieces of output, "X = 1" and "X = 2".

The execution of a Prolog program can happen in two modes: interpreted and compiled. Compilation is discussed in the next subsection. Interpretation essentially amounts to running

the BACK-CHAIN algorithm from Section 9.4, with the program as the knowledge base. We say "essentially," because Prolog interpreters contain a variety of improvements designed to maximize speed. Here we consider only two.

## CHOICE POINT

First, instead of constructing the list of all possible answers for each subgoal before continuing to the next, Prolog interpreters generate one answer and a "promise" to generate the rest when the current answer has been fully explored. This promise is called a **choice point**. When the depth-first search completes its exploration of the possible solutions arising from the current answer and backs up to the choice point, the choice point is expanded to yield a new answer for the subgoal and a new choice point. This approach saves both time and space. It also provides a very simple interface for debugging because at all times there is only a single solution path under consideration.

## TRAIL

Second, our simple implementation of BACK-CHAIN spends a good deal of time generating substitutions and applying them to query lists. Prolog eliminates the need for a *substitution* data type by implementing logic variables that can remember their current binding. At any point in time every variable in the program is either unbound or is bound to some value. Together, these variables and values implicitly define a substitution. Of course, there is only one such substitution at a time, but that is all we need. The substitution is the right one for the current path in the search tree. Extending the path can only add new variable bindings, because an attempt to add a different binding for an already-bound variable results in a failure of unification. When a path in the search fails, Prolog will back up to a previous choice point, and then it may have to unbind some variables. This is done by keeping track of all the variables that have been bound in a stack called the **trail**. As each new variable is bound by UNIFY-VAR, the variable is pushed onto the trail stack. When a goal fails and it is time to back up to a previous choice point, each of the variables is unbound as it is removed from the trail.

## OPEN-CODE

## Compilation of logic programs

It is possible to make a reasonably efficient Prolog interpreter by following the guidelines in the previous subsection. But interpreting programs in any language, including Prolog, is necessarily slower than running compiled code. This is because the interpreter always behaves as if it has never seen the program before. A Prolog interpreter must do database retrieval to find sentences that match the goal, and analysis of sentence structure to decide what subgoals to generate. All serious heavy-duty Prolog programming is done with compiled code. The great advantage of compilation is that when it is time to execute the inference process, we can use inference routines specifically designed for the sentences in the knowledge base. Prolog basically generates a miniature theorem prover for each different predicate, thereby eliminating much of the overhead of interpretation. It is also possible to **open-code** the unification routine for each different call, thereby avoiding explicit analysis of term structure. (For details of open-coded unification, see Warren *et al.* (1977).)

The instruction sets of today's computers give a poor match with Prolog's semantics, so most Prolog compilers compile into an intermediate language rather than directly into machine language. The most popular intermediate language is the Warren Abstract Machine, or WAM, named after David H. D. Warren, one of the implementors of the first Prolog compiler. The WAM

is an abstract instruction set that is suitable for Prolog and can be either interpreted or translated into machine language. Other compilers translate Prolog into a high-level language such as Lisp or C, and then use that language's compiler to translate to machine language. For example, the definition of the *Member* predicate can be compiled into the code shown in Figure 10.4.

```
procedure MEMBER(item, list, continuation)
    trail ← GLOBAL-TRAIL-POINTER()
    if UNIFY([item | NEW-VARIABLE()], list) then CALL(continuation)
    RESET-TRAIL(trail)
    rest ← NEW-VARIABLE()
    if UNIFY(list, [NEW-VARIABLE() | rest]) then MEMBER(item, rest, continuation)
```

**Figure 10.4** Pseudocode representing the result of compiling the *Member* predicate. The function NEW-VARIABLE returns a new variable, distinct from all other variables so far used. The procedure CALL(*continuation*) continues execution with the specified continuation.

There are several points worth mentioning:

- Rather than having to search the knowledge base for *Member* clauses, the clauses are built into the procedure and the inferences are carried out simply by calling the procedure.
- As described earlier, the current variable bindings are kept on a trail. The first step of the procedure saves the current state of the trail, so that it can be restored by RESET-TRAIL if the first clause fails. This will undo any bindings generated by the first call to UNIFY.
- The trickiest part is the use of **continuations** to implement choice points. You can think of a continuation as packaging up a procedure and a list of arguments that together define what should be done next whenever the current goal succeeds. It would not do just to return from a procedure like MEMBER when the goal succeeds, because it may succeed in several ways, and each of them has to be explored. The continuation argument solves this problem because it can be called each time the goal succeeds. In the MEMBER code, if *item* unifies with the first element of the *list*, then the MEMBER predicate has succeeded. We then CALL the continuation, with the appropriate bindings on the trail, to do whatever should be done next. For example, if the call to MEMBER were at the top level, the continuation would print the bindings of the variables.

Before Warren's work on compilation of inference in Prolog, logic programming was too slow for general use. Compilers by Warren and others allowed Prolog to achieve speeds of up to 50,000 LIPS (logical inferences per second) on standard 1990-model workstations. More recently, application of modern compiler technology, including type inference, open-coding, and interprocedural data-flow analysis has allowed Prolog to reach speeds of several million LIPS, making it competitive with C on a variety of standard benchmarks (Van Roy, 1990). Of course, the fact that one can write a planner or natural language parser in a few dozen lines of Prolog makes it somewhat more desirable than C for prototyping most small-scale AI research projects.

## Other logic programming languages

Although Prolog is the only accepted standard for logic programming, there are many other useful systems, each extending the basic Prolog model in different ways.

OR-PARALLELISM

AND-PARALLELISM

CONSTRAINT LOGIC PROGRAMMING

TYPE PREDICATES

The parallelization of Prolog is an obvious direction to explore. If we examine the work done by a Prolog program, there are two principal sources of parallelism. The first, called **OR-parallelism**, comes from the possibility of a goal unifying with many different literals and implications in the knowledge base. Each gives rise to an independent branch in the search space that can lead to a potential solution, and all such branches can be solved in parallel. The second, called **AND-parallelism**, comes from the possibility of solving each conjunct in the body of an implication in parallel. And-parallelism is more difficult to achieve, because solutions for the whole conjunction require consistent bindings for all the variables. Each conjunctive branch must communicate with the other branches to ensure a global solution. A number of projects have been successful in achieving a degree of parallel inference, but the most advanced is probably the PIM (Parallel Inference Machine) project, part of the Fifth Generation Computing Systems project in Japan. PIM has achieved speeds of 64 million LIPS.

Prolog can be enriched, rather than just accelerated, by generalizing the notion of the binding of a variable. Prolog's logic variables are very useful because they allow a programmer to generate a partial solution to a problem, leaving some of the variables unbound, and then later fill in the values for those variables. Unfortunately, there is no way in Prolog to specify *constraints* on values: for example, to say that  $X < 3$ , and then later in the computation determine the exact value of  $X$ . The **constraint logic programming** (CLP) formalism extends the notions of variables and unification to allow such constraints. Consider this definition of a triangle based on the lengths of the three sides:

$$\text{Triangle}(x, y, z) \Leftarrow (x > 0) \wedge (y > 0) \wedge (z > 0) \wedge (x + y > z) \wedge (y + z > x) \wedge (x + z > y)$$

In either Prolog or CLP, this definition can be used to confirm  $\text{Triangle}(3, 4, 5)$ . But only in CLP would the query  $\text{Triangle}(x, 4, 5)$  give the binding specified by the constraint  $\{x > 1 \wedge x < 9\}$ ; in standard Prolog, this query would fail.

As well as using arithmetical constraints on variables, it is possible to use logical constraints. For example, we can insist that a particular variable refer to a *Person*. In standard Prolog, this can only be done by inserting the conjunct  $\text{Person}(p)$  into the body of a clause. Then, when the clause is used, the system will attempt to solve the remainder of the clause with  $p$  bound to each different person in the knowledge base. In languages such as Login and Life, literals containing **type predicates** such as *Person* are implemented as constraints. Inference is delayed until the constraints need to be resolved. Thus,  $\text{Person}(p)$  just means that the variable  $p$  is constrained to be a person; it does not generate alternative bindings for  $p$ . The use of types can simplify programs, and the use of constraints can speed up their execution.

## Advanced control facilities

Going back to our census knowledge base, consider the query "What is the income of the spouse of the president?" This might be stated in Prolog as

$$\text{Income}(s, t) \wedge \text{Married}(s, p) \wedge \text{Occupation}(p, \text{President})$$

This query will be expensive to compute, because it must enumerate all person/income pairs, then fetch the spouse for each person (failing on those that are not married, and looping on anyone who is married multiple times), and finally check for the one person whose occupation is president. Conjunctive queries like this often can be answered more efficiently if we first spend some time to reorder the conjuncts to reduce the expected amount of computation. For this query, a better ordering is

$$\text{Occupation}(p, \text{President}) \wedge \text{Married}(s, p) \wedge \text{Income}(s, i)$$

This yields the same answer, but with no backtracking at all, assuming that the *Occupation* and *Married* predicates are indexed by their second arguments.

## METAREASONING

This reordering process is an example of **metareasoning**, or reasoning about reasoning. As with constraint satisfaction search (Section 3.7), the heuristic we are using for conjunct ordering is to put the most constraining conjuncts first. In this case it is clear that only one  $p$  satisfies *Occupation*( $p$ , President), but it is not always easy to predict in advance how many solutions there will be to a predicate. Even if it were, it would not be a good idea to try all  $n!$  permutations of an  $n$ -place conjunction for large  $n$ . Languages such as MRS (Genesereth and Smith, 1981; Russell, 1985) allow the programmer to write metarules to decide which conjuncts are tried first. For example, the user could write a rule saying that the goal with the fewest variables should be tried first.

Some systems change the way backtracking is done rather than attempting to reorder conjuncts. Consider the problem of finding all people  $x$  who come from the same town as the president. One inefficient ordering of this query is:

$$\text{Resident}(p, \text{town}) \wedge \text{Resident}(x, \text{town}) \wedge \text{Occupation}(p, \text{President})$$

## CHRONOLOGICAL BACKTRACKING

Prolog would try to solve this by enumerating all residents  $p$  of any town, then enumerating all residents  $x$  of that town, and then checking if  $p$  is the president. When the *Occupation*( $p$ , President) goal fails, Prolog backtracks to the *most recent* choice point, which is the *Resident*( $x$ , town) goal. This is called **chronological backtracking**; although simple, it is sometimes inefficient. Clearly, generating a new  $x$  cannot possibly help  $p$  become president!

## BACKJUMPING

The technique of **backjumping** avoids such pointless repetition. In this particular problem, a backjumping search would backtrack two steps to *Resident*( $p$ , town) and generate a new binding for  $p$ . Discovering where to backjump to at compilation time is easy for a compiler that keeps global dataflow information. Sometimes, in addition to backjumping to a reasonable spot, the system will cache the combination of variables that lead to the dead end, so that they will not be repeated again in another branch of the search. This is called **dependency-directed backtracking**. In practice, the overhead of storing all the dead ends is usually too great—as with heuristic search, memory is often a stronger constraint than time. In practice, there are many more backjumping systems than full dependency-directed backtracking systems.

## DEPENDENCY-DIRECTED BACKTRACKING

The final kind of metareasoning is the most complicated: being able to remember a previously computed inference rather than having to derive it all over again. This is important because most logical reasoning systems are given a series of related queries. For example, a logic-based agent repeatedly ASKS its knowledge base the question "what should I do now?" Answering this question will involve subgoals that are similar or identical to ones answered the previous time around. The agent could just store every conclusion that it is able to prove, but this would soon exhaust memory. There must be some guidance to decide which conclusions are

worth storing and which should be ignored, either because they are easy to recompute or because they are unlikely to be asked again. Chapter 21 discusses these issues in the general context of an agent trying to take advantage of its previous reasoning experiences.

## 10.4 THEOREM PROVERS

Theorem provers (also known as automated reasoners) differ from logic programming languages in two ways. First, most logic programming languages only handle Horn clauses, whereas theorem provers accept full first-order logic. Second, Prolog programs intertwine logic and control. The programmer's choice in writing  $A \Leftarrow B \wedge C$  instead of  $A \Leftarrow C \wedge B$  affects the execution of the program. In theorem provers, the user can write either of these, or another form such as  $\neg B \Leftarrow C \wedge \neg A$ , and the results will be exactly the same. Theorem provers still need control information to operate efficiently, but this information is kept distinct from the knowledge base, rather than being part of the knowledge representation itself. Most of the research in theorem provers is in finding control strategies that are generally useful. In Section 9.6 we covered three generic strategies: unit preference, linear input resolution, and set of support.

### Design of a theorem prover

In this section, we describe the theorem prover OTTER (Organized Techniques for Theorem-proving and Effective Research) (McCune, 1992), with particular attention to its control strategy. In preparing a problem for OTTER, the user must divide the knowledge into four parts:

- A set of clauses known as the **set of support** (or *sos*), which defines the important facts about the problem. Every resolution step resolves a member of the set of support against another axiom, so the search is focused on the set of support.
- A **set of usable axioms** that are outside the set of support. These provide background knowledge about the problem area. The boundary between what is part of the problem (and thus in *sos*) and what is background (and thus in the usable axioms) is up to the user's judgment.
- A set of equations known as **rewrites** or **demodulators**. Although demodulators are equations, they are always applied in the left to right direction. Thus, they define a canonical form into which all terms will be simplified. For example, the demodulator  $x + 0 = x$  says that every term of the form  $x + 0$  should be replaced by the term  $x$ .
- A set of parameters and clauses that defines the control strategy. In particular, the user specifies a heuristic function to control the search and a filtering function that eliminates some subgoals as uninteresting.

OTTER works by continually resolving an element of the set of support against one of the usable axioms. Unlike Prolog, it uses a form of best-first search. Its heuristic function measures the "weight" of each clause, where lighter clauses are preferred. The exact choice of heuristic is up to the user, but generally, the weight of a clause should be correlated with its size and/or difficulty.

Unit clauses are usually treated as very light, so that the search can be seen as a generalization of the unit preference strategy. At each step, OTTER moves the "lightest" clause in the set of support to the usable list, and adds to the usable list some immediate consequences of resolving the lightest clause with elements of the usable list. OTTER halts when it has found a refutation or when there are no more clauses in the set of support. The algorithm is shown in more detail in Figure 10.5.

```

procedure OTTER(sos, usable)
  inputs: sos, a set of support—clauses defining the problem (a global variable)
           usable, background knowledge potentially relevant to the problem

  repeat
    clause  $\leftarrow$  the lightest member of sos
    move clause from sos to usable
    PROCESS(INFER(clause, usable), sos)
  until sos - [ ] or a refutation has been found

function INFER(clause, usable) returns clauses
  resolve clause with each member of usable
  return the resulting clauses after applying FILTER

procedure PROCESS(clauses, sos)
  for each clause in clauses do
    clause  $\leftarrow$  SIMPLIFY(clause)
    merge identical literals
    discard clause if it is a tautology
    sos  $\leftarrow$  [clause | sos]
    if clause has no literals then a refutation has been found
    if clause has one literal then look for unit refutation
  end
```

**Figure 10.5** Sketch of the OTTER theorem prover. Heuristic control is applied in the selection of the "lightest" clause, and in the FILTER function that eliminates uninteresting clauses from consideration.

## Extending Prolog

An alternative way to build a theorem prover is to start with a Prolog compiler and extend it to get a sound and complete reasoner for full first-order logic. This was the approach taken in the Prolog Technology Theorem Prover, or PTTP (Stickel, 1988). PTTP includes five significant changes to Prolog to restore completeness and expressiveness:

- The occurs check is put back into the unification routine to make it sound.

LOCKING

- The depth-first search is replaced by an iterative deepening search. This makes the search strategy complete and takes only a constant factor more time.
- Negated literals (such as  $\neg P(x)$ ) are allowed. In the implementation, there are two separate routines, one trying to prove  $P$  and one trying to prove  $\neg P$ .
- A clause with  $n$  atoms is stored as  $n$  different rules. For example,  $A \Leftarrow B A C$  would also be stored as  $\neg B \Leftarrow C A \neg A$  and as  $\neg C \Leftarrow B A \neg A$ . This technique, known as **locking**, means that the current goal need only be unified with the head of each clause but still allows for proper handling of negation.
- Inference is made complete (even for non-Horn clauses) by the addition of the linear input resolution rule: If the current goal unifies with the negation of one of the goals on the stack, then the goal can be considered solved. This is a way of reasoning by contradiction. Suppose we are trying to prove  $P$  and that the current goal is  $\neg P$ . This is equivalent to saying that  $\neg P \Rightarrow P$ , which entails  $P$ .

Despite these changes, PTTP retains the features that make Prolog fast. Unifications are still done by modifying variables directly, with unbinding done by unwinding the trail during backtracking. The search strategy is still based on input resolution, meaning that every resolution is against one of the clauses given in the original statement of the problem (rather than a derived clause). This makes it feasible to compile all the clauses in the original statement of the problem.

The main drawback of PTTP is that the user has to relinquish all control over the search for solutions. Each inference rule is used by the system both in its original form and in the contrapositive form. This can lead to unintuitive searches. For example, suppose we had the rule

$$(f(x,y) = f(a,b)) \Leftarrow (x = a) \wedge (y = b)$$

As a Prolog rule, this is a reasonable way to prove that two  $f$  terms are equal. But PTTP would also generate the contrapositive:

$$(x \neq a) \Leftarrow (f(x,y) \neq f(a,b)) \wedge (y = b)$$

It seems that this is a wasteful way to prove that any two terms  $x$  and  $a$  are different.

PROOF-CHECKER

SOCRATIC  
REASONER

## Theorem provers as assistants

So far, we have thought of a reasoning system as an independent agent that has to make decisions and act on its own. Another use of theorem provers is as an assistant, providing advice to, say, a mathematician. In this mode the mathematician acts as a supervisor, mapping out the strategy for determining what to do next and asking the theorem prover to fill in the details. This alleviates the problem of semi-decidability to some extent, because the supervisor can cancel a query and try another approach if the query is taking too much time. A theorem prover can also act as a **proof-checker**, where the proof is given by a human as a series of fairly large steps; the individual inferences required to show that each step is sound are filled in by the system.

A **Socratic reasoner** is a theorem prover whose ASK function is incomplete, but which can always arrive at a solution if asked the right series of questions. Thus, Socratic reasoners make good assistants, provided there is a supervisor to make the right series of calls to ASK. ONTIC (McAllester, 1989) is an example of a Socratic reasoning system for mathematics.

of an agent—on each cycle, we add the percepts to the knowledge base and run the forward chainer, which chooses an action to perform according to a set of condition-action rules.

Theoretically, we could implement a production system with a theorem prover, using resolution to do forward chaining over a full first-order knowledge base. A more restricted language, on the other hand, can provide greater efficiency because the branching factor is reduced. The typical production system has these features:

- The system maintains a knowledge base called the **working memory**. This contains a set of positive literals with no variables.
- The system also maintains a separate **rule memory**. This contains a set of inference rules, each of the form  $p_1 \wedge p_2 \wedge \dots \Rightarrow act_1 \wedge act_2 \wedge \dots$ , where the  $p_i$  are literals, and the  $act_i$  are actions to take when the  $p_i$  are all satisfied. Allowable actions are adding and deleting elements from the working memory, and possibly other actions (such as printing a value).
- In each cycle, the system computes the subset of rules whose left-hand side is satisfied by the current contents of the working memory. This is called the **match phase**.
- The system then decides which of the rules should be executed. This is called the **conflict resolution phase**.
- The final step in each cycle is to execute the action(s) in the chosen rule(s). This is called the **act phase**.

## Match phase

Unification addresses the problem of matching a pair of literals, where either literal can contain variables. We can use unification in a straightforward way to implement a forward-chaining production system, but this is very inefficient. If there are  $w$  elements in working memory and  $r$  rules each with  $n$  elements in the left-hand side, and solving a problem requires  $c$  cycles, then the naive match algorithm must perform  $wrnc$  unifications. A simple expert system might have  $w = 100, r = 200, n = 5, c = 1000$ , so this is a hundred million unifications. The **rete** algorithm<sup>4</sup> used in the OPS-5 production system was the first to seriously address this problem. The rete algorithm is best explained by example. Suppose we have the following rule memory:

$$\begin{aligned} A(x) \wedge B(x) \wedge C(y) &\Rightarrow \text{add } D(x) \\ A(x) \wedge \text{BOO } A \wedge D(x) &\Rightarrow \text{add } E(x) \\ A(x) \wedge B(x) \wedge A \wedge E(x) &\Rightarrow \text{delete } A(x) \end{aligned}$$

and the following working memory:

$$\{A(1), A(2), B(2), B(3), B(4), C(5)\}$$

The rete algorithm first compiles the rule memory into the network shown in Figure 10.6. In this diagram, the circular nodes represent fetches (not unifications) to working memory. Under node A, the working memory elements A(1) and A(2) are fetched and stored. The square nodes indicate unifications. Of the six possible  $A \times B$  combinations at the  $A = B$  node, only A(2) and B(2) satisfy the unification. Finally, rectangular boxes indicate actions. With the initial working

<sup>4</sup> Rete is Latin for net. It rhymes with treaty.

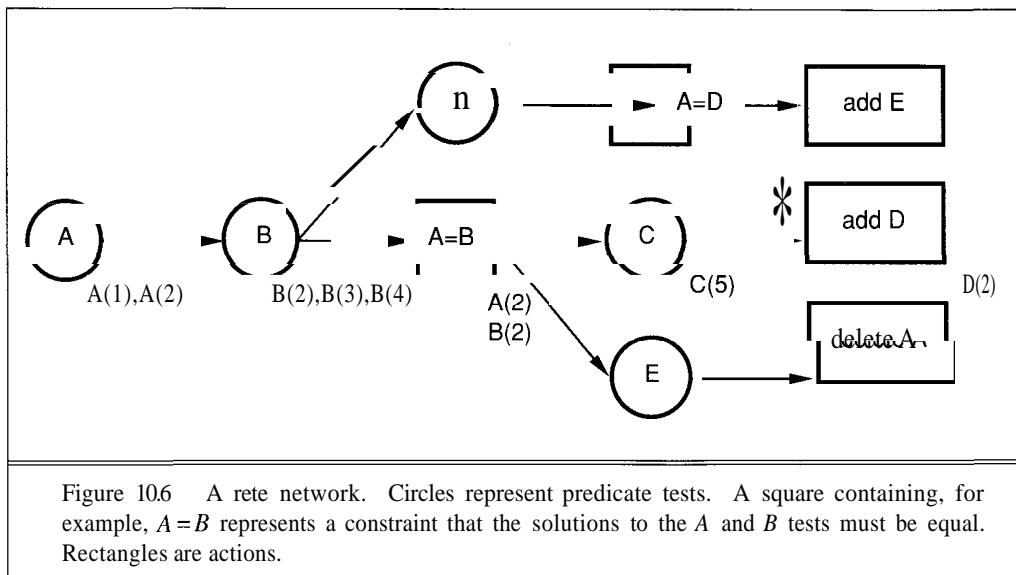


Figure 10.6 A rete network. Circles represent predicate tests. A square containing, for example,  $A = B$  represents a constraint that the solutions to the  $A$  and  $B$  tests must be equal. Rectangles are actions.

memory the “*add D*” rule is the only one that fires, resulting in the addition of the sentence  $D(2)$  to working memory.

One obvious advantage of the rete network is that it eliminates duplication between rules. All three of the rules start with a conjunction of  $A$  and  $B$ , and the network allows that part to be shared. The second advantage of rete networks is in eliminating duplication over time. Most production systems make only a few changes to the knowledge base on each cycle. This means that most of the tests at cycle  $t+1$  will give the same result as at cycle  $t$ . The rete network modifies itself after each addition or deletion, but if there are few changes, then the cost of each update will be small relative to the whole job of maintaining the indexing information. The network thus represents the saved intermediate state in the process of testing for satisfaction of a set of conjuncts. In this case, adding  $D(2)$  will result in the activation of the “*add E*” rule, but will not have any effect on the rest of the network. Adding or deleting an  $A$ , however, will have a bigger effect that needs to be propagated through much of the network.

## Conflict resolution phase

Some production systems execute the actions of all rules that pass the match phase. Other production systems treat these rules only as suggestions, and use the conflict resolution phase to decide which of the suggestions to accept. This phase can be thought of as the control strategy. Some of the strategies that have been used are as follows:

- *No duplication.* Do not execute the same rule on the same arguments twice.
- *Recency.* Prefer rules that refer to recently created working memory elements.

- *Specificity.* Prefer rules that are more specific.<sup>5</sup> For example, the second of these two rules would be preferred:

$Mammal(x) \Rightarrow \text{add Legs}(x, 4)$

$Mammal(x) \wedge Human(x) \Rightarrow \text{add Legs}(x, 2)$

- *Operation priority.* Prefer actions with higher priority, as specified by some ranking. For example, the second of the following two rules should probably have a higher priority:

$ControlPanel(p) \wedge Dusty(p) \Rightarrow Action(Dust(p))$

$ControlPanel(p) \wedge MeltdownLightOn(p) \Rightarrow Action(Evacuate)$

## Practical uses of production systems

Forward-chaining production systems formed the foundation of much early work in AI. In particular, the XCON system (originally called R1 (McDermott, 1982)) was built using a production system (rule-based) architecture. XCON contains several thousand rules for designing configurations of computer components for customers of the Digital Equipment Corporation. It was one of the first clear commercial successes in the emerging field of expert systems. Many other similar systems have been built using the same underlying technology, which has been implemented in the general-purpose language OPS-5. A good deal of work has gone into designing matching algorithms for production system languages, as we have seen; implementations on parallel hardware have also been attempted (Acharya *et al.*, 1992).

ARCHITECTURES

Production systems are also popular in **cognitive architectures**—that is, models of human reasoning—such as ACT (Anderson, 1983) and SOAR (Laird *et al.*, 1987). In such systems, the "working memory" of the system models human short-term memory, and the productions are part of long-term memory. Both ACT and SOAR have sophisticated mechanisms for conflict resolution, and for saving the results of expensive reasoning in the form of new productions. These can be used to avoid reasoning in future situations (see also Section 21.2).

## 10.6 FRAME SYSTEMS AND SEMANTIC NETWORKS

EXISTENTIAL  
GRAPHS

In 1896, seven years after Peano developed what is now the standard notation for first-order logic, Charles Peirce proposed a graphical notation called **existential graphs** that he called "the logic of the future." Thus began a long-running debate between advocates of "logic" and advocates of "semantic networks." What is unfortunate about this debate is that it obscured the underlying unity of the field. It is now accepted that every semantic network or frame system could just as well have been defined as sentences in a logic, and most accept that it could be first-order logic.<sup>6</sup> (We will show how to execute this translation in detail.) The important thing with any representation language is to understand the semantics, and the proof theory; the details of the syntax are less important. *Whether the language uses strings or nodes and links, and whether it*

<sup>5</sup> For more on the use of specificity to implement default reasoning, see Chapter 14.

<sup>6</sup> There are a few problems having to do with handling exceptions, but they too can be handled with a little care.



*is called a semantic network or a logic, has no effect on its meaning or on its implementation.*

Having said this, we should also say that the format of a language can have a *significant* effect on its clarity for a human reader. Some things are easier to understand in a graphical notation; some are better shown as strings of characters. Fortunately, there is no need to choose one or the other; the skilled AI practitioner can translate back and forth between notations, choosing the one that is best for the task at hand, but drawing intuitions from other notations. Some systems, such as the CYC system mentioned in Chapter 8, provide both kinds of interfaces.

Besides the appeal of pretty node-and-link diagrams, semantic networks have been successful for the same reason that Prolog is more successful than full first-order logic theorem provers: because most semantic network formalisms have a very simple execution model. Programmers can build a large network and still have a good idea about what queries will be efficient, because (a) it is easy to visualize the steps that the inference procedure will go through, and (b) the query language is so simple that difficult queries cannot be posed. This may be the reason why many of the pioneering researchers in commonsense ontology felt more comfortable developing their theories with the semantic network approach.

## Syntax and semantics of semantic networks

Semantic networks concentrate on categories of objects and the relations between them. It is very natural to draw the link

*Cats* *Subset* *Mammals*

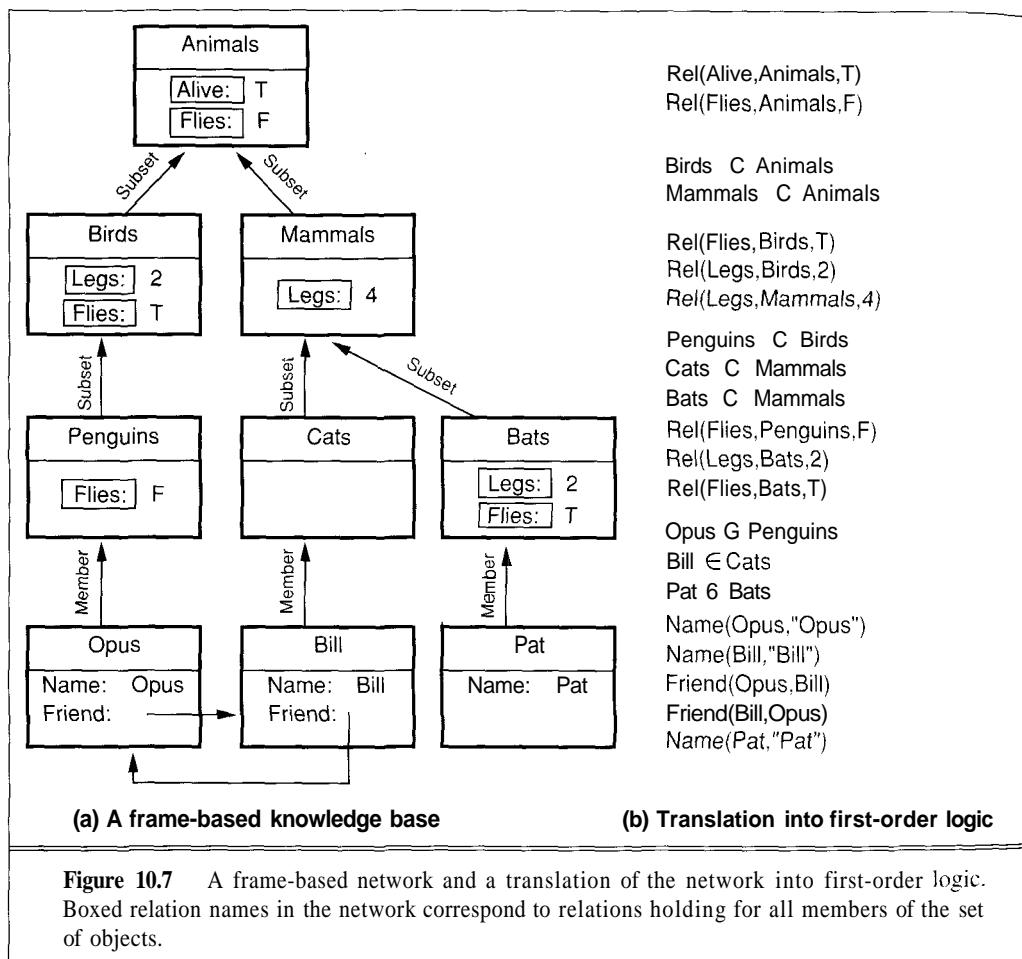
to say that cats are mammals. Of course, it is just as easy to write the logical sentence *Cats C Mammals*, but when semantic networks were first used in AI (around 1961), this was not widely appreciated; people thought that in order to use logic they would have to write

$\forall x \text{ Cat}(x) \Rightarrow \text{Mammal}(x)$

which seemed a lot more intimidating. It was also felt that  $\forall x$  did not allow exceptions, but that *Subset* was somehow more forgiving.<sup>7</sup>

We now recognize that semantics is more important than notation. Figure 10.7 gives an example of a typical frame-based network, and a translation of the network into first-order logic. This network can be used to answer the query "How many legs does Opus have?" by following the chain of *Member* and *Subset* links from Opus to Penguins to Birds, and seeing that birds have two legs. This is an example of **inheritance**, as described in Section 8.4. That is clear enough, but what happens, when, say, there are two different chains to two different numbers of legs? Ironically, semantic networks sometimes lack a clear semantics. Often the user is left to induce the semantics of the language from the behavior of the program that implements it. Consequently, users often think of semantic networks at the implementation level rather than the logical level or the knowledge level.

<sup>7</sup> In many systems, the name *IsA* was given to both subset and set-membership links, in correspondence with English usage: "a cat is a mammal" and "Fifi is a cat." This can lead directly to inconsistencies, as pointed out by Drew McDermott (1976) in his article "Artificial Intelligence Meets Natural Stupidity." Some systems also failed to distinguish between properties of members of a category and properties of the category as a whole.



**Figure 10.7** A frame-based network and a translation of the network into first-order logic. Boxed relation names in the network correspond to relations holding for all members of the set of objects.

The semantics of a simple semantic network language can be stated by providing first-order logical equivalents for assertions in the network language. We first define a version in which exceptions are not allowed. In addition to *Subset* and *Member* links, we find that there is a need for at least three other kinds of links: one that says a relation  $R$  holds between two objects,  $A$  and  $B$ ; one that says  $R$  holds between every element of the class  $A$  and the object  $B$ ; and one that says that  $R$  holds between every element of  $A$  and some element of  $B$ . The five standard link types are summarized in Figure 10.8.<sup>8</sup> Notice that a theorem prover or logic programming language could take the logical translations of the links and do inheritance by ordinary logical inference. A semantic network system uses special-purpose algorithms for following links, and therefore can be faster than general logical inference.

<sup>8</sup> Because assertions of the form  $A \sqsubseteq R B$  are so common, we use the abbreviation  $\text{Rel}(R, A, B)$  as syntactic sugar in the logical translation (Figure 10.7).

Link Type	Semantics	Example
$A \xrightarrow{\text{Subset}} B$	$ACB$	$Cats \subset C \text{ Mammals}$
$A \xrightarrow{\text{Member}} B$	$A \in B$	$Bill \in G \text{ Cats}$
$A \xrightarrow{R} B$	$R(A, B)$	$Bill \xrightarrow{\text{Age}} 12$
$A \boxed{R} B$	$\forall x \ x \in A \Rightarrow R(x, B)$	$Birds \boxed{\xrightarrow{\text{Legs}}} 2$
$A \boxed{\boxed{R}} B$	$\forall x \ \exists y \ x \in A \Rightarrow y \in B \wedge R(x, y)$	$Birds \boxed{\boxed{\xrightarrow{\text{Parent}}}} Birds$

Figure 10.8 Link types in semantic networks, and their meanings.

## Inheritance with exceptions

DEFAULT VALUE

As we saw in Chapter 8, natural kinds are full of exceptions. The diagram in Figure 10.7 says that mammals have 4 legs, but it also says that bats, which are mammals, have 2 legs. According to the straightforward logical semantics, this is a contradiction. To fix the problem, we will change the semantic translation of a boxed- $R$  link from  $A$  to  $B$  to mean that every member of  $A$  must have an  $R$  relation to  $B$  unless there is some intervening  $A'$  for which  $Rel(R, A', B')$ . Then Figure 10.7 will unambiguously mean that bats have 2 legs, not 4. Notice that  $Rel(R, A, B)$  no longer means that every  $A$  is related by  $R$  to  $B$ ; instead it means that  $B$  is a **default value** for the  $R$  relation for members of  $A$ , but the default can be overridden by other information.

It may be intuitive to think of inheritance with exceptions by following links in a diagram, but it is also possible—and instructive—to define the semantics in first-order logic. The first step in the logical translation is to **reify** relations: a relation  $R$  becomes an object, not a predicate. That means that  $Rel(R, A, B)$  is just an ordinary atomic sentence, not an abbreviation for a complex sentence. It also means that we can no longer write  $R(x, B)$ , because  $R$  is an object, not a predicate. We will use  $Val(R, x, B)$  to mean that the equivalent of an  $R(x, B)$  relation is explicitly asserted in the semantic network, and  $Holds(R, x, B)$  to mean that  $R(x, B)$  can be inferred. We then can define *Holds* by saying that a relation  $R$  holds between  $x$  and  $b$  if either there is an explicit *Val* predication or there is a *Rel* on some parent class  $p$  of which  $x$  is an element, and there is no *Rel* on any intervening class  $i$ . (A class  $i$  is intervening if  $x$  is an element of  $i$  and  $i$  is a subset of  $p$ .) In other words:

$$\begin{aligned} \forall r, x, b \ Holds(r, x, b) &\Leftrightarrow \\ Val(r, x, b) \vee (\exists p \ x \in p \ A \ Rel(r, p, b) \ A \ \neg InterveningRel(x, p, r)) \\ \forall x, p, r \ InterveningRel(x, p, r) &\Leftrightarrow \\ \exists i \ Intervening(x, i, p) \ A \ \exists b' \ Rel(r, i, b') \\ \forall a, i, p \ Intervening(x, i, p) &\Leftrightarrow (x \in i) \wedge (i \subset p) \end{aligned}$$

Note that the  $C$  symbol means proper subset (e.g.,  $i \subset p$  means that  $i$  is a subset of  $p$  and is not equal to  $p$ ). The next step is to recognize that it is important not only to know what *Rel* and *Val* relations hold, but also what ones *do not* hold. Suppose we are trying to find the  $n$  that satisfies *Holds(Legs, Opus, n)*. We know *Rel(Legs, Birds, 2)* and we know Opus is a bird, but the definition of *Holds* does not allow us to infer anything unless we can prove there is no *Rel(Legs, i, b)* for  $i = Penguins$  or any other intervening category. If the knowledge base only

contains positive *Rel* atoms (i.e., *Rel(Legs, Birds, 2)* A *Rel(Flies, Birds, T)*), then we are stuck. Therefore, the translation of a semantic network like Figure 10.7 should include sentences that say that the *Rel* and *Val* relations that are shown are the only ones that are true:

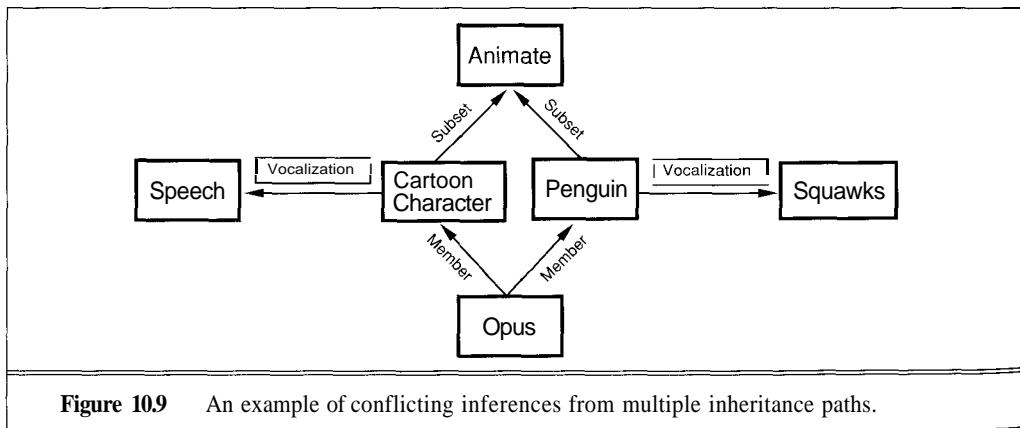
$$\begin{aligned}\forall r, a, b \ Rel(r, a, b) &\Leftrightarrow [r, a, b] \in \{[\text{Alive}, \text{Animal}, T], [\text{Flies}, \text{Animal}, F], \dots\} \\ \forall r, a, b \ Val(r, a, b) &\Leftrightarrow [r, a, b] \notin \{[\text{Friend}, \text{Opus}, \text{Bill}], [\text{Friend}, \text{Bill}, \text{Opus}], \dots\}\end{aligned}$$

## Multiple inheritance

MULTIPLE  
INHERITANCE

Some semantic network systems allow **multiple inheritance**—that is, an object can belong to more than one category and can therefore inherit properties along several different paths. In some cases, this will work fine. For example, some people might belong to both the categories *Billionaire* and *PoloPlayer*, in which case we can infer that they are rich and can ride a horse.

It is possible, however, for two inheritance paths to produce conflicting answers. An example of this difficulty is shown in Figure 10.9. Opus is a penguin, and therefore speaks only in squawks. Opus is a cartoon character, and therefore speaks English.<sup>9</sup> In the simple logical translation given earlier, we would be able to infer both conclusions, which, with appropriate background knowledge, would lead to a contradiction. Without additional information indicating some preference for one path, there is no way to resolve the conflict.



**Figure 10.9** An example of conflicting inferences from multiple inheritance paths.

## Inheritance and change

A knowledge base is not of much use to an agent unless it can be expanded. In systems based on first-order logic, we use **TELL** to add a new sentence to the knowledge base, and we enjoy the

<sup>9</sup> The classical example of multiple inheritance conflict is called the "Nixon diamond." It arises from the observation that Richard Nixon was both a Quaker (and hence a pacifist) and a Republican (and hence not a pacifist). Because of its potential for controversy, we will avoid this particular example. The other canonical example involves a bird called Tweety, about whom the less said the better.

property of **monotonicity**: if  $P$  follows from  $KB$ , then it still follows when  $KB$  is augmented by  $\text{TELL}(KB, S)$ . In other words,

**if**  $KB \vdash P$  **then**  $(KB \text{ A } S) \vdash P$

NONMONOTONIC

Inheritance with exceptions is **nonmonotonic**: from the semantic network in Figure 10.7 it follows that Bill has 4 legs, but if we were to add the new statement  $\text{Rel}(\text{Legs}, \text{Cats}, 3)$ , then it no longer follows that Bill has 4 legs. There are two ways to deal with this.

First, we could switch from first-order logic to a **nonmonotonic logic** that explicitly deals with default values. Nonmonotonic logics allow you to say that a proposition  $P$  should be treated as true until additional evidence allows you to prove that  $P$  is false. There has been quite a lot of interesting theoretical work in this area, but so far its impact on practical systems has been smaller than other approaches, so we will not address it in this chapter.

Second, we could treat the addition of the new statement as a RETRACT followed by a TELL. Given the way we have defined  $\text{Rel}$ , this makes perfect sense. We do not make statements of the form  $\text{TELL}(KB, \text{Rel}(R, A, B))$ . Instead, we make one big equivalence statement of the form

$\text{TELL}(KB, \forall r, a, b \text{ Rel}(r, a, b) \Leftrightarrow \dots)$

where the  $\dots$  indicate all the possible  $\text{Rel}$ 's. So to add  $\text{Rel}(\text{Legs}, \text{Cats}, 3)$ , we would have to remove the old equivalence statement and replace it by a new one. Once we have altered the knowledge base by removing a sentence from it (and not just adding a new one) we should not be surprised at the nonmonotonicity. Section 10.8 discusses implementations of RETRACT.

## Implementation of semantic networks

Once we have decided on a meaning for our networks, we can start to implement the network. Of course, we could choose to implement the network with a theorem prover or logic programming language, and in some cases this would be the best choice. But for networks with simple semantics, a more straightforward implementation is possible. A node in a network is represented by a data structure with fields for the basic taxonomic connections: which categories it is a member of; what elements it has; what immediate subsets and supersets. It also has fields for other relations in which it participates. The RELS-IN and RELS-OUT fields handle ordinary (unboxed) links, and the ALL-RELS-IN and ALL-RELS-OUT fields handle boxed links. Here is the data type definition for nodes:

```
datatype SEM-NET-NODE
  components: NAME, MEMBERSHIPS, ELEMENTS, SUPERS, SUBS,
             RELS-IN, RELS-OUT, ALL-RELS-IN, ALL-RELS-OUT
```

Each of the four REL-fields is organized as a table indexed by the relation. We use the function  $\text{LOOKUP}(key, table)$  to find the value associated with a key in a table. So, if we have the two links  $Opus \xrightarrow{\text{Friend}} Bill$  and  $Opus \xrightarrow{\text{Friend}} Steve$ , then  $\text{LOOKUP}(\text{Friend}, \text{RELS-OUT}(Opus))$  gives us the set  $\{Bill, Steve\}$ .

The code in Figure 10.10 implements everything you need in order to ASK the network whether subset, membership, or other relations hold between two objects. Each of the functions simply follows the appropriate links until it finds what it is looking for, or runs out of links. The code does not handle double-boxed links, nor does it handle exceptions. Also, the code that TELLS the network about new relations is not shown, because it is straightforward.

The code can be extended with other functions to answer other questions. One problem with this approach is that it is easy to become carried away with the data structures and forget their underlying semantics. For example, we could easily define a NUMBER-OF-SUBKINDS function that returns the length of the list in the SUBS slot. For Figure 10.7, NUMBER-OF-SUBKINDS(*Animal*) = 2. This may well be the answer the user wanted, but its logical status is dubious. First of all, it

```

function MEMBER?(element,category) returns True or False
  for each c in MEMBERSHIPS[element] do
    if SUBSET?(c, category) then return True
  return False

function SUBSET?(sub, super) returns True or False
  if sub = super then return True
  for each c in SUPERS[sub] do
    if SUBSET?(c, super) then return True
  return False

function RELATED-TO?(source,relation, destination) returns True or False
  if relation appears in RELS-OUT(source) then
    return MEMBER([relation,destination], RELS-OUT(node))
  else for each c in MEMBERSHIPS(source) do
    if ALL-RELATED-TO?(c, relation, destination) then return True
  end
  return False

function ALL-RELATED-TO?(source, relation, destination) returns True or False
  if relation appears in ALL-RELS-OUT(source) then
    return MEMBER([relation,destination], ALL-RELS-OUT(node))
  else for each c in SUPERS(category) do
    if ALL-RELATED-TO?(c, relation, destination) then return True
  end
  return False

```

**Figure 10.10** Basic routines for inheritance and relation testing in a simple exception-free semantic network. Note that the function MEMBER? is defined here to operate on semantic network nodes, while the function MEMBER is a utility that operates on sets.

is likely that there are species of animals that are not represented in the knowledge base. Second, it may be that some nodes denote the same object. Perhaps *Dog* and *Chien* are two nodes with an equality link between them. Do these count as one or two? Finally, is *Dog-With-Black-Ears* a kind of animal? How about *Dog-On-My-Block-Last-Thursday*? It is easy to answer these questions based on what is stored in the knowledge base, but it is better to have a clear semantics so that the questions can be answered about the world, rather than about the current state of the internal representation.

## Expressiveness of semantic networks

The networks we have discussed so far are extremely limited in their expressiveness. For example, it is not possible to represent negation (Opus does not ride a bicycle), disjunction (Opus appears in either the *Times* or the *Dispatch*), or quantification (all of Opus' friends are cartoon characters). These constructs are essential in many domains.

Some semantic networks extend the notation to allow all of first-order logic. Peirce's original existential graphs, **partitioned semantic networks** (Hendrix, 1975), and SNEPS (Shapiro, 1979) all take this approach. A more common approach retains the limitations on expressiveness and uses **procedural attachment** to fill in the gaps. Procedural attachment is a technique where a function written in a programming language can be stored as the value of some relation, and used to answer ASK calls about that relation (and sometimes TELL calls as well).

What do semantic networks provide in return for giving up expressiveness? We have already seen two advantages: they are able to capture inheritance information in a modular way, and their simplicity makes them easy to understand. Efficiency is often claimed as a third advantage: because inference is done by following links, rather than retrieving sentences from a knowledge base and performing unifications, it can operate with only a few machine cycles per inference step. But if we look at the kinds of computations done by compiled Prolog programs, we see there is not much difference. A compiled Prolog program for a set of subset and set-membership sentences, combined with general properties of categories, does almost the same computations as a semantic network.

PARTITIONED  
SEMANTIC  
NETWORKS

PROCEDURAL  
ATTACHMENT

## 10.7 DESCRIPTION LOGICS



SUBSUMPTION  
CLASSIFICATION

The syntax of first-order logic is designed to make it easy to say things about objects. *Description logics are designed to focus on categories and their definitions*. They provide a reasonably sophisticated facility for defining categories in terms of existing relations, with much greater expressiveness than typical semantic network languages. The principal inference tasks are **subsumption**—checking if one category is a subset of another based on their definitions—and **classification**—checking if an object belongs to a category. In some description logics, objects are also viewed as categories defined by the object's description and (presumably) containing only one member. This way of looking at representation is a significant departure from the object-centered view that is most compatible with first-order logical syntax.

The CLASSIC language (Borgida *et al.*, 1989) is a typical description logic. The syntax of CLASSIC descriptions is shown in Figure 10.11.<sup>10</sup> For example, to say that bachelors are unmarried, adult males we would write

$$\text{Bachelor} = \text{And}(\text{Unmarried}, \text{Adult}, \text{Male})$$

The equivalent in first-order logic would be

$$\forall x \text{ Bachelor}(x) \Leftrightarrow \text{Unmarried}(x) \wedge \text{Adult}(x) \wedge \text{Male}(x)$$

Notice that the description logic effectively allows direct logical operations on predicates, rather than having to first create sentences to be joined by connectives. Any description in CLASSIC can be written in first-order logic, but some descriptions are more straightforward in CLASSIC. For example, to describe the set of men with at least three sons who are all unemployed and married to doctors, and at most two daughters who are all professors in physics or chemistry departments, we would use

$$\begin{aligned} & \text{And}(\text{Man}, \text{AtLeast}(3, \text{Son}), \text{AtMost}(2, \text{Daughter}), \\ & \quad \text{All}(\text{Son}, \text{And}(\text{Unemployed}, \text{Married}, \text{All}(\text{Spouse}, \text{Doctor}))), \\ & \quad \text{All}(\text{Daughter}, \text{And}(\text{Professor}, \text{Fills}(\text{Department}, \text{Physics}, \text{Chemistry})))) \end{aligned}$$

We leave it as an exercise to translate this into first-order logic.

$\text{Concept} \equiv \text{Thing} \mid \text{ConceptName}$ $\mid \text{And}(\text{Concept}, \dots)$ $\mid \text{All}(\text{RoleName}, \text{Concept})$ $\mid \text{AtLeast}(\text{Integer}, \text{RoleName})$ $\mid \text{AtMost}(\text{Integer}, \text{RoleName})$ $\mid \text{Fills}(\text{Role Name}, \text{Individual Name}, \dots)$ $\mid \text{SameAs}(\text{Path}, \text{Path})$ $\mid \text{OneOf}(\text{Individual Name}, \dots)$
$\text{Path} \rightarrow [\text{RoleName}, \dots]$

**Figure 10.11** The syntax of descriptions in a subset of the CLASSIC language.

Perhaps the most important aspect of description logics is the emphasis on tractability of inference. A problem instance is solved by describing it and asking if it is subsumed by one of several possible solution categories. In standard first-order logic systems, predicting the solution time is often impossible. It is often left to the user to engineer the representation to detour around sets of sentences that seem to be causing the system to take several weeks to solve a problem. The thrust in description logics, on the other hand, is to ensure that subsumption-testing can be solved in time polynomial in the size of the problem description. The CLASSIC language satisfies this condition, and is currently the most comprehensive language to do so.

<sup>10</sup> Notice that the language does *not* allow one to simply state that one concept, or category, is a subset of another. This is a deliberate policy: subsumption between categories must be derivable from some aspects of the descriptions of the categories. If not, then something is missing from the descriptions.

This sounds wonderful in principle, until one realizes that it can only have one of two consequences: hard problems either cannot be stated at all, or require exponentially large descriptions! However, the tractability results do shed light on what sorts of constructs cause problems, and thus help the user to understand how different representations behave. For example, description logics usually lack *negation* and *disjunction*. These both force first-order logical systems to essentially go through an exponential case analysis in order to ensure completeness. For the same reason, they are excluded from Prolog. CLASSIC only allows a limited form of disjunction in the *Fills* and *OneOf* constructs, which allow disjunction over explicitly enumerated individuals but not over descriptions. With disjunctive descriptions, nested definitions can lead easily to an exponential number of alternative routes by which one category can subsume another.

### Practical uses of description logics

Because they combine clear semantics and simple logical operations, description logics have become popular with both the theoretical and practical AI communities. Applications have included financial management (Mays *et al.*, 1987), database interfaces (Beck *et al.*, 1989), and software information systems (Devanbu *et al.*, 1991). Because of the gradual extension of the class of tractable languages, and a better understanding of what kinds of constructs cause intractability, the efficiency of description logic systems has improved by several orders of magnitude over the last decade.

## 10.8 MANAGING RETRACTIONS, ASSUMPTIONS, AND EXPLANATIONS

We have said a great deal about TELL and ASK, but so far very little about RETRACT. Most logical reasoning systems, regardless of their implementation, have to deal with RETRACT. As we have seen, there are three reasons for retracting a sentence. It may be that a fact is no longer important, and we want to forget about it to free up space for other purposes. It may be that the system is tracking the current state of the world (without worrying about past situations) and that the world changes. Or it may be that the system assumed (or determined) that a fact was true, but now wants to assume (or comes to determine) that it is actually false. In any case, we want to be able to retract a sentence from the knowledge base without introducing any inconsistencies, and we would like the interaction with the knowledge base as a whole (the cycle of TELL, ASK and RETRACT requests) to be efficient.

It takes a little experience to appreciate the problem. First, it is important to understand the distinction between  $\text{RETRACT}(KB, P)$  and  $\text{TELL}(KB, \neg P)$ . Assuming that the knowledge base already contains  $P$ , adding  $\neg P$  with TELL will allow us to conclude both  $P$  and  $\neg P$ , whereas removing  $P$  with RETRACT will allow us to conclude neither  $P$  nor  $\neg P$ . Second, if the system does any forward chaining, then RETRACT has some extra work to do. Suppose the knowledge base was told  $P$  and  $P \Rightarrow Q$ , and used that to infer  $Q$  and add it to the knowledge base. Then  $\text{RETRACT}(KB, P)$  must remove both  $P$  and  $Q$  to keep the knowledge base consistent. However, if there is some other independent reason for believing  $Q$  (perhaps both  $R$  and  $R \Rightarrow Q$  have been

TRUTH  
MAINTENANCETRUTH  
MAINTENANCE  
SYSTEM

EXPLANATIONS

ASSUMPTIONS

JTMS

asserted), then  $Q$  does not have to be removed after all. The process of keeping track of which additional propositions need to be retracted when we retract  $P$  is called **truth maintenance**.

The simplest approach to truth maintenance is to use chronological backtracking (see page 309). In this approach, we keep track of the order in which sentences are added to the knowledge base by numbering them from  $P_1$  to  $P_n$ . When the call  $\text{RETRACT}(P_i)$  is made, the system reverts to the state just before  $P_i$  was added. If desired, the sentences  $P_{i+1}$  through  $P_n$  can then be added again. This is simple, and it guarantees that the knowledge base will be consistent, but it means that retraction is  $O(n)$ , where  $n$  is the size of the knowledge base. We would prefer a more efficient approach that does not require us to duplicate all the work for  $P_{i+1}$  to  $P_n$ .

A **truth maintenance system** or **TMS** is a program that keeps track of dependencies between sentences so that retraction (and some other operations) will be more efficient. A TMS actually performs four important jobs. First, a TMS enables dependency-directed backtracking, to avoid the inefficiency of chronological backtracking.

A second and equally important job is to provide **explanations** of propositions. A proof is one kind of explanation—if we ask, "Explain why you believe  $P$  is true?" then a proof of  $P$  is a good explanation. If a proof is not possible, then a good explanation is one that involves **assumptions**. For example, if we ask, "Explain why the car won't start," there may not be enough evidence to prove anything, but a good explanation is, "If we assume that there is gas in the car and that it is reaching the cylinders, then the observed absence of activity proves that the electrical system must be at fault." Technically, an explanation  $E$  of a sentence  $P$  is defined as a set of sentences such that  $E$  entails  $P$ . The sentences in  $E$  must either be known to be true (i.e., they are in the knowledge base), or they must be known to be assumptions that the problem-solver has made. To avoid having the whole knowledge base as an explanation, we will insist that  $E$  is minimal, that is, that there is no proper subset of  $E$  that is also an explanation.

The ability to deal with assumptions and explanations is critical for the third job of a TMS: doing default reasoning. In a taxonomic system that allows exceptions, stating that Opus is a penguin does not sanction an irrefutable inference that Opus has two legs, because additional information about Opus might override the derived belief. A TMS can deliver the explanation that Opus, being a penguin, has two legs *provided he is not an abnormal penguin*. Here, the lack of abnormality is made into an explicit assumption. Finally, TMSs help in dealing with inconsistencies. If adding  $P$  to the knowledge base results in a logical contradiction, a TMS can help pinpoint an explanation of what the contradiction is.

There are several types of TMSs. The simplest is the justification-based truth maintenance system or **JTMS**. In a JTMS, each sentence in the knowledge base is annotated with a justification that identifies the sentences from which it was inferred, if any. For example, if  $Q$  is inferred by Modus Ponens from  $P$ , then the set of sentences  $\{P, P \Rightarrow Q\}$  could serve as a justification of the sentence  $Q$ . Some sentences will have more than one justification. Justifications are used to do selective retractions. If after adding  $P_1$  through  $P_n$  we get a call to  $\text{RETRACT}(P_i)$ , then the JTMS will remove from the knowledge base exactly those sentences for which  $P_i$  is a required part of every justification. So, if a sentence  $Q$  had  $\{P_i, P_i \Rightarrow Q\}$  as its only justification, it would be removed; if it had the additional justification  $\{P_i, P_i \vee R \Rightarrow Q\}$ , then it would still be removed; but if it also had the justification  $\{R, P_i \vee R \Rightarrow Q\}$ , then it would be spared.

In most JTMS implementations, it is assumed that sentences that are considered once will probably be considered again, so rather than removing a sentence from the knowledge base when

it loses all justification, we merely mark the sentence as being *out* of the knowledge base. If a subsequent assertion restores one of the justifications, then we mark the sentence as being back *in*. In this way, the JTMS retains all of the inference chains that it uses, and need not rederive sentences when a justification becomes valid again.

To solve the car diagnosis problem with a JTMS, we would first assume (that is, assert) that there is gas in the car and that it is reaching the cylinders. These sentences would be labelled as *in*. Given the right background knowledge, the sentence representing the fact that the car will not start would also become labelled *in*. We could then ask the JTMS for an explanation. On the other hand, if it turned out that the assumptions were not sufficient (i.e., they did not lead to "car won't start" being *in*), then we would retract the original assumptions and make some new ones. We still have a search problem—the TMS does only part of the job.

ATMS

The JTMS was the first type of TMS, but the most popular type is the ATMS or assumption-based truth maintenance system. The difference is that a JTMS represents one consistent state of the world at a time. The maintenance of justifications allows you to quickly move from one state to another by making a few retractions and assertions, but at any time only one state is represented. An ATMS represents *all* the states that have ever been considered at the same time. Whereas a JTMS simply labels each sentence as being *in* or *out*, an ATMS keeps track, for each sentence, of which assumptions would cause the sentence to be true. In other words, each sentence has a label that consists of a set of assumption sets. The sentence holds just in those cases where all the assumptions in one of the assumption sets hold.

To solve problems with an ATMS, we can make assumptions (such as  $P_i$  or "gas in car") in any order we like. Instead of retracting assumptions when one line of reasoning fails, we just assert all the assumptions we are interested in, even if they contradict each other. We then can check a particular sentence to determine the conditions under which it holds. For example, the label on the sentence  $Q$  would be  $\{\{P_i\}, \{R\}\}$ , meaning that  $Q$  is true under the assumption that  $P_i$  is true or under the assumption that  $R$  is true. A sentence that has the empty set as one of its assumption sets is necessarily true—it is true with no assumptions at all. On the other hand, a sentence with no assumption sets is just false.

The algorithms used to implement truth maintenance systems are a little complicated, and we do not cover them here. The computational complexity of the truth maintenance problem is at least as great as that of propositional inference—that is, NP-hard. Therefore, you should not expect truth maintenance to be a panacea (except for trivially small problems). But when used carefully (for example, with an informed choice about what is an assumption and what is a fact that can not be retracted), a TMS can be an important part of a logical system.

## 10.9 SUMMARY

---

This chapter has provided a connection between the conceptual foundations of knowledge representation and reasoning, explained in Chapters 6 through 9, and the practical world of actual reasoning systems. We emphasize that real understanding of these systems can only be obtained by trying them out.

We have described implementation techniques and characteristics of four major classes of logical reasoning systems:

- Logic programming systems and theorem provers.
- Production systems.
- Semantic networks.
- Description logics.

We have seen that there is a trade-off between the expressiveness of the system and its efficiency. Compilation can provide significant improvements in efficiency by taking advantage of the fact that the set of sentences is fixed in advance. Usability is enhanced by providing a clear semantics for the representation language, and by simplifying the execution model so that the user has a good idea of the computations required for inference.

---

## BIBLIOGRAPHICAL AND HISTORICAL NOTES

Work on indexing and retrieval in knowledge bases appears in the literatures of both AI and databases. The two major texts on AI programming (Charniak *et al.*, 1987; Norvig, 1992) discuss the topic in depth. The text by Forbus and de Kleer (1993) also covers much of this ground. The standard reference on management of databases and knowledge bases is (Ullman, 1989). Jack Minker was a major pioneer in the development of the theory of deductive databases (Gallaire and Minker, 1978; Minker, 1988). Colomb (1991) presents some interesting ideas about using hardware to aid indexing of Prolog programs.

As mentioned in Chapter 9, unification was foreshadowed by Herbrand (1930), and formally introduced by Robinson (1965) in the same article that unveiled the resolution inference rule. Extending work by Boyer and Moore (1972), Martelli and Montanari (1976) and Paterson and Wegman (1978) developed unification algorithms that run in linear time and space via sharing of structure among representations of terms. Unification is surveyed by Knight (1989) and by Lassez *et al.* (1988). Shieber (1986) covers the use of unification in natural language processing.

Prolog was developed, and the first interpreter written, by the French researcher Alain Colmerauer in 1972 (Roussel, 1975; Colmerauer *et al.*, 1973); Colmerauer (1985) also gives an English-language survey of Prolog. Much of the theoretical background was developed by Robert Kowalski (1974; 1979b; 1979a) in collaboration with Colmerauer. Kowalski (1988) and Cohen (1988) provide good historical overviews of the origins of Prolog. *Foundations of Logic Programming* (Lloyd, 1987) is a theoretical analysis of the underpinnings of Prolog and other logic programming languages. Ait-Kaci (1991) gives a clear exposition of the Warren Abstract Machine (WAM) model of computation (Warren, 1983).

Recently, much of the effort in logic programming has been aimed toward increasing efficiency by building information about specific domains or specific inference patterns into the logic programming language. The language LOGIN (Ait-Kaci and Nasr, 1986) incorporates efficient handling of inheritance reasoning. Constraint logic programming (CLP) is based on the use of constraint satisfaction, together with a background theory, to solve constraints on

variables (Roach *et al.*, 1990), rather than the simple equality propagation used in normal unification. (Herbrand's original formulation had also used constraining equations rather than syntactic matching.) CLP is analyzed theoretically in (Jaffar and Lassez, 1987). Jaffar *et al.* (1992a) work specifically in the domain of the real numbers, using a logic programming language called CLP(R). Concurrent CLP is addressed by Saraswat (1993). Jaffar *et al.* (1992b) present the Constraint Logic Abstract Machine (CLAM), a WAM-like abstraction designed to aid in the analysis of CLP(R). Ait-Kaci and Podelski (1993) describe a sophisticated constraint logic programming language called LIFE, which combines constraint logic programming with functional programming and with inheritance reasoning (as in LOGIN). Prolog III (Colmerauer, 1990) builds in several assorted types of reasoning into a Prolog-like language. Volume 58 (1992) of the journal *Artificial Intelligence* is devoted primarily to constraint-based systems. Kohn (1991) describes an ambitious project to use constraint logic programming as the foundation for a real-time control architecture, with applications to fully automatic pilots.

Aside from the development of constraint logic and other advanced logic programming languages, there has been considerable effort to speed up the execution of Prolog by highly optimized compilation and the use of parallel hardware, especially in the Japanese Fifth Generation computing project. Van Roy (1990) examines some of the issues involved in fast execution on serial hardware. Feigenbaum and Shrobe (1993) provide a general account and evaluation of the Fifth Generation project. The Fifth Generation's parallel hardware prototype was the PIM, or Parallel Inference Machine (Taki, 1992). Logic programming of the PIM was based on the formalism of guarded Horn clauses (Ueda, 1985) and the GHC and KL1 languages that grew out of it (Furukawa, 1992). A number of applications of parallel logic programming are covered by Nitta *et al.* (1992). Other languages for parallel logic programming include Concurrent Prolog (Shapiro, 1983) and PARLOG (Clark and Gregory, 1986).

Logic programming is not the only paradigm of programming that has been prominent in AI. **Functional programming** models programs not as collections of logical clauses but as descriptions of mathematical functions. Functional programming is based on the **lambda** calculus (Church, 1941) and combinatory **logic** (Schönfinkel, 1924; Curry and Feys, 1958), two sophisticated mathematical notations for describing and reasoning about functions. The earliest functional programming language, dating from 1958, was **Lisp**, which is due to John McCarthy. Its history and prehistory is described in detail in (McCarthy, 1978). Incidentally, McCarthy denies (p. 190) that Lisp was intended as an actual implementation of the lambda-calculus (as has often been asserted), although it does borrow certain features. Lisp stands for LISt Processing, the use of **linked lists** whose elements are connected by pointers (rather than by proximity in the machine's address space, as arrays are) to create data structures of great flexibility. The list processing technique predicated Lisp and functional programming (Newell and Shaw, 1957; Gelernter *et al.*, 1960). After its invention, Lisp proliferated into a wide variety of dialects, partly because the language had been designed to be easy to modify and extend. In the past two decades, there has been an effort to reunify the language as **Common Lisp**, described in great detail by Steele (1990). Both of the two major AI programming texts mentioned above assume the use of Lisp. A number of other functional programming languages have been developed around a small, clean core of definitions. These include SCHEME, DYLAN, and ML.

The so-called **problem-solving languages** were precursors of logic programming in that they attempted to incorporate inference-like mechanisms, although they were not logic program-

ming languages as such and had control structures other than backtracking. PLANNER (Hewitt, 1969), although never actually implemented, was a very complex language that used automatic backtracking mechanisms analogous to the Prolog control structure. A subset known as MICRO-PLANNER (Sussman and Winograd, 1970) was implemented and used in the SHRDLU natural language understanding system (Winograd, 1972). The CONNIVER language (Sussman and McDermott, 1972) allowed finer programmer control over backtracking than MICRO-PLANNER. CONNIVER was used in the HACKER (Sussman, 1975) and BUILD (Fahlman, 1974) planning systems. QLISP (Sacerdoti *et al.*, 1976) used pattern matching to initiate function calls, as Prolog does; it was used in the NOAH planning system (Sacerdoti, 1975; Sacerdoti, 1977). More recently, POPLOG (Sloman, 1985) has attempted to incorporate several programming languages, including Lisp, Prolog, and POP-11 (Barrett *et al.*, 1985), into an integrated system.

Reasoning systems with metalevel capabilities were first proposed by Hayes (1973), but his GOLUX system was never built (a fate that also befell Doyle's (1980) ambitious SEAN system). AMORD (de Kleer *et al.*, 1977) put some of these ideas into practice, as did TEIRESIAS (Davis, 1980) in the field of rule-based expert systems. In the area of logic programming systems, MRS (Gensesereth and Smith, 1981; Russell, 1985) provided extensive metalevel facilities. Dincbas and Le Pape (1984) describe a similar system called METALOG. The work of David E. Smith (1989) on controlling logical inference builds on MRS. Alan Bundy's (1983) PRESS system used logical reasoning at the metalevel to guide the use of equality reasoning in solving algebra and trigonometry problems. It was able to attain humanlike performance on the British A-level exams for advanced precollege students, although equality reasoning had previously been thought to be a very difficult problem for automated reasoning systems. Guard *et al.* (1969) describe the early SAM theorem prover, which helped to solve an open problem in lattice theory. Wos and Winker (1983) give an overview of the contributions of AURA theorem prover toward solving open problems in various areas of mathematics and logic. McCune (1992) follows up on this, recounting the accomplishments of AURA'S successor OTTER in solving open problems. McAllester (1989) describes the ONTIC expert assistant system for mathematics research.

A *Computational Logic* (Boyer and Moore, 1979) is the basic reference on the Boyer-Moore theorem prover. Stickel (1988) covers the Prolog Technology Theorem Prover (PTTP), which incorporates the technique of locking introduced by Boyer (1971).

Early work in automated program synthesis was done by Simon (1963), Green (1969a), and Manna and Waldinger (1971). The transformational system of Burstall and Darlington (1977) used equational reasoning with recursion equations for program synthesis. Barstow (1979) provides an early book-length treatment. RAPTS (Paige and Henglein, 1987) takes an approach that views automated synthesis as an extension of the process of compilation. KIDS (Smith, 1990) is one of the strongest modern systems; it operates as an expert assistant. Manna and Waldinger (1992) give a tutorial introduction to the current state of the art, with emphasis on their own deductive approach. *Automating Software Design* (Lowry and McCartney, 1991) is an anthology; the articles describe a number of current approaches.

There are a number of textbooks on logic programming and Prolog. *Logic for Problem Solving* (Kowalski, 1979b) is an early text on logic programming in general, with a number of exercises. Several textbooks on Prolog are available (Clocksin and Mellish, 1987; Sterling and Shapiro, 1986; O'Keefe, 1990; Bratko, 1990). Despite focusing on Common Lisp, Norvig (1992) gives a good deal of basic information about Prolog, as well as suggestions for implementing

Prolog interpreters and compilers in Common Lisp. Several textbooks on automated reasoning were mentioned in Chapter 9. Aside from these, a unique text by Bundy (1983) provides reasonably broad coverage of the basics while also providing treatments of more advanced topics such as meta-level inference (using PRESS as one case study) and the use of higher-order logic. The *Journal of Logic Programming* and the *Journal of Automated Reasoning* are the principal journals for logic programming and theorem proving respectively. The major conferences in these fields are the annual International Conference on Automated Deduction (CADE) and International Conference on Logic Programming.

Aside from classical examples like the semantic networks used in Shastric Sanskrit grammar described in Chapter 8, or Peirce's existential graphs as described by Roberts (1973), modern work on semantic networks in AI began in the 1960s with the work of Quillian (1961; 1968). Charniak's (1972) thesis served to underscore the full extent to which heterogeneous knowledge on a wide variety of topics is essential for the interpretation of natural language discourse.

Minsky's (1975) so-called "frames paper" served to place knowledge representation on the map as a central problem for AI. The specific formalism suggested by Minsky, however, that of so-called "frames," was widely criticized as, at best, a trivial extension of the techniques of object-oriented programming, such as inheritance and the use of default values (Dahl *et al.*, 1970; Birtwistle *et al.*, 1973), which predated Minsky's frames paper. It is not clear to what extent the latter papers on object-oriented programming were influenced in turn by early AI work on semantic networks.

The question of semantics arose quite acutely with respect to Quillian's semantic networks (and those of others who followed his approach), with their ubiquitous and very vague "ISA links," as well as other early knowledge representation formalisms such as that of MERLIN (Moore and Newell, 1973) with its mysterious "flat" and "cover" operations. Woods' (1975) famous article "What's In a Link?" drew the attention of AI researchers to the need for precise semantics in knowledge representation formalisms. Brachman (1979) elaborated on this point and proposed solutions. Patrick Hayes's (1979) "The Logic of Frames" cut even deeper by claiming that most of whatever content such knowledge representations *did* have was merely sugar-coated logic: "Most of 'frames' is just a new syntax for parts of first-order logic." Drew McDermott's (1978b) "Tarskian Semantics, or, No Notation Without Denotation!" argued that the kind of semantical analysis used in the formal study of first-order logic, based on Tarski's definition of truth, should be the standard for *all* knowledge representation formalisms. Measuring all formalisms by the "logic standard" has many advocates but remains a controversial idea; notably, McDermott himself has reversed his position in "A Critique of Pure Reason" (McDermott, 1987). NETL (Fahlman, 1979) was a sophisticated semantic network system whose ISA links (called "virtual copy" or VC links) were based more on the notion of "inheritance" characteristic of frame systems or of object-oriented programming languages than on the subset relation, and were much more precisely defined than Quillian's links from the pre-Woods era. NETL is particularly intriguing because it was intended to be implemented in parallel hardware to overcome the difficulty of retrieving information from large semantic networks. David Touretzky (1986) subjects inheritance to rigorous mathematical analysis. Selman and Levesque (1993) discuss the complexity of inheritance with exceptions, showing that in most formulations it is NP-complete.

The development of description logics is merely the most recent stage in a long line of research aimed at finding useful subsets of first-order logic for which inference is computationally

tractable. Hector Levesque and Ron Brachman (1987) showed that certain logical constructs, notably certain uses of disjunction and negation, were primarily responsible for the intractability of logical inference. Building on the KL-ONE system (Schmolze and Lipkis, 1983), a number of systems have been developed whose designs incorporate the results of theoretical complexity analysis, most notably KRYPTON (Brachman *et al.*, 1983) and Classic (Borgida *et al.*, 1989). The result has been a marked increase in the speed of inference, and a much better understanding of the interaction between complexity and expressiveness in reasoning systems. On the other hand, as Doyle and Patil (1991) argue, restricting the expressiveness of a language either makes it impossible to solve certain problems, or encourages the user to circumvent the language restrictions using nonlogical means.

The study of truth maintenance systems began with the TMS (Doyle, 1979) and RUP (McAllester, 1980) systems, both of which were essentially JTMSs. The ATMS approach was described in a series of papers by Johan de Kleer (1986c; 1986a; 1986b). *Building Problem Solvers* (Forbus and de Kleer, 1993) explains in depth how TMSs can be used in AI applications.

---

## EXERCISES

**10.1** Recall that inheritance information in semantic networks can be captured logically by suitable implication sentences. In this exercise, we will consider the efficiency of using such sentences for inheritance.

- a. Consider the information content in a used-car catalogue such as Kelly's "Blue Book": that, for example, 1973 Dodge Vans are worth \$575. Suppose all this information (for 11,000 models) is encoded as logical rules, as suggested in the chapter. Write down three such rules, including that for 1973 Dodge Vans. How would you use the rules to find the value of a *particular* car (e.g., JB, which is a 1973 Dodge Van) given a backward-chaining theorem prover such as Prolog?
- b. Compare the time efficiency of the backward-chaining method for solving this problem with the inheritance method used in semantic nets.
- c. Explain how forward chaining allows a logic-based system to solve the same problem efficiently, assuming that the KB contains only the 11,000 rules about price.
- d. Describe a situation in which neither forward nor backward chaining on the rules will allow the price query for an individual car to be handled efficiently.
- e. Can you suggest a solution enabling this type of query to be solved efficiently in all cases in logic systems? (*Hint:* Remember that two cars of the same category have the same price.)

**10.2** The following Prolog code defines a relation R.

```
R( [ ] , X , X ) .  
R( [ A | X ] , Y , [ A | Z ] ) :- R( X , Y , Z )
```

- a. Show the proof tree and solution obtained for the queries

```
R( [ 1,2 ] , L , [ 1,2,3,4 ] ) and R( L , M , [ 1,2,3,4 ] )
```

- b. What standard list operation does R represent?  
c. Define the Prolog predicate `last(L, X)` (X is the last element of list L) using R and no other predicates.



10.3 In this exercise, we will look at sorting in Prolog.

- Write Prolog clauses that define the predicate `sorted(L)`, which is true if and only if list L is sorted in ascending order.
- Write a Prolog definition for the predicate `perm(L, M)`, which is true if and only if L is a permutation of M.
- Define `sort(L, M)` (M is a sorted version of L) using `perm` and `sorted`.
- Run `sort` on longer and longer lists until you lose patience. What is the time complexity of your program?
- Write a faster sorting algorithm, such as insertion sort or quicksort, in Prolog.



10.4 In this exercise, we will look at the recursive application of rewrite rules using logic programming. A rewrite rule (or demodulator in OTTER terminology) is an equation with a specified direction. For example, the rewrite rule  $x + 0 \rightarrow x$  suggests replacing any expression that matches  $x + 0$  with the expression  $x$ . The application of rewrite rules is a central part of mathematical reasoning systems, for example, in expression simplification and symbolic differentiation. We will use the predicate `Rewrite(x, y)` to represent rewrite rules. For example, the earlier rewrite rule is written as `Rewrite(x + 0, x)`. We will also need some way to define primitive terms that cannot be further simplified. For example, we can use `Primitive(0)` to say that 0 is a primitive term.

- Write a definition of a predicate `Simplify(xy)`, that is true when y is a simplified version of x; that is, no further rewrite rules are applicable to any subexpression of y.
- Write a collection of rules for simplification of expressions involving arithmetic operators, and apply your simplification algorithm to some sample expressions.
- Write a collection of rewrite rules for symbolic differentiation, and use them along with your simplification rules to differentiate and simplify expressions involving arithmetic expressions, including exponentiation.

10.5 In this exercise, we will consider the implementation of search algorithms in Prolog. Suppose that `successor(X, Y)` is true when state Y is a successor of state X; and that `goal(X)` is true when X is a goal state. Write a definition for `solve(X, P)`, which means that P is a path (list of states) beginning with X, ending in a goal state, and consisting of a sequence of legal steps as defined by `successor`. You will find that depth-first search is the easiest way to do this. How easy would it be to add heuristic search control?

**10.6** Why do you think that Prolog includes no heuristics for guiding the search for a solution to the query?

**10.7** Assume we put into a logical database a segment of the U.S. census data listing the age, city of residence, date of birth, and mother of every person, and where the constant symbol for each person is just their social security number. Thus, Ron's age is given by `Age(443-65-1282, 76)`.

Which of the indexing schemes S1–S5 following enable an efficient solution for which of the queries Q1-Q4 (assuming normal backward chaining).

- ◊ **S1:** an index for each atom in each position.
  - 0 **S2:** an index for each first argument.
  - ◊ **S3:** an index for each predicate atom.
  - ◊ **S4:** an index for each *combination* of predicate and first argument.
  - ◊ **S5:** an index for each *combination* of predicate and second argument, and an index for each first argument (nonstandard).
- 0 **Q1:** *Age(443-44-4321, x)*  
0 **Q2:** *ResidesIn(x, Houston)*  
0 **Q3:** *Mother(x, y)*  
0 **Q4:** *Age(x, 34) A ResidesIn(x, TinyTownUSA)*

10.8 We wouldn't want a semantic network to contain both *Age(Bill, 12)* and *Age(Bill, 10)*, but it's fine if it contains both *Friend(Bill, Opus)* and *Friend(Bill, Steve)*. Modify the functions in Figure 10.10 so that they make the distinction between logical functions and logical relations, and treat each properly.

10.9 The code repository contains a logical reasoning system whose components can be replaced by other versions. Re-implement some or all of the following components, and make sure that the resulting system works using the circuit example from Chapter 8.

- a. Basic data types and access functions for sentences and their components.
- b. STORE and FETCH for atomic sentences (disregarding efficiency).
- c. Efficient indexing mechanisms for STORE and FETCH.
- d. A unification algorithm.
- e. A forward-chaining algorithm.
- f. A backward-chaining algorithm using iterative deepening.

# Part IV

## ACTING LOGICALLY

In Part II, we saw that an agent cannot always select actions based solely on the percepts that are available at the moment, or even the internal model of the current state. We saw that **problem-solving agents** are able to plan ahead—to consider the consequences of *sequences* of actions—before acting. In Part III, we saw that a **knowledge-based agent** can select actions based on explicit, logical representations of the current state and the effects of actions. This allows the agent to succeed in complex, inaccessible environments that are too difficult for a problem-solving agent.

In Part IV, we put these two ideas together to build **planning agents**. At the most abstract level, the task of planning is the same as problem solving. Planning can be viewed as a type of problem solving in which the agent uses beliefs about actions and their consequences to search for a solution over the more abstract space of plans, rather than over the space of situations. Planning algorithms can also be viewed as special-purpose theorem provers that reason efficiently with axioms describing actions.

Chapter 11 introduces the basic ideas of planning, including the need to divide complex problems into **subgoals** whose solutions can be combined to provide a solution for the complete problem. Chapter 12 extends these ideas to more expressive representations of states and actions, and discusses real-world planning systems. Chapter 13 considers the execution of plans, particularly for cases in which unknown contingencies must be handled.

# 11 PLANNING

*In which we see how an agent can take advantage of problem structure to construct complex plans of action.*

## PLANNING AGENT

In this chapter, we introduce the basic ideas involved in planning systems. We begin by specifying a simple **planning agent** that is very similar to a problem-solving agent (Chapter 3) in that it constructs plans that achieve its goals, and then executes them. Section 11.2 explains the limitations of the problem-solving approach, and motivates the design of planning systems. The planning agent differs from a problem-solving agent in its representations of goals, states, and actions, as described in Section 11.4. The use of explicit, logical representations enables the planner to direct its deliberations much more sensibly. The planning agent also differs in the way it represents and searches for solutions. The remainder of the chapter describes in detail the basic **partial-order planning** algorithm, which searches through the space of plans to find one that is guaranteed to succeed. The additional flexibility gained from the partially ordered plan representation allows a planning agent to handle quite complicated domains.

## 11.1 A SIMPLE PLANNING AGENT

When the world state is accessible, an agent can use the percepts provided by the environment to build a complete and correct model of the current world state. Then, given a goal, it can call a suitable planning algorithm (which we will call IDEAL-PLANNER) to generate a plan of action. The agent can then execute the steps of the plan, one action at a time.

The algorithm for the simple planning agent is shown in Figure 11.1. This should be compared with the problem-solving agent shown in Figure 3.1. The planning algorithm IDEAL-PLANNER can be any of the planners described in this chapter or Chapter 12. We assume the existence of a function STATE-DESCRIPTION, which takes a percept as input and returns an initial state description in the format required by the planner, and a function MAKE-GOAL-QUERY, which is used to ask the knowledge base what the next goal should be. Note that the agent must deal with the case where the goal is infeasible (it just ignores it and tries another), and the case

```

function SIMPLE-PLANNING-AGENT(percept) returns an action
  static: KB, a knowledge base (includes action descriptions)
    p, a plan, initially NoPlan
    t, a counter, initially 0, indicating time
  local variables: G, a goal
    current, a current state description

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  current  $\leftarrow$  STATE-DESCRIPTION(KB, t)
  if p = NoPlan then
    G  $\leftarrow$  ASK(KB, MAKE-GOAL-QUERY(t))
    p  $\leftarrow$  IDEAL-PLANNER(current, G, KB)
  if p = NoPlan or p is empty then action  $\leftarrow$  NoOp
  else
    action  $\leftarrow$  FIRST(p)
    p  $\leftarrow$  REST(p)
  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t  $\leftarrow$  t + 1
  return action

```

**Figure 11.1** A simple planning agent. The agent first generates a goal to achieve, and then constructs a plan to achieve it from the current state. Once it has a plan, it keeps executing it until the plan is finished, then begins again with a new goal.

where the complete plan is in fact empty, because the goal is already true in the initial state. The agent interacts with the environment in a minimal way—it uses its percepts to define the initial state and thus the initial goal, but thereafter it simply follows the steps in the plan it has constructed. In Chapter 13, we discuss more sophisticated agent designs that allow more interaction between the world and the planner during plan execution.

## 11.2 FROM PROBLEM SOLVING TO PLANNING

Planning and problem solving are considered different subjects because of the differences in the representations of goals, states, and actions, and the differences in the representation and construction of action sequences. In this section, we first describe some of the difficulties encountered by a search-based problem-solving approach, and then introduce the methods used by planning systems to overcome these difficulties.

Recall the basic elements of a search-based problem-solver:

- **Representation of actions.** Actions are described by programs that generate successor state descriptions.
- **Representation of states.** In problem solving, a complete description of the initial state is

given, and actions are represented by a program that generates complete state descriptions. Therefore, all state representations are complete. In most problems, a state is a simple data structure: a permutation of the pieces in the eight puzzle, the position of the agent in a route-finding problem, or the position of the six people and the boat in the missionaries and cannibals problem. State representations are used only for successor generation, heuristic function evaluation, and goal testing.

- **Representation of goals.** The only information that a problem-solving agent has about its goal is in the form of the goal test and the heuristic function. Both of these can be applied to states to decide on their desirability, but they are used as "black boxes." That is, the problem-solving agent cannot "look inside" to select actions that might be useful in achieving the goal.
- **Representation of plans.** In problem solving, a solution is a sequence of actions, such as "Go from Arad to Sibiu to Fagaras to Bucharest." During the construction of solutions, search algorithms consider only unbroken sequences of actions beginning from the initial state (or, in the case of bidirectional search, ending at a goal state).

Let us see how these design decisions affect an agent's ability to solve the following simple problem: "Get a quart of milk and a bunch of bananas and a variable-speed cordless drill." Treating this as a problem-solving exercise, we need to specify the initial state: the agent is at home but without any of the desired objects, and the operator set: all the things that the agent can do. We can optionally supply a heuristic function: perhaps the number of things that have not yet been acquired.

Figure 11.2 shows a very small part of the first two levels of the search space for this problem, and an indication of the path toward the goal. The actual branching factor would be in the thousands or millions, depending on how actions are specified, and the length of the solution could be dozens of steps. Obviously, there are too many actions and too many states to consider. The real difficulty is that the heuristic evaluation function can only choose among states to decide which is closer to the goal; it cannot eliminate actions from consideration. Even if the evaluation function could get the agent into the supermarket, the agent would then resort to a guessing game. The agent makes guesses by considering actions—buying an orange, buying tuna fish, buying corn flakes, buying milk—and the evaluation function ranks these guesses—bad, bad, bad, good. The agent then knows that buying milk is a good thing, but has no idea what to try next and must start the guessing process all over again.

The fact that the problem-solving agent considers sequences of actions starting from the initial state also contributes to its difficulties. It forces the agent to decide first what to do in the initial state, where the relevant choices are essentially to go to any of a number of other places. Until the agent has figured out *how* to obtain the various items—by buying, borrowing, leasing, growing, manufacturing, stealing—it cannot really decide where to go. The agent therefore needs a more flexible way of structuring its deliberations, so that it can work on whichever part of the problem is most likely to be solvable given the current information.

The first key idea behind planning is to "*open up*" the representation of states, goals, and actions. Planning algorithms use descriptions in some formal language, usually first-order logic or a subset thereof. States and goals and goals are represented by sets of sentences, and actions are represented by logical descriptions of preconditions and effects. This enables the planner to



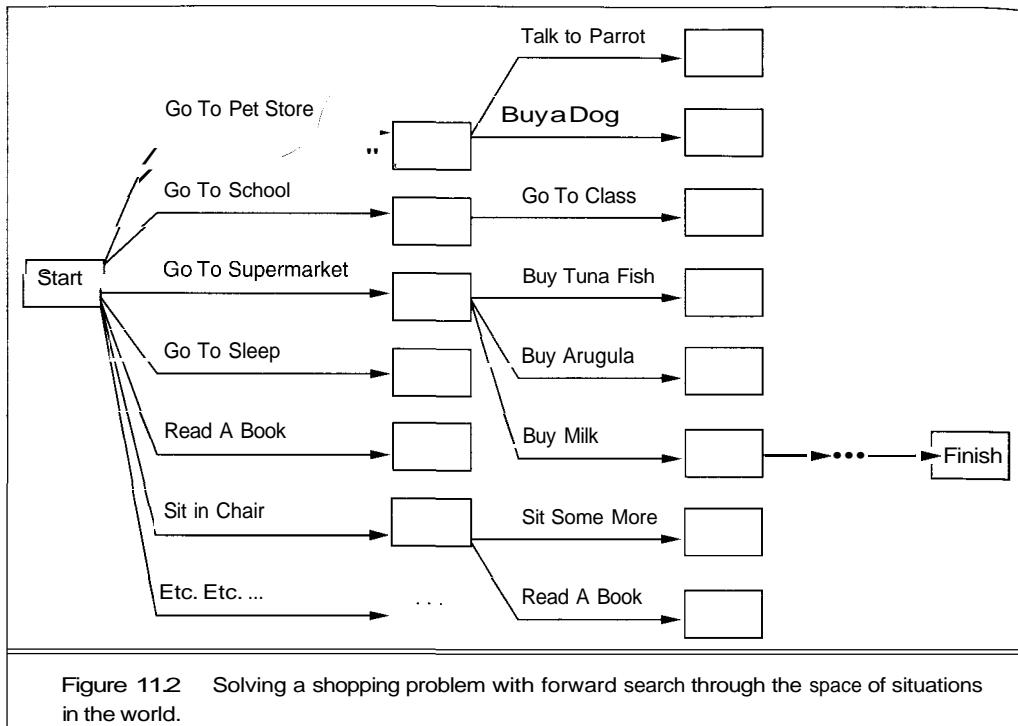


Figure 11.2 Solving a shopping problem with forward search through the space of situations in the world.

make direct connections between states and actions. For example, if the agent knows that the goal is a conjunction that includes the conjunct *Have(Milk)*, and that *Buy(x)* achieves *Have(x)*, then the agent knows that it is worthwhile to consider a plan that includes *Buy(Milk)*. It need not consider irrelevant actions such as *Buy(WhippingCream)* or *GoToSleep*.

The second key idea behind planning is that *the planner is free to add actions to the plan wherever they are needed, rather than in an incremental sequence starting at the initial state*. For example, the agent may decide that it is going to have to *Buy(Milk)*, even before it has decided where to buy it, how to get there, or what to do afterwards. There is no necessary connection between the order of planning and the order of execution. By making "obvious" or "important" decisions first, the planner can reduce the branching factor for future choices and reduce the need to backtrack over arbitrary decisions. Notice that the representation of states as sets of logical sentences plays a crucial role in making this freedom possible. For example, when adding the action *Buy(Milk)* to the plan, the agent can represent the state in which the action is executed as, say, *At(Supermarket)*. This actually represents an entire class of states—states with and without bananas, with and without a drill, and so on. Search algorithms that require complete state descriptions do not have this option.

The third and final key idea behind planning is that *most parts of the world are independent of most other parts*. This makes it feasible to take a conjunctive goal like "get a quart of milk and a bunch of bananas and a variable-speed cordless drill" and solve it with a divide-and-conquer strategy. A subplan involving going to the supermarket can be used to achieve the first two

conjuncts, and another subplan (e.g., either going to the hardware store or borrowing from a neighbor) can be used to achieve the third. The supermarket subplan can be further divided into a milk subplan and a bananas subplan. We can then put all the subplans together to solve the whole problem. This works because there is little interaction between the two subplans: going to the supermarket does not interfere with borrowing from a neighbor, and buying milk does not interfere with buying bananas (unless the agent runs out of some resource, like time or money).

Divide-and-conquer algorithms are efficient because it is almost always easier to solve several small sub-problems rather than one big problem. However, divide-and-conquer fails in cases where the cost of combining the solutions to the sub-problems is too high. Many puzzles have this property. For example, the goal state in the eight puzzle is a conjunctive goal: to get tile 1 in position A *and* tile 2 in position B *and* ... up to tile 8. We could treat this as a planning problem and plan for each subgoal independently, but the reason that puzzles are "tricky" is that it is difficult to put the subplans together. It is easy to get tile 1 in position A, but getting tile 2 in position B is likely to move tile 1 out of position. For tricky puzzles, the planning techniques in this chapter will not do any better than problem-solving techniques of Chapter 4. Fortunately, the real world is a largely benign place where subgoals tend to be nearly independent. If this were not the case, then the sheer size of the real world would make successful problem solving impossible.

## 11.3 PLANNING IN SITUATION CALCULUS

Before getting into planning techniques in detail, we present a formulation of planning as a logical inference problem, using situation calculus (see Chapter 7). A planning problem is represented in situation calculus by logical sentences that describe the three main parts of a problem:

- **Initial state:** An arbitrary logical sentence about a situation  $S_0$ . For the shopping problem, this might be<sup>1</sup>

$$\text{At}(\text{Home}, S_0) \wedge \neg\text{Have}(\text{Milk}, S_0) \wedge \neg\text{Have}(\text{Bananas}, S_0) \wedge \neg\text{Have}(\text{Drill}, S_0)$$

- **Goal state:** A logical query asking for suitable situations. For the shopping problem, the query would be

$$\exists s \text{ At}(\text{Home}, s) \wedge \text{Have}(\text{Milk}, s) \wedge \text{Have}(\text{Bananas}, s) \wedge \text{Have}(\text{Drill}, s)$$

- **Operators:** A set of descriptions of actions, using the action representation described in Chapter 7. For example, here is a successor-state axiom involving the *Buy(Milk)* action:

$$\begin{aligned} \forall a, s \text{ Have}(\text{Milk.Result}(a, s)) &\Leftrightarrow [(a = \text{Buy}(\text{Milk}) \wedge \text{At}(\text{Supermarket}, s) \\ &\quad \vee (\text{Have}(\text{Milk}, s) \wedge a \neq \text{Drop}(\text{Milk}))] \end{aligned}$$

Recall that situation calculus is based on the idea that actions transform states: *Result(a, s)* names the situation resulting from executing action *a* in situation *s*. For the purposes of planning, it

<sup>1</sup> A better representation might be along the lines of  $\neg\exists m \text{ Milk}(m) \wedge \text{Have}(m, S_0)$ , but we have chosen the simpler notation to facilitate the explanation of planning methods. Notice that partial state information is handled automatically by a logical representation, whereas problem-solving algorithms required a special multiple-state representation.

will be useful to handle action sequences as well as single actions. We will use  $\text{Result}'(l, s)$  to mean the situation resulting from executing the sequence of actions / starting in  $s$ .  $\text{Result}'$  is defined by saying that an empty sequence of actions has no effect on a situation, and the result of a nonempty sequence of actions is the same as applying the first action, and then applying the rest of the actions from the resulting situation:

$$\begin{aligned} \forall s \text{ } \text{Result}'([ ], s) &= s \\ \forall a, p, s \text{ } \text{Result}'([a|p], s) &= \text{Result}'(p, \text{Result}(a, s)) \end{aligned}$$

A solution to the shopping problem is a plan  $p$  that when applied to the start state  $S_0$  yields a situation satisfying the goal query. In other words, a  $p$  such that

$$\begin{aligned} \text{At}(\text{Home}, \text{Result}'(p, S_0)) \wedge \text{Have}(\text{Milk}, \text{Result}'(p, S_0)) \wedge \text{Have}(\text{Bananas}, \text{Result}'(p, S_0)) \\ \wedge \text{Have}(\text{Drill}, \text{Result}'(p, S_0)) \end{aligned}$$

If we hand this query to ASK, we end up with a solution such as

$$\begin{aligned} p = [ &\text{Go}(\text{SuperMarket}), \text{Buy}(\text{Milk}), \text{Buy}(\text{Banana}), \\ &\text{Go}(\text{HardwareStore}), \text{Buy}(\text{Drill}), \text{Go}(\text{Home}) ] \end{aligned}$$

From the theoretical point of view, there is little more to say. We have a formalism for expressing goals and plans, and we can use the well-defined inference procedure of first-order logic to find plans. It is true that there are some limitations in the expressiveness of situation calculus, as discussed in Section 8.4, but situation calculus is sufficient for most planning domains.

Unfortunately, a good theoretical solution does not guarantee a good practical solution. We saw in Chapter 3 that problem solving takes time that is exponential in the length of the solution in the worst case, and in Chapter 9, we saw that logical inference is only semidecidable. If you suspect that planning by unguided logical inference would be inefficient, you're right. Furthermore, the inference procedure gives us no guarantees about the resulting plan  $p$  other than that it achieves the goal. In particular, note that if  $p$  achieves the goal, then so do  $[\text{Nothing}|p]$  and  $[A, A^{-1}|p]$ , where  $\text{Nothing}$  is an action that makes no changes (or at least no relevant changes) to the situation, and  $A^{-1}$  is the inverse of  $A$  (in the sense that  $s = \text{Result}(A^{-1}, \text{Result}(A, s))$ ). So we may end up with a plan that contains irrelevant steps if we use unguided logical inference.

To make planning practical we need to do two things: (1) Restrict the language with which we define problems. With a restrictive language, there are fewer possible solutions to search through. (2) Use a special-purpose algorithm called a **planner** rather than a general-purpose theorem prover to search for a solution. The two go hand in hand: every time we define a new problem-description language, we need a new planning algorithm to process the language. The remainder of this chapter and Chapter 12 describe a series of planning languages of increasing complexity, along with planning algorithms for these languages. Although we emphasize the algorithms, it is important to remember that we are always dealing with a logic: a formal language with a well-defined syntax, semantics, and proof theory. The proof theory says what can be inferred about the results of action sequences, and therefore what the legal plans are. The algorithm enables us to find those plans. The idea is that the algorithm can be designed to process the restricted language more efficiently than a resolution theorem prover.



## 11.4 BASIC REPRESENTATIONS FOR PLANNING

The "classical" approach that most planners use today describes states and operators in a restricted language known as the STRIPS language,<sup>2</sup> or in extensions thereof. The STRIPS language lends itself to efficient planning algorithms, while retaining much of the expressiveness of situation calculus representations.

### Representations for states and goals

In the STRIPS language, states are represented by conjunctions of function-free ground literals, that is, predicates applied to constant symbols, possibly negated. For example, the initial state for the milk-and-bananas problem might be described as

$$\text{At(Home)} \wedge \neg \text{Have(Milk)} \wedge \neg \text{Have(Bananas)} \wedge \neg \text{Have(Drill)} \wedge \dots$$

As we mentioned earlier, a state description does not need to be complete. An incomplete state description, such as might be obtained by an agent in an inaccessible environment, corresponds to a set of possible complete states for which the agent would like to obtain a successful plan. Many planning systems instead adopt the convention—analogous to the "negation as failure" convention used in logic programming—that if the state description does not mention a given positive literal then the literal can be assumed to be false.

Goals are also described by conjunctions of literals. For example, the shopping goal might be represented as

$$\text{At(Home)} \wedge \text{Have(Milk)} \wedge \text{Have(Bananas)} \wedge \text{Have(Drill)}$$

Goals can also contain variables. For example, the goal of being at a store that sells milk would be represented as

$$\text{At}(x) \wedge \text{Sells}(x, \text{Milk})$$

As with goals given to theorem provers, the variables are assumed to be existentially quantified. However, one must distinguish clearly between a goal given to a planner and a query given to a theorem prover. The former asks for a sequence of actions that *makes the goal true if executed*, and the latter asks whether the query sentence *is true* given the truth of the sentences in the knowledge base.

Although representations of initial states and goals are used as inputs to planning systems, it is quite common for the planning process itself to maintain only implicit representations of states. Because most actions change only a small part of the state representation, it is more efficient to keep track of the changes. We will see how this is done shortly.

<sup>2</sup> Named after a pioneering planning program known as the STanford Research Institute Problem Solver. There are two unfortunate things about the name STRIPS. First, the organization no longer uses the name "Stanford" and is now known as SRI International. Second, the program is what we now call a planner, not a problem solver, but when it was developed in 1970, the distinction had not been articulated. Although the STRIPS planner has long since been superseded, the STRIPS language for describing actions has been invaluable, and many "STRIPS-like" variants have been developed.

## Representations for actions

Our STRIPS operators consist of three components:

ACTION DESCRIPTION

- **The action description** is what an agent actually returns to the environment in order to do something. Within the planner it serves only as a name for a possible action.
- **The precondition** is a conjunction of atoms (positive literals) that says what must be true before the operator can be applied.
- **The effect** of an operator is a conjunction of literals (positive or negative) that describes how the situation changes when the operator is applied.<sup>3</sup>

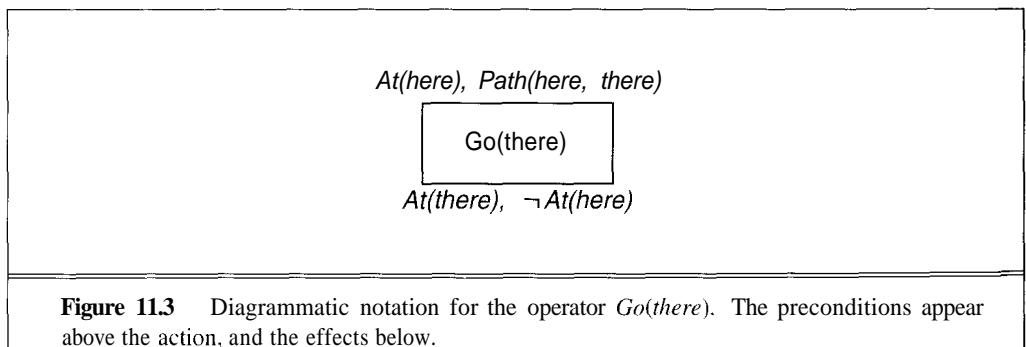
PRECONDITION

EFFECT

Here is an example of the syntax we will use for forming a STRIPS operator for going from one place to another:

$$\begin{aligned} Op(\text{ACTION:} & Go(\text{there}), \text{PRECOND:} At(\text{here}) \wedge \\ & \text{EFFECT:} At(\text{there}) \wedge \neg At(\text{here})) \end{aligned}$$

(We will also use a graphical notation to describe operators, as shown in Figure 11.3.) Notice that there are no explicit situation variables. Everything in the precondition implicitly refers to the situation immediately before the action, and everything in the effect implicitly refers to the situation that is the result of the action.



**Figure 11.3** Diagrammatic notation for the operator *Go(there)*. The preconditions appear above the action, and the effects below.

OPERATOR SCHEMA

An operator with variables is known as an **operator schema**, because it does not correspond to a single executable action but rather to a family of actions, one for each different instantiation of the variables. Usually, only fully instantiated operators can be executed; our planning algorithms will ensure that each variable has a value by the time the planner is done. As with state descriptions, the language of preconditions and effects is quite restricted. The precondition must be a conjunction of positive literals, and the effect must be a conjunction of positive and/or negative literals. All variables are assumed universally quantified, and there can be no additional quantifiers. In Chapter 12, we will relax these restrictions.

APPLICABLE

We say that an operator  $o$  is **applicable** in a state  $s$  if there is some way to instantiate the variables in  $o$  so that every one of the preconditions of  $o$  is true in  $s$ , that is, if  $\text{Precond}(o) \models s$ . In the resulting state, all the positive literals in  $\text{Effect}(o)$  hold, as do all the literals that held in  $s$ ,

<sup>3</sup> The original version of STRIPS divided the effects into an **add list** and a **delete list**.

except for those that are negative literals in  $\text{Effect}(o)$ . For example, if the initial situation includes the literals

$\text{At}(\text{Home}), \text{Path}(\text{Home}, \text{Supermarket}), \dots$

then the action  $\text{Go}(\text{Supermarket})$  is applicable, and the resulting situation contains the literals

$\neg\text{At}(\text{Home}), \text{At}(\text{Supermarket}), \text{Path}(\text{Home}, \text{Supermarket}), \dots$

## Situation Space and Plan Space

In Figure 11.2, we showed a search space of *situations* in the world (in this case the shopping world). A path through this space from the initial state to the goal state constitutes a plan for the shopping problem. If we wanted, we could take a problem described in the STRIPS language and solve it by starting at the initial state and applying operators one at a time until we reached a state that includes all the literals in the goal. We could use any of the search methods of Part II. An algorithm that did this would clearly be considered a problem solver, but we could also consider it a planner. We would call it a **situation space** planner because it searches through the space of possible situations, and a **progression** planner because it searches forward from the initial situation to the goal situation. The main problem with this approach is the high branching factor and thus the huge size of the search space.

SITUATION SPACE  
PROGRESSION

REGRESSION

PARTIAL PLAN

REFINEMENT OPERATORS

One way to try to cut the branching factor is to search backwards, from the goal state to the initial state; such a search is called **regression** planning. This approach is *possible* because the operators contain enough information to regress from a partial description of a result state to a partial description of the state before an operator is applied. We cannot get complete descriptions of states this way, but we don't need to. The approach is *desirable* because in typical problems the goal state has only a few conjuncts, each of which has only a few appropriate operators, whereas the initial state usually has many applicable operators. (An operator is appropriate to a goal if the goal is an effect of the operator.) Unfortunately, searching backwards is complicated somewhat by the fact that we often have to achieve a conjunction of goals, not just one. The original STRIPS algorithm was a situation-space regression planner that was incomplete (it could not always find a plan when one existed) because it had an inadequate way of handling the complication of conjunctive goals. Fixing this incompleteness makes the planner very inefficient.

In summary, the nodes in the search tree of a situation-space planner correspond to situations, and the path through the search tree is the plan that will be ultimately returned by the planner. Each branch point adds another step to either the beginning (regression) or end (progression) of the plan.

An alternative is to search through the space of *plans* rather than the space of *situations*. That is, we start with a simple, incomplete plan, which we call a **partial plan**. Then we consider ways of expanding the partial plan until we come up with a complete plan that solves the problem. The operators in this search are operators on plans: adding a step, imposing an ordering that puts one step before another, instantiating a previously unbound variable, and so on. The solution is the final plan, and the path taken to reach it is irrelevant.

Operations on plans come in two categories. **Refinement operators** take a partial plan and add constraints to it. One way of looking at a partial plan is as a representation for a set

MODIFICATION OPERATOR

of complete, fully constrained plans. Refinement operators eliminate some plans from this set, but they never add new plans to it. Anything that is not a refinement operator is a **modification operator**. Some planners work by constructing potentially incorrect plans, and then "debugging" them using modification operators. In this chapter, we use only refinement operators.

## Representations for plans

If we are going to search through a space of plans, we need to be able to represent them. We can settle on a good representation for plans by considering partial plans for a simple problem: putting on a pair of shoes. The goal is the conjunction of *RightShoeOn* & *LeftShoeOn*, the initial state has no literals at all, and the four operators are

$$\begin{aligned}Op(\text{ACTION:} &\text{RightShoe}, \text{PRECOND:} \text{RightSockOn}, \text{EFFECT:} \text{RightShoeOn}) \\Op(\text{ACTION:} &\text{RightSock}, \text{EFFECT:} \text{RightSockOn}) \\Op(\text{ACTION:} &\text{LeftShoe}, \text{PRECOND:} \text{LeftSockOn}, \text{EFFECT:} \text{LeftShoeOn}) \\Op(\text{ACTION:} &\text{LeftSock}, \text{EFFECT:} \text{LeftSockOn})\end{aligned}$$

LEAST COMMITMENT

A partial plan for this problem consists of the two steps *RightShoe* and *LeftShoe*. But which step should come first? Many planners use the principle of **least commitment**, which says that one should only make choices about things that you currently care about, leaving the other choices to be worked out later. This is a good idea for programs that search, because if you make a choice about something you don't care about now, you are likely to make the wrong choice and have to backtrack later. A least commitment planner could leave the ordering of the two steps unspecified. When a third step, *RightSock*, is added to the plan, we want to make sure that putting on the right sock comes before putting on the right shoe, but we do not care where they come with respect to the left shoe. A planner that can represent plans in which some steps are ordered (before or after) with respect to each other and other steps are unordered is called a **partial order** planner. The alternative is a **total order** planner, in which plans consist of a simple list of steps. A totally ordered plan that is derived from a plan  $P$  by adding ordering constraints is called a **linearization of  $P$** .

PARTIAL ORDER

TOTAL ORDER

LINEARIZATION

FULLY INSTANTIATED PLANS

The socks-and-shoes example does not show it, but planners also have to commit to bindings for variables in operators. For example, suppose one of your goals is *Have(Milk)*, and you have the action *Buy(item, store)*. A sensible commitment is to choose this action with the variable *item* bound to *Milk*. However, there is no good reason to pick a binding for *store*, so the principle of least commitment says to leave it unbound and make the choice later. Perhaps another goal will be to buy an item that is only available in one specialty store. If that store also carries milk, then we can bind the variable *store* to the specialty store at that time. By delaying the commitment to a particular store, we allow the planner to make a good choice later. This strategy can also help prune out bad plans. Suppose that for some reason the branch of the search space that includes the partially instantiated action *Buy(Milk, store)* leads to a failure for some reason unrelated to the choice of store (perhaps the agent has no money). If we had committed to a particular store, then the search algorithm would force us to backtrack and consider another store. But if we have not committed, then there is no choice to backtrack over and we can discard this whole branch of the search tree without having to enumerate any of the stores. Plans in which every variable is bound to a constant are called **fully instantiated plans**.

PLAN

In this chapter, we will use a representation for plans that allows for deferred commitments about ordering and variable binding. A **plan** is formally defined as a data structure consisting of the following four components:

CAUSAL LINKS

- A set of plan steps. Each step is one of the operators for the problem.
- A set of step ordering constraints. Each ordering constraint is of the form  $S_i \prec S_j$ , which is read as “ $S_i$  before  $S_j$ ” and means that step  $S_i$  must occur sometime before step  $S_j$  (but not necessarily immediately before).<sup>4</sup>
- A set of variable binding constraints. Each variable constraint is of the form  $v = x$ , where  $v$  is a variable in some step, and  $x$  is either a constant or another variable.
- **A set of causal links.**<sup>5</sup> A causal link is written as  $S_i \xrightarrow{c} S_j$  and read as “ $S_i$  achieves  $c$  for  $S_j$ ”. Causal links serve to record the purpose(s) of steps in the plan: here a purpose of  $S_i$  is to achieve the precondition  $c$  of  $S_j$ .

The initial plan, before any refinements have taken place, simply describes the unsolved problem. It consists of two steps, called *Start* and *Finish*, with the ordering constraint *Start X Finish*. Both *Start* and *Finish* have null actions associated with them, so when it is time to execute the plan, they are ignored. The *Start* step has no preconditions, and its effect is to add all the propositions that are true in the initial state. The *Finish* step has the goal state as its precondition, and no effects. By defining a problem this way, our planners can start with the initial plan and manipulate it until they come up with a plan that is a solution. The shoes-and-socks problem is defined by the four operators given earlier and an initial plan that we write as follows:

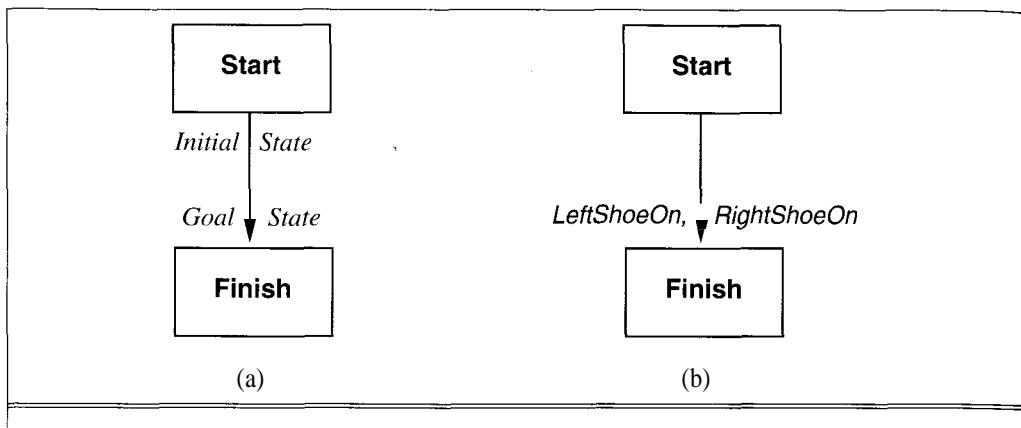
```
Plan(STEPS:{ S1: Op(ACTION:Start),
               S2: Op(ACTION:Finish,
                      PRECOND:RightShoeOn A LeftShoeOn)},
      ORDERINGS: {S1 < S2},
      BINDINGS: {},
      LINKS: {})
```

As with individual operators, we will use a graphical notation to describe plans (Figure 11.4(a)). The initial plan for the shoes-and-socks problem is shown in Figure 11.4(b). Later in the chapter we will see how this notation is extended to deal with more complex plans.

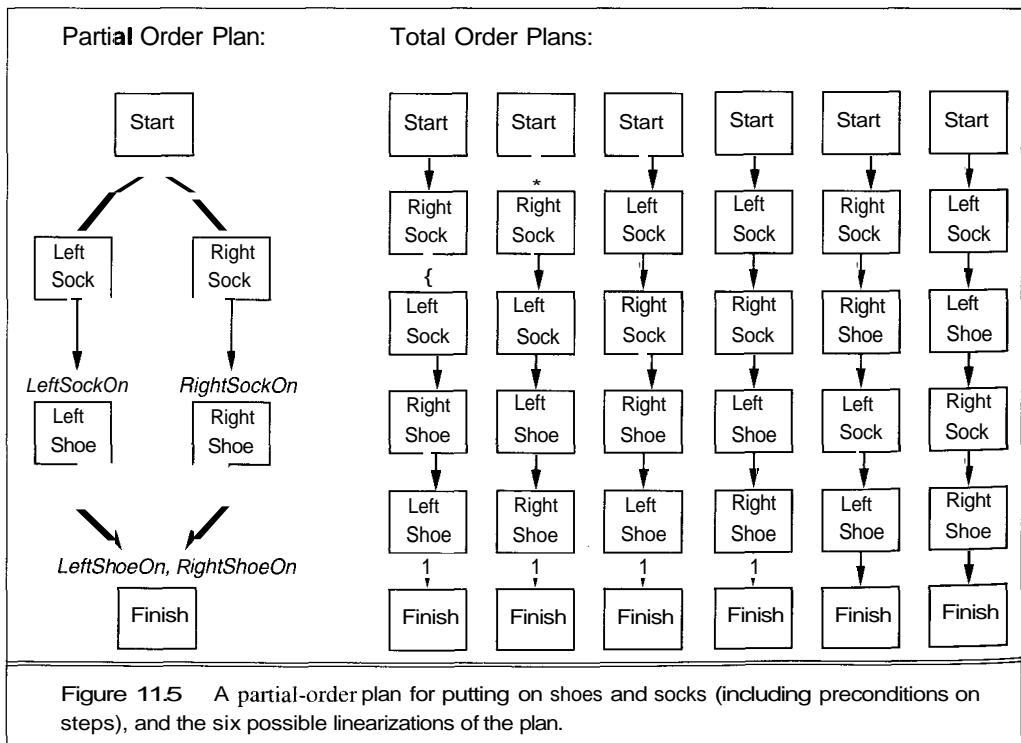
Figure 11.5 shows a partial-order plan that is a solution to the shoes-and-socks problem, and six linearizations of the plan. This example shows that the partial-order plan representation is powerful because it allows a planner to ignore ordering choices that have no effect on the correctness of the plan. As the number of steps grows, the number of possible ordering choices grows exponentially. For example, if we added a hat and a coat to the problem, which interact neither with each other nor with the shoes and socks, then there would still be one partial plan that represents all the solutions, but there would be 180 linearizations of that partial plan. (Exercise 11.1 asks you to derive this number).

<sup>4</sup> We use the notation  $A \prec B \prec C$  to mean  $(A \text{ X } B) \text{ A } (B \prec C)$ .

<sup>5</sup> Some authors call causal links **protection intervals**.



**Figure 11.4** (a) Problems are defined by partial plans containing only *Start* and *Finish* steps. The initial state is entered as the effects of the *Start* step, and the goal state is the precondition of the *Finish* step. Ordering constraints are shown as arrows between boxes. (b) The initial plan for the shoes-and-socks problem.



**Figure 11.5** A partial-order plan for putting on shoes and socks (including preconditions on steps), and the six possible linearizations of the plan.

## Solutions

SOLUTION

COMPLETE PLAN  
ACHIEVED

CONSISTENT PLAN

A **solution** is a plan that an agent can execute, and that guarantees achievement of the goal. If we wanted to make it really easy to check that a plan is a solution, we could insist that only fully instantiated, totally ordered plans can be solutions. But this is unsatisfactory for three reasons. First, for problems like the one in Figure 11.5, it is more natural for the planner to return a partial-order plan than to arbitrarily choose one of the many linearizations of it. Second, some agents are capable of performing actions in parallel, so it makes sense to allow solutions with parallel actions. Lastly, when creating plans that may later be combined with other plans to solve larger problems, it pays to retain the flexibility afforded by the partial ordering of actions. Therefore, we allow partially ordered plans as solutions using a simple definition: a **solution** is a **complete, consistent** plan. We need to define these terms.

A **complete plan** is one in which every precondition of every step is **achieved** by some other step. A step achieves a condition if the condition is one of the effects of the step, and if no other step can possibly cancel out the condition. More formally, a step  $S_i$  achieves a precondition  $c$  of the step  $S_j$  if (1)  $S_i \prec S_j$  and  $c \in \text{EFFECTS}(S_i)$ ; and (2) there is no step  $S_k$  such that  $(\neg c) \in \text{EFFECTS}(S_k)$ , where  $S_i \prec S_k \prec S_j$  in some linearization of the plan.

A **consistent plan** is one in which there are no contradictions in the ordering or binding constraints. A contradiction occurs when both  $S_i \prec S_j$  and  $S_j \prec S_i$  hold or both  $v = A$  and  $v = B$  hold (for two different constants  $A$  and  $B$ ). Both  $\prec$  and  $=$  are transitive, so, for example, a plan with  $S_1 \prec S_2$ ,  $S_2 \prec S_3$ , and  $S_3 \prec S_1$  is inconsistent.

The partial plan in Figure 11.5 is a solution because all the preconditions are achieved. From the preceding definitions, it is easy to see that any linearization of a solution is also a solution. Hence the agent can execute the steps in any order consistent with the constraints, and still be assured of achieving the goal.

## 11.5 A PARTIAL-ORDER PLANNING EXAMPLE



In this section, we sketch the outline of a partial-order regression planner that searches through plan space. The planner starts with an initial plan representing the start and finish steps, and on each iteration adds one more step. If this leads to an inconsistent plan, it backtracks and tries another branch of the search space. *To keep the search focused, the planner only considers adding steps that serve to achieve a precondition that has not yet been achieved.* The causal links are used to keep track of this.

We illustrate the planner by returning to the problem of getting some milk, a banana, and a drill, and bringing them back home. We will make some simplifying assumptions. First, the *Go* action can be used to travel between any two locations. Second, the description of the *Buy* action ignores the question of money (see Exercise 11.2). The initial state is defined by the following operator, where *HWS* means hardware store and *SM* means supermarket:

*Op(ACTION:Start, EFFECT:At(Home) A Sells(HWS, Drill)  
A Sells(SM,Milk), Sells(SM,Banana))*

The goal state is defined by a *Finish* step describing the objects to be acquired and the final destination to be reached:

*Op(ACTION:Finish,*  
*PRECOND:Have(Drill)A Have(Milk)A Have(Banana) A At(Home))*

The actions themselves are defined as follows:

*Op(ACTION:Go(there), PRECOND:At(here),  
   EFFECT:At(there) A  $\neg$ At(here))*  
*Op(ACTION:Buy(x), PRECOND:At(store)A Sells(store, x),  
   EFFECT: Have(x))*

Figure 11.6 shows a diagram of the initial plan for this problem. We will develop a solution to the problem step by step, showing at each point a figure illustrating the partial plan at that point in the development. As we go along, we will note some of the properties we require for the planning algorithm. After we finish the example, we will present the algorithm in detail.

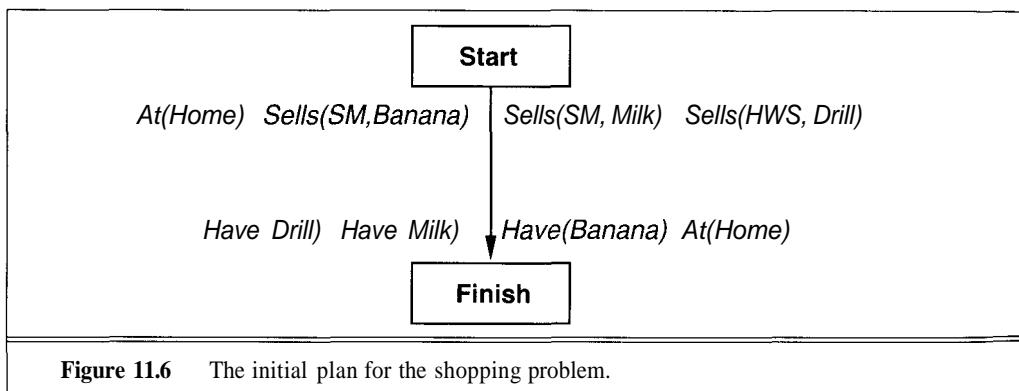
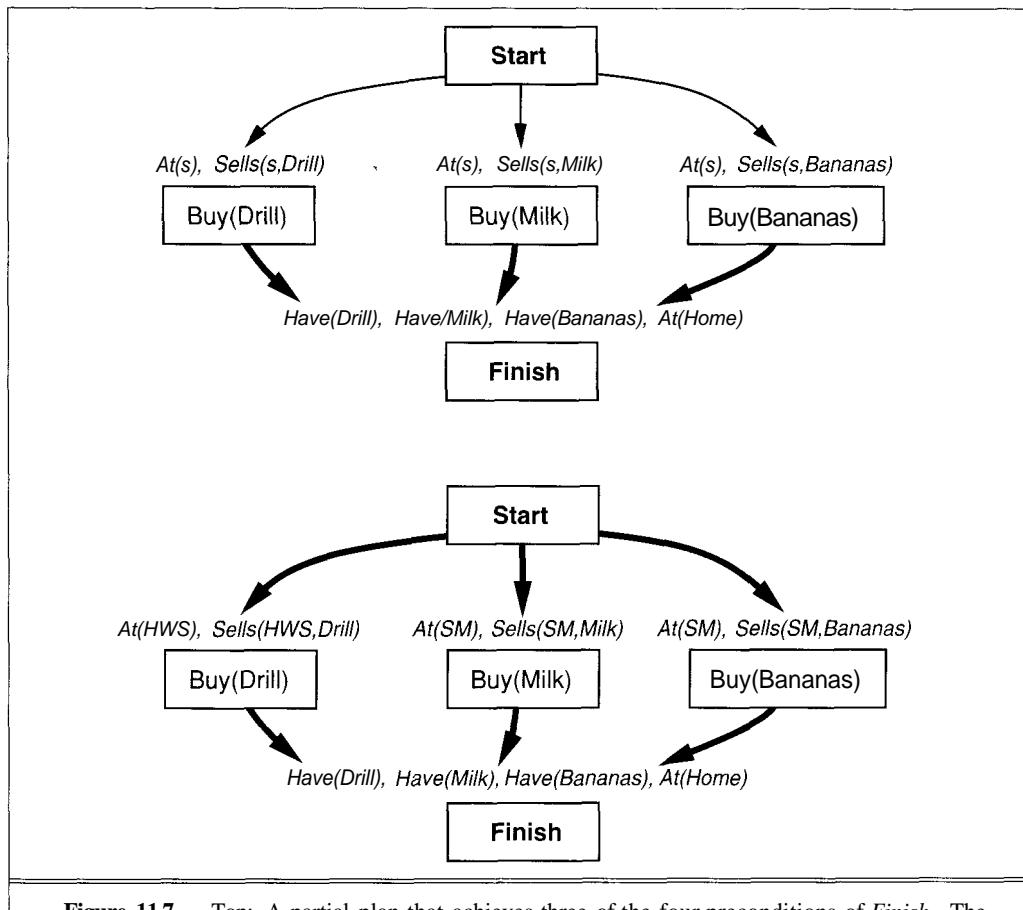


Figure 11.6 The initial plan for the shopping problem.

The first thing to notice about Figure 11.6 is that there are many possible ways in which the initial plan can be elaborated. Some choices will work, and some will not. As we work out the solution to the problem, we will show some correct choices and some incorrect choices. For simplicity, we will start with some correct choices. In Figure 11.7 (top), we have selected three *Buy* actions to achieve three of the preconditions of the *Finish* action. In each case there is only one possible choice because the operator library offers no other way to achieve these conditions.

The bold arrows in the figure are causal links. For example, the leftmost causal link in the figure means that the step *Buy(Drill)* was added in order to achieve the *Finish* step's *Have(Drill)* precondition. The planner will make sure that this condition is maintained by **protecting** it: if a step might delete the *Have(Drill)* condition, then it will not be inserted between the *Buy(Drill)* step and the *Finish* step. Light arrows in the figure show ordering constraints. By definition, all actions are constrained to come after the *Start* action. Also, all causes are constrained to come before their effects, so you can think of each bold arrow as having a light arrow underneath it.

The second stage in Figure 11.7 shows the situation after the planner has chosen to achieve the *Sells* preconditions by linking them to the initial state. Again, the planner has no choice here because there is no other operator that achieves *Sells*.

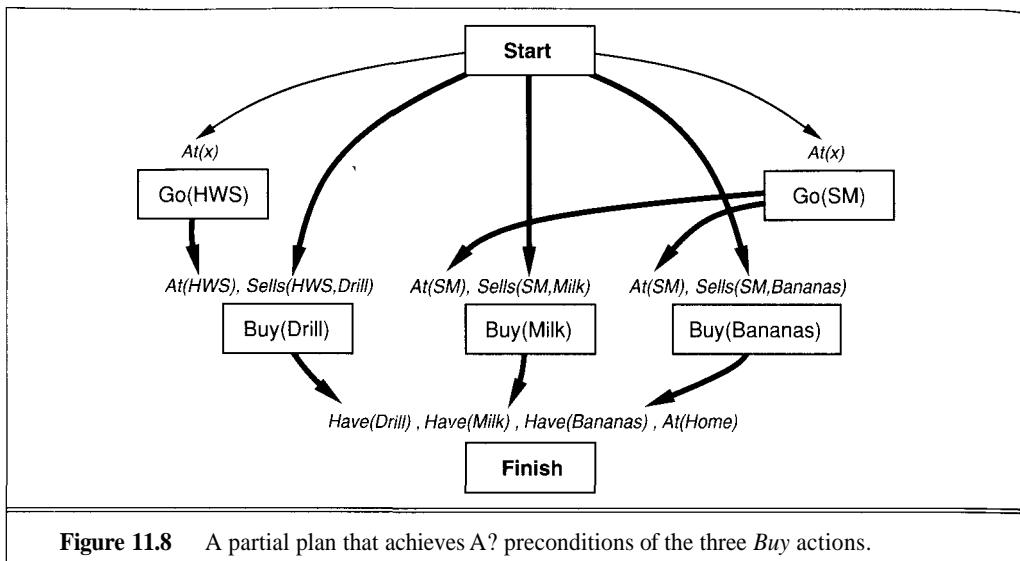


**Figure 11.7** Top: A partial plan that achieves three of the four preconditions of *Finish*. The heavy arrows show causal links. Bottom: Refining the partial plan by adding causal links to achieve the *Sells* preconditions of the *Buy* steps.

Although it may not seem like we have done much yet, this is actually quite an improvement over what we could have done with the problem-solving approach. First, out of all the things that one can buy, and all the places that one can go, we were able to choose just the *right Buy* actions and just the right places, without having to waste time considering the others. Then, once we have chosen the actions, we need not decide how to order them; a partial-order planner can make that decision later.

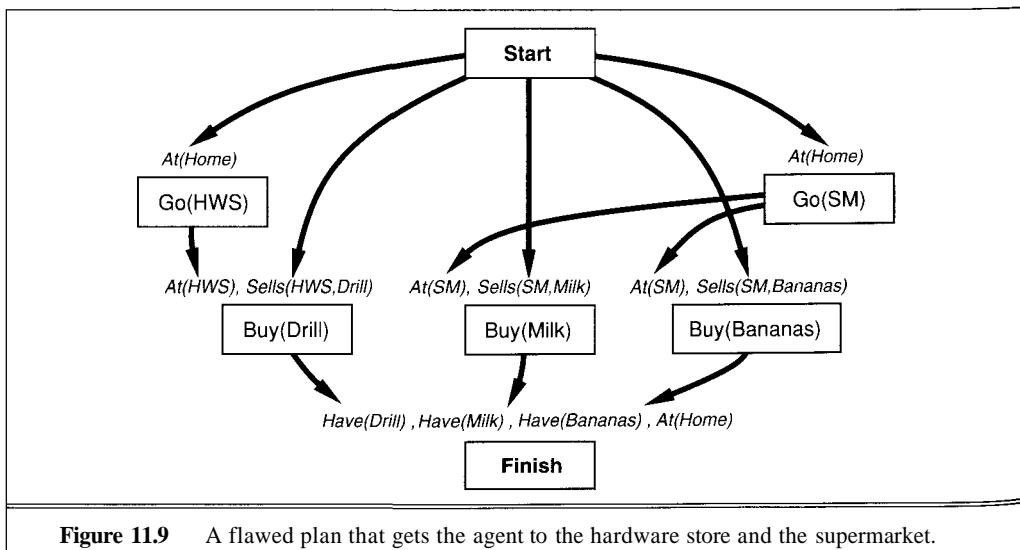
In Figure 11.8, we extend the plan by choosing two *Go* actions to get us to the hardware store and supermarket, thus achieving the *At* preconditions of the *Buy* actions.

So far, everything has been easy. A planner could get this far without having to do any search. Now it gets harder. The two *Go* actions have unachieved preconditions that interact with each other, because the agent cannot be *At* two places at the same time. Each *Go* action has a precondition  $At(x)$ , where  $x$  is the location that the agent was at before the *Go* action. Suppose



the planner tries to achieve the preconditions of *Go(HWS)* and *Go(SM)* by linking them to the *At(Home)* condition in the initial state. This results in the plan shown in Figure 11.9.

Unfortunately, this will lead to a problem. The step *Go(HWS)* adds the condition *At(HWS)*, but it also deletes the condition *At(Home)*. So if the agent goes to the hardware store, it can no longer go from home to the supermarket. (That is, unless it introduces another step to go back home from the hardware store—but the causal link means that the start step, not some other step,



I  
T  
PROTECTED LINKS  
THREATS  
  
DEMOTION  
PROMOTION

achieves the *At(Home)* precondition.) On the other hand, if the agent goes to the supermarket first, then it cannot go from home to the hardware store.

At this point, we have reached a dead end in the search for a solution, and must back up and try another choice. The interesting part is seeing how *a planner could notice that this partial plan is a dead end without wasting a lot of time on it*. The key is that the causal links in a partial plan are **protected links**. A causal link is protected by ensuring that **threats**—that is, steps that might delete (or **clobber**) the protected condition—are ordered to come before or after the protected link. Figure 11.10(a) shows a threat: The causal link  $S_1 \rightarrow S_2$  is threatened by the new step  $S_3$  because one effect of  $S_3$  is to delete  $c$ . The way to resolve the threat is to add ordering constraints to make sure that  $S_3$  does not intervene between  $S_1$  and  $S_2$ . If  $S_3$  is placed before  $S_1$  this is called **demotion** (see Figure 11.10(b)), and if it is placed after  $S_2$ , it is called **promotion** (see Figure 11.10(c)).

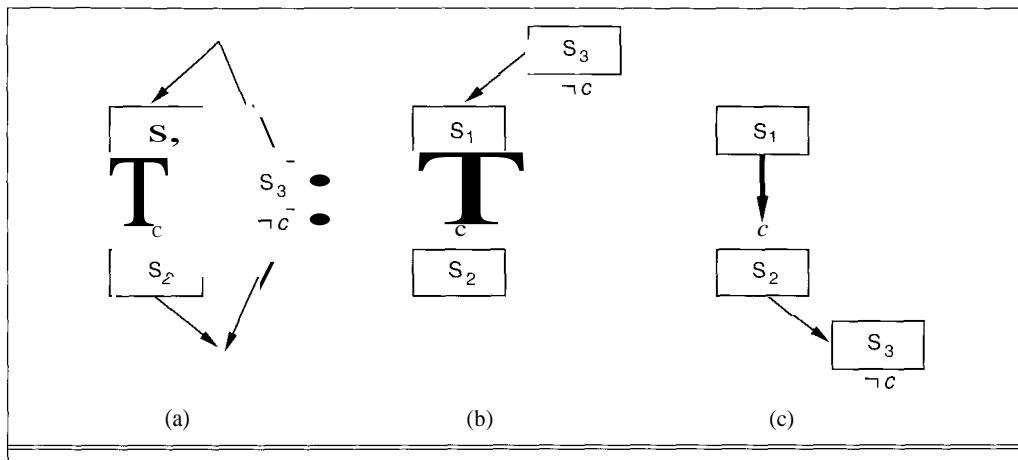
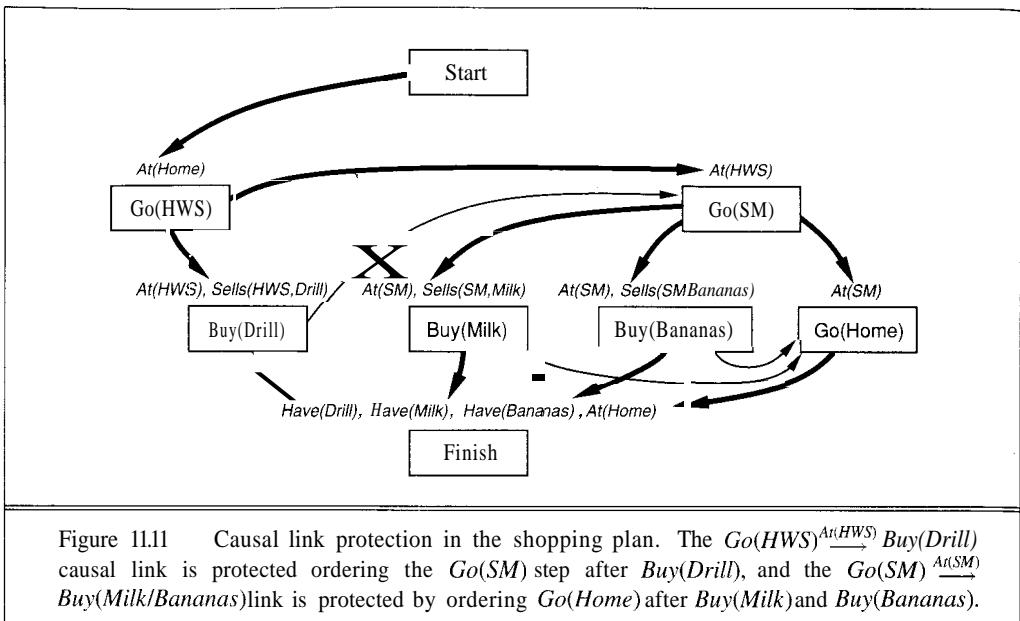


Figure 11.10 Protecting causal links. In (a), the step  $S_3$  threatens a condition  $c$  that is established by  $S_1$  and protected by the causal link from  $S_1$  to  $S_2$ . In (b),  $S_3$  has been demoted to come before  $S_1$ , and in (c) it has been promoted to come after  $S_2$ .

In Figure 11.9, there is no way to resolve the threat that each *Go* step poses to the other. Whichever *Go* step comes first will delete the *At(Home)* condition on the other step. Whenever the planner is unable to resolve a threat by promotion or demotion, it gives up on the partial plan and backs up to a try a different choice at some earlier point in the planning process.

Suppose the next choice is to try a different way to achieve the *At(x)* precondition of the *Go(SM)* step, this time by adding a causal link from *Go(HWS)* to *Go(SM)*. In other words, the plan is to go from home to the hardware store and then to the supermarket. This introduces another threat. Unless the plan is further refined, it will allow the agent to go from the hardware store to the supermarket without first buying the drill (which was why it went to the hardware store in the first place). However much this might resemble human behavior, we would prefer our planning agent to avoid such forgetfulness. Technically, the *Go(SM)* step threatens the *At(HWS)* precondition of the *Buy(Drill)* step, which is protected by a causal link. The threat is resolved by constraining *Go(SM)* to come after *Buy(Drill)*. Figure 11.11 shows this.



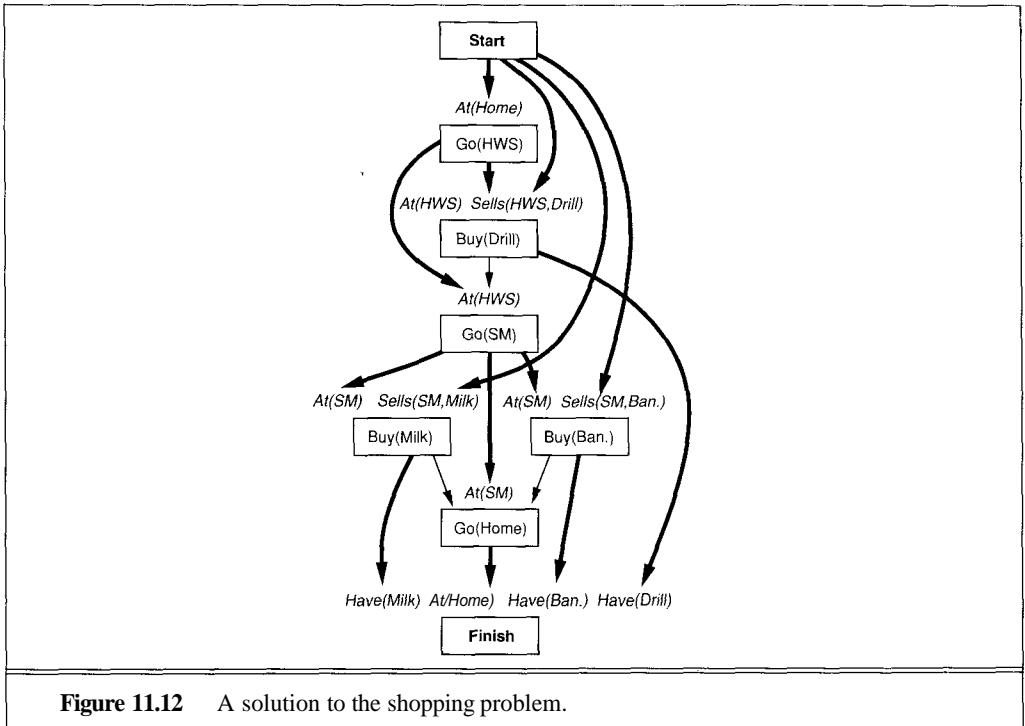
Only the  $\text{At(Home)}$  precondition of the *Finish* step remains unachieved. Adding a  $\text{Go(Home)}$  step achieves it, but introduces an  $\text{At}(x)$  precondition that needs to be achieved.<sup>6</sup> Again, the protection of causal links will help the planner decide how to do this:

- If it tries to achieve  $\text{At}(x)$  by linking to  $\text{At(Home)}$  in the initial state, there will be no way to resolve the threats caused by  $\text{go(HWS)}$  and  $\text{Go(SM)}$ .
- If it tries to link  $\text{At}(x)$  to the  $\text{Go(HWS)}$  step, there will be no way to resolve the threat posed by the  $\text{Go(SM)}$  step, which is already constrained to come after  $\text{Go(HWS)}$ .
- A link from  $\text{Go(SM)}$  to  $\text{At}(x)$  means that  $x$  is bound to  $\text{SM}$ , so that now the  $\text{Go(Home)}$  step deletes the  $\text{At(SM)}$  condition. This results in threats to the  $\text{At(SM)}$  preconditions of  $\text{Buy(Milk)}$  and  $\text{Buy(Bananas)}$ , but these can be resolved by ordering  $\text{Go(Home)}$  to come after these steps (Figure 11.11).

Figure 11.12 shows the complete solution plan, with the steps redrawn to reflect the ordering constraints on them. The result is an almost totally ordered plan; the only ambiguity is that  $\text{Buy(Milk)}$  and  $\text{Buy(Bananas)}$  can come in either order.

Let us take stock of what our partial-order planner has accomplished. It can take a problem that would require many thousands of search states for a problem-solving approach, and solve it with only a few search states. Moreover, the least commitment nature of the planner means it only needs to search at all in places where subplans interact with each other. Finally, the causal links allow the planner to recognize when to abandon a doomed plan without wasting a lot of time expanding irrelevant parts of the plan.

<sup>6</sup> Notice that the  $\text{Go(Home)}$  step also has the effect  $\neg\text{At}(x)$ , meaning that the step will delete an  $\text{At}$  condition for some location yet to be decided. This is a **possible threat** to protected conditions in the plan such as  $\text{At(SM)}$ , but we will not worry about it for now. Possible threats are dealt with in Section 11.7.



**Figure 11.12** A solution to the shopping problem.

## 11.6 A PARTIAL-ORDER PLANNING ALGORITHM

In this section, we develop a more formal algorithm for the planner sketched in the previous section. We call the algorithm POP, for Partial-Order Planner. The algorithm appears in Figure 11.13. (Notice that POP is written as a *nondeterministic* algorithm, using **choose** and **fail** rather than explicit loops. Nondeterministic algorithms are explained in Appendix B.)

POP starts with a minimal partial plan, and on each step extends the plan by achieving a precondition  $c$  of a step  $S_{need}$ . It does this by choosing some operator—either from the existing steps of the plan or from the pool of operators—that achieves the precondition. It records the causal link for the newly achieved precondition, and then resolves any threats to causal links. The new step may threaten an existing causal link or an existing step may threaten the new causal link. If at any point the algorithm fails to find a relevant operator or resolve a threat, it backtracks to a previous choice point. An important subtlety is that the selection of a step and precondition in SELECT-SUBGOAL is *not* a candidate for backtracking. The reason is that every precondition needs to be considered eventually, and the handling of preconditions is commutative: handling  $c_1$  and then  $c_2$  leads to exactly the same set of possible plans as handling  $c_2$  and then  $c_1$ . So we

```

function POP(initial, goal, operators) returns plan
  plan  $\leftarrow$  MAKE-MINIMAL-PLAN(initial, goal)
  loop do
    if SOLUTION?(plan) then return plan
     $S_{need}, c \leftarrow$  SELECT-SUBGOAL(plan)
    CHOOSE-OPERATOR(plan, operators, Sneed, c)
    RESOLVE-THREATS(plan)
  end

function SELECT-SUBGOAL(plan) returns Sneed, c
  pick a plan step Sneed from STEPS(plan)
  with a precondition c that has not been achieved
  return Sneed, c

procedure CHOOSE-OPERATOR(plan, operators, Sneed, c)
  choose a step Sadd from operators or STEPS(plan) that has c as an effect
  if there is no such step then fail
  add the causal link  $S_{add} \xrightarrow{c} S_{need}$  to LINKS(plan)
  add the ordering constraint  $S_{add} \prec S_{need}$  to ORDERINGS(plan)
  if Sadd is a newly added step from operators then
    add Sadd to STEPS(plan)
    addStart  $\prec S_{add} \prec$  Finish to ORDERINGS(plan)
  procedure RESOLVE-THREATS(plan)
  for each Sthreat that threatens a link  $S_i \xrightarrow{c} S_j$  in LINKS(plan) do
    choose either
      Promotion: Add  $S_{threat} \prec S_i$  to ORDERINGS(plan)
      Demotion: Add  $S_j \prec S_{threat}$  to ORDERINGS(plan)
    if not CONSISTENT(plan) then fail
  end

```

**Figure 11.13** The partial-order planning algorithm, POP.

can just pick a precondition and move ahead without worrying about backtracking. The pick we make affects only the speed, and not the possibility, of finding a solution.

Notice that POP is a regression planner, because it starts with goals that need to be achieved and works backwards to find operators that will achieve them. Once it has achieved all the preconditions of all the steps, it is done; it has a solution. POP *is sound and complete*. Every plan it returns is in fact a solution, and if there is a solution, then it will be found (assuming a breadth-first or iterative deepening search strategy). At this point, we suggest that the reader return to the example of the previous section, and trace through the operation of POP in detail.



## 11.7 PLANNING WITH PARTIALLY INSTANTIATED OPERATORS

POSSIBLE THREAT

The version of POP in Figure 11.13 outlines the algorithm, but leaves some details unspecified. In particular, it does not deal with variable binding constraints. For the most part, all this entails is being diligent about keeping track of binding lists and unifying the right expressions at the right time. The implementation techniques of Chapter 10 are applicable here.

There is one substantive decision to make: in RESOLVE-THREATS, should an operator that has the effect, say,  $\neg At(x)$  be considered a threat to the condition  $At(Home)$ ? Currently we can distinguish between threats and non-threats, but this is a **possible threat**. There are three main approaches to dealing with possible threats:

- **Resolve now with an equality constraint:** Modify RESOLVE-THREATS so that it resolves all possible threats as soon as they are recognized. For example, when the planner chooses the operator that has the effect  $\neg At(x)$ , it would add a binding such as  $x = HWS$  to make sure it does not threaten  $At(Home)$ .
- **Resolve now with an inequality constraint:** Extend the language of variable binding constraints to allow the constraint  $x \neq Home$ . This has the advantage of being a lower commitment—it does not require an arbitrary choice for the value of  $x$ —but it is a little more complicated to implement, because the unification routines we have used so far all deal with equalities, not inequalities.
- **Resolve later:** The third possibility is to ignore possible threats, and only deal with them when they become *necessary* threats. That is, RESOLVE-THREATS would not consider  $\neg At(x)$  to be a threat to  $At(Home)$ . But if the constraint  $x = Home$  were ever added to the plan, then the threat would be resolved (by promotion or demotion). This approach has the advantage of being low commitment, but has the disadvantage of making it harder to decide if a plan is a solution.

Figure 11.14 shows an implementation of the changes to CHOOSE-OPERATOR, along with the changes to RESOLVE-THREATS that are necessary for the third approach. It is certainly possible (and advisable) to do some bookkeeping so that RESOLVE-THREATS will not need to go through a triply nested loop on each call.

When partially instantiated operators appear in plans, the criterion for solutions needs to be refined somewhat. In our earlier definition (page 349), we were concerned mainly with the question of partial ordering; a solution was defined as a partial plan such that all linearizations are guaranteed to achieve the goal. With partially instantiated operators, we also need to ensure that all instantiations will achieve the goal. We therefore extend the definition of achievement for a step in a plan as follows:

A step  $S_i$  **achieves** a precondition  $c$  of the step  $S_j$  if (1)  $S_i \prec S_j$  and  $S_i$  has an effect that necessarily unifies with  $c$ ; and (2) there is no step  $S_k$  such that  $S_i \prec S_k \prec S_j$  in some linearization of the plan, and  $S_k$  has an effect that possibly unifies with  $\neg c$ .

The POP algorithm can be seen as constructing a proof that the each precondition of the goal step is achieved. CHOOSE-OPERATOR comes up with the  $S_i$  that achieves (1), and RESOLVE-THREATS makes sure that (2) is satisfied by promoting or demoting possible threats. The tricky part is that

```

procedure CHOOSE-OPERATOR(plan, operators, Sneed, c)
  choose a step Sadd from operators or STEPS(plan) that has cadd as an effect
    such that u = UNIFY(C, cadd, BINDINGS(plan))
  if there is no such step
    then fail
    add u to BINDINGS(plan)
    add Sadd  $\xrightarrow{c}$  Sneed to LINKS(plan)
    add Sadd  $\prec$  Sneed to ORDERINGS(plan)
  if Sadd is a newly added step from operators then
    add Sadd to STEPS(plan)
    add Start  $\prec$  Sadd  $\prec$  Finish to ORDERINGS(plan)

procedure RESOLVE-THREATS(plan)
  for each Si  $\xrightarrow{c}$  Sj in LINKS(plan) do
    for each Sthreat in STEPS(plan) do
      for each c' in EFFECT(Sthreat) do
        if SUBST(BINDINGS(plan), c) = SUBST(BINDINGS(plan),  $\neg c'$ ) then
          choose either
            Promotion: Add Sthreat  $\prec$  Si to ORDERINGS(plan)
            Demotion: Add Sj  $\prec$  Sthreat to ORDERINGS(plan)
        if not CONSISTENT(plan)
          then fail
      end
    end
  end

```

**Figure 11.14** Support for partially instantiated operators in POP.

if we adopt the "resolve-later" approach, then there will be possible threats that are not resolved *yet*. We therefore need some way of checking that these threats are all gone before we **return** the plan. It turns out that if the initial state contains no variables and if every operator mentions all its variables in its precondition, then any complete plan generated by POP is guaranteed to be fully instantiated. Otherwise we will need to change the function SOLUTION? to check that **there** are no uninstantiated variables and choose bindings for them if there are. If this is done, then POP is guaranteed to be a sound planner in all cases.

It is harder to see that POP is complete—that is, finds a solution whenever one exists—but again it comes down to understanding how the algorithm mirrors the definition of achievement. The algorithm generates every possible plan that satisfies part (1), and then filters out those plans that do not satisfy part (2) or that are inconsistent. Thus, if there is a plan that is a solution, POP will find it. So if you accept the definition of **solution** (page 349), you should accept that POP is a sound and complete planner.

## 11.8 KNOWLEDGE ENGINEERING FOR PLANNING

The methodology for solving problems with the planning approach is very much like the general knowledge engineering guidelines of Section 8.2:

- Decide what to talk about.
- Decide on a vocabulary of conditions (literals), operators, and objects.
- Encode operators for the domain.
- Encode a description of the specific problem instance.
- Pose problems to the planner and get back plans.

We will cover each of these five steps, demonstrating them in two domains.

### The blocks world

**What to talk about:** The main consideration is that operators are so restricted in what they can express (although Chapter 12 relaxes some of the restrictions). In this section we show how to define knowledge for a classic planning domain: the blocks world. This domain consists of a set of cubic blocks sitting on a table. The blocks can be stacked, but only one block can fit directly on top of another. A robot arm can pick up a block and move it to another position, either on the table or on top of another block. The arm can only pick up one block at a time, so it cannot pick up a block that has another one on it. The goal will always be to build one or more stacks of blocks, specified in terms of what blocks are on top of what other blocks. For example, a goal might be to make two stacks, one with block *A* on *B*, and the other with *C* on *D*.

**Vocabulary:** The objects in this domain are the blocks and the table. They are represented by constants. We will use *On*(*b*,*x*) to indicate that block *b* is on *x*, where *x* is either another block or the table. The operator for moving block *b* from a position on top of *x* to a position on top *y* will be *Move*(*b*,*x*,*y*). Now one of the preconditions on moving *b* is that no other block is on it. In first-order logic this would be  $\neg\exists x \text{ } On(x, b)$  or alternatively  $\forall x \text{ } \neg On(x, b)$ . But our language does not allow either of these forms, so we have to think of something else. The trick is to invent a predicate to represent the fact that no block is on *b*, and then make sure the operators properly maintain this predicate. We will use *Clear*(*x*) to mean that nothing is on *x*.

**Operators:** The operator *Move* moves a block *b* from *x* to *y* if both *b* and *y* are clear, and once the move is made, *x* becomes clear but *y* is clear no longer. The formal description of *Move* is as follows:

*Op(ACTION:Move(*b*,*x*, *y*),*  
*PRECOND:* *On*(*b*,*x*) A *Clear*(*b*) A *Clear*(*y*),  
*EFFECT:* *On*(*b*,*y*) A *Clear*(*x*) A  $\neg On(b, x)$  A  $\neg Clear(y))$

Unfortunately, this operator does not maintain *Clear* properly when *x* or *y* is the table. When *x* = *Table*, this operator has the effect *Clear(Table)*, but the table should not become clear, and when *y* = *Table*, it has the precondition *Clear(Table)*, but the table does not have to be clear to

move a block onto it. To fix this, we do two things. First, we introduce another operator to move a block  $b$  from  $x$  to the table:

```
Op(ACTION:MoveToTable(b,x),
    PRECOND:On(b,x) A Clear(b),
    EFFECT:On(b,Table) A Clear(x)A  $\neg$ On(b,x))
```

Second, we take the interpretation of  $Clear(x)$  to be "there is a clear space on  $x$  to hold a block." Under this interpretation,  $Clear(Table)$  will always be part of the initial situation, and it is proper that  $Move(b, Table, y)$  has the effect  $Clear(Table)$ . The only problem is that nothing prevents the planner from using  $Move(b, x, Table)$  instead of  $MoveToTable(b, x)$ . We could either live with this problem—it will lead to a larger-than-necessary search space, but will not lead to incorrect answers—or we could introduce the predicate  $Block$  and add  $Block(b)$  A  $Block(y)$  to the precondition of  $Move$ .

Finally, there is the problem of spurious operations like  $Move(B, C, C)$ , which should be a no-op, but which instead has contradictory effects. It is common to ignore problems like this, because they tend not to have any effect on the plans that are produced. To really fix the problem, we need to be able to put inequalities in the precondition:  $b \neq x \neq y$ .

## Shakey's world

The original STRIPS program was designed to control Shakey,<sup>7</sup> a robot that roamed the halls of SRI in the early 1970s. It turns out that most of the work on STRIPS involved simulations where the actions performed were just printing to a terminal, but occasionally Shakey would actually move around, grab, and push things, based on the plans created by STRIPS. Figure 11.15 shows a version of Shakey's world consisting of four rooms lined up along a corridor, where each room has a door and a light switch.

Shakey can move from place to place, push movable objects (such as boxes), climb on and off of rigid objects (such as boxes), and turn light switches on and off. We will develop the vocabulary of literals along with the operators:

1. Go from current location to location  $y$ :  $Go(y)$

This is similar to the  $Go$  operator used in the shopping problem, but somewhat restricted.

The precondition  $At(Shakey, x)$  establishes the current location, and we will insist that  $x$  and  $y$  be *In* the same room:  $In(x, r)$  A  $In(y, r)$ . To allow Shakey to plan a route from room to room, we will say that the door between two rooms is *In* both of them.

2. Push an object  $b$  from location  $x$  to location  $y$ :  $Push(b, x, y)$

Again we will insist that the locations be in the same room. We introduce the predicate  $Pushable(b)$ , but otherwise this is similar to  $Go$ .

3. Climb up onto a box:  $Climb(b)$ .

We introduce the predicate  $On$  and the constant  $Floor$ , and make sure that a precondition of  $Go$  is  $On(Shakey, Floor)$ . For  $Climb(b)$ , the preconditions are that Shakey is A? the same place as  $b$ , and  $b$  must be *Climbable*.

<sup>7</sup> Shakey's name comes from the fact that its motors made it a little unstable when it moved.

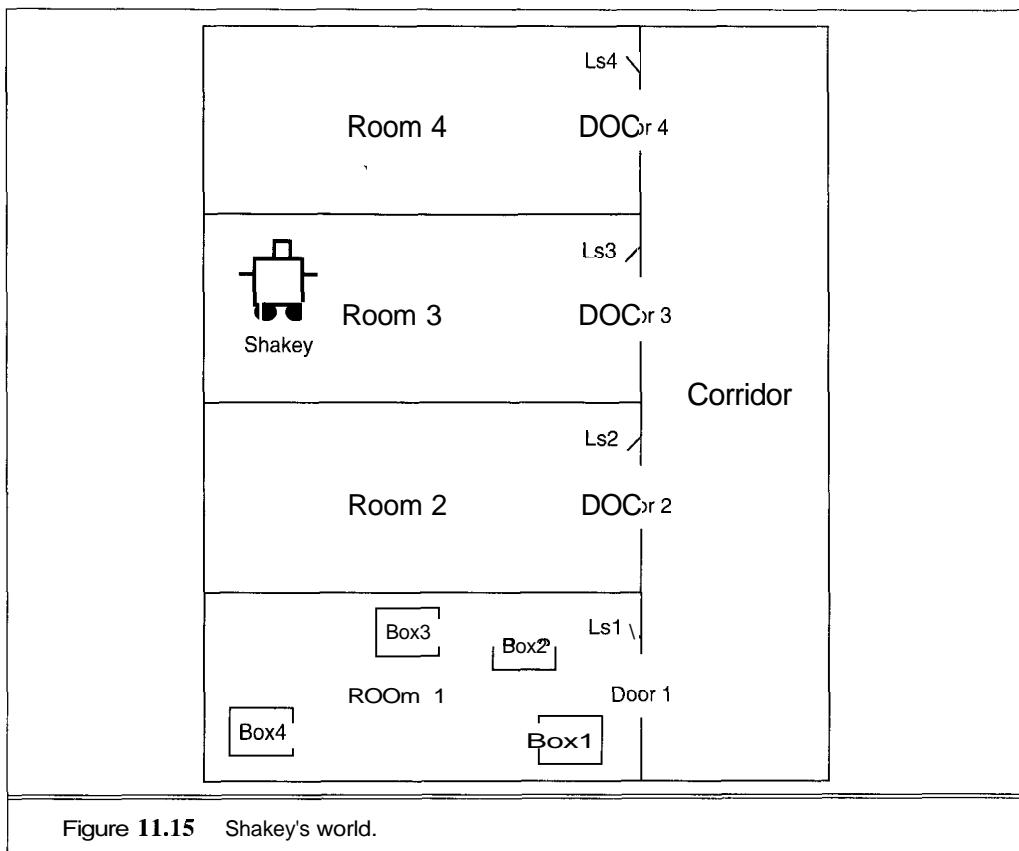


Figure 11.15 Shakey's world.

4. Climb down from a box: *Down(b)*.  
This just undoes the effects of a *Climb*.
5. Turn a light switch on: *TurnOn(ls)*.  
Because Shakey is short, this can only be done when Shakey is on top of a box that is at the light switch's location.<sup>8</sup>
6. Turn a light switch off: *TurnOff(ls)*.  
This is similar to *TurnOn*. Note that it would not be possible to represent toggling a light switch as a STRIPS action, because there are no conditionals in the language to say that the light becomes on if it was off and off if it was on. (Section 12.4 will add conditionals to the language.)

In situation calculus, we could write an axiom to say that every box is pushable and climbable. But in STRIPS, we have to include individual literals for each box in the initial state. We also have to include the complete map of the world in the initial state, in terms of what objects are *In* which

<sup>8</sup> Shakey was never dexterous enough to climb on a box or toggle a switch, but STRIPS was capable of finding plans using these actions.

rooms, and which locations they are *At*. We leave this, and the specification of the operators, as an exercise.

In conclusion, it is possible to represent simple domains with STRIPS operators, but it requires ingenuity in coming up with the right set of operators and predicates in order to stay within the syntactic restrictions that the language imposes.

## 11.9 SUMMARY

In this chapter, we have defined the planning problem and shown that situation calculus is expressive enough to deal with it. Unfortunately, situation calculus planning using a general-purpose theorem prover is very inefficient. Using a restricted language and special-purpose algorithms, planning systems can solve quite complex problems. Thus, planning comes down to an exercise in finding a language that is just expressive enough for the problems you want to solve, but still admits a reasonably efficient algorithm. The points to remember are as follows:

- Planning agents use lookahead to come up with actions that will contribute to goal achievement. They differ from problem-solving agents in their use of more flexible representations of states, actions, goals, and plans.
- The STRIPS language describes actions in terms of their preconditions and effects. It captures much of the expressive power of situation calculus, but not all domains and problems that can be described in the STRIPS language.
- It is not feasible to search through the space of situations in complex domains. Instead we search through the space of plans, starting with a minimal plan and extending it until we find a solution. For problems in which most subplans do not interfere with each other, this will be efficient.
- The principle of least commitment says that a planner (or any search algorithm) should avoid making decisions until there is a good reason to make a choice. Partial-ordering constraints and uninstantiated variables allow us to follow a least-commitment approach.
- The causal link is a useful data structure for recording the purposes for steps. Each causal link establishes a protection interval over which a condition should not be deleted by another step. Causal links allow early detection of unresolvable conflicts in a partial plan, thereby eliminating fruitless search.
- The POP algorithm is a sound and complete algorithm for planning using the STRIPS representation.
- The ability to handle partially instantiated operators in POP reduces the need to commit to concrete actions with fixed arguments, thereby improving efficiency.

---

## BIBLIOGRAPHICAL AND HISTORICAL NOTES

The roots of AI planning lie partly in problem solving through state-space search and associated techniques such as problem reduction and means-ends analysis, especially as embodied in Newell and Simon's GPS, and partly in theorem proving and situation calculus, especially as embodied in the QA3 theorem proving system (Green, 1969b). Planning has also been historically motivated by the needs of robotics. STRIPS (Fikes and Nilsson, 1971), the first major planning system, illustrates the interaction of these three influences. STRIPS was designed as the planning component of the software for the Shakey robot project at SRI International. Its overall control structure was modeled on that of GPS, and it used a version of QA3 as a subroutine for establishing preconditions for actions. Lifschitz (1986) offers careful criticism and formal analysis of the STRIPS system. Bylander (1992) shows simple planning in the fashion of STRIPS to be PSPACE-complete. Fikes and Nilsson (1993) give a historical retrospective on the STRIPS project and a survey of its relationship to more recent planning efforts.

LINEAR  
NONLINEAR  
NONINTERLEAVED

For several years, terminological confusion has reigned in the field of planning. Some authors (Genesereth and Nilsson, 1987) use the term **linear** to mean what we call totally ordered, and **nonlinear** for partially ordered. Sacerdoti (1975), who originated the term, used "linear" to refer to a property that we will call **noninterleaved**. Given a set of subgoals, a noninterleaved planner can find plans to solve each subgoal, but then it can only combine them by placing all the steps for one subplan before or after all the steps of the others. Many early planners of the 1970s were noninterleaved, and thus were incomplete—they could not always find a solution when one exists. This was forcefully driven home by the Sussman Anomaly (see Exercise 11.4), found during experimentation with the HACKER system (Sussman, 1975). (The anomaly was actually found by Alien Brown, not by Sussman himself, who thought at the time that assuming linearity to begin with was often a workable approach.) HACKER introduced the idea of protecting subgoals, and was also an early example of plan learning.

Goal regression planning, in which steps in a totally ordered plan are reordered so as to avoid conflict between subgoals, was introduced by Waldinger (1975) and also used by Warren's (1974) WARPLAN. WARPLAN is also notable in that it was the first planner to be written using a logic programming language (Prolog), and is one of the best examples of the remarkable economy that can sometimes be gained by using logic programming: WARPLAN is only 100 lines of code, a small fraction of the size of comparable planners of the time. INTERPLAN (Tate, 1975b; Tate, 1975a) also allowed arbitrary interleaving of plan steps to overcome the Sussman anomaly and related problems.

The construction of partially ordered plans (then called **task networks**) was pioneered by the NOAH planner (Sacerdoti, 1975; Sacerdoti, 1977), and thoroughly investigated in Tate's (1977) NONLIN system, which also retained the clear conceptual structure of its predecessor INTERPLAN. INTERPLAN and NONLIN provide much of the grounding for the work described in this chapter and the next, particularly in the use of causal links to detect potential protection violations. NONLIN was also the first planner to use an explicit algorithm for determining the truth or falsity of conditions at various points in a partially specified plan.

TWEAK (Chapman, 1987) formalizes a generic, partial-order planning system. Chapman provides detailed analysis, including proofs of completeness and intractability (NP-hardness and

undecidability) of various formulations of the planning problem and its subcomponents. The POP algorithm described in the chapter is based on the SNLP algorithm (Soderland and Weld, 1991), which is an implementation of the planner described by McAllester and Rosenblitt (1991). Weld contributed several useful suggestions to the presentation in this chapter.

A number of important papers on planning were presented at the Timberline workshop in 1986, and its proceedings (Georgeff and Lansky, 1986) are an important source. *Readings in Planning* (Alien *et al.*, 1990) is a comprehensive anthology of many of the best articles in the field, including several good survey articles. *Planning and Control* (Dean and Wellman, 1991) is a good general introductory textbook on planning, and is particularly remarkable because it makes a particular effort to integrate classical AI planning techniques with classical and modern control theory, metareasoning, and reactive planning and execution monitoring. Weld (1994) provides an excellent survey of modern planning algorithms.

Planning research has been central to AI since its inception, and papers on planning are a staple of mainstream AI journals and conferences, but there are also specialized conferences devoted exclusively to planning, like the Timberline workshop, the 1990 DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control, or the International Conferences on AI Planning Systems.

---

## EXERCISES

**11.1** Define the operator schemata for the problem of putting on shoes and socks and a hat and coat, assuming that there are no preconditions for putting on the hat and coat. Give a partial-order plan that is a solution, and show that there are 180 different linearizations of this solution.

**11.2** Let us consider a version of the milk/banana/drill shopping problem in which money is included, at least in a simple way.

- a. Let CC denote a credit card that the agent can use to buy any object. Modify the description of *Buy* so that the agent has to have its credit card in order to buy anything.
- b. Write a *PickUp* operator that enables the agent to *Have* an object if it is portable and at the same location as the agent.
- c. Assume that the credit card is at home, but *Have(CC)* is initially false. Construct a partially ordered plan that achieves the goal, showing both ordering constraints and causal links.
- d. Explain in detail what happens during the planning process when the agent explores a partial plan in which it leaves home without the card.

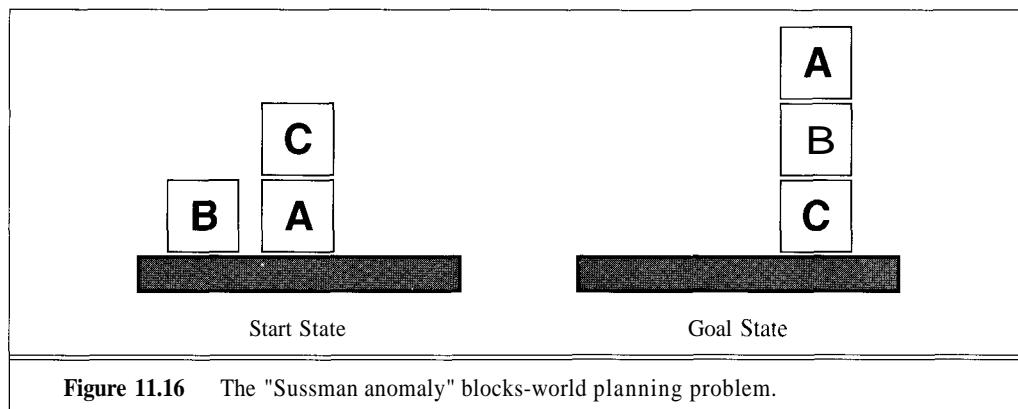
**11.3** There are many ways to characterize planners. For each of the following dichotomies, explain what they mean, and how the choice between them affects the efficiency and completeness of a planner.

- a. Situation space vs. plan space.
- b. Progressive vs. regressive.

- c. Refinement vs. debugging.
- d. Least commitment vs. more commitment.
- e. Bound variables vs. unbound variables.
- f. Total order vs. partial order.
- g. Interleaved vs. noninterleaved.
- h. Unambiguous preconditions vs. ambiguous preconditions.**
- i. Systematic vs. unsystematic.

SUSSMAN ANOMALY

**11.4** Figure 11.16 shows a blocks-world planning problem known as the **Sussman anomaly**. The problem was considered anomalous because the noninterleaved planners of the early 1970s could not solve it. Encode the problem using STRIPS operators, and use POP to solve it.



**Figure 11.16** The "Sussman anomaly" blocks-world planning problem.

**11.5** Suppose that you are the proud owner of a brand new time machine. That means that you can perform actions that affect situations in the past. What changes would you have to make to the planners in this chapter to accommodate such actions?



**11.6** The POP algorithm shown in the text is a regression planner, because it adds steps whose effects satisfy unsatisfied conditions in the plan. Progression planners add steps whose preconditions are satisfied by conditions known to be true in the plan. Modify POP so that it works as a progression planner, and compare its performance to the original on several problems of your choosing.

**11.7** In this exercise, we will look at planning in Shakey's world.

- a. Describe Shakey's six actions in situation calculus notation.
- b. Translate them into the STRIPS language.
- c. Either manually or using a partial-order planner, construct a plan for Shakey to get *Box2* into *Room2* from the starting configuration in Figure 11.15.
- d. Suppose Shakey has  $n$  boxes in a room and needs to move them all into another room. What is the complexity of the planning process in terms of  $n$ ?

**11.8** POP is a nondeterministic algorithm, and has a choice about which operator to add to the plan at each step and how to resolve each threat. Can you think of any domain-independent heuristics for ordering these choices that are likely to improve POP's efficiency? Will they help in Shakey's world? Are there any additional, domain-dependent heuristics that will improve the efficiency still further?

**11.9** In this exercise we will consider the monkey-and-bananas problem, in which there is a monkey in a room with some bananas hanging out of reach from the ceiling, but a box is available that will enable the monkey to reach the bananas if he climbs on it. Initially, the monkey is at A, the bananas at B, and the box at C. The monkey and box have height *Low*, but if the monkey climbs onto the box he will have height *High*, the same as the bananas. The actions available to the monkey include *Go* from one place to another, *Push* an object from one place to another, *Climb* onto an object, and *Grasp* an object. Grasping results in holding the object if the monkey and object are in the same place at the same height.

- a. Write down the initial state description in predicate calculus.
- b. Write down STRIPS-style definitions of the four actions, providing at least the obvious preconditions.
- c. Suppose the monkey wants to fool the scientists, who are off to tea, by grabbing the bananas but leaving the box in its original place. Write this as a general goal (i.e., not assuming the box is necessarily at C) in the language of situation calculus. Can this goal be solved by a STRIPS-style system?
- d. Your axiom for pushing is probably incorrect, because if the object is too heavy, its position will remain the same when the *Push* operator is applied. Is this an example of the frame problem or the qualification problem?

# 12 PRACTICAL PLANNING

*In which planning algorithms meet the real world and survive, albeit with some significant modifications.*

## 12.1 PRACTICAL PLANNERS

Chapter 11 showed how a partial-order planner's search through the space of plans can be more efficient than a problem-solver's search through the space of situations. On the other hand, the POP planner can only handle problems that are stated in the STRIPS language, and its search process is so unguided that it can still only be used for small problems. In this chapter we begin by surveying existing planners that operate in complex, realistic domains. This will help to pinpoint the weaknesses of POP and suggest the necessary extensions. We then show how the planning language and algorithms of Chapter 11 can be extended and the search focused to handle domains like these.

### **Spacecraft assembly, integration, and verification**

OPTIMUM-AIV is a planner that is used by the European Space Agency to help in the assembly, integration, and verification (AIV) of spacecraft. The system is used both to generate plans and to monitor their execution. During monitoring, the system reminds the user of upcoming activities, and can suggest repairs to the plan when an activity is performed late, cancelled, or reveals something unexpected. In fact, the ability to quickly replan is the principal objective of OPTIMUM-AIV. The system does not execute the plans; that is done by humans with standard construction and test equipment.

In complex projects like this, it is common to use scheduling tools from operations research (OR) such as PERT charts or the critical path method. These tools essentially take a *hand-constructed* complete partial-order plan and generate an optimal schedule for it. Actions are

treated as objects that take up time and have ordering constraints; their effects are ignored. This avoids the need for knowledge engineering, and for one-shot problems it may be the most appropriate solution. For most practical applications, however, there will be many related problems to solve, so it is worth the effort to describe the domain and then have the plans automatically generated. This is especially important during the *execution* of plans. If a step of a plan fails, it is often necessary to replan quickly to get the project back on track. PERT charts do not contain the causal links and other information needed to see how to fix a plan, and human replanning is often too slow.

The success of real-world AI systems requires integration into the environment in which they operate. It is vital that a planner be able to access existing databases of project information in whatever format they might have, and that the planner's input and output representations be in a form that is both expressive and easily understood by users. The STRIPS language is insufficient for the AIV domain because it cannot express four key concepts:

1. **Hierarchical plans:** Obviously, launching a spacecraft is more complicated than shopping for groceries. One way to handle the increased complexity is to specify plans at varying levels of detail. The top-level plan might be: prepare booster rocket, prepare capsule, load cargo, and launch. There might be a dozen intermediate levels before we finally get down to the level of executable actions: insert nut *A* into hole *B* and fasten with bolt *C*. Adding the ability to represent hierarchical plans can make the difference between feasible and infeasible computation, and it can make the resulting plan easier to understand. It also allows the user to provide guidance to the planner in the form of a partially specified, abstract plan for which the planner can fill in the details.
2. **Complex conditions:** STRIPS operators are essentially propositional. True, they do allow variables, but the variables are used in a very limited way. For example, there is no universal quantification, and without it we cannot describe the fact that the *Launch* operator causes *all* the objects that are in the spacecraft to go into orbit. Similarly, STRIPS operators are unconditional: we cannot express the fact that if all systems are go, then the *Launch* will put the spacecraft into orbit, otherwise it will put it into the ocean.
3. **Time:** Because the STRIPS language is based on situation calculus it assumes that all actions occur instantaneously, and that one action follows another with no break in between. Real-world projects need a better model of time. They must represent the fact that projects have deadlines (the spacecraft must be launched on June 17), actions have durations (it takes 6 hours to test the *XYZ* assembly), and steps of plans may have time windows (the machine that tests the *XYZ* assembly is available from May 1 to June 1 (except weekends), but it must be reserved one week ahead of time). Indeed, the major contribution of traditional OR techniques is to satisfy time constraints for a complete partial-order plan.
4. **Resources:** A project normally has a budget that cannot be exceeded, so the plan must be constrained to spend no more money than is available. Similarly, there are limits on the number of workers that are available, and on the number of assembly and test stations. Resource limitations may be placed on the number of things that may be used at one time (e.g., people) or on the total amount that may be used (e.g., money). Action descriptions must incorporate resource consumption and generation, and planning algorithms must be able to handle constraints on resources efficiently.

OPTIMUM-AIV is based on the open planning architecture O-PLAN (Currie and Tate, 1991). O-PLAN is similar to the POP planner of Chapter 11, except that it is augmented to accept a more expressive language that can represent time, resources, and hierarchical plans. It also accepts heuristics for guiding the search and records its reasons for each choice, which makes it easier to replan when necessary. O-PLAN has been applied to a variety of problems, including software procurement planning at Price Waterhouse, back axle assembly process planning at Jaguar Cars, and complete factory production planning at Hitachi.

## Job shop scheduling

The problem that a factory solves is to take in raw materials and components, and assemble them into finished products. The problem can be divided into a planning task (deciding what assembly steps are going to be performed) and a scheduling task (deciding when and where each step will be performed). In many modern factories, the planning is done by hand and the scheduling is done with an automated tool.

O-PLAN is being used by Hitachi for job shop planning and scheduling in a system called TOSCA. A typical problem involves a product line of 350 different products, 35 assembly machines, and over 2000 different operations. The planner comes up with a 30-day schedule for three 8-hour shifts a day. In general, TOSCA follows the partial-order, least-commitment planning approach. It also allows for "low-commitment" decisions: choices that impose constraints on the plan or on a particular step. For example, the system might choose to schedule an action to be carried out on a class of machine without specifying any particular one.

Factories with less diversity of products often follow a fixed plan, but still have a need for automated scheduling. The ISIS system (Fox and Smith, 1984) was developed specifically for scheduling. It was first tested at the Westinghouse turbine component plant in Winston-Salem, NC. The plant makes thousands of different turbine blades, and for each one, there are one or more plans, called process routings. When an order comes in, one of the plans is chosen and a time for it is scheduled. The time depends on the criticality of the order: whether it is an urgent replacement for a failed blade in service, a scheduled maintenance part that has plenty of lead time but must arrive on time, or just a stock order to build up the reserves.

Traditional scheduling methods such as PERT are capable of finding a feasible ordering of steps subject to time constraints, but it turns out that human schedulers using PERT spend 80% to 90% of their time communicating with other workers to discover what the real constraints are. A successful automated scheduler needs to be able to represent and reason with these additional constraints. Factors that are important include the cost of raw materials on hand, the value of finished but unshipped goods, accurate forecasts of future needs, and minimal disruption of existing procedures. ISIS uses a hierarchical, least-commitment search to find high-quality plans that satisfy all of these requirements.

## Scheduling for space missions

Planning and scheduling systems have been used extensively in planning space missions as well as in constructing spacecraft. There are two main reasons for this. First, spacecraft are very

expensive and sometimes contain humans, and any mistake can be costly and irrevocable. Second, space missions take place in space, which does not contain many other agents to mess up the expected effects of actions. Planners have been used by the ground teams for the Hubble space telescope and at least three spacecraft: Voyager, UOSAT-II, and ERS-1. In each case, the goal is to orchestrate the observational equipment, signal transmitters, and attitude- and velocity-control mechanisms, in order to maximize the value of the information gained from observations while obeying resource constraints on time and energy.

Mission scheduling often involves very complex temporal constraints, particularly those involving periodic events. For example, ERS-1, the European Earth Resource Observation satellite, completes an orbit every 100 minutes, and returns to the same point every 72 hours. An observation of a particular point on the earth's surface thus can be made at any one of a number of times, each separated by 72 hours, but at no other time. Satellites also have resource constraints on their power output: they cannot exceed a fixed maximum output, and they must be sure not to discharge too much power over a period of time. Other than that, a satellite can be considered as a job-shop scheduling problem, where the telescopes and other instruments are the machines, and the observations are the products. PlanERS-1 is a planner based on O-PLAN that produces observation plans for the ERS-1.

The Hubble space telescope (HST) is a good example of the need for automated planning tools. After it was launched in April 1990, the primary mirror was found to be out of focus. Using Bayesian techniques for image reconstruction (see Chapter 24), the ground team was able to compensate for the defect to a degree, enabling the HST to deliver novel and important data on Pluto, a gravitational lens, a supernova, and other objects. In 1993, shuttle astronauts repaired most of the problems with the primary mirror, opening up the possibility of a new set of observations. The ground team is constantly learning more about what the HST can and cannot do, and it would be impossible to update the observation plans to reflect this ever-increasing knowledge without automated planning and scheduling tools.

Any astronomer can submit a proposal to the HST observing committee. Proposals are classified as high priority (which are almost always executed and take up about 70% of the available observing time), low priority (which are scheduled as time allows), or rejected. Proposals are received at the rate of about one per day, which means there are more proposals than can be executed. Each proposal includes a machine-readable specification of which instrument should be pointed at which celestial object, and what kind of exposure should be made. Some observations can be done at any time, whereas others are dependent on factors such as the alignment of planets and whether the HST is in the earth's shadow. There are some constraints that are unique to this domain. For example, an astronomer may request periodic observations of a quasar over a period of months or years subject to the constraint that each observation be taken under the same shadow conditions.

The HST planning system is split into two parts. A long-term scheduler, called SPIKE, first schedules observations into one-week segments. The heuristic is to assign high-priority proposals so that they can all be executed within the scheduled segment, and then to pack each segment with extra low-priority proposals until they are about 20% above capacity. This is done a year or more ahead of time. A multiyear schedule with 5000 observations or so can be created in less than an hour, so replanning is easy. After each segment is scheduled, a short-term planner, SPSS, does the detailed planning of each segment, filling in the time between high-priority tasks

with as many low-priority ones as possible. The system also calculates the commands for the platform attitude controls so that the observation plan can be executed. It can check the feasibility of proposals and detailed schedules much faster than human experts.

### Buildings, aircraft carriers, and beer factories

SIPE (System for Interactive Planning and Execution monitoring) was the first planner to deal with the problem of replanning, and the first to take some important steps toward expressive operators. It is similar to O-PLAN in the range of its features and in its applicability. It is not in everyday practical use, but it has been used in demonstration projects in several domains, including planning operations on the flight deck of an aircraft carrier and job-shop scheduling for an Australian beer factory. Another study used SIPE to plan the construction of multistory buildings, one of the most complex domains ever tackled by a planner.

SIPE allows deductive rules to operate over states, so that the user does not have to specify all relevant literals as effects of each operator. It allows for an inheritance hierarchy among object classes and for an expressive set of constraints. This means it is applicable to a wide variety of domains, but its generality comes at a cost. For example, in the building construction domain, it was found that SIPE needed time  $O(n^{2.5})$  for an  $n$ -story building. This suggests a high degree of interaction between stories, when in fact the degree of interaction should be much lower: if you remember to build from the ground up and make sure that the elevator shafts line up, it should be possible to get performance much closer to  $O(n)$ .

The examples in this and the preceding sections give an idea of the state of the art of planning systems. They are good enough to model complex domains, but have not yet gained practical acceptance beyond a few pilot projects. Clearly, to achieve the degree of flexibility and efficiency needed to exceed the capabilities of human planners armed with traditional scheduling tools, we need to go far beyond the limited STRIPS language.

## 12.2 HIERARCHICAL DECOMPOSITION

---

The grocery shopping example of Chapter 11 produced solutions at a rather high level of abstraction. A plan such as

[*Go(Supermarket), Buy(Milk), Buy(Bananas), Go(Home)*]

is a good high-level description of what to do, but it is a long way from the type of instructions that can be fed directly to the agent's effectors. Thus, it is insufficient for an agent that wants to actually *do* anything. On the other hand, a low-level plan such as

[*Forward(1cm), Turn(\ deg), Forward(1cm), ...*]

would have to be many thousands of steps long to solve the shopping problem. The space of plans of that length is so huge that the techniques of Chapter 11 would probably not find a solution in a reasonable amount of time.

HIERARCHICAL  
DECOMPOSITION  
ABSTRACT  
OPERATOR

PRIMITIVE  
OPERATOR

To resolve this dilemma, all the practical planners we surveyed have adopted the idea of **hierarchical decomposition**:<sup>1</sup> that an **abstract operator** can be decomposed into a group of steps that forms a plan that implements the operator. These decompositions can be stored in a library of plans and retrieved as needed.

Consider the problem of building a frame house. The abstract operator *Build(House)* can be decomposed into a plan that consists of the four steps *Obtain Permit*, *Hire Builder*, *Construction*, and *Pay Builder*, as depicted in Figure 12.1. The steps of this plan may in turn be further decomposed into even more specific plans. We show a decomposition of the *Construction* step. We could continue decomposing until we finally get down to the level of *Hammer(Nail)*. The plan is complete when every step is a **primitive operator**—one that can be directly executed by the agent. Hierarchical decomposition is most useful when operators can be decomposed in more than one way. For example, the *Build Walls* operator can be decomposed into a plan involving wood, bricks, concrete, or vinyl.

To make the idea of hierarchical planning work, we have to do two things. (1) Provide an extension to the STRIPS language to allow for nonprimitive operators. (2) Modify the planning algorithm to allow the replacement of a nonprimitive operator with its decomposition.

## Extending the language

To incorporate hierarchical decomposition, we have to make two additions to the description of each problem domain.

First, we partition the set of operators into primitive and nonprimitive operators. In our domain, we might define *Hammer(Nail)* to be primitive and *Build(House)* to be nonprimitive. In general, the distinction between primitive and nonprimitive is relative to the agent that will execute the plan. For the general contractor, an operator such as *Install(FloorBoards)* would be primitive, because all the contractor has to do is order a worker to do the installation. For the worker, *Install(FloorBoards)* would be nonprimitive, and *Hammer(Nail)* would be primitive.

Second, we add a set of decomposition methods. Each method is an expression of the form *Decompose(o,p)*, which means that a nonprimitive operator that unifies with *o* can be decomposed into a plan *p*. Here is decomposition of *Construction* from Figure 12.1:

```

Decompose(Construction,
  Plan(STEPS:{S1 : Build(Foundation) S2 : Build(Frame),
```

*S<sub>3</sub> : Build(Roof), S<sub>4</sub> : Build(Walls),*

*S<sub>5</sub> : Build(Interior)})*

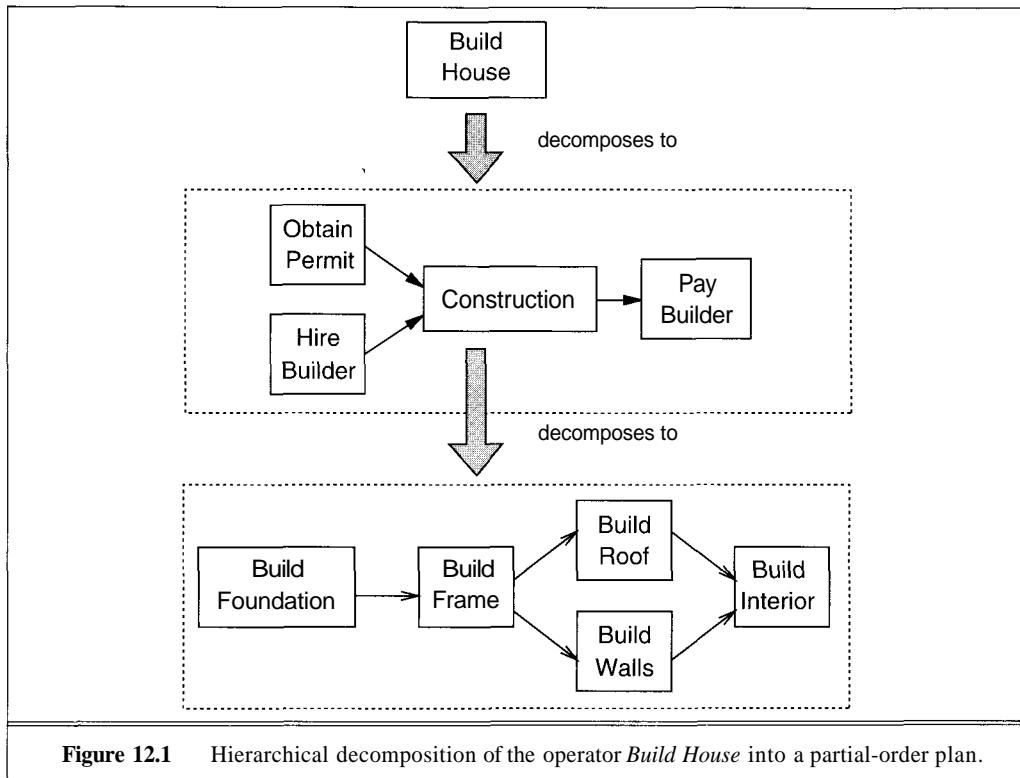
*ORDERINGS:{S<sub>1</sub> < S<sub>2</sub> < S<sub>3</sub> < S<sub>5</sub>, S<sub>2</sub> < S<sub>4</sub> < S<sub>5</sub>},*

*BINDINGS:{},*

*LINKS:{S<sub>1</sub> →<sub>Foundation</sub> S<sub>2</sub>, S<sub>2</sub> →<sub>Frame</sub> S<sub>3</sub>, S<sub>2</sub> →<sub>Frame</sub> S<sub>4</sub>, S<sub>3</sub> →<sub>Roof</sub> S<sub>5</sub>, S<sub>4</sub> →<sub>Walls</sub> S<sub>5</sub>}}))*

A *decomposition method* is like a subroutine or macro definition for an operator. As such, it is important to make sure that the decomposition is a correct implementation of the operator. We

<sup>1</sup> For hierarchical decomposition, some authors use the term **operator reduction** (reducing a high-level operator to a set of lower-level ones), and some use **operator expansion** (expanding a macro-like operator into the structure that implements it). This kind of planning is also called **hierarchical task network** planning.



say that a plan  $p$  correctly implements an operator  $o$  if it is a complete and consistent plan for the problem of achieving the effects of  $o$  given the preconditions of  $o$ :

1.  $p$  must be consistent. (There is no contradiction in the ordering or variable binding constraints of  $p$ .)
2. Every effect of  $o$  must be asserted by at least one step of  $p$  (and is not denied by some other, later step of  $p$ ).
3. Every precondition of the steps in  $p$  must be achieved by a step in  $p$  or be one of the preconditions of  $o$ .

This guarantees that it is possible to replace a nonprimitive operator with its decomposition and have everything hook up properly. Although one will need to check for possible threats arising from interactions between the newly introduced steps and conditions and the existing steps and conditions, there is no need to worry about interactions among the steps of the decomposition itself. Provided there is not too much interaction *between* the different parts of the plan, hierarchical planning allows very complex plans to be built up cumulatively from simpler subplans. It also allows plans to be generated, saved away, and then re-used in later planning problems.

## Modifying the planner

We can derive a hierarchical decomposition planner, which we call HD-POP, from the POP planner of Figure 11.13. The new algorithm is shown in Figure 12.2. There are two principal changes. First, as well as finding ways to achieve unachieved conditions in the plan, the algorithm must find a way to decompose nonprimitive operators. Because both kinds of refinements must be carried out to generate a complete, primitive plan, there is no need to introduce a backtracking choice of one or the other. HD-POP simply does one of each on each iteration; more sophisticated strategies can be used to reduce the branching factor. Second, the algorithm takes a *plan* as input, rather than just a goal. We have already seen that a goal can be represented as a *Start-Finish* plan, so this is compatible with the POP approach. However, it will often be the case that the user has some idea of what general kinds of activities are needed, and allowing more extensive plans as inputs means that this kind of guidance can be provided.

```

function HD-POP(plan, operators, methods) returns plan
  inputs: plan, an abstract plan with start and goal steps (and possibly other steps)

  loop do
    if SOLUTION?(plan) then return plan
     $S_{need}, C \leftarrow \text{SELECT-SUB-GOAL}(\textit{plan})$ 
    CHOOSE-OPERATOR(plan, operators,  $S_{need}$ , c)
     $S_{nonprim} \leftarrow \text{SELECT-NONPRIMITIVE}(\textit{plan})$ 
    CHOOSE-DECOMPOSITION(plan, methods,  $S_{nonprim}$ )
    RESOLVE-THREATS(plan)
  end

```

**Figure 12.2** A hierarchical decomposition partial-order planning algorithm, HD-POP. On each iteration of the loop we first achieve an unachieved condition (CHOOSE-OPERATOR), then decompose a nonprimitive operator (CHOOSE-DECOMPOSITION), then resolve threats.

To make HD-POP work, we have to change SOLUTION? to check that every step of the plan is primitive. The other functions from Figure 11.J.3 remain unchanged. There are two new procedures: SELECT-NONPRIMITIVE arbitrarily selects a nonprimitive step from the plan. The function CHOOSE-DECOMPOSITION picks a decomposition method for the plan and applies it. If *method* is chosen as the decomposition for the step  $S_{nonprim}$ , then the fields of the *plan* are altered as follows:

- STEPS: Add all the steps of *method* to the plan, but remove  $S_{nonprim}$ .
- BINDINGS: Add all the variable binding constraints of *method* to the plan. Fail if this introduces a contradiction.
- ORDERINGS: Following the principle of least commitment, we replace each ordering constraint of the form  $S_a \prec S_{nonprim}$  with constraint(s) that order  $S_a$  before the *latest* step(s) of *method*. That is, if  $S_m$  is a step of *method*, and there is no other  $S_j$  in *method* such that  $S_m \prec S_j$ , then add the constraint  $S_a \prec S_m$ . Similarly, replace each constraint of the form

$S_{nonprim} \prec S_z$  with constraint(s) that order  $S_z$  after the *earliest* step(s) of *method*. We then call RESOLVE-THREATS to add any additional ordering constraints that may be needed.

- **LINKS:** It is easier to match up causal links with the right substeps of *method* than it is to match up ordering constraints. If  $S_i \rightarrow S_z$ ,  $S_{nonprim}$  was a causal link in *plan*, replace it by a set of links  $S_i \rightarrow c$ ,  $S_m$ , where each  $S_m$  is a step of *method* that has  $c$  as a precondition, and there is no earlier step of *method* that has  $c$  as a precondition. (If there are several such steps with  $c$  as a precondition, then put in a causal link for each one. If there are none, then the causal link from  $S_i$  can be dropped, because  $c$  was an unnecessary precondition of  $S_{nonprim}$ .) Similarly, for each link  $S_{nonprim} \rightarrow S_j$  in *plan*, replace it with a set of links  $S_m \rightarrow c$ ,  $S_j$ , where  $S_m$  is a step of *method* that has  $c$  as an effect and there is no later step of *method* with  $c$  as an effect.

In Figure 12.3, we show a more detailed diagram of the decomposition of a plan step, in the context of a larger plan. Notice that one of the causal links that leads into the nonprimitive step *Build House* ends up being attached to the first step of the decomposition, but the other causal link is attached to a later step.

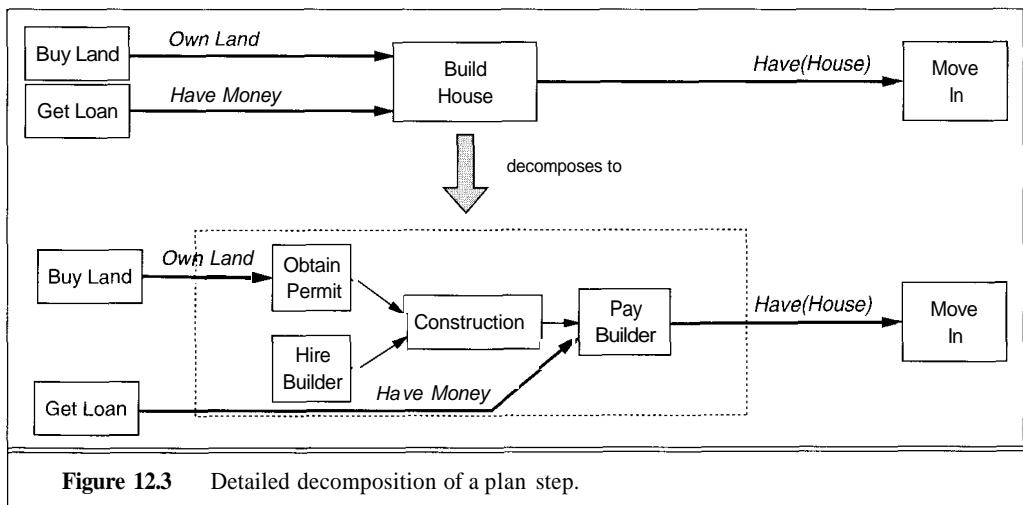


Figure 12.3 Detailed decomposition of a plan step.

## 12.3 ANALYSIS OF HIERARCHICAL DECOMPOSITION

Hierarchical decomposition *seems* like a good idea, on the same grounds that subroutines or macros are a good idea in programming: they allow the programmer (or knowledge engineer) to specify the problem in pieces of a reasonable size. The pieces can be combined hierarchically to create large plans, without incurring the enormous combinatorial cost of constructing large plans from primitive operators. In this section, we will make this intuitive idea more precise.

ABSTRACT  
SOLUTIONDOWNWARD  
SOLUTION

UPWARD SOLUTION

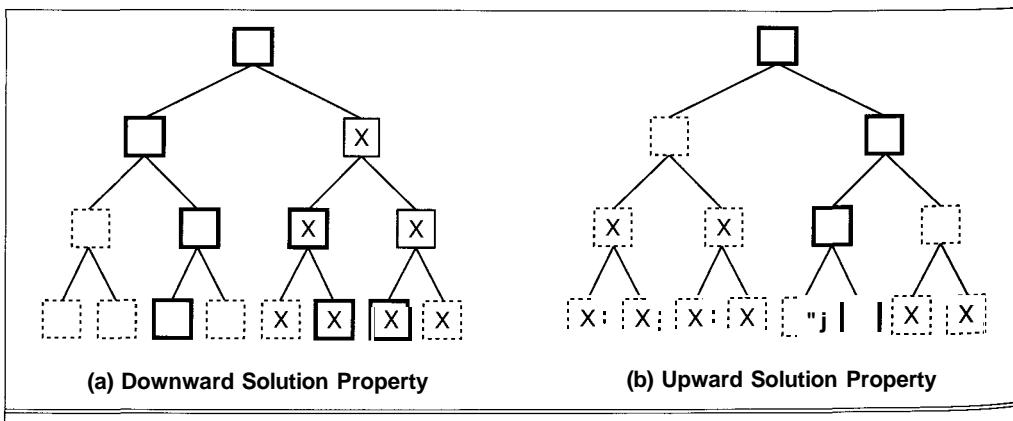
Suppose that there is a way to string four abstract operators together to build a house—for example, the four operators shown in Figure 12.1. We will call this an **abstract solution**—a plan that contains abstract operators, but is consistent and complete. Finding a small abstract solution, if one exists, should not be too expensive. Continuing this process, we should be able to obtain a primitive plan without too much backtracking.

This assumes, however, that in finding an abstract solution, and in rejecting other abstract plans as inconsistent, one is doing useful work. One would *like* the following properties to hold:

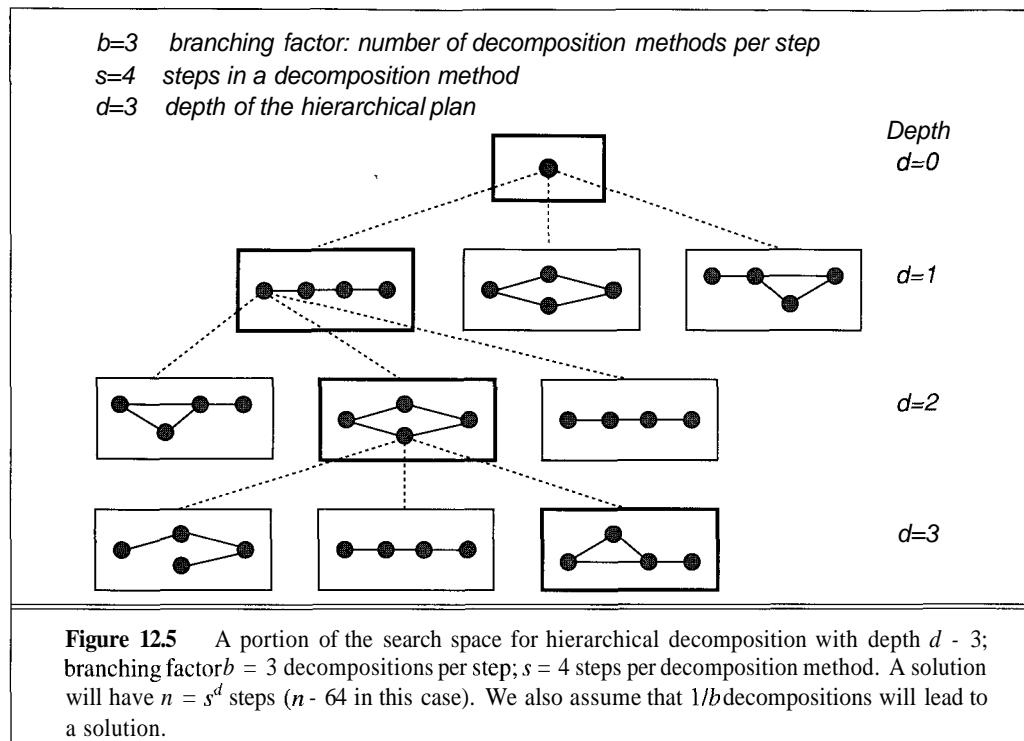
- If  $p$  is an abstract solution, then there is a primitive solution of which  $p$  is an abstraction. If this property holds, then once an abstract solution is found we can prune away all other abstract plans from the search tree. This property is the **downward solution** property.
- If an abstract plan is inconsistent, then there is no primitive solution of which it is an abstraction. If this property holds, then we can prune away all the descendants of any inconsistent abstract plan. This is called the **upward solution** property because it also means that all complete abstractions of primitive solutions are abstract solutions.

Figure 12.4 illustrates these two notions graphically. Each box represents an entire plan (not just a step), and each arc represents a decomposition step in which an abstract operator is expanded. At the top is a very abstract plan, and at the bottom are plans with only primitive steps. The boxes with bold outlines are (possibly abstract) solutions, and the ones with dotted outlines are inconsistent. Plans marked with an "X" need not be examined by the planning algorithm. (The figure shows only complete or inconsistent plans, leaving out consistent but incomplete plans and the achievement steps that are applied to them.)

To get a more quantitative feel for how these properties affect the search, we need a simplified model of the search space. Assume that there is at least one solution with  $n$  primitive steps, and that the time to resolve threats and handle constraints is negligible: all we will be concerned with is the time it takes to choose the right set of steps. Figure 12.5 defines the search space in terms of the parameters  $b$ ,  $s$ , and  $d$ .



**Figure 12.4** The upward and downward solution properties in plan space (abstract plans on top, primitive plans on the bottom). Bold outlined boxes are solutions, dotted outlines are inconsistent, and boxes marked with an "X" can be pruned away in a left-to-right search.



A nonhierarchical planner would have to generate an  $n$ -step plan, choosing among  $b$  possibilities for each one. (We are also assuming that the number of decompositions per nonprimitive step,  $b$ , is the same as the number of applicable new operators for an open precondition of a primitive step.) Thus, it takes time  $O(b^n)$  in the worst case. *With a hierarchical planner, we can adopt the strategy of only searching decompositions that lead to abstract solutions.* (In our simplified model, exactly 1 of every  $b$  decompositions is a solution. In more realistic models, we need to consider what to do when there are zero or more than one solution.) The planner has to look at  $sb$  steps at depth  $d = 1$ . At depth  $d = 2$ , it looks at another  $sb$  steps for each step it decomposes, but it only has to decompose  $1/b$  of them, for a total of  $bs^2$ . Thus, the total number of plans considered is

$$\sum_{i=1}^d bs^i = O(bs^d)$$

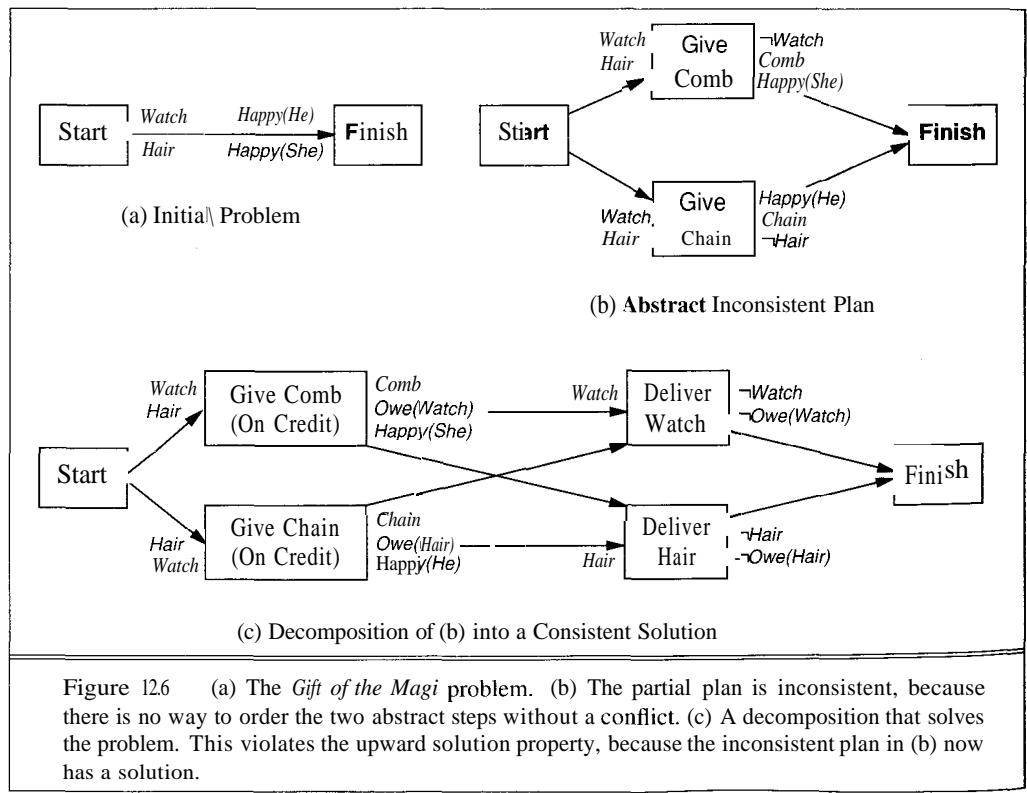
To give you an idea of the difference, for the parameter values in Figure 12.5, a nonhierarchical planner has to inspect  $3 \times 10^{30}$  plans, whereas a hierarchical planner looks at only 576.

The upward and downward solution properties seem to be enormously powerful. At first glance, it may seem that they are necessary consequences of the correctness conditions for decompositions (page 373). In fact, neither property is guaranteed to hold. Without these properties, or some reasonable substitute, a hierarchical planner does no better than a nonhierarchical planner in the worst case (although it may do better in the average case).

Figure 12.6 shows an example where the upward solution property does not hold. That is, the abstract solution is inconsistent, but there is a decomposition that solves the problem. The problem is taken from the O. Henry story *The Gift of the Magi*. A poor couple has only two prized possessions; he a gold watch and she her beautiful long hair. They each plan to buy presents to make the other happy. He decides to trade his watch to buy a fancy comb for her hair, and she decides to sell her hair to get a gold chain for his watch. As Figure 12.6(b) shows, the resulting abstract plan is inconsistent. However, it is still possible to decompose this inconsistent plan into a consistent solution, if the right decomposition methods are available. In Figure 12.6(c) we decompose the "Give Comb" step with an "installment plan" method. In the first step of the decomposition, the husband takes possession of the comb, and gives it to his wife, while agreeing to deliver the watch in payment at a later date. In the second step, the watch is handed over and the obligation is fulfilled. A similar method decomposes the "Give Chain" step. As long as both giving steps are ordered before the delivery steps, this decomposition solves the problem.

UNIQUE MAIN SUBACTION

One way to guarantee the upward solution property is to make sure that each decomposition method satisfies the **unique main subaction** condition: that there is one step of the decomposed plan to which all preconditions and effects of the abstract operator are attached. In the Magi example, the unique main subaction condition does not hold. It does not hold in Figure 12.3 either, although it would be if *Own Land* were a precondition of the Pay Builder step. Sometimes



it is worthwhile to do some preprocessing of the decomposition methods to put them in a form that satisfies the unique main subaction condition so that we can freely cut off search when we hit an inconsistent abstract plan without fear of missing any solutions.

It is important to remember that even when the upward solution property fails to hold, it is still a reasonable heuristic to prefer applications of decomposition to consistent plans rather than inconsistent ones. Similarly, even when the downward solution property is violated, it makes more sense to pursue the refinement of abstract solutions than that of inconsistent plans, even though the abstract solution may not lead to a real solution. (Exercise 12.4 asks you to find an example of the violation of the downward solution property.)

## Decomposition and sharing

In CHOOSE-DECOMPOSITION, we just merge each step of the decomposition into the existing plan. This is appropriate for a divide-and-conquer approach—we solve each subproblem separately, and then combine it into the rest of the solution. But often the only solution to a problem involves combining the two solutions by *sharing* steps rather than by joining distinct sets of steps. For example, consider the problem "enjoy a honeymoon and raise a baby." A planner might choose the decomposition "get married and go on honeymoon" for the first subproblem and "get married and have a baby" for the second, but the planner could get into a lot of trouble if the two "get married" steps are different. Indeed, if a precondition to "get married" is "not married," and divorce is not an option, then there is no way that a planner can merge the two subplans without sharing steps. Hence a sharing mechanism is required for a hierarchical planner to be complete.

Sharing can be implemented by adding a choice point in CHOOSE-DECOMPOSITION for every operator in a decomposition: either a new step is created to instantiate the operator or an existing step is used. This is exactly analogous to the existing choice point in CHOOSE-OPERATOR. Although this introduces a lot of additional choice points, many of them will have only a single alternative if no operators are available for sharing. Furthermore, it is a reasonable heuristic to prefer sharing to non-sharing.

Many hierarchical planners use a different mechanism to handle this problem: they merge decompositions without sharing but allow **critics** to modify the resulting plan. A critic is a function that takes a plan as input and returns a modified plan with some conflict or other anomaly corrected. Theoretically, using critics is no more or less powerful than putting in all the choice points, but it can be easier to manage the search space with a well-chosen set of critics.

Note that the choice of sharing versus merging steps has an effect on the efficiency of planning, as well as on completeness. An interesting example of the costs and benefits of sharing occurs in optimizing compilers. Consider the problem of compiling  $\sin(x) + \cos(x)$  for a sequential computer. Most compilers accomplish this by merging two separate subroutine calls in a trivial way: all the steps of sin come before any of the steps of cos (or vice versa). If we allowed sharing instead of merging, we could actually get a more efficient solution, because the two computations have many steps in common. Most compilers do not do this because it would take too much time to consider all the possible shared plans. Instead, most compilers take the critic approach: a peephole optimizer is just a kind of critic.

## Decomposition versus approximation

The literature on planning is confusing because authors have not agreed on their terminology. There are two completely separate ideas that have gone under the name **hierarchical planning**.

**Hierarchical decomposition**, as we have seen, is the idea that an abstract, nonprimitive operator can be decomposed into a more complex network of steps. The abstraction here is on the granularity with which operators interact with the world. Second, **abstraction hierarchies** capture the idea that a single operator can be planned with at different levels of abstraction. At the primitive level, the operator has a full set of preconditions and effects; at higher levels, the planner ignores some of these details.

To avoid confusion, we will use the term **approximation hierarchy** for this kind of abstraction, because the "abstract" version of an operator is slightly incorrect, rather than merely abstract. An approximation hierarchy planner takes an operator and partitions its preconditions according to their **criticality level**, for example:

```

Op(ACTION:Buy(x),
    EFFECT:Have(x) A  $\neg$ Have(Money),
    PRECOND:1:Sells(store,x)A
        2:At(store)A
        3: Have(Money))

```

Conditions labelled with lower numbers are considered more critical. In the case of buying something, there is not much one can do if the store does not sell it, so it is vital to choose the right store. An approximation hierarchy planner first solves the problem using only the preconditions of criticality 1. The solution would be a plan that buys the right things at the right stores, but does not worry about how to travel between stores or how to pay for the goods. The idea is that it will be easy to find an abstract solution like this, because most of the pesky details are ignored. Once a solution is found, it can be expanded by considering the preconditions at criticality level 2. Then we get a plan that takes travel into consideration, but still does not worry about paying. We keep expanding the solution in this way until all the preconditions are satisfied.

In the framework we have presented, we do not really need to change the language or the planner to support approximation hierarchy planning. All we have to do is provide the right decomposition methods and abstract operators. First, we define *Buy*<sub>1</sub>, which has just the precondition at criticality level 1, and has a single decomposition method to *Buy*<sub>2</sub>, which has preconditions at criticality 1 and 2, and so on. Clearly, any domain generated this way has the unique main subaction property—the decomposition has only one step, and it has all the preconditions and effects of the abstract operator. Therefore, the upward solution property holds.

If we add a heuristic that causes the planning algorithm to backtrack to choice points involving low-number preconditions first, then we get an approximation hierarchy planner using our standard hierarchical decomposition planner. Thus, the criticality levels used in approximation hierarchy planning can be seen as providing control information to guide the planning search. In this sense, criticality levels are a rather crude tool because they do not allow the criticality to depend on context. (For example, money might be a critical precondition if one is buying a house.) One promising approach is to replace criticality levels with descriptions of how likely the operator is to succeed in various combinations of circumstances.

APPROXIMATION  
HIERARCHY

CRITICALITY LEVEL

## 12.4 MORE EXPRESSIVE OPERATOR DESCRIPTIONS

Hierarchical planning addresses the problem of efficiency, but we still need to make our representation language more expressive in order to broaden its applicability. The extensions include allowing for the effects of an action to depend on the circumstances in which it is executed; allowing disjunctive and negated preconditions; and allowing universally quantified preconditions and effects. We conclude the section with a planning algorithm, POP-DUNC (Partial-Order Planner with Disjunction, Universal quantification, Negation and Conditional effects).

### Conditional effects

CONDITIONAL EFFECTS

Operators with **conditional effects** have different effects depending on what the world is like when they are executed. We will return to the blocks world of Section 11.8 for some of our examples. Conceptually, the simple blocks world has only one real action—moving a block from one place to another—but we were forced to introduce two operators in order to maintain the *Clear* predicate properly:

$$\begin{aligned} &Op(\text{ACTION:Move}(b, x, v), \\ &\quad \text{PRECOND: } On(b, x) \wedge Clear(b) \wedge Clear(y), \\ &\quad \text{EFFECT: } On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y)) \end{aligned}$$

$$\begin{aligned} &Op(\text{ACTION:MoveToTable}(b, x), \\ &\quad \text{PRECOND: } On(b, x) \wedge Clear(b), \\ &\quad \text{EFFECT: } On(b, \text{Table}) \wedge Clear(x) \wedge \neg On(b, x)) \end{aligned}$$

Suppose that the initial situation includes *On(A, B)* and we have the goal *Clear(B)*. We can achieve the goal by moving A off B, but unfortunately we are forced to choose whether we want to move A to the table or to somewhere else. This introduces a premature commitment and can lead to inefficiency.

We can eliminate the premature commitment by extending the operator language to include conditional effects. In this case, we can define a single operator *Move(b, x, y)* with a conditional effect that says, "if y is not the table then an effect is  $\neg Clear(y)$ ." We will use the syntax "*effect when condition*" to denote this, where *effect* and *condition* are both literals or conjunctions of literals. We place this syntax in the EFFECT slot of an operator, but it is really a combination of a precondition and an effect: the *effect* part refers to the situation that is the result of the operator, and the *condition* part refers to the situation before the operator is applied. Thus, "*Q when P*" is not the same as the logical statement  $P \Rightarrow Q$ ; rather it is equivalent to the situation calculus statement  $P(s) \Rightarrow Q(\text{Result}(act, s))$ . The conditional *Move* operator is written as follows:

$$\begin{aligned} &Op(\text{ACTION:Move}(b, x, y), \\ &\quad \text{PRECOND: } On(b, x) \wedge Clear(b) \wedge Clear(y), \\ &\quad \text{EFFECT: } On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \\ &\qquad \quad \wedge \neg Clear(y) \text{ when } y \neq \text{Table}) \end{aligned}$$

Now we have to incorporate this new syntax into the planner. Two changes are required. First, in SELECT-SUB-GOAL, we have to decide if a precondition  $c$  in a conditional effect of the form  $e \text{ when } c$  should be considered as a candidate for selection. The answer is that if the effect  $e$  supplies a condition that is protected by a causal link, then we should consider selecting  $c$ , but not until the causal link is there. This is because the causal link means that the plan will not work unless  $c$  is also true. Considering the operator just shown, the planner would usually have no need to establish  $y \neq \text{Table}$  because  $\neg \text{Clear}(y)$  is not usually needed as a precondition of some other step in the plan. Thus, the planner can usually use the table if it needs somewhere to put a block that does not need to be anywhere special.

## CONFRONTATION

Second, we have another possibility for RESOLVE-THREAT. Any step that has the effect  $(\neg c' \text{ when } p)$  is a possible threat to the causal link  $S_i \xrightarrow{c} S_j$  whenever  $c$  and  $c'$  unify. We can resolve the threat by making sure that  $p$  does not hold. We call this technique **confrontation**. In the blocks world, if we need a given block to be clear in order to carry out some step, then it is possible for the  $\text{Move}(b, x, y)$  operator to threaten this condition if  $y$  is uninstantiated. However, the threat only occurs if  $y \neq \text{Table}$ ; confrontation removes the threat by setting  $y = \text{Table}$ . A version of RESOLVE-THREATS that incorporates confrontation appears in Figure 12.7.

```

procedure RESOLVE-THREATS(plan)
  for each  $S_i \xrightarrow{c} S_j$  in LINKS(plan) do
    for each  $S_{\text{threat}}$  in STEPS(plan) do
      for each  $c'$  in EFFECT( $S_{\text{threat}}$ ) do
        if SUBST(BINDINGS(plan),  $c$ ) = SUBST(BINDINGS(plan),  $\neg c'$ ) then
          choose either
            Promotion: Add  $S_{\text{threat}} \prec S_i$  to ORDERINGS(plan)
            Demotion: Add  $S_j \prec S_{\text{threat}}$  to ORDERINGS(plan)
            Confrontation: if  $c'$  is really of the form  $(c' \text{ when } p)$  then
              CHOOSE-OPERATOR(plan, operators,  $S_{\text{threat}}$ ,  $\neg p$ )
              RESOLVE-THREATS(plan)
            if not CONSISTENT(plan) then fail
          end
        end
      end

```

**Figure 12.7** A version of RESOLVE-THREATS with the confrontation technique for resolving conditional effects.

## Negated and disjunctive goals

The confrontation technique calls CHOOSE-OPERATOR with the goal  $\neg p$ . This is something new: so far we have insisted that all goals (preconditions) be positive literals. Dealing with negated literals as goals does not add much complexity: we still just have to check for effects that match

DISJUNCTIVE  
PRECONDITIONSDISJUNCTIVE  
EFFECTSUNIVERSALLY  
QUANTIFIED  
PRECONDITIONSUNIVERSALLY  
QUANTIFIED  
EFFECTS

the goal. We do have to make sure that our unification function allows  $p$  to match  $\neg\neg p$ . We also have to treat the initial state specially: we do not want to specify all the conditions that are false in the initial state, so we say that a goal of the form  $\neg p$  can be matched either by an explicit effect that unifies with  $\neg p$  or by the initial state, if it does not contain  $p$ .

While we are at it, it is easy to add **disjunctive preconditions**. In SELECT-SUB-GOAL, if we choose a step with a precondition of the form  $p \vee q$ , then we nondeterministically choose to return either  $p$  or  $q$  and reserve the other one as a backtrack point. Of course, any operator with the precondition  $p \vee q$  could be replaced with two operators, one with  $p$  as a precondition and one with  $q$ , but then the planner would have to commit to one or the other. Keeping the conditions in a single operator allows us to delay making the commitment.

Whereas disjunctive preconditions are easy to handle, **disjunctive effects** are very difficult to incorporate. They change the environment from deterministic to nondeterministic. A disjunctive effect is used to model random effects, or effects that are not determined by the preconditions of the operator. For example, the operator  $Flip(coin)$  would have the disjunctive effect  $Heads(coin) \vee Tails(coin)$ . In some cases, a single plan can guarantee goal achievement even in the face of disjunctive effects. For example, an operator such as  $TurnHeadsSideUp(coin)$  will coerce the world into a known state even after a flip. In most cases, however, the agent needs to develop a different plan to handle each possible outcome. We develop algorithms for this kind of planning in Chapter 13.

## Universal quantification

In defining the blocks world, we had to introduce the condition  $Clear(b)$ . In this section, we extend the language to allow **universally quantified preconditions**. Instead of writing  $Clear(b)$  as a precondition, we can use  $\forall x \text{ Block}(x) \Rightarrow \neg\text{On}(x, b)$  instead. Not only does this relieve us of the burden of making each operator maintain the  $Clear$  predicate, but it also allows us to handle more complex domains, such as a blocks world with different size blocks.

We also allow for **universally quantified effects**. For example, in the shopping domain we could define the operator  $Carry(bag, x, y)$  so that it has the effect that all objects that are in the bag are at  $y$  and are no longer at  $x$ . There is no way to define this operator without universal quantification. Here is the syntax we will use:

$$\begin{aligned} &Op(\text{ACTION}: \text{Carry}(bag, x, y), \\ &\quad \text{PRECOND}: \text{Bag}(bag) \text{ A } \text{At}(bag, x), \\ &\quad \text{EFFECT}: \text{At}(bag, y), \neg\text{At}(bag, x) \text{ A} \\ &\quad \quad \forall i \text{ Item}(i) \Rightarrow (\text{At}(i, y) \text{ A } \neg\text{At}(i, x)) \text{ when } \text{In}(i, bag)) \end{aligned}$$

Although this looks like full first-order logic with quantifiers and implications, it is not. The syntax—and the corresponding semantics—is strictly limited. We will only allow worlds with a finite, static, typed universe of objects, so that the universally quantified condition can be satisfied by enumeration. The description of the initial state must mention all the objects and give each one a **type**, specified as a unary predicate. For example,  $\text{Bag}(B) \text{ A } \text{Item}(I_1) \text{ A } \text{Item}(I_2) \text{ A } \text{Item}(B)$ . Notice that is possible for an object to have more than one type:  $B$  is both a bag and an item. The **static universe** requirement means that the objects mentioned in the initial state cannot change type or be destroyed, and no new objects can be created. That is, no operator except  $Start$  can

have  $Bag(x)$  or  $\neg Bag(x)$  as an effect. With this semantics of objects in mind, we can extend the syntax of preconditions and effects by allowing the form

$$\forall x \ T(x) \Rightarrow C(x)$$

where  $T$  is a unary type predicate on  $x$ , and  $C$  is a condition involving  $x$ . The finite, static, typed universe means that we can always expand this form into an equivalent conjunctive expression with no quantifiers:

$$MX \ T(x) \Rightarrow C(x) = C(x_1) \wedge \dots \wedge C(x_n)$$

where  $x_1, \dots, x_n$  are the objects in the initial state that satisfy  $T(x)$ . Here is an example:

Initial State:  $Bag(B) \wedge Milk(M_1) \wedge Milk(M_2) \wedge Milk(M_3)$

Expression:  $\forall x Milk(x) \Rightarrow In(x, B)$

Expansion:  $In(M_1, B) \wedge In(M_2, B) \wedge In(M_3, B)$

In our planner, we will expand universally quantified goals to eliminate the quantifier. This can be inefficient because it can lead to large conjunctions that need to be satisfied, but there is no general solution that does any better.

For universally quantified effects, we are better off. We do not need to expand out the effect because we may not care about many of the resulting conjuncts. Instead we leave the universally quantified effect as it is, but make sure that RESOLVE-THREATS can notice that a universally quantified effect is a threat and that CHOOSE-OPERATOR can notice when a universally quantified effect can be used as a causal link.

Some domains are dynamic in that objects are created or destroyed, or change their type over time. Plans in which objects are first made, then used, seem quite natural. Our insistence on a static set of objects might seem to make it impossible to handle such domains. In fact, we can often finesse the problem by specifying broad static types in the universally quantified expressions and using additional dynamic unary predicates to make finer discriminations. In particular, we can use the dynamic predicate to distinguish between potential and actual objects. We begin with a supply of *non-Actual* objects, and object creation is handled by operators that make objects *Actual*. Suppose that in the house-building domain there are two possible sites for the house. In the initial state we could include  $House(H_1) \wedge House(H_2)$ , where we take  $House$  to mean that its argument is a possible house: something that might exist in some points of some plans, but not in others. In the initial state neither  $Actual(H_1)$  nor  $Actual(H_2)$  holds, but that can change: the  $Build(x)$  operator has  $Actual(x)$  as an effect, and the  $Demolish(x)$  operator has  $\neg Actual(x)$  as an effect.

If there were an infinite number of possible houses (or just a million), then this approach would not work. But objects that can potentially exist in large, undifferentiated quantities can often be treated as resources, which are covered in the next section.

## A planner for expressive operator descriptions

We now combine all the extensions to create our POP-DUNC algorithm. Because the top level of POP-DUNC is identical to that of POP (Figure 11.13), we will not repeat it here. Figure 12.8 shows the parts of POP-DUNC that differ from the original. SELECT-SUB-GOAL is modified to expand out universally quantified preconditions and to choose one of two possible ways to satisfy

**function** SELECT-SUB-GOAL(*plan*) *returns plan, precondition conjunct*

pick a plan step  $S_{need}$  from STEPS(*plan*) with a precondition conjunct  $c$  that has not been achieved

if  $c$  is a universally quantified expression **then**  
**return**  $S_{need}$ , EXPANSION( $c$ )  
**else if**  $c$  is a disjunction  $c_1 \vee c_2$  **then**  
**return**  $S_{need}$ , choose( $c_1, c_2$ )  
**else return**  $S_{need}, c$

**procedure** CHOOSE-OPERATOR(*plan, operators, S<sub>need</sub>, c*)

**choose** a step  $S_{add}$  from *operators* or STEPS(*plan*) that has  $c_{add}$  as an effect  
such that  $u = \text{UNIFY}(c, c_{add}, \text{BINDINGS}(\text{plan}))$

if there is no such step **then fail**

$u' \leftarrow u$  without the universally quantified variables of  $c_{add}$

add  $u'$  to BINDINGS(*plan*)

add  $S_{add} \xrightarrow{c} S_{need}$  to LINKS(*plan*)

add  $S_{add} \prec S_{need}$  to ORDERINGS(*plan*)

if  $S_{add}$  is a newly added step from *operators* **then**

add  $S_{add}$  to STEPS(*plan*)

add Start  $\prec S_{add} \prec$  Finish to ORDERINGS(*plan*)

**procedure** RESOLVE-THREATS(*plan*)

**for each**  $S_i \xrightarrow{c} S_j$  in LINKS(*plan*) **do**  
**for each**  $S_{threat}$  in STEPS(*plan*) **do**  
**for each**  $c'$  in EFFECT( $S_{threat}$ ) **do**  
**if** SUBST(BINDINGS(*plan*),  $c$ ) = SUBST(BINDINGS(*plan*),  $\neg c'$ ) **then**  
**choose either**  
*Promotion:* Add  $S_{threat} \prec S_i$  to ORDERINGS(*plan*)  
*Demotion:* Add  $S_j \prec S_{threat}$  to ORDERINGS(*plan*)  
*Confrontation:* if  $c'$  is really of the form ( $c'$  when  $p$ ) **then**  
CHOOSE-OPERATOR(*plan, operators, S<sub>threat</sub>,  $\neg p$* )  
RESOLVE-THREATS(*plan*)  
**if not** CONSISTENT(*plan*) **then fail**  
**end**  
**end**  
**end**

**Figure 12.8** The relevant components of POP-DUNC.

a disjunctive precondition. CHOOSE-OPERATOR is modified only slightly, to handle universally quantified variables properly. RESOLVE-THREATS is modified to include confrontation as one method of resolving a threat from a conditional effect.

## 12.5 RESOURCE CONSTRAINTS

In Chapter 11, we tackled the problem of shopping, but ignored money. In this section, we consider how to handle money and other resources properly. To do this, we need a language in which we can express a precondition such as *Have*(\$1.89), and we need a planning algorithm that will handle this efficiently. The former is theoretically possible with what we already have. We can represent each coin and bill in the world in the initial state: *Dollar*( $d_1$ ) A *Dollar*( $d_2$ ) A *Quarter*( $q_1$ ) A .... We can then add decomposition methods to enumerate the ways it is possible to have a dollar. We have to be careful about inequality constraints, because we would not want the goal *Have*(\$2.00) to be satisfied by *Have*( $d_1$ ) A *Have*( $d_1$ ). The final representation would be a little unintuitive and extremely verbose, but we could do it.

The problem is that the representation is totally unsuited for planning. Let us say we pose the problem *Have*(\$1,000,000), and the best the planner can come up with is a plan that generates \$1000. The planner would then backtrack looking for another plan. For every step that achieved, say, *Have*( $d_1$ ), the planner would have to consider *Have*( $d_2$ ) instead. The planner would end up generating all combinations of all the coins and bills in the world that total \$1000. Clearly, this is a waste of search time, and it fails to capture the idea that it is the quantity of money you have that is important, not the identity of the coins and bills.

### Using measures in planning

The solution is to introduce numeric-valued **measures** (see Chapter 8). Recall that a measure is an *amount* of something, such as money or volume. Measures can be referred to by logical terms such as \$(1.50) or *Gallons*(6) or *GasLevel*. Measure functions such as *Volume* apply to objects such as *GasInCar* to yield measures: *GasLevel* = *Volume*(*GasInCar*) = *Gallons*(6). In planning problems, we are usually interested in amounts that change over time. A situation calculus representation would therefore include a situation argument (e.g., *GasLevel*( $s$ )), but as usual in planning we will leave the situation implicit. We will call expressions such as *GasLevel* **measure fluents**.

Planners that use measures typically require them to be "declared" up front with associated range information. For example, in a shopping problem, we might want to state that the amount of money the agent has, *Cash*, must be nonnegative; that the amount of gas in the tank, *GasLevel*, can range up to 15 gallons; that the price of gas ranges from \$1.00 to \$1.50 per gallon; and that the price of milk ranges from \$1.00 to \$1.50 per quart:

$$\begin{aligned} \$0 < \text{Cash} \\ \text{Gallons}(0) < \text{GasLevel} < \text{Gallons}(15) \\ \$1.00 < \text{UnitPrice(Gas)} \times \text{Gallons}(1) < \$1.50 \\ \$1.00 < \text{UnitPrice(Milk)} \times \text{Quarts}(1) < \$1.50 \end{aligned}$$

Measures such as the price of gas are realities with which the planner must deal, but over which it has little control. Other measures, such as *Cash* and *GasLevel*, are treated as **resources** that can be produced and consumed. That is, there are operators such as *Drive* that require and consume

the *GasLevel* resource, and there are operators such as *FillUp* that produce more of the *GasLevel* resource (while consuming some of the *Cash* resource).

To represent this, we allow inequality tests involving measures in the precondition of an operator. In the effect of an operator, we allow numeric assignment statements, where the left-hand side is a measure fluent and the right-hand side is an arithmetic expression involving measures. This is like an assignment and not like an equality in that the right-hand side refers to values *before* the action and the left-hand side refers to the value of the measure fluent *after* the action. As usual, the initial state is described by the effects of the start action:

$$\begin{aligned} &Op(\text{ACTION:Start}, \\ &\quad \text{EFFECT: } Cash \leftarrow \$12.50 \text{A} \\ &\quad \quad GasLevel \leftarrow Gallons(5) \text{A} \\ &\quad \vdots \end{aligned}$$

The *Buy* action reduces the amount of *Cash* one has:

$$\begin{aligned} &Op(\text{ACTION:Buy}(x, store), \\ &\quad \text{EFFECT: } Have(x) \text{A} \quad Cash \leftarrow Cash - Price(x, store)) \end{aligned}$$

Getting gas can be described by an abstract *Fillup* operator:

$$\begin{aligned} &Op(\text{ACTION:Fillup}(GasLevel), \\ &\quad \text{EFFECT: } GasLevel \leftarrow Gallons(15) \text{A} \\ &\quad \quad Cash \leftarrow Cash - (UnitPrice(Gas) \times (Gallons(15) - GasLevel))) \end{aligned}$$

The declared upper and lower bounds serve as implicit preconditions for each operator. For example, *Buy*(*x*) has the implicit precondition *Cash* > *Price*(*x*) to ensure that the quantity will be within range after the action. It takes some reasonably sophisticated algebraic manipulation code to automatically generate these preconditions, but it is less error-prone to do that than to require the user to explicitly write down the preconditions.

These examples should give you an idea of the versatility and power of using measures to reason about consumable resources. Although practical planners such as SIPE, O-PLAN, and DEVISER all have mechanisms for resource allocation, the theory behind them has not been formulated in a clean way, and there is disagreement about just what should be represented and how it should be reasoned with. We will sketch an outline of one approach.

It is a good idea to plan for scarce resources first. This can be done using an abstraction hierarchy of preconditions, as in Section 12.3, or by a special resource mechanism. Either way, it is desirable to delay the choice of a causal link for the resource measures. That is, when planning to buy a quart of milk, it is a good idea to check if *Cash* > (*Quarts*(1) x *UnitPrice*(*Milk*, *store*)), but it would be premature to establish a causal link to that precondition from either the start state or some other step. The idea is to first pick the steps of the plan and do a rough check to see if the resource requirements are satisfiable. If they are, then the planner can continue with the normal mechanism of resolving threats for all the preconditions.

An easy way of doing the rough check on resources is to keep track of the minimum and maximum possible values of each quantity at each step in the plan. For example, if in the initial state we have *Cash* ← \$(12.50) and in the description of measures we have \$(0.50) < *UnitPrice*(*Bananas*, *store*) x *Pounds*(1) < \$(1.00) and \$(1.00) < *UnitPrice*(*Milk*, *store*) x *Quarts*(1) < \$(1.50), then a plan with the steps *Buy*(*Milk*) and *Buy*(*Bananas*) will have the

range  $\$(10.00) < \text{Cash} < \$(11.00)$  at the finish. If we started out with less than  $\$(1.50)$ , then this approach would lead to failure quickly, without having to try all permutations of *Buy* steps at all combinations of stores.

There is a trade-off in deciding how much of the resource quantity information we want to deal with in the rough check. We could implement a full constraint satisfaction problem solver for arbitrary arithmetic inequalities (making sure we tie this process in with the normal unification of variables). However, with complicated domains, we can end up spending just as long solving the constraint satisfaction problem as we would have spent resolving all the conflicts.

## Temporal constraints

In most ways, time can be treated like any other resource. The initial state specifies a start time for the plan, for example, *Time* — 8:30. (Here 8:30 is a shorthand for *Minutes*( $8 \times 60 + 30$ ).) We then can say how much time each operation consumes. (In the case of time, of course, consumption means *adding* to the amount.) Suppose it takes 10 seconds to pump each gallon of gas, and 3 minutes to do the rest of the *Fillup* action. Then an effect of *Fillup* is

$$\text{Time} \leftarrow \text{Time} + \text{Minutes}(3) + (\text{Seconds}(10)/\text{Gallons}(1)) \times (\text{Gallons}(15) - \text{GasLevel})$$

It is handy to provide the operator *Wait*(*x*), which has the effect  $\text{Time} \leftarrow \text{Time} + x$  and no other preconditions or effects (at least in static domains).

There are two ways in which time differs from other resources. First, actions that are executed in parallel consume the maximum of their respective times rather than the sum. Second, constraints on the time resource have to be consistent with ordering constraints. That is, if  $S_i \prec S_j$  is one of the ordering constraints, then *Time* at  $S_i$  must be less than *Time* at  $S_j$ .

Another important constraint is that time never goes backward. This implies that no operators generate time instead of consuming it. Thus, if the goal state specifies a deadline (a maximum time), and you have a partial plan whose steps require more time than is allowed, you can backtrack immediately, without considering any completions of the plan. (See Exercise 12.9 for more on this.)

## 12.6 SUMMARY

---

In this chapter we have seen several ways to extend the planning language—the representation of states and operators—to allow the planner to handle more complex, realistic domains. Each extension requires a corresponding change to the planning algorithm, and there is often a difficult trade-off between expressiveness and worst-case efficiency. However, when the more expressive representations are used wisely, they can lead to an increase in efficiency.

We have tried to present the field of planning in a way that emphasizes the best aspects of progress in AI. The field started with a vague set of requirements (to control a robot) and after some experimenting with an initially promising but ultimately intractable approach (situation calculus and theorem proving) settled down to a well-understood paradigm (STRIPS-style planning). From

there, progress was made by a series of implementations, and formalizations of ever more ambitious variations on the basic planning language.

- The STRIPS language is too restricted for complex, realistic domains, but can be extended in several ways.
- Planners based on extended STRIPS-like languages and partial-order least-commitment algorithms have proven capable of handling complex domains such as spacecraft missions and manufacturing.
- **Hierarchical decomposition** allows nonprimitive operators to be included in plans, with a known decomposition into more primitive steps.
- Hierarchical decomposition is most effective when it serves to prune the search space. Pruning is guaranteed when either the **downward solution** property (every abstract solution can be decomposed into a primitive solution) or **upward solution** property (inconsistent abstract plans have no primitive solutions) holds.
- We can make the planning language closer to situation calculus by allowing **conditional effects** (the effect of an operator depends on what is true when it is executed) and **universal quantification** (the precondition and effect can refer to all objects of a certain class).
- Many actions consume **resources**, such as money, gas, or raw materials. It is convenient to treat these as numeric measures in a pool rather than try to reason about, say, each individual coin and bill in the world. Actions can generate and consume resources, and it is usually cheap and effective to check partial plans for satisfaction of resource constraints before attempting further refinements.
- Time is one of the most important resources. With a few exceptions, time can be handled with the general mechanisms for manipulating resource measures.

---

## BIBLIOGRAPHICAL AND HISTORICAL NOTES

**MACROPS**

Abstract and hierarchical planning was introduced in the ABSTRIPS system (Sacerdoti, 1974), a variant of STRIPS. (Actually, the facility in STRIPS itself for learning **macrops**—"macro-operators" consisting of a sequence of bottom-level steps—could be considered the first mechanism for hierarchical planning.) Hierarchy was also used in the LAWALY system (Siklossy and Dreussi, 1973). Wilkins (1986) discusses some ambiguities in the meaning of the term "hierarchical planning." Yang (1990) explains the "unique main subaction" property in the context of abstraction planning. Erol, Hendler, and Nau (1994) present a complete hierarchical decomposition planner to which our HD-POP owes a great deal.

Work on deciding what hierarchical plans are worth knowing about and how to adapt previously constructed plans to novel situations goes under the name adaptive planning (Alterman, 1988) or case-based planning (Hammond, 1989).

Continuous time was first dealt with by DEVISER (Vere, 1983). A more recent planner focusing on the treatment of continuous time constraints is FORBIN (Dean *et al.*, 1990). NONLIN+ (Tate and Whiter, 1984) and SIPE (Wilkins, 1988; Wilkins, 1990) could reason

about the allocation of limited resources to various plan steps. MOLGEN (Stefik, 1981b; Stefik, 1981a) allowed reasoning about objects that are only partially specified (via constraints). GARI (Descotte and Latombe, 1985) allowed hierarchical reasoning with constraints, in that some lower-level constraints are given lower priorities and may be violated at some stages during plan construction. O-PLAN (Bell and Tate, 1985) had a uniform, general representation for constraints on time and resources.

Classical situation calculus had allowed the full predicate calculus for describing the preconditions and effects of plan steps. Later formalisms have tried to get much of this expressiveness back, without sacrificing the efficiency gained by the use of STRIPS operators. The ADL planning formalism (Pednault, 1986) allows for multiagent planning and avoids the problems with the STRIPS formalism that were pointed out by Lifschitz (1986). PEDESTAL (McDermott, 1991) was the first (partial) implementation of ADL. UCPOP (Penberthy and Weld, 1992; Barrett *et al.*, 1993) is a more complete implementation of ADL that also allows for partial-order planning. POP-DUNC is based largely on this work. Weld (1994) describes UCPOP and gives a general introduction to partial-order planning with actions having conditional effects. Kambham-pati and Nau (1993) analyze the applicability of Chapman's (1987) NP-hardness results to actions with conditional effects. Wolfgang Bibel (1986) has attempted to revive the use of full predicate calculus for planning, counting on advances in theorem proving to avoid the intractability that this led to in the early days of planning.

The improvements in the richness of the representations used in modern planners have helped make them more nearly equal to the challenges of real-world planning tasks than was STRIPS. Several systems have been used to plan for scientific experimentation and observation. MOLGEN was used in the design of scientific experiments in molecular genetics. T-SCHED (Drabble, 1990) was used to schedule mission command sequences for the UOSAT-II satellite. PLAN-ERS1 (Fuchs *et al.*, 1990), based on O-PLAN, was used for observation planning at the European Space Agency; SPIKE (Johnston and Adorf, 1992) was used for observation planning at NASA for the Hubble space telescope. Manufacturing has been another fertile application area for advanced planning systems. OPTIMUM-AIV (Aarup *et al.*, 1994), based on O-PLAN, has been used in spacecraft assembly at the European Space Agency. ISIS (Fox *et al.*, 1981; Fox, 1990) has been used for job shop scheduling at Westinghouse. GARI planned the machining and construction of mechanical parts. FORBIN was used for factory control. NONLIN+ was used for naval logistics planning. SIPE has had a number of applications, including planning for aircraft carrier flight deck operations.

---

## EXERCISES

- 12.1** Give decompositions for the *Hire Builder* and *Obtain Permit* steps in Figure 12.1, and show how the decomposed subplans connect into the overall plan.
- 12.2 Rework the previous exercise using an approximation hierarchy. That is, assign criticality levels to each precondition of each step. How did you decide which preconditions get higher criticality levels?

**12.3** Give an example in the house-building domain of two abstract subplans that cannot be merged into a consistent plan without sharing steps. (*Hint:* Places where two physical parts of the house come together are also places where two subplans tend to interact.)

**12.4** Construct an example of the violation of the downward solution property. That is, find an abstract solution such that, when one of the steps is decomposed, the plan becomes inconsistent in that one of its threats cannot be resolved.

**12.5** Prove that the upward solution property always holds for approximation hierarchy planning (see page 380). You may use Tenenberg (1988) for hints.

**12.6** Add existential quantifiers (3) to the plan language, using whatever syntax restrictions you find reasonable, and extend the planner to accommodate them.

**12.7** Write operators for the shopping domain that will enable the planner to achieve the goal of having three oranges by grabbing a bag, going to the store, grabbing the oranges, paying for them, and returning home. Model money as a resource. Use universal quantification in the operators, and show that the original contents of the bag will still be there at the end of the plan.

**12.8** We said in Section 11.6 that the SELECT-SUB-GOAL part of the POP algorithm was *not* a backtrack point—that we can work on subgoals in any order without affecting completeness (although the choice certainly has an effect on efficiency). When we change the SELECT-SUB-GOAL part to handle hierarchical decomposition, do we need to make it a backtrack point?

**12.9** Some domains have resources that are monotonically decreasing or increasing. For example, time is monotonically increasing, and if there is a *Buy* operator, but no *Earn*, *Beg*, *Borrow*, or *Steal*, then money is monotonically decreasing. Knowing this can cut the search space: if you have a partial plan whose steps require more money than is available, then you can avoid considering any of the possible completions of the plan.

- a. Explain how to determine if a measure is monotonic, given a set of operator descriptions.
- b. Design an experiment to analyze the efficiency gains resulting from the use of monotonic resources in planning.

**12.10** Some of the operations in standard programming languages can be modelled as actions that change the state of the world. For example, the assignment operation changes the contents of a memory location; the print operation changes the state of the output stream. A program consisting of these operations can also be considered as a plan, whose goal is given by the specification of the program. Therefore, planning algorithms can be used to construct programs that achieve a given specification.

- a. Write an operator schema for the assignment operator (assigning the value of one variable to another).
- b. Show how object creation can be used by a planner to produce a plan for exchanging the values of two variables using a temporary variable.

# 13 PLANNING AND ACTING

*In which planning systems must face up to the awful prospect of actually having to take their own advice.*

The assumptions required for flawless planning and execution, given the algorithms in the previous chapters, are that the world be accessible, static, and deterministic—just as for our simple search methods. Furthermore, the action descriptions must be correct and complete, describing all the consequences exactly. We described a planning agent in this ideal case in Chapter 11; the resemblance to the simple problem-solving agent of Chapter 3 was no coincidence.

In real-world domains, agents have to deal with both incomplete and incorrect information. Incompleteness arises because the world is inaccessible; for example, in the shopping world, the agent may not know where the milk is kept unless it asks. Incorrectness arises because the world does not necessarily match the agent's model of it; for example, the price of milk may have doubled overnight, and the agent's wallet may have been pickpocketed.

There are two different ways to deal with the problems arising from incomplete and incorrect information:

CONDITIONAL  
PLANNING

CONTINGENCY  
SENSING ACTIONS

EXECUTION  
MONITORING

REPLANNING

- ◊ **Conditional planning:** Also known as **contingency planning**, conditional planning deals with incomplete information by constructing a conditional plan that accounts for each possible situation or **contingency** that could arise. The agent finds out which part of the plan to execute by including **sensing actions** in the plan to test for the appropriate conditions. For example, the shopping agent might want to include a sensing action in its shopping plan to check the price of some object in case it is too expensive. Conditional planning is discussed in Section 13.1.
- ◊ **Execution monitoring:** The simple planning agent described in Chapter 11 executes its plan "with its eyes closed"—once it has a plan to execute, it does not use its percepts to select actions. Obviously this is a very fragile strategy when there is a possibility that the agent is using incorrect information about the world. By monitoring what is happening while it executes the plan, the agent can tell when things go wrong. It can then do **replanning** to find a way to achieve its goals from the new situation. For example, if the agent discovers that it does not have enough money to pay for all the items it has picked up, it can return some and replace them with cheaper versions. In Section 13.2 we look

at a simple replanning agent that implements this strategy. Section 13.3 elaborates on this design to provide a full integration of planning and execution.

DEFERRING

Execution monitoring is related to conditional planning in the following way. An agent that builds a plan and then executes it while watching for errors is, in a sense, taking into account the possible conditions that constitute execution errors. Unlike a conditional planner, however, the execution monitoring agent is actually **deferring** the job of dealing with those conditions until they actually arise. The two approaches of course can be combined by planning for some contingencies and leaving others to be dealt with later if they occur. In Section 13.4, we will discuss when one might prefer to use one or the other approach.

## 13.1 CONDITIONAL PLANNING

We begin by looking at the nature of conditional plans and how an agent executes them. This will help to clarify the relationship between sensing actions in the plan and their effects on the agent's knowledge base. We then explain how to construct conditional plans.

### The nature of conditional plans

Let us consider the problem of fixing a flat tire. Suppose we have the following three action schemata:

$$\begin{aligned} &Op(\text{ACTION:} Remove(x), \\ &\quad \text{PRECOND: } On(x), \\ &\quad \text{EFFECT: } Off(x) \wedge ClearHub(x) \wedge \neg On(x)) \\ &Op(\text{ACTION:} PutOn(x), \\ &\quad \text{PRECOND: } Off(x) \wedge ClearHub(x), \\ &\quad \text{EFFECT: } On(x) \wedge \neg ClearHub(x) \wedge \neg Off(x)) \\ &Op(\text{ACTION:} Inflate(x), \\ &\quad \text{PRECOND: } Intact(x) \wedge Flat(x), \\ &\quad \text{EFFECT: } Inflated(x) \wedge \neg Flat(x)) \end{aligned}$$

If our goal is to have an inflated tire on the wheel:

$$On(x) \wedge Inflated(x)$$

and the initial conditions are

$$Inflated(Spare) \wedge Intact(Spare) \wedge Off(Spare) \wedge On(Tire_1) \wedge Flat(Tire_1)$$

then any of the standard planners described in the previous chapters would be able to come up with the following plan:

$$[Remove(Tire_1), PutOn(Spare)]$$

If the absence of *Intact(Tire<sub>1</sub>)* in the initial state really means that the tire is not intact (as the standard planners assume), then this is all well and good. But suppose we have incomplete

knowledge of the world—the tire may be flat because it is punctured, or just because it has not been pumped up lately. Because changing a tire is a dirty and time-consuming business, it would be better if the agent could execute a conditional plan: if  $Tire_1$  is intact, then inflate it. If not, remove it and put on the spare. To express this formally, we can extend our original notation for plan steps with a conditional step  $If(<Condition>, <ThenPart>, <ElsePart>)$ . Thus, the tire-fixing plan now includes the step

$$If(Intact(Tire_1), [Inflate(Tire_1)], [Remove(Tire_1), PutOn(Spare)])$$

Thus, the conditional planning agent can sometimes do better than the standard planning agents described earlier. Furthermore, there are cases where a conditional plan is the only possible plan. If the agent does not know if its spare tire is flat or inflated, then the standard planner will fail, whereas the conditional planner can insert a second conditional step that inflates the spare if necessary. Lastly, if there is a possibility that both tires have holes, then neither planner can come up with a guaranteed plan. In this case, a conditional planner can plan for all the cases where success is possible, and insert a *Fail* action on those branches where no completion is possible.

Plans that include conditional steps are executed as follows. When the conditional step is executed, the agent first tests the condition against its knowledge base. It then continues executing either the then-part or the else-part, depending on whether the condition is true or false. The then-part and the else-part can themselves be plans, allowing arbitrary nesting of conditionals. The conditional planning agent design is shown in Figure 13.1. Notice that it deals with nested conditional steps by following the appropriate conditional branches until it finds a real action to do. (The conditional planning algorithm itself, CPOP, will be discussed later.)

The crucial part of executing conditional plans is that the agent must, at the time of execution of a conditional step, be able to decide the truth or falsehood of the condition—that is, *the condition must be known to the agent* at that point in the plan. If the agent does not know if  $Tire_1$  is intact or not, it cannot execute the previously shown plan. What the agent knows at any point is of course determined by the sequence of percepts up to that point, the sequence of actions carried out, and the agent's initial knowledge. In this case, the initial conditions in the knowledge base do not say anything about  $Intact(Tire_1)$ . Furthermore, the agent may have no actions that cause  $Intact(Tire_1)$  to become true.<sup>1</sup> *To ensure that a conditional plan is executable, the agent must insert actions that cause the relevant conditions to become known by the agent.*

Facts become known to the agent through its percepts, so what we mean by the previous remark is that the agent must act in such a way as to make sure it receives the appropriate percepts. For example, one way to come to know that a tire is intact is to put some air into it and place one's listening device in close proximity. A hissing percept then enables the agent to infer that the tire is not intact.<sup>2</sup> Suppose we use the name *CheckTire(x)* to refer to an action that establishes the state of the tire  $x$ . This is an example of a **sensing action**.

Using the situation calculus description of sensing actions described in Chapter 8, we would write

$$\forall x, s \quad Tire(x) \Rightarrow KnowsWhether("Intact(x)", Result(CheckTire(x), s))$$

<sup>1</sup> Note that if, for example, a *Patch(Tire\_1)* action were available, then a standard plan could be constructed.

<sup>2</sup> Agents without sound percepts can wet the tire. A bubbling visual percept then suggests the tire is compromised.

```

function CONDITIONAL-PLANNING-AGENT(percept)returns an action
  static: KB, a knowledge base (includes action descriptions)
    p, a plan, initially NoPlan
    t, a counter, initially 0, indicating time
    G, a goal

  TELL(KB,MAKE-PERCEPT-SENTENCE(percept, t))
  current  $\leftarrow$  STATE-DESCRIPTION(KB,t)
  if p = NoPlan then p  $\leftarrow$  CPOP(current,G, KB)
  if p = NoPlan or p is empty then action  $\leftarrow$  NoOp
  else
    action  $\leftarrow$  FIRST(p)
    while CONDITIONAL?i(action) do
      if ASK(KB,CONDITION-PART[action]) then p  $\leftarrow$  APPEND(THEN-PART[action], REST(p))
      else p  $\leftarrow$  APPEND(ELSE-PART[action], REST(p))
      action  $\leftarrow$  FIRST(p)
    end
    p  $\leftarrow$  REST(p)
  TELL(KB,MAKE-ACTION-SENTENCE(action, 0))
  f  $\leftarrow$  t + 1
  return action

```

**Figure 13.1** A conditional planning agent.

In our action schema format, we would write

*Op*(ACTION:*CheckTire*(*x*),  
 PRECOND:*Tire*(*x*),  
 EFFECT:*KnowsWhether*("*Intact(x)*") )

Notice that as well as having knowledge effects, a sensing action can have ordinary effects. For example, if the *CheckTire* action uses the water method, then the tire will become wet. Sensing actions can also have preconditions that need to be established. For example, we might need to fetch the pump in order to put some air in the tire in order to check it. A *conditional planner* therefore will sometimes create plans that involve carrying out ordinary actions for the purpose of obtaining some needed information.



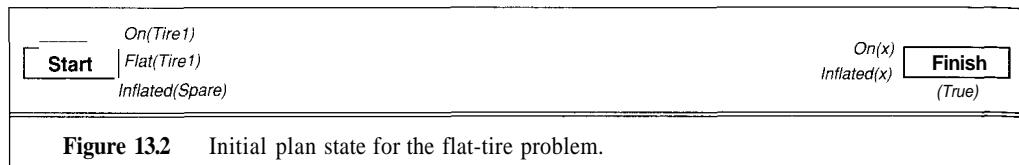
## An algorithm for generating conditional plans

CONTEXT

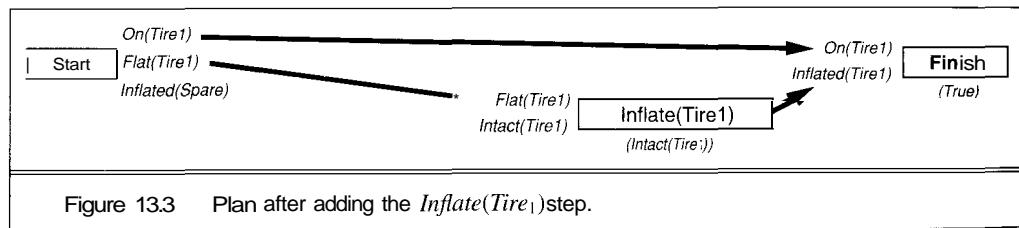
The process of generating conditional plans is much like the planning process described in Chapter 11. The main additional construct is the **context** of a step in the plan. A step's context is simply the union of the conditions that must hold in order for the step to be executed—essentially, it describes the "branch" on which the step lies. For example, the action *Inflate(Tire<sub>1</sub>)* in the earlier plan has a context *Intact(Tire<sub>1</sub>)*. Once it is established that a step has a certain context, then subsequent steps in the plan inherit that context. Because it cannot be the case that two steps

with distinct contexts can both be executed, such steps cannot interfere with each other. Contexts are therefore essential for keeping track of which steps can establish or violate the preconditions of which other steps. An example will make this clear.

The flat-tire plan begins with the usual start and finish steps (Figure 13.2). Notice that the finish step has the context *True*, indicating that no assumptions have been made so far.



There are two open conditions to be resolved: *On(x)* and *Inflated(x)*. The first is satisfied by adding a link from the start step, with the unifier  $\{x/Tire_1\}$ . The second is satisfied by adding the step *Inflate(Tire<sub>1</sub>)*, which has preconditions *Flat(Tire<sub>1</sub>)* and *Intact(Tire<sub>1</sub>)* (see Figure 13.3).



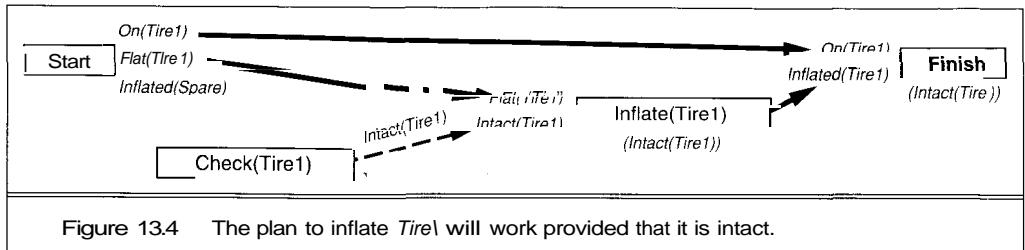
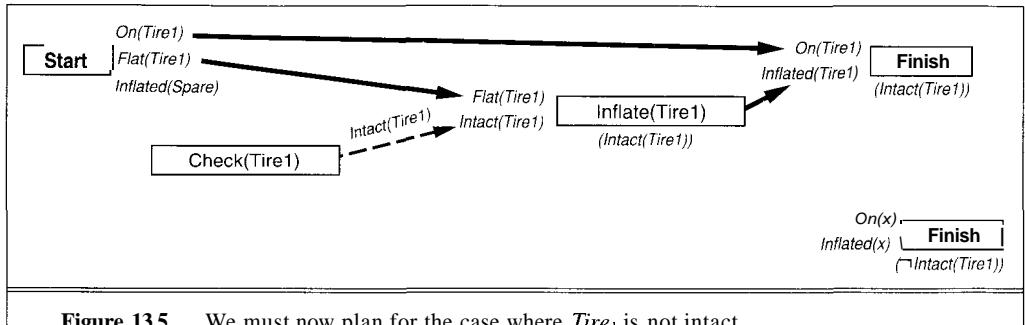
The open condition *Flat(Tire<sub>1</sub>)* is satisfied by adding a link from the start step. The interesting part is what to do with the *Intact(Tire<sub>1</sub>)* condition. In the statement of the problem there are no actions that can make the tire intact—that is, no action schema with the effect *Intact(x)*. At this point a standard causal-link planner would abandon this plan and try another way to achieve the goal. There is, however, an action *CheckTire(x)* that allows one to *know* the truth value of a proposition that unifies with *Intact(Tire<sub>1</sub>)*. If (and this is sometimes a big if) the outcome of checking the tire is that the tire is known to be intact, then the *Inflate(Tire<sub>1</sub>)* step can be achieved. We therefore add the *CheckTire* step to the plan with a **conditional link** (shown as a dotted arrow in Figure 13.4) to the *Inflate(Tire<sub>1</sub>)* step. The *CheckTire* step is called a **conditional step** because it will become a branch point in the final plan. The inflate step and the finish step now acquire a context label stating that they are assuming the outcome *Intact(Tire<sub>1</sub>)* rather than  $\neg$ *Intact(Tire<sub>1</sub>)*. Because *CheckTire* has no preconditions in our simple formulation, the plan is complete given the context of the finish step.

Obviously, we cannot stop here. We need a plan that works in both cases. The conditional planner ensures this by adding a second copy of the original finish step, labelled with a context that is the negation of the existing context (see Figure 13.5).<sup>3</sup> In this way, the planner covers an

CONDITIONAL LINK

CONDITIONAL STEP

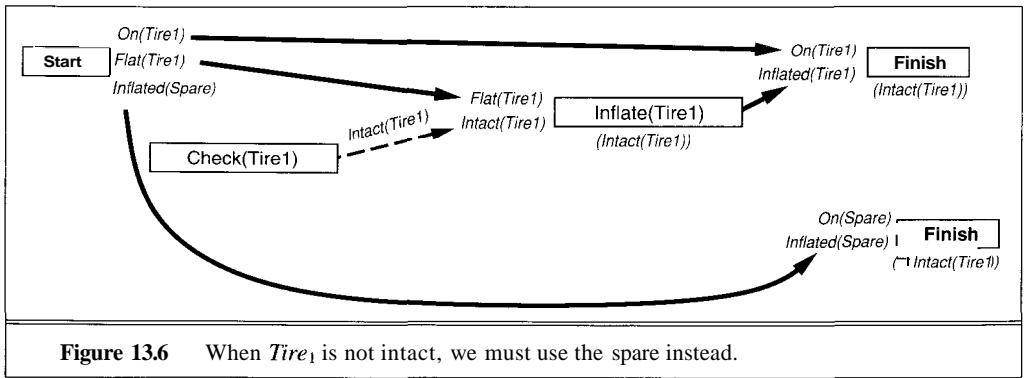
<sup>3</sup> If the solution of this new branch requires further context assumptions, then a third copy of the finish step will be added whose context is the negation of the disjunction of the existing finish steps. This continues until no more assumptions are needed.

Figure 13.4 The plan to inflate  $Tire_1$  will work provided that it is intact.Figure 13.5 We must now plan for the case where  $Tire_1$  is not intact.

exhaustive set of possibilities—for every possible outcome, there is a corresponding finish step, and a path to get to the finish step.

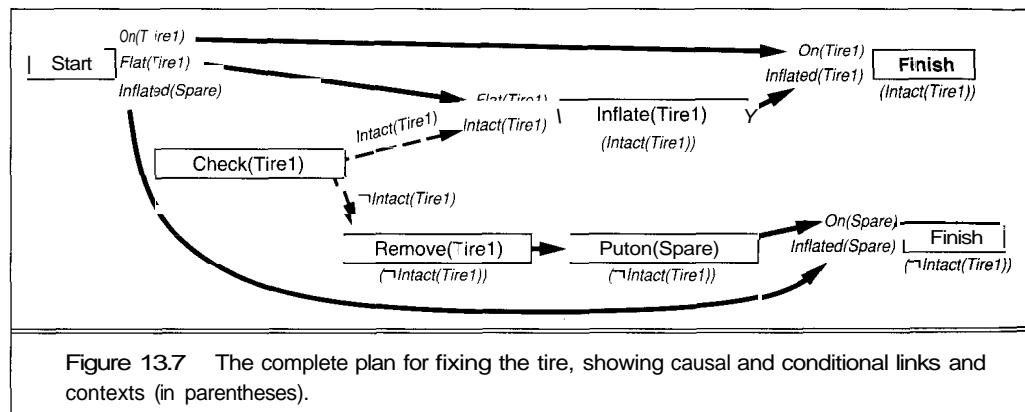
Now we need to solve the goal when  $Tire_1$  has a hole in it. Here the context is very useful. If we were to try to add the step  $Inflate(Tire_1)$  to the plan, we would immediately see that the precondition  $Intact(Tire_1)$  is inconsistent with the context  $\neg Intact(Tire_1)$ . Thus, the only ways to satisfy the  $Inflated(x)$  condition are to link it to the start step with the unifier  $\{x/spare\}$  or to add an  $Inflate$  step for the spare. Because the latter leads to a dead end (because the spare is not flat), we choose the former. This leads to the plan state in Figure 13.6.

The steps  $Remove(Tire_1)$  and  $PutOn(Spare)$  are now added to the plan to satisfy the condition  $On(Spare)$ , using standard causal-link addition. Initially, the steps would have a *True*

Figure 13.6 When  $Tire_1$  is not intact, we must use the spare instead.

## CONDITIONING

context, because it has not yet been established that they can only be executed under certain circumstances. This means that we have to check how the steps interact with other steps in the plan. In particular, the *Remove(Tire1)* step threatens the causal link protecting *On(Tire1)* in the first finish step (the one with the context *(Intact(Tire1))*). In a standard causal-link planner, the only solution would be to promote or demote the *Remove(Tire1)* step so that it cannot interfere. In the conditional planner, we can also resolve the threat by **conditioning** the step so that its context becomes incompatible with the context of the step whose precondition it is threatening (in this case, the first finish step). Conditioning is achieved by finding a conditional step that has a possible outcome that would make the threatening step's context incompatible with the causal link's context. In this case, the *CheckTire* step has a possible outcome  $\neg\text{Intact}(\text{Tire1})$ . If we make a conditional link from the *CheckTire* step to the *Remove(Tire1)* step, then the remove step is no longer a threat. The new context is inherited by the *PutOn(Spare)* step, and the plan is now complete (Figure 13.7).



The algorithm is called CPOP (for Conditional Partial-Order Planner). It builds on the POP algorithm, and extends it by incorporating contexts, multiple finish steps and the conditioning process for resolving potential threats. It is shown in Figure 13.8.

## Extending the plan language

## PARAMETERIZED PLANS

The conditional steps we used in the previous section had only two possible outcomes. In some cases, however, a sensing action can have any number of outcomes. For example, checking the color of some object might result in a sentence of the form *Color(x,c)* being known for some value of *c*. Sensing actions of this type can be used in **parameterized plans**, where the exact actions to be carried out will not be known until the plan is executed. For example, suppose we have a goal such as

*Color(Chair,c) A Color(Table,c)*

```

function CPOP(initial, goals, operators) returns plan
  plan  $\leftarrow$  MAKE-PLAN(initial, goals)
  loop do

    Termination:
    if there are no unsatisfied preconditions
      and the contexts of the finish steps are exhaustive
      then return plan

    Alternative context generation:
    if the plans for existing finish steps are complete and have contexts  $C_1 \dots C_n$  then
      add a new finish step with a context  $\neg(C_1 \vee \dots \vee C_n)$ 
      this becomes the current context

    Subgoal selection and addition:
    find a plan step  $S_{need}$  with an open precondition  $c$ 

    Action selection:
    choose a step  $S_{add}$  from operators or STEPS(plan) that adds  $c$  or
      knowledge of  $c$  and has a context compatible with the current context
    if there is no such step
      then fail
    add  $S_{add} \dashv S_{need}$  to LINKS(plan)
    add  $S_{add} < S_{need}$  to ORDERINGS(plan)
    if  $S_{add}$  is a newly added step then
      add  $S_{add}$  to STEPS(plan)
      add  $Start < S_{add} < Finish$  to ORDERINGS(plan)

    Threat resolution:
    for each step  $S_{threat}$  that potentially threatens any causal link  $S_i \dashv c \dashv 5\}$ 
      with a compatible context do
        choose one of
        Promotion: Add  $S_{threat} < S_i$  to ORDERINGS(plan)
        Demotion: Add  $S_j < S_{threat}$  to ORDERINGS(plan)
        Conditioning:
          find a conditional step  $S_{cond}$  possibly before both  $S_{threat}$  and  $S_j$ , where
          1. the context of  $S_{cond}$  is compatible with the contexts of  $S_{threat}$  and  $S_j$ ;
          2. the step has outcomes consistent with  $S_{threat}$  and  $S_j$ , respectively
          add conditioning links for the outcomes from  $S_{cond}$  to  $S_{threat}$  and  $S_j$ 
          augment and propagate the contexts of  $S_{threat}$  and  $S_j$ 

        if no choice is consistent
          then fail
      end
    end

```

**Figure 13.8** The CPOP algorithm for constructing conditional plans.

("the chair and table are the same color"). The chair is initially unpainted, and we have some paints and a paintbrush. Then we might use the plan

$[SenseColor(Table), KnowsWhat("Color(Table, \underline{c}))], GetPaint(c), Paint(Chair, c)]$

The last two steps are parameterized, because until execution the agent will not know the value of  $c$ . We call  $c$  a **runtime variable**, as distinguished from normal planning variables whose values are known as soon as the plan is made. The step *SenseColor(Table)* will have the effect of providing percepts sufficient to allow the agent to deduce the color of the table. Then the action *KnowsWhat("Color(Table, \underline{c}))* is executed simply by querying the knowledge base to establish a value for the variable  $c$ . This value can be used in subsequent steps such as *GetPaint(c)*, or in conditional steps such as *If( $c = Green, [\dots], [\dots]$ )*. Sensing actions are defined just as in the binary-condition case:

$Op(ACTION:SenseColor(x),$   
**EFFECT:** *Knows What("Color( $\underline{x}, \underline{c}$ )")*)

In situation calculus, the action would be described by

$\forall x, s \exists c \ KnowsWhat("Color(\underline{x}, \underline{c})", Result(SenseColor(x), s))$

The variables  $x$  and  $c$  are treated differently by the planner. Logically,  $c$  is existentially quantified in the situation calculus representation, and thus must be treated as a Skolem function of the object being sensed and the situation in which it is sensed. In the planner, runtime variables like  $c$  unify only with ordinary variables, and not with constants or with each other. This corresponds exactly to what would happen with Skolem functions.

When we have the ability to discover that certain facts *are* true, as well as the ability to cause them to *become* true, then we may wish to have some control over which facts are changed and which are preserved. For example, the goal of having the table and chair the same color can be achieved by painting them both black, regardless of what color the table is at the start. This can be prevented by protecting the table's color so that the agent has to sense it, rather than painting over it. A **maintenance goal** can be used to specify this:

*Color(Chair, c) A Color(Table, c) A Maintain(Color(Table, x))*

The *Maintain* goal will ensure that no action is inserted in the plan that has an ordinary causal effect that changes the color of the table. This is done in a causal-link planner by adding a causal link from the start step to the finish step protecting the table's initial color.

Plans with conditionals start to look suspiciously like programs. Moreover, executing such plans starts to look rather like interpreting a program. The similarity becomes even stronger when we include loops in plans. A loop is like a conditional, except that when the condition holds, a portion of the plan is repeated. For example, we might include a looping step to make sure the chair is painted properly:

*While(Knows("UnevenColor(Chair)")[Paint(Chair, c), CheckColor(Chair)])*

Techniques for generating plans with conditionals and loops are almost identical to those for generating programs from logical specifications (so-called **automatic programming**). Even a standard planner can do automatic programming of a simple kind if we encode as STRIPS operators the actions corresponding to assignment statements, procedure calls, printing, and so on.

## 3.2 A SIMPLE REPLANNING AGENT

As long as the world behaves exactly as the action descriptions describe it, then executing a plan in the ideal or incomplete-information cases will always result in goal achievement. As each step is executed, the world state will be as predicted—as long as nothing goes wrong.

"Something going wrong" means that the world state after an action is not as predicted. More specifically, the remaining plan segment will fail if any of its preconditions is not met. The preconditions of a plan segment (as opposed to an individual step) are all those preconditions of the steps in the segment that are not established by other steps in the segment. It is straightforward to annotate a plan at each step with the preconditions required for successful completion of the remaining steps. In terms of the plan description adopted in Chapter 11, the required conditions are just the propositions protected by all the causal links beginning at or before the current step and ending at or after it. Then we can detect a potential failure by comparing the current preconditions with the state description generated from the percept sequence. This is the standard model of execution monitoring, first used by the original STRIPS planner. STRIPS also introduced the **triangle table**, an efficient representation for fully annotated plans.

TRIANGLE TABLE

ACTION MONITORING

A second approach is to check the preconditions of each action as it is executed, rather than checking the preconditions of the entire remaining plan. This is called **action monitoring**. As well as being simpler and avoiding the need for annotations, this method fits in well with realistic systems where an individual action failure can be recognized. For example, if a robot agent issues a command to the motor subsystem to move two meters forward, the subsystem can report a failure if the robot bumps into an obstacle that materialized unexpectedly. On the other hand, action monitoring is less effective than execution monitoring, because it does not look ahead to see that an unexpected current state will cause an action failure some time in the future. For example, the obstacle that the robot bumped into might have been knocked off the table by accident much earlier in the plan. An agent using execution monitoring could have realized the problem and picked it up again.

Action monitoring is also useful when a goal is serendipitously achieved. That is, if someone or something else has already changed the world so that the goal is achieved, action monitoring notices this and avoids wasting time by going through the rest of the plan.

These forms of monitoring require that the percepts provide enough information to tell if a plan or action is about to fail. In an inaccessible world where the relevant conditions are not perceivable, more complicated strategies are needed to cope with undetected but potentially serious deviations from expectations. This issue is beyond the scope of the current chapter.

We can divide the causes of plan failure into two kinds, depending on whether it is possible to anticipate the possible contingencies:

BOUNDED INDETERMINACY

**0 Bounded indeterminacy:** In this case, actions can have unexpected effects, but the possible effects can be enumerated and described as part of the action description axiom. For example, the result of opening a can of paint can be described as the disjunction of having paint available, having an empty can, or spilling the paint. Using a combination of CPOP and the "D" (disjunctive) part of POP-DUNC we can generate conditional plans to deal with this kind of indeterminacy.

- ◊ **Unbounded indeterminacy:** In this case, the set of possible unexpected outcomes is too large to be completely enumerated. This would be the case in very complex and/or dynamic domains such as driving, economic planning, and military strategy. In such cases, we can plan for at most a limited number of contingencies, and must be able to *replan* when reality does not behave as expected.

The next subsection describes a simple method for replanning based on trying to get the plan "back on track" as quickly as possible. Section 13.3 describes a more comprehensive approach that deals with unexpected conditions as an integral part of the decision-making process.

## Simple replanning with execution monitoring

One approach to replanning based on execution monitoring is shown in Figure 13.9. The simple planning agent is modified so that it keeps track of both the remaining plan segment  $p$  and the complete plan  $q$ . Before carrying out the first action of  $p$ , it checks to see whether the preconditions of the  $p$  are met. If not, it calls CHOOSE-BEST-CONTINUATION to choose some point in the complete plan  $q$  such that the plan  $p'$  from that point to the end of  $q$  is easiest to achieve from the current state. The new plan is to first achieve the preconditions of  $p'$  and then execute it.

Consider how REPLANNING-AGENT will perform the task of painting the chair to match the table. Suppose that the motor subsystem responsible for the painting action is imperfect and sometimes leaves small areas unpainted. Then after the *Paint(Chair, c)* action is done, the execution-monitoring part will check the preconditions for the rest of the plan; the preconditions

```

function REPLANNING-AGENT(percept) returns an action
  static: KB, a knowledge base (includes action descriptions)
    p, an annotated plan, initially NoPlan
    q, an annotated plan, initially NoPlan
    G, a goal

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  current ← STATE-DESCRIPTION(KB, t)
  if p = NoPlan then
    p ← PLANNER(current, G, KB)
    q ← p
    if p = NoPlan or p is empty then return NoOp
    if PRECONDITIONS(p) not currently true in KB then
      p' ← CHOOSE-BEST-CONTINUATION(current, q)
      p ← APPEND(PLANNER(current, PRECONDITIONS(p'), KB), p')
      q ← p
      action ← FIRST(p)
      p ← REST(p)
    return action
  
```

Figure 13.9 An agent that does execution monitoring and replanning.

are just the goal conditions because the remaining plan  $p$  is now empty, and the agent will detect that the chair is not all the same color as the table. Looking at the original plan  $q$ , the current state is identical to the precondition before the chair-painting step, so the agent will now try to paint over the bare spots. This behavior will cycle until the chair is completely painted.

Suppose instead that the agent runs out of paint during the painting process. This is not envisaged by the action description for *Paint*, but it will be detected because the chair will again not be completely painted. At this point, the current state matches the precondition of the plan beginning with *GetPaint*, so the agent will go off and get a new can of paint before continuing.

Consider again the agent's behavior in the first case, as it paints and repaints the chair. Notice that the *behavior* is identical to that of a conditional planning agent running the looping plan shown earlier. *The difference lies in the time at which the computation is done and the information is available to the computation process.* The conditional planning agent reasons explicitly about the possibility of uneven paint, and prepares for it even though it may not occur. The looping behavior results from a looping plan. The replanning agent assumes at planning time that painting succeeds, but during execution checks on the results and plans just for those contingencies that actually arise. The looping behavior results not from a looping plan but from the interaction between action failures and a persistent replanner.

We should mention the question of **learning** in response to failed expectations about the results of actions. Consider a plan for the painting agent that includes an action to open a door (perhaps to the paint store). If the door sticks a little, the replanning agent will try again until the door opens. But if the door is locked, the agent has a problem. Of course, if the agent already knows about locked doors, then *Unlocked* will be a precondition of opening the door, and the agent will have inserted a *CheckIfLocked* action that observes the state of the door, and perhaps a conditional branch to fetch the key. But if the agent does not know about locked doors, it will continue pulling on the door indefinitely. What we would like to happen is for the agent to learn that its action description is wrong; in this case, there is a missing precondition. We will see how this kind of learning can take place in Chapter 21.



### 13.3 FULLY INTEGRATED PLANNING AND EXECUTION

SITUATED PLANNING AGENT

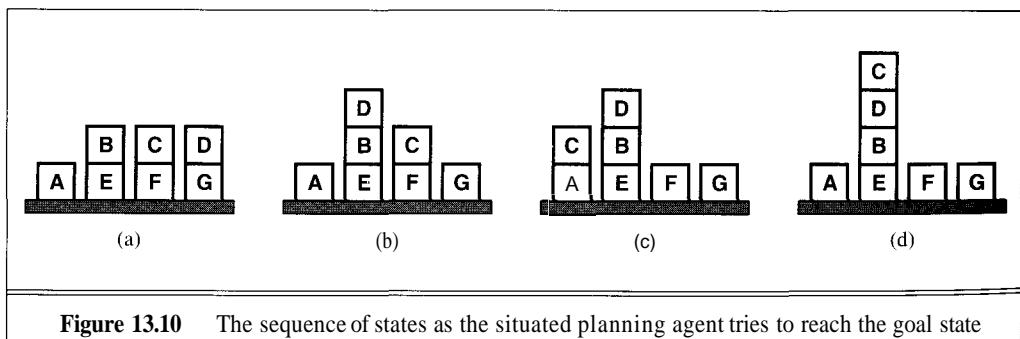
In this section, we describe a more comprehensive approach to plan execution, in which the planning and execution processes are fully integrated. Rather than thinking of the planner and execution monitor as separate processes, one of which passes its results to the other, we can think of them as a single process in a **situated planning agent**.<sup>4</sup> The agent is thought of as always being *part of the way through* executing a plan—the grand plan of living its life. Its activities include executing some steps of the plan that are ready to be executed; refining the plan to resolve any of the standard deficiencies (open conditions, potential clobbering, and so on); refining the plan in the light of additional information obtained during execution; and fixing the

<sup>4</sup> The word "situated," which became popular in AI in the late 1980s, is intended to emphasize that the process of deliberation takes place in an agent that is directly connected to an environment. In this book all the agents are "situated," but the situated planning agent integrates deliberation and action to a greater extent than some of the other designs.

plan in the light of unexpected changes in the environment, which might include recovering from execution errors or removing steps that have been made redundant by serendipitous occurrences. Obviously, when it first gets a new goal the agent will have no actions ready to execute, so it will spend a while generating a partial plan. It is quite possible, however, for the agent to begin execution before the plan is complete, especially when it has independent subgoals to achieve. The situated agent continuously monitors the world, updating its world model from new percepts even if its deliberations are still continuing.

As in the discussion of the conditional planner, we will first go through an example and then give the planning algorithm. We will keep to the formulation of steps and plans used by the partial-order planner POP, rather than the more expressive languages used in Chapter 12. It is, of course, possible to incorporate more expressive languages, as well as conditional planning techniques, into a situated planner.

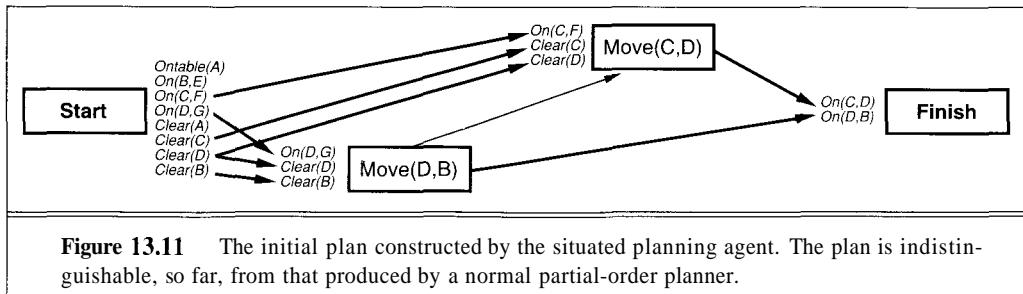
The example we will use is a version of the blocks world. The start state is shown in Figure 13.10(a), and the goal is  $On(C,D) \wedge On(D,B)$ . The action we will need is  $Move(x,y)$ , which moves block  $x$  onto block  $y$ , provided both are clear. Its action schema is

$$\begin{aligned} Op(\text{ACTION: } & Move(x,y), \\ \text{PRECOND: } & Clear(x) \wedge Clear(y) \wedge On(x,z), \\ \text{EFFECT: } & On(x,y) \wedge Clear(z) \wedge \neg On(x,z) \wedge \neg Clear(y)) \end{aligned}$$


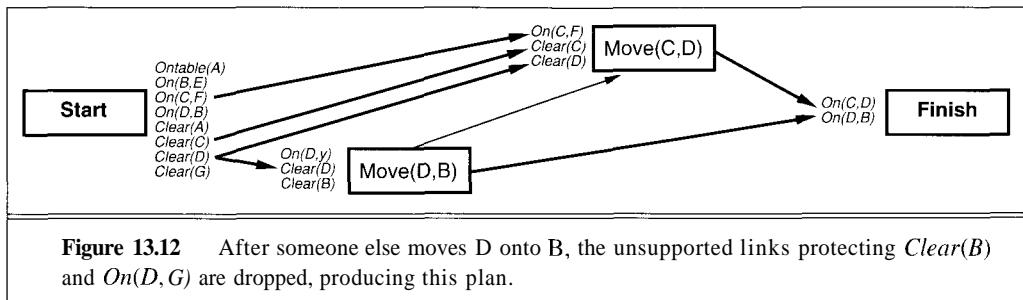
**Figure 13.10** The sequence of states as the situated planning agent tries to reach the goal state  $On(C,D) \wedge On(D,B)$  as shown in (d). The start state is (a). At (b), another agent has interfered, putting  $D$  on  $B$ . At (c), the agent has executed  $Move(C,D)$  but has failed, dropping  $C$  on  $A$  instead. It retries  $Move(C,D)$ , reaching the goal state (d).

The agent first constructs the plan shown in Figure 13.11. Notice that although the preconditions of both actions are satisfied by the initial state, there is an ordering constraint putting  $Move(D,B)$  before  $Move(C,D)$ . This is needed to protect the condition  $Clear(D)$  until  $Move(D,B)$  is completed.

At this point, the plan is ready to be executed, but nature intervenes. An external agent moves  $D$  onto  $B$  (perhaps the agent's teacher getting impatient), and the world is now in the state shown in Figure 13.10(b). Now  $Clear(B)$  and  $On(D,G)$  are no longer true in the initial state, which is updated from the new percept. The causal links that were supplying the preconditions  $Clear(B)$  and  $On(D,G)$  for the  $Move(D,B)$  action become invalid, and must be removed from the plan. The new plan is shown in Figure 13.12. Notice that two of the preconditions for  $Move(D,B)$



**Figure 13.11** The initial plan constructed by the situated planning agent. The plan is indistinguishable, so far, from that produced by a normal partial-order planner.



**Figure 13.12** After someone else moves D onto B, the unsupported links protecting *Clear(B)* and *On(D, G)* are dropped, producing this plan.

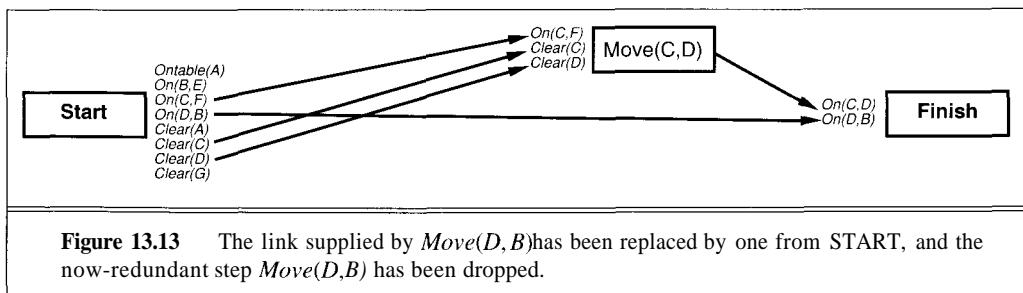
are now open, and the precondition *On(D, y)* is now uninstantiated because there is no reason to assume the move will be from G any more.

Now the agent can take advantage of the "helpful" interference by noticing that the causal link protecting *On(D, B)* and supplied by *Move(D, B)* can be replaced by a direct link from START. This process is called **extending** a causal link, and is done whenever a condition can be supplied by an earlier step instead of a later one without causing a new threat.

EXTENDING

REDUNDANT STEP

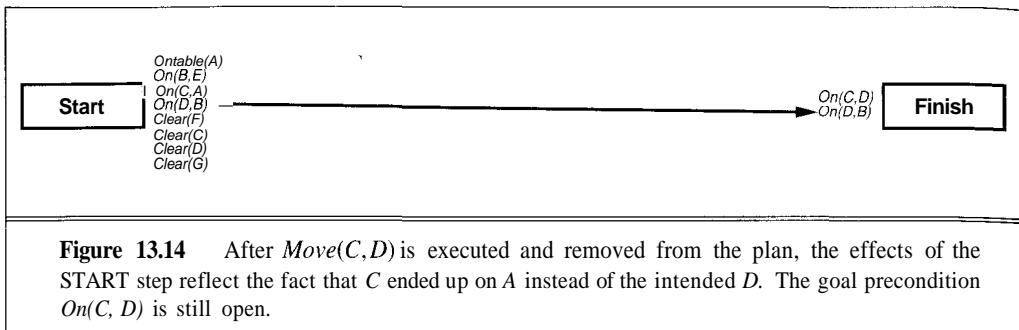
Once the old link from *Move(D, B)* is removed, the step no longer supplies any causal links at all. It is now a **redundant step**. All redundant steps are dropped from the plan, along with any links supplying them. This gives us the plan shown in Figure 13.13.



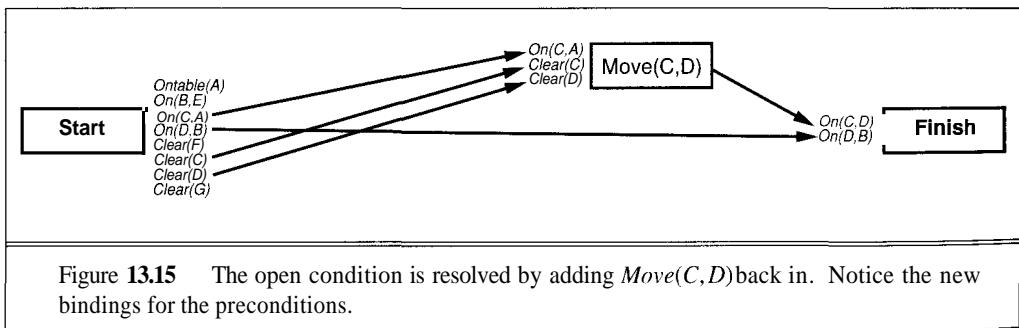
**Figure 13.13** The link supplied by *Move(D, B)* has been replaced by one from START, and the now-redundant step *Move(D, B)* has been dropped.

Now the step *Move(C, D)* is ready to be executed, because all of its preconditions are satisfied by the START step, no other steps are necessarily before it, and it does not threaten any other link in the plan. The step is removed from the plan and executed. Unfortunately, the agent is clumsy and drops C onto A instead of D, giving the state shown in Figure 13.10(c). The new

plan state is shown in Figure 13.14. Notice that although there are now no actions in the plan, there is still an open condition for the FINISH step.

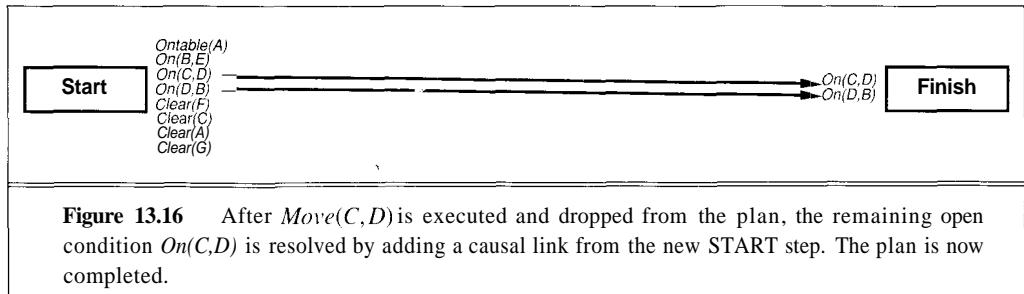


The agent now does the same planning operations as a normal planner, adding a new step to satisfy the open condition. Once again,  $Move(C, D)$  will satisfy the goal condition. Its preconditions are satisfied in turn by new causal links from the START step. The new plan appears in Figure 13.15.



Once again,  $Move(C, D)$  is ready for execution. This time it works, resulting in the goal state shown in Figure 13.10(d). Once the step is dropped from the plan, the goal condition  $On(C, D)$  becomes open again. Because the START step is updated to reflect the new world state, however, the goal condition can be satisfied immediately by a link from the START step. This is the normal course of events when an action is successful. The final plan state is shown in Figure 13.16. Because all the goal conditions are satisfied by the START step and there are no remaining actions, the agent resets the plan and looks for something else to do.

The complete agent design is shown in Figure 13.17 in much the same form as used for POP and CPOP, although we abbreviate the part in common (resolving standard flaws). One significant structural difference is that planning and acting are the same "loop" as implemented by the coupling between agent and environment. After each plan modification, an action is returned (even if it is a *NoOp*) and the world model is updated from the new percept. We assume that each action finishes executing before the next percept arrives. To allow for extended execution with a completion signal, an executed action must remain in the plan until it is completed.



## 13.4 DISCUSSION AND EXTENSIONS

We have arrived at an agent design that addresses many of the issues arising in real domains:

- The agent can use explicit domain descriptions and goals to control its behavior.
- By using partial-order planning, it can take advantage of problem decomposition to deal with complex domains without necessarily suffering exponential complexity.
- By using the techniques described in Chapter 12, it can handle domains involving conditional effects, universally quantified effects, object creation and deletion, and ramifications. It can also use "canned plans" to achieve subgoals.
- It can deal with errors in its domain description, and, by incorporating conditional planning, it can plan to obtain information when more is needed.
- It can deal with a dynamically changing world by incrementally fixing its plan as it detects errors and unfulfilled preconditions.

Clearly, this is progress. It is, however, a good idea to examine each advance in capabilities and try to see where it breaks down.

### Comparing conditional planning and replanning

Looking at conditional planning, we see that almost all actions in the real world have a variety of possible outcomes besides the expected outcome. The number of possible conditions that must be planned for grows exponentially with the number of steps in the plan. Given that only one set of conditions will actually occur, this seems rather wasteful as well as impractical. Many of the events being planned for have only an infinitesimal chance of occurring.

Looking at replanning, we see that the planner is basically assuming no action failures, and then fixing problems as they arise during execution. This too has its drawbacks. The planner may produce very "fragile" plans, which are very hard to fix if anything goes wrong. For example, the entire existence of "spare tires" is a result of conditional planning rather than replanning. If the agent does not plan for a puncture, then it will not see the need for a spare tire. Unfortunately, without a spare tire, even the most determined replanning agent might be faced with a long walk.

```

function SITUATED-PLANNING-AGENT(percept) returns an action
  static: KB, a knowledge base (includes action descriptions)
    p, a plan, initially NoPlan
    t, a counter, initially 0, indicating time
    G, a goal

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  current  $\leftarrow$  STATE-DESCRIPTION(KB, t)
  EFFECTS(START(p))  $\leftarrow$  current
  if p = NoPlan then
    G  $\leftarrow$  ASK(KB, MAKE-GOAL-QUERY(t))
    p  $\leftarrow$  MAKE-PLAN(current, G, KB)
  action  $\leftarrow$  NoOp (the default)

  Termination:
  if there are no open preconditions and p has no steps other than START and FTNISH then
    p  $\leftarrow$  NoPlan and skip remaining steps

  Resolving standard flaws:
  resolve any open condition by adding a causal link from any existing
    possibly prior step or a new step
  resolve potential threats by promotion or demotion

  Remove unsupported causal links:
  if there is a causal link START  $\xrightarrow{c}$  S protecting a proposition c
    that no longer holds in START then
      remove the link and any associated bindings

  Extend causal links back to earliest possible step:
  if there is a causal link Sj  $\xrightarrow{c}$  Sk such that
    another step Si exists with Si < Sj and the link Si  $\xrightarrow{c}$  Sk is safe then
      replace 5)  $\xrightarrow{c}$  Sk with Si  $\xrightarrow{c}$  Sk

  Remove redundant actions:
  remove any step S that supplies no causal links

  Execute actions when ready for execution:
  if a step S in the plan other than FINISH satisfies the following:
    (a) all preconditions satisfied by START;
    (b) no other steps necessarily between START and S; and
    (c) S does not threaten any causal link in p then
      add ordering constraints to force all other steps after S
      remove S from p, and all causal links to and from S
      action  $\leftarrow$  the action in S

  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t  $\leftarrow$  t + 1
  return action

```

Figure 13.17 A situated planning agent.

Conditional planning and replanning are really two extremes of a continuous spectrum. One way to construct intermediate systems is to specify disjunctive outcomes for actions where more than one outcome is reasonably likely. Then the agent can insert a sensing action to see which outcome occurred and construct a conditional plan accordingly. Other contingencies are dealt with by replanning. Although this approach has its merits, it requires the agent designer to decide which outcomes need to be considered. This also means that the decision must be made once for each action schema, rather than depending on the particular context of the action. In the case of the provision of spare tires, for example, it is clear that the decision as to which contingencies to plan for depends not just on the likelihood of occurrence—after all, punctures are quite rare—but also on the cost of an action failure. An unlikely condition needs to be taken into account if it would result in catastrophe (e.g., a puncture when driving across a remote desert). Even if a conditional plan can be constructed, it might be better to plan around the suspect action altogether (e.g., by bringing two spare tires or crossing the desert by camel).

What all this suggests is that when faced with a complex domain and incomplete and incorrect information, the agent needs a way to assess the likelihoods and costs of various outcomes. Given this information, it should construct a plan that maximizes the probability of success and minimizes costs, while ignoring contingencies that are unlikely or are easy to deal with. Part V of this book deals with these issues in depth.

## Coercion and abstraction

Although incomplete and incorrect information is the normal situation in real domains, there are techniques that still allow an agent to make quite complex, long-range plans without requiring the full apparatus of reasoning about likelihoods.

COERCION

The first method an agent can apply is **coercion**, which reduces uncertainty about the world by forcing it into a known state regardless of the initial state. A simple example is provided by the table-painting problem. Suppose that some aspects of the world are permanently inaccessible to the agent's senses—for example, it may have only a black and white camera. In this case, the agent can pick up a can of paint and paint both the chair and the table from the same can. This achieves the goal and reduces uncertainty. Furthermore, if the agent can read the label on the can, it will even know the color of the chair and table.

AGGREGATION

A second technique is **abstraction**. Although we have discussed abstraction as a tool for handling complexity (see Chapter 12), it also allows the agent to ignore details of a problem about which it may not have exact and complete knowledge. For example, if the agent is currently in London and plans to spend a week in Paris, it has a choice as to whether to plan the trip at an abstract level (fly out on Sunday, return the following Saturday) or a detailed level (take flight BA 216 and then taxi number 13471 via the Boulevard Peripherique). At the abstract level, the agent has actions such as *Fly(London,Paris)* that are reasonably certain to work. Even with delays, oversold flights, and so on, the agent will still get to Paris. At the detailed level, there is missing information (flight schedules, which taxi will turn up, Paris traffic conditions) and the possibility of unexpected situations developing that would lead to a particular flight being missed.

**Aggregation** is another useful form of abstraction for dealing with large numbers of objects. For example, in planning its cash flows, the U.S. Government assumes a certain number

of taxpayers will send in their tax returns by any given date. At the level of individual taxpayers, there is almost complete uncertainty, but at the aggregate level, the system is reliable. Similarly, in trying to pour water from one bottle into another using a funnel, it would be hopeless to plan the path of each water molecule because of uncertainty as well as complexity, yet the pouring as an aggregated action is very reliable.

The discussion of abstraction leads naturally to the issue of the connection between plans and physical actions. Our agent designs have assumed that the actions returned by the planner (which are concrete instances of the actions described in the knowledge base) are directly executable in the environment. A more sophisticated design might allow planning at a higher level of abstraction and incorporate a "motor subsystem" that can take an action from the plan and generate a sequence of primitive actions to be carried out. The subsystem might, for example, generate a speech signal from an utterance description returned by the planner; or it might generate stepper-motor commands to turn the wheels of a robot to carry out a "move" action in a motion plan. We discuss the connection between planning and motor programs in more detail in Chapter 25. Note that the subsystem might itself be a planner of some sort; its goal is to find a sequence of lower-level actions that achieve the effects of the higher-level action specified in the plan. In any case, the actions generated by the higher-level planner must be capable of execution independently of each other, because the lower-level motor system cannot allow for interactions or interleavings among the subplans that implement different actions.

## 13.5 SUMMARY

---

The world is not a tidy place. When the unexpected or unknown occurs, an agent needs to do something to get back on track. This chapter shows how conditional planning and replanning can help an agent recover.

- Standard planning algorithms assume complete and correct information. Many domains violate this assumption.
- Incomplete information can be dealt with using sensing actions to obtain the information needed. **Conditional plans** include different subplans in different contexts, depending on the information obtained.
- Incorrect information results in unsatisfied preconditions for actions and plans. **Execution monitoring** detects violations of the preconditions for successful completion of the plan. **Action monitoring** detects actions that fail.
- A simple **replanning agent** uses execution monitoring and splices in subplans as needed.
- A more comprehensive approach to plan execution involves incremental modifications to the plan, including execution of steps, as conditions in the environment evolve.
- **Abstraction** and **coercion** can overcome the uncertainty inherent in most real domains.

---

## BIBLIOGRAPHICAL AND HISTORICAL NOTES

Early planners, which lacked conditionals and loops, sometimes resorted to a coercive style in response to environmental uncertainty. Sacerdoti's NOAH used coercion in its solution to the "keys and boxes" problem, a planning challenge problem in which the planner knows little about the initial state. Mason (1993) argued that sensing often can and should be dispensed with in robotic planning, and describes a sensorless plan that can move a tool into a specific position on a table by a sequence of tilting actions *regardless* of the initial position.

WARPLAN-C (Warren, 1976), a variant of WARPLAN, was one of the earliest planners to use conditional actions. Olawski and Gini (1990) lay out the major issues involved in conditional planning. Recent systems for partial-order conditional planning include UWL (Etzioni *et al.*, 1992) and CNLP (Peot and Smith, 1992), on which CPOP is based). C-BURIDAN (Draper *et al.*, 1994) handles conditional planning for actions with probabilistic outcomes, thereby connecting to the work on Markov decision problems described in Chapter 17.

There is a close relation between conditional planning and automated program synthesis, for which there are a number of references in Chapter 10. The two fields have usually been pursued separately because of the enormous difference in typical cost between execution of machine instructions and execution of actions by robot vehicles or manipulators. Linden (1991) attempts explicit cross-fertilization between the two fields.

The earliest major treatment of execution monitoring was PLANEX (Fikes *et al.*, 1972), which worked with the STRIPS planner to control the robot Shakey. PLANEX used triangle tables to allow recovery from partial execution failure without complete replanning. Shakey's model of execution is discussed further in Chapter 25. The NASL planner (McDermott, 1978a) treated a planning problem simply as a specification for carrying out a complex action, so that execution and planning were completely unified. It used theorem proving to reason about these complex actions. IPEM (Integrated Planning, Execution, and Monitoring) (Ambros-Ingerson and Steel, 1988), which was the first system to smoothly integrate partial-order planning and planning execution, forms the basis for the discussion in this chapter.

A system that contains an explicitly represented agent function, whether implemented as a table or a set of condition-action rules, need not worry about unexpected developments in the environment. All it has to do is to execute whatever action its function recommends for the state in which it finds itself (or in the case of inaccessible environments, the percept sequence to date). The field of **reactive planning** aims to take advantage of this fact, thereby avoiding the complexities of planning in dynamic, inaccessible environments. "Universal plans" (Schoppers, 1987) were developed as a scheme for reactive planning, but turned out to be a rediscovery of the idea of **policies** in Markov decision processes. Brooks's (1986) subsumption architecture (also discussed in Chapter 25) uses a layered finite state machine to represent the agent function, and stresses the use of minimal internal state. Another important manifestation of the reactive planning paradigm is Pengi (Agre and Chapman, 1987), designed as a response to the criticism of classical AI planning in Chapman (1987). Ginsberg (1989) made a spirited attack on reactive planning, including intractability results for some formulations of the reactive planning problem. For an equally spirited response, see Schoppers (1989).

---

## EXERCISES

**13.1** Consider how one might use a planning system to play chess.

- a. Write action schemata for legal moves. Make sure to include in the state description some way to indicate whose move it is. Will basic STRIPS actions suffice?
- b. Explain how the opponent's moves can be handled by conditional steps.
- c. Explain how the planner would represent and achieve the goal of winning the game.
- d. How might we use the planner to do a finite-horizon lookahead and pick the best move, rather than planning for outright victory?
- e. How would a replanning approach to chess work? What might be an appropriate way to combine conditional planning and replanning for chess?

**13.2** Discuss the application of conditional planning and replanning techniques to the vacuum world and wumpus world.



**13.3** Represent the actions for the flat-tire domain in the appropriate format, formulate the initial and goal state descriptions, and use the POP algorithm to solve the problem.



**13.4** This exercise involves the use of POP to *actually* fix a flat tire (in simulation).



- a. Build an environment simulator for the flat-tire world. Your simulator should be able to update the state of the environment according to the actions taken by the agent. The easiest way to do this is to take the postconditions directly from the operator descriptions and use TELL and RETRACT to update a logical knowledge base representing the world state.
- b. Implement a planning agent for your environment, and show that it fixes the tire.

**13.5** In this exercise, we will add nondeterminism to the environment from Exercise 13.4.

- a. Modify your environment so that with probability 0.1, an action fails—that is, one of the effects does not occur. Show an example of a plan not working because of an action failure.
- b. Modify your planning agent to include a simple replanning capability. It should call POP to construct a repair plan to get back to the desired state along the solution path, execute the repair plan (calling itself recursively, of course, if the repair plan fails), and then continue executing the original plan from there. (You may wish to start by having failed actions do nothing at all, so that this recursive repair method automatically results in a “loop-until-success” behavior; this will probably be easier to debug!)
- c. Show that your agent can fix the tire in this new environment.

**13.6** **Softbots** construct and execute plans in software environments. One typical task for softbots is to find copies of technical reports that have been published at some other institution. Suppose that the softbot is given the task "Get me the most recent report by X on topic Y." Relevant actions include logging on to a library information system and issuing queries, using an Internet directory to find X's institution, sending email to X; connecting to X's institution by `ftp`, and so on. Write down formal representations for a representative set of actions, and discuss what sort of planning and execution algorithms would be needed.

# Part V

## UNCERTAIN KNOWLEDGE AND REASONING

Parts III and IV covered the **logical agent** approach to AI. We used first-order logic as the language to represent facts, and we showed how standard inference procedures and planning algorithms can derive new beliefs and hence identify desirable actions. In Part V, we reexamine the very foundation of the logical approach, describing how it must be changed to deal with the often unavoidable problem of uncertain information. **Probability theory** provides the basis for our treatment of systems that reason under uncertainty. Also, because actions are no longer certain to achieve goals, agents will need ways of weighing up the desirability of goals and the likelihood of achieving them. For this, we use **utility theory**. Probability theory and utility theory together constitute **decision theory**, which allows us to build rational agents for uncertain worlds.

Chapter 14 covers the basics of probability theory, including the representation language for uncertain beliefs. **Belief networks**, a powerful tool for representing and reasoning with uncertain knowledge, are described in detail in Chapter 15, along with several other formalisms for handling uncertainty. Chapter 16 develops utility theory and decision theory in some depth. Finally, Chapter 17 describes the full decision-theoretic agent design for uncertain environments, thereby generalizing the planning methods of Part IV.

# 14      UNCERTAINTY

*In which we see what an agent should do when not all is crystal clear.*

## 14.1 ACTING UNDER UNCERTAINTY



UNCERTAINTY

One problem with first-order logic, and thus with the logical-agent approach, is that *agents almost never have access to the whole truth about their environment*. Some sentences can be ascertained directly from the agent's percepts, and others can be inferred from current and previous percepts together with knowledge about the properties of the environment. In almost every case, however, even in worlds as simple as the wumpus world in Chapter 6, there will be important questions to which the agent cannot find a categorical answer. The agent must therefore act under **uncertainty**. For example, a wumpus agent often will find itself unable to discover which of two squares contains a pit. If those squares are en route to the gold, then the agent might have to take a chance and enter one of the two squares.

Uncertainty can also arise because of incompleteness and incorrectness in the agent's understanding of the properties of the environment. The **qualification problem**, mentioned in Chapter 7, says that many rules about the domain will be incomplete, because there are too many conditions to be explicitly enumerated, or because some of the conditions are unknown. Suppose, for example, that the agent wants to drive someone to the airport to catch a flight, and is considering a plan  $A_{90}$  that involves leaving home 90 minutes before the flight departs and driving at a reasonable speed. Even though the airport is only about 15 miles away, the agent will not be able to reach a definite conclusion such as "Plan  $A_{90}$  will get us to the airport in time," but rather only the weaker conclusion "Plan  $A_{90}$  will get us to the airport in time, as long as my car doesn't break down or run out of gas, and I don't get into an accident, and there are no accidents on the bridge, and the plane doesn't leave early, and there's no earthquake, . . ."<sup>1</sup> A logical

<sup>1</sup> Conditional planning can overcome uncertainty to some extent, but only if the agent's sensing actions can obtain the required information, and if there are not too many different contingencies.

agent therefore will not believe that plan  $A_{90}$  will necessarily achieve the goal, and that makes it difficult for the logical agent to conclude that plan  $A_{90}$  is the right thing to do.

Nonetheless, let us suppose that  $A_{90}$  is in fact the right thing to do. What do we mean by saying this? As we discussed in Chapter 2, we mean that out of all the possible plans that could be executed,  $A_{90}$  is expected to maximize the agent's performance measure, given the information it has about the environment. The performance measure includes getting to the airport in time for the flight, avoiding a long, unproductive wait at the airport, and avoiding speeding tickets along the way. The information the agent has cannot guarantee any of these outcomes for  $A_{90}$ , but it can provide some degree of belief that they will be achieved. Other plans, such as  $A_{120}$ , might increase the agent's belief that it will get to the airport on time, but also increase the likelihood of a long wait. *The right thing to do, the rational decision, therefore, depends on both the relative importance of various goals and the likelihood that, and degree to which, they will be achieved.* The remainder of this section sharpens up these ideas, in preparation for the development of the general theories of uncertain reasoning and rational decisions that we present in this and subsequent chapters.



## Handling uncertain knowledge

In this section, we look more closely at the nature of uncertain knowledge. We will use a simple diagnosis example to illustrate the concepts involved. Diagnosis—whether for medicine, automobile repair, or whatever—is a task that almost always involves uncertainty. If we tried to build a dental diagnosis system using first-order logic, we might propose rules such as

$$\forall p \ Symptom(p, \text{Toothache}) \Rightarrow Disease(p, \text{Cavity})$$

The problem is that this rule is wrong. Not all patients with toothaches have cavities; some of them may have gum disease, or impacted wisdom teeth, or one of several other problems:

$$\begin{aligned} \forall p \ Symptom(p, \text{Toothache}) \Rightarrow \\ Disease(p, \text{Cavity}) \vee Disease(p, \text{GumDisease}) \vee Disease(p, \text{ImpactedWisdom}) \dots \end{aligned}$$

Unfortunately, in order to make the rule true, we have to add an almost unlimited list of possible causes. We could try turning the rule into a causal rule:

$$\forall p \ Disease(p, \text{Cavity}) \Rightarrow Symptom(p, \text{Toothache})$$

But this rule is not right either; not all cavities cause pain. The only way to fix the rule is to make it logically exhaustive: to extend the left-hand side to cover all possible reasons why a cavity might or might not cause a toothache. Even then, for the purposes of diagnosis, one must also take into account the possibility that the patient may have a toothache and a cavity that are unconnected.

Trying to use first-order logic to cope with a domain like medical diagnosis thus fails for three main reasons:

- ◊ **Laziness:** It is too much work to list the complete set of antecedents or consequents needed to ensure an exceptionless rule, and too hard to use the enormous rules that result.
- 0 **Theoretical ignorance:** Medical science has no complete theory for the domain.



◊ **Practical ignorance:** Even if we know all the rules, we may be uncertain about a particular patient because all the necessary tests have not or cannot be run.

The connection between toothaches and cavities is just not a logical consequence in either direction. This is typical of the medical domain, as well as most other judgmental domains: law, business, design, automobile repair, gardening, dating, and so on. The agent's knowledge can at best provide only a **degree of belief** in the relevant sentences. Our main tool for dealing with degrees of belief will be **probability theory**, which assigns a numerical degree of belief between 0 and 1 to sentences.<sup>2</sup>

*Probability provides a way of summarizing the uncertainty that comes from our laziness and ignorance.* We may not know for sure what afflicts a particular patient, but we believe that there is, say, an 80% chance—that is, a probability of 0.8—that the patient has a cavity if he or she has a toothache. This probability could be derived from statistical data—80% of the toothache patients seen so far have had cavities—or from some general rules, or from a combination of evidence sources. The 80% summarizes those cases in which all the factors needed for a cavity to cause a toothache are present, as well as other cases in which the patient has both toothache and cavity but the two are unconnected. The missing 20% summarizes all the other possible causes of toothache that we are too lazy or ignorant to confirm or deny.

A probability of 0 for a given sentence corresponds to an unequivocal belief that the sentence is false, while a probability of 1 corresponds to an unequivocal belief that the sentence is true. Probabilities between 0 and 1 correspond to intermediate degrees of belief in the truth of the sentence. The sentence itself is *in fact* either true or false. It is important to note that a degree of belief is different from a degree of truth. A probability of 0.8 does not mean "80% true" but rather an 80% degree of belief—that is, a fairly strong expectation. If an agent assigns a probability of 0.8 to a sentence, then the agent expects that in 80% of cases that are indistinguishable from the current situation as far as the agent's knowledge goes, the sentence will turn out to be actually true. Thus, probability theory makes the same ontological commitment as logic, namely, that facts either do or do not hold in the world. Degree of truth, as opposed to degree of belief, is the subject of **fuzzy logic**, which is covered in Section 15.6.

Before we plunge into the details of probability, let us pause to consider the status of probability statements such as "The probability that the patient has a cavity is 0.8." In propositional and first-order logic, a sentence is true or false depending on the interpretation and the world; it is true just when the fact it refers to is the case. Probability statements do not have quite the same kind of semantics.<sup>3</sup> This is because the probability that an agent assigns to a proposition depends on the percepts that it has received to date. In discussing uncertain reasoning, we call this the **evidence**. For example, suppose that the agent has drawn a card from a shuffled pack. Before looking at the card, the agent might assign a probability of 1/52 to its being the ace of spades. After looking at the card, an appropriate probability for the same proposition would be 0 or 1. Thus, an assignment of probability to a proposition is analogous to saying whether or not a given logical sentence (or its negation) is entailed by the knowledge base, rather than whether or not it

<sup>2</sup> Until recently, it was thought that probability theory was too unwieldy for general use in AI, and many approximations and alternatives to probability theory were proposed. Some of these will be covered in Section 15.6.

<sup>3</sup> The *objectivist* view of probability, however, claims that probability statements *are* true or false in the same way as logical sentences. In Section 14.5, we discuss this claim further.

is true. Just as entailment status can change when more sentences are added to the knowledge base, probabilities can change when more evidence is acquired.<sup>4</sup>

All probability statements must therefore indicate the evidence with respect to which the probability is being assessed. As the agent receives new percepts, its probability assessments are updated to reflect the new evidence. Before the evidence is obtained, we talk about **prior** or **unconditional** probability; after the evidence is obtained, we talk about **posterior** or **conditional** probability. In most cases, an agent will have some evidence from its percepts, and will be interested in computing the conditional probabilities of the outcomes it cares about given the evidence it has. In some cases, it will also need to compute conditional probabilities with respect to the evidence it has plus the evidence it expects to obtain during the course of executing some sequence of actions.

## Uncertainty and rational decisions

The presence of uncertainty changes radically the way in which an agent makes decisions. A logical agent typically has a single (possibly conjunctive) goal, and executes any plan that is guaranteed to achieve it. An action can be selected or rejected on the basis of whether or not it achieves the goal, regardless of what other actions achieve. When uncertainty enters the picture, this is no longer the case. Consider again the  $A_{90}$  plan for getting to the airport. Suppose it has a 95% chance of succeeding. Does this mean it is a rational choice? Obviously, the answer is "Not necessarily." There might be other plans, such as  $A_{120}$ , with higher probabilities of success. If it is vital not to miss the flight, then it might be worth risking the longer wait at the airport. What about  $A_{1440}$ , a plan that involves leaving home 24 hours in advance? In most circumstances, this is not a good choice, because although it almost guarantees getting there on time, it involves an intolerable wait.

PREFERENCES

UTILITY THEORY

To make such choices, an agent must first have **preferences** between the different possible outcomes of the various plans. A particular outcome is a completely specified state, including such factors as whether or not the agent arrives in time, and the length of the wait at the airport. We will be using **utility theory** to represent and reason with preferences. The term **utility** is used here in the sense of "the quality of being useful," not in the sense of the electric company or water works. Utility theory says that every state has a degree of usefulness, or utility, to an agent, and that the agent will prefer states with higher utility.

The utility of a state is relative to the agent whose preferences the utility function is supposed to represent. For example, the payoff functions for games in Chapter 5 are utility functions. The utility of a state in which White has won a game of chess is obviously high for the agent playing White, but low for the agent playing Black. Or again, some players (including the authors) might be happy with a draw against the world champion, whereas other players (including the former world champion) might not. There is no accounting for taste or preferences: you might think that an agent who prefers jalapeño-bubble-gum ice cream to chocolate-chocolate-chip is odd or even misguided, but you could not say the agent is irrational.

<sup>4</sup> This is quite different from a sentence becoming true or false as the world changes. Handling a changing world using probabilities requires the same kinds of mechanisms—situations, intervals and events—as we used in Chapter 8 for logical representations.

It is also interesting that utility theory allows for altruism. It is perfectly consistent for an agent to assign high utility to a state where the agent itself suffers a concrete loss but others profit. Here, "concrete loss" must denote a reduction in "personal welfare" of the kind normally associated with altruism or selfishness—wealth, prestige, comfort, and so on—rather than a loss of utility *per se*. Therefore, utility theory is necessarily "selfish" only if one equates a preference for the welfare of others with selfishness; conversely, altruism is only inconsistent with the principle of utility maximization if one's goals do *not* include the welfare of others.

## DECISION THEORY

Preferences, as expressed by utilities, are combined with probabilities in the general theory of rational decisions called **decision theory**:

$$\text{Decision theory} = \text{probability theory} + \text{utility theory}$$

The fundamental idea of decision theory is that *an agent is rational if and only if it chooses the action that yields the highest expected utility, averaged over all the possible outcomes of the action*. This is called the principle of Maximum Expected Utility (MEU). Probabilities and utilities are therefore combined in the evaluation of an action by weighting the utility of a particular outcome by the probability that it occurs. We saw this principle in action in Chapter 5, where we examined optimal decisions in backgammon. We will see that it is in fact a completely general principle.

## Design for a decision-theoretic agent

The structure of an agent that uses decision theory to select actions is identical, at an abstract level, to that of the logical agent described in Chapter 6. Figure 14.1 shows what needs to be done. In this chapter and the next, we will concentrate on the task of computing probabilities for current states and for the various possible outcomes of actions. Chapter 16 covers utility theory in more depth, and Chapter 17 fleshes out the complete agent architecture.

```
function DT-AGENT(percept) returns an action
  static: a set probabilistic beliefs about the state of the world
    calculate updated probabilities for current state based on
      available evidence including current percept and previous action
    calculate outcome probabilities for actions,
      given action descriptions and probabilities of current states
    select action with highest expected utility
      given probabilities of outcomes and utility information
    return action
```

**Figure 14.1** A decision-theoretic agent that selects rational actions. The steps will be fleshed out in the next four chapters.

## 14.2 BASIC PROBABILITY NOTATION

Now that we have set up the general framework for a rational agent, we will need a formal language for representing and reasoning with uncertain knowledge. Any notation for describing degrees of belief must be able to deal with two main issues: the nature of the sentences to which degrees of belief are assigned, and the dependence of the degree of belief on the agent's state of knowledge. The version of probability theory we present uses an extension of propositional logic for its sentences. The dependence on experience is reflected in the syntactic distinction between prior probability statements, which apply before any evidence is obtained, and conditional probability statements, which include the evidence explicitly.

### Prior probability

UNCONDITIONAL  
PRIOR PROBABILITY

We will use the notation  $P(A)$  for the **unconditional or prior probability** that the proposition  $A$  is true. For example, if  $Cavity$  denotes the proposition that a particular patient has a cavity,

$$P(Cavity) = 0.1$$

means that *in the absence of any other information*, the agent will assign a probability of 0.1 (a 10% chance) to the event of the patient's having a cavity. It is important to remember that  $P(A)$  can only be used when there is no other information. As soon as some new information  $B$  is known, we have to reason with the conditional probability of  $A$  given  $B$  instead of  $P(A)$ . Conditional probabilities are covered in the next section.

RANDOM VARIABLES

The proposition that is the subject of a probability statement can be represented by a proposition symbol, as in the  $P(A)$  example. Propositions can also include equalities involving so-called **random variables**. For example, if we are concerned about the random variable  $Weather$ , we might have

$$\begin{aligned} P(Weather = Sunny) &= 0.7 \\ P(Weather = Rain) &= 0.2 \\ P(Weather = Cloudy) &= 0.08 \\ P(Weather = Snow) &= 0.02 \end{aligned}$$

DOMAIN

Each random variable  $X$  has a **domain** of possible values  $\langle x_1, \dots, x_n \rangle$  that it can take on.<sup>5</sup> We will usually deal with discrete sets of values, although continuous random variables will be discussed briefly in Chapter 15. We can view proposition symbols as random variables as well, if we assume that they have a domain  $\langle true, false \rangle$ . Thus, the expression  $P(Cavity)$  can be viewed as shorthand for  $P(Cavity = true)$ . Similarly,  $P(\neg Cavity)$  is shorthand for  $P(Cavity = false)$ . Usually, we will use the letters  $A$ ,  $B$ , and so on for Boolean random variables, and the letters  $X$ ,  $Y$ , and so on for multivalued variables.

Sometimes, we will want to talk about the probabilities of all the possible values of a random variable. In this case, we will use an expression such as  $\mathbf{P}(Weather)$ , which denotes a

<sup>5</sup> In probability, the variables are capitalized, while the values are lowercase. This is unfortunately the reverse of logical notation, but it is the tradition.

*vector* of values for the probabilities of each individual state of the weather. Given the preceding values, for example, we would write

$$\mathbf{P}(\text{Weather}) = \langle 0.7, 0.2, 0.08, 0.02 \rangle$$

PROBABILITY DISTRIBUTION

This statement defines a **probability distribution** for the random variable *Weather*.

We will also use expressions such as  $\mathbf{P}(\text{Weather}, \text{Cavity})$  to denote the probabilities of all combinations of the values of a set of random variables. In this case,  $\mathbf{P}(\text{Weather}, \text{Cavity})$  denotes a  $4 \times 2$  table of probabilities. We will see that this notation simplifies many equations.

We can also use logical connectives to make more complex sentences and assign probabilities to them. For example,

$$P(\text{Cavity} \wedge \neg \text{Insured}) = 0.06$$

says there is an 6% chance that a patient has a cavity and has no insurance.

CONDITIONAL  
POSTERIOR

## Conditional probability

Once the agent has obtained some evidence concerning the previously unknown propositions making up the domain, prior probabilities are no longer applicable. Instead, we use **conditional or posterior** probabilities, with the notation  $P(A|B)$ . This is read as "the probability of *A* given that *all we know is B*." For example,

$$P(\text{Cavity}|\text{Toothache}) = 0.8$$

indicates that if a patient is observed to have a toothache, and no other information is yet available, then the probability of the patient having a cavity will be 0.8. It is important to remember that  $P(A|B)$  can only be used when all we know is *B*. As soon as we know *C*, then we must compute  $P(A|B \wedge C)$  instead of  $P(A|B)$ . A prior probability  $P(A)$  can be thought of as a special case of conditional probability  $P(A|)$ , where the probability is conditioned on no evidence.

We can also use the P notation with conditional probabilities.  $P(X|Y)$  is a two-dimensional table giving the values of  $P(X=x_i|Y=y_j)$  for each possible  $i, j$ . Conditional probabilities can be defined in terms of unconditional probabilities. The equation

$$P(A|B) = \frac{P(A \wedge B)}{P(B)} \tag{14.1}$$

holds whenever  $P(B) > 0$ . This equation can also be written as

$$P(A \wedge B) = P(A|B)P(B)$$

PRODUCT RULE

which is called the **product rule**. The product rule is perhaps easier to remember: it comes from the fact that for *A* and *B* to be true, we need *B* to be true, and then *A* to be true given *B*. We can also have it the other way around:

$$P(A \wedge B) = P(B|A)P(A)$$

In some cases, it is easier to reason in terms of prior probabilities of conjunctions, but for the most part, we will use conditional probabilities as our vehicle for probabilistic inference.

We can also extend our P notation to handle equations like these, providing a welcome degree of conciseness. For example, we might write

$$\mathbf{P}(X, Y) = \mathbf{P}(X|Y)\mathbf{P}(Y)$$

which denotes a set of equations relating the corresponding individual entries in the tables (*not* a matrix multiplication of the tables). Thus, one of the equations might be

$$P(X = x_1 \wedge Y = y_2) = P(X = x_1 | Y = y_2)P(Y = y_2)$$

In general, if we are interested in the probability of a proposition  $A$ , and we have accumulated evidence  $B$ , then the quantity we must calculate is  $P(A|B)$ . Sometimes we will not have this conditional probability available directly in the knowledge base, and we must resort to probabilistic inference, which we describe in later sections.

As we have already said, probabilistic inference does not work like logical inference. It is tempting to interpret the statement  $P(A|B) = 0.8$  to mean "whenever  $B$  is true, conclude that  $P(A)$  is 0.8." This is wrong on two counts: first,  $P(A)$  always denotes the prior probability of  $A$ , not the posterior given some evidence; second, the statement  $P(A|B) = 0.8$  is only applicable when  $B$  is the only available evidence. When additional information  $C$  is available, we must calculate  $P(A|B \wedge C)$ , which may bear little relation to  $P(A|B)$ . In the extreme case,  $C$  might tell us directly whether  $A$  is true or false. If we examine a patient who complains of toothache, and discover a cavity, then we have additional evidence *Cavity*, and we conclude (trivially) that  $P(\text{Cavity}|\text{Toothache} \wedge \text{Cavity}) = 1.0$ .

## 14.3 THE AXIOMS OF PROBABILITY

In order to define properly the semantics of statements in probability theory, we will need to describe how probabilities and logical connectives interact. We take as given the properties of the connectives themselves, as defined in Chapter 6. As for probabilities, it is normal to use a small set of axioms that constrain the probability assignments that an agent can make to a set of propositions. The following axioms are in fact sufficient:

1. All probabilities are between 0 and 1.  

$$0 < P(A) < 1$$
2. Necessarily true (i.e., valid) propositions have probability 1, and necessarily false (i.e., unsatisfiable) propositions have probability 0.  

$$P(\text{True}) = 1 \quad P(\text{False}) = 0$$
3. The probability of a disjunction is given by  

$$P(A \vee B) = P(A) + P(B) - P(A \wedge B)$$

The first two axioms serve to define the probability scale. The third is best remembered by reference to the Venn diagram shown in Figure 14.2. The figure depicts each proposition as a set, which can be thought of as the set of all possible worlds in which the proposition is true. The total probability of  $A \vee B$  is seen to be the sum of the probabilities assigned to  $A$  and  $B$ , but with  $P(A \wedge B)$  subtracted out so that those cases are not counted twice.

From these three axioms, we can derive all other properties of probabilities. For example, if we let  $B$  be  $\neg A$  in the last axiom, we obtain an expression for the probability of the negation of

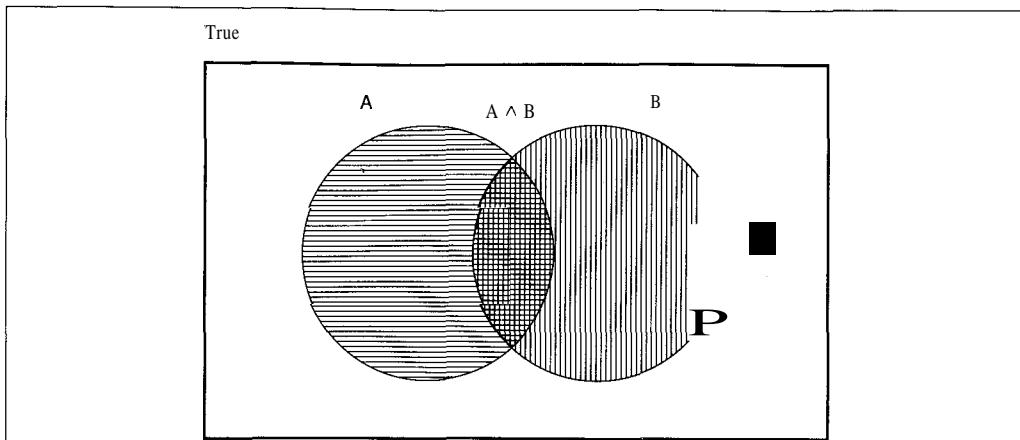


Figure 14.2 A Venn diagram showing the propositions  $A$ ,  $B$ ,  $A \vee B$  (the union of  $A$  and  $B$ ), and  $A \wedge B$  (the intersection of  $A$  and  $B$ ) as sets of possible worlds.

a proposition in terms of the probability of the proposition itself:

$$\begin{aligned}
 P(A \vee \neg A) &= P(A) + P(\neg A) - P(A \wedge \neg A) && \text{(by 3 with } B = \neg A\text{)} \\
 P(\text{True}) &= P(A) + P(\neg A) - P(\text{False}) && \text{(by logical equivalence)} \\
 1 &= P(A) + P(\neg A) && \text{(by 2)} \\
 P(\neg A) &= 1 - P(A) && \text{(by algebra)}
 \end{aligned}$$

### Why the axioms of probability are reasonable

The axioms of probability can be seen as restricting the set of probabilistic beliefs that an agent can hold. This is somewhat analogous to the logical case, where a logical agent cannot simultaneously believe  $A$ ,  $B$ , and  $\neg(A \wedge B)$ , for example. There is, however, an additional complication. In the logical case, the semantic definition of conjunction means that at least one of the three beliefs just mentioned *must be false in the world*, so it is unreasonable for an agent to believe all three. With probabilities, on the other hand, statements refer not to the world directly, but to the agent's own state of knowledge. Why, then, can an agent not hold the following set of beliefs, given that these probability assignments clearly violate the third axiom?

$$\begin{aligned}
 P(A) &= 0.4 \\
 P(B) &= 0.3 \\
 P(A \wedge B) &= 0.0 \\
 P(A \vee B) &= 0.8
 \end{aligned} \tag{14.2}$$

This kind of question has been the subject of decades of intense debate between those who advocate the use of probabilities as the only legitimate form for degrees of belief, and those who advocate alternative approaches. Here, we give one argument for the axioms of probability, first stated in 1931 by Bruno de Finetti.

The key to de Finetti's argument is the connection between degree of belief and actions. The idea is that if an agent has some degree of belief in a proposition,  $A$ , then the agent should be able to state odds at which it is indifferent to a bet for or against  $A$ . Think of it as a game between two agents: Agent 1 states "my degree of belief in event  $A$  is 0.4." Agent 2 is then free to choose whether to bet for or against  $A$ , at stakes that are consistent with the stated degree of belief. That is, Agent 2 could choose to bet that  $A$  will occur, betting \$4 against Agent 1's \$6. Or Agent 2 could bet \$6 against \$4 that  $A$  will not occur.<sup>6</sup> If an agent's degrees of belief do not accurately reflect the world, then you would expect it would tend to lose money over the long run, depending on the skill of the opposing agent.



But de Finetti proved something much stronger: *if Agent 1 expresses a set of degrees of belief that violate the axioms of probability theory then there is a betting strategy for Agent 2 that guarantees that Agent 1 will lose money*. So if you accept the idea that an agent should be willing to "put its money where its probabilities are," then you should accept that it is irrational to have beliefs that violate the axioms of probability.

One might think that this betting game is rather contrived. For example, what if one refuses to bet? Does that scupper the whole argument? The answer is that the betting game is an abstract model for the decision-making situation in which every agent is *unavoidably* involved at every moment. Every action (including inaction) is a kind of bet, and every outcome can be seen as a payoff of the bet. One can no more refuse to bet than one can refuse to allow time to pass.

We will not provide the proof of de Finetti's theorem (see Exercise 14.15), but we will show an example. Suppose that Agent 1 has the set of degrees of belief from Equation (14.2). If Agent 2 chooses to bet \$4 on  $A$ , \$3 on  $B$ , and \$2 on  $\neg(A \vee B)$ , then Figure 14.3 shows that Agent 1 always loses money, regardless of the outcomes for  $A$  and  $B$ .

Agent 1		Agent 2		Outcome for Agent 1			
Proposition	Belief	Bet	Stakes	$A \wedge B$	$A \wedge \neg B$	$\neg A \wedge S$	$\neg A \wedge \neg B$
$A$	0.4	$A$	4 to 6	-6	-6	4	4
$B$	0.3	$B$	3 to 7	-7	3	-7	3
$A \vee B$	0.8	$\neg(A \vee B)$	2 to 8	2	2	2	-8
				-11	-1	-1	-1

Figure 14.3 Because Agent 1 has inconsistent beliefs, Agent 2 is able to devise a set of bets that guarantees a loss for Agent 1, no matter what the outcome of  $A$  and  $B$ .

Other strong philosophical arguments have been put forward for the use of probabilities, most notably those of Cox (1946) and Carnap (1950). The world being the way it is, however, practical demonstrations sometimes speak louder than proofs. The success of reasoning systems based on probability theory has been much more effective in making converts. We now look at how the axioms can be deployed to make inferences.

<sup>6</sup> One might argue that the agent's preferences for different bank balances are such that the possibility of losing \$1 is not counterbalanced by an equal possibility of winning \$1. We can make the bet amounts small enough to avoid this problem, or we can use the more sophisticated treatment due to Savage (1954) to circumvent this issue altogether.

JOINT PROBABILITY DISTRIBUTION

ATOMIC EVENT

## The joint probability distribution

In this section, we define the **joint probability distribution** (or "joint" for short), which completely specifies an agent's probability assignments to all propositions in the domain (both simple and complex).

A probabilistic model of a domain consists of a set of random variables that can take on particular values with certain probabilities. Let the variables be  $X_1 \dots X_n$ . An **atomic event** is an assignment of particular values to all the variables—in other words, a complete specification of the state of the domain.

The joint probability distribution  $\mathbf{P}(X_1, \dots, X_n)$  assigns probabilities to all possible atomic events. Recall that  $\mathbf{P}(X_i)$  is a one-dimensional vector of probabilities for the possible values of the variable  $X_i$ . Then the joint is an  $n$ -dimensional table with a value in every cell giving the probability of that specific state occurring. Here is a joint probability distribution for the trivial medical domain consisting of the two Boolean variables *Toothache* and *Cavity*:

	<i>Toothache</i>	$\neg$ <i>Toothache</i>
<i>Cavity</i>	0.04	0.06
$\neg$ <i>Cavity</i>	0.01	0.89

Because the atomic events are mutually exclusive, any conjunction of atomic events is necessarily false. Because they are collectively exhaustive, their disjunction is necessarily true. Hence, from the second and third axioms of probability, the entries in the table sum to 1. In the same way, the joint probability distribution can be used to compute any probabilistic statement we care to know about the domain, by expressing the statement as a disjunction of atomic events and adding up their probabilities. Adding across a row or column gives the unconditional probability of a variable, for example,  $P(\text{Cavity}) = 0.06 + 0.04 = 0.10$ . As another example:

$$P(\text{Cavity} \vee \text{Toothache}) = 0.04 + 0.01 + 0.06 = 0.11$$

Recall that we can make inferences about the probabilities of an unknown proposition  $A$ , given evidence  $B$ , by calculating  $P(A|B)$ . A query to a probabilistic reasoning system will therefore ask for the value of a particular conditional probability. Conditional probabilities can be found from the joint using Equation (14.1):

$$P(\text{Cavity}|\text{Toothache}) = \frac{P(\text{Cavity} \wedge \text{Toothache})}{P(\text{Toothache})} = \frac{0.04}{0.04 + 0.01} = 0.80$$

Of course, in a realistic problem, there might be hundreds or thousands of random variables to consider, not just two. In general it is not practical to define all the  $2^n$  entries for the joint probability distribution over  $n$  Boolean variables, but it is important to remember that if we could define all the numbers, then we could read off any probability we were interested in.

Modern probabilistic reasoning systems sidestep the joint and work directly with conditional probabilities, which are after all the values that we are interested in. In the next section, we introduce a basic tool for this task.

## 14.4 BAYES' RULE AND ITS USE

Recall the two forms of the product rule:

$$P(A \wedge B) = P(A|B)P(B) *$$

$$P(A \wedge B) = P(B|A)P(A)$$

Equating the two right-hand sides and dividing by  $P(A)$ , we get

$$P(B|A) = \frac{P(A|B)P(B)}{P(A)} \quad (14.3)$$

### BAYES' RULE

This equation is known as **Bayes' rule** (also Bayes' law or Bayes' theorem).<sup>7</sup> This simple equation underlies all modern AI systems for probabilistic inference. The more general case of multivalued variables can be written using the P notation as follows:

$$P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)}$$

where again this is to be taken as representing a set of equations relating corresponding elements of the tables. We will also have occasion to use a more general version conditionalized on some background evidence  $E$ :

$$P(Y|X, E) = \frac{P(X|Y, E)P(Y|E)}{P(X|E)} \quad (14.4)$$

The proof of this form is left as an exercise.

### Applying Bayes' rule: The simple case

On the surface, Bayes' rule does not seem very useful. It requires three terms—a conditional probability and two unconditional probabilities—just to compute one conditional probability.

Bayes' rule is useful in practice because there are many cases where we do have good probability estimates for these three numbers and need to compute the fourth. In a task such as medical diagnosis, we often have conditional probabilities on causal relationships and want to derive a diagnosis. A doctor knows that the disease meningitis causes the patient to have a stiff neck, say, 50% of the time. The doctor also knows some unconditional facts: the prior probability of a patient having meningitis is 1/50,000, and the prior probability of any patient having a stiff neck is 1/20. Letting  $S$  be the proposition that the patient has a stiff neck and  $M$  be the proposition that the patient has meningitis, we have

$$P(S|M) = 0.5$$

$$P(M) = 1/50000$$

$$P(S) = 1/20$$

$$P(M|S) = \frac{P(S|M)P(M)}{P(S)} = \frac{0.5 \times 1/50000}{1/20} = 0.0002$$

<sup>7</sup> According to rule 1 on page 1 of Strunk and White's *The Elements of Style*, it should be Bayes's rather than Bayes'. The latter is, however, more commonly used.

That is, we expect only one in 5000 patients with a stiff neck to have meningitis. Notice that even though a stiff neck is strongly indicated by meningitis (probability 0.5), the probability of meningitis in the patient remains small. This is because the prior on stiff necks is much higher than that for meningitis.

 One obvious question to ask is why one might have available the conditional probability in one direction but not the other. In the meningitis case, perhaps the doctor knows that 1 out of 5000 patients with stiff necks has meningitis, and therefore has no need to use Bayes' rule. Unfortunately, *diagnostic knowledge is often more tenuous than causal knowledge*. If there is a sudden epidemic of meningitis, the unconditional probability of meningitis,  $P(M)$ , will go up. The doctor who derived  $P(M|S)$  from statistical observation of patients before the epidemic will have no idea how to update the value, but the doctor who computes  $P(M|S)$  from the other three values will see that  $P(M|S)$  should go up proportionately to  $P(M)$ . Most importantly, the causal information  $P(S|M)$  is *unaffected* by the epidemic, because it simply reflects the way meningitis works. The use of this kind of direct causal or model-based knowledge provides the crucial robustness needed to make probabilistic systems feasible in the real world.

## Normalization

Consider again the equation for calculating the probability of meningitis given a stiff neck:

$$P(M|S) = \frac{P(S|M)P(M)}{P(S)}$$

Suppose we are also concerned with the possibility that the patient is suffering from whiplash  $W$  given a stiff neck:

$$P(W|S) = \frac{P(S|W)P(W)}{P(S)}$$

Comparing these two equations, we see that in order to compute the **relative likelihood** of meningitis and whiplash, given a stiff neck, we need not assess the prior probability  $P(S)$  of a stiff neck. To put numbers on the equations, suppose that  $P(S|W) = 0.8$  and  $P(W) = 1/1000$ . Then

$$\begin{aligned} P(M|S) &= P(S|M)P(M) = 0.5 \times 1/50000 = \frac{1}{100000} \\ P(W|S) &= P(S|W)P(W) = 0.8 \times 1/1000 = \frac{80}{100000} \end{aligned}$$

That is, whiplash is 80 times more likely than meningitis, given a stiff neck.

In some cases, relative likelihood is sufficient for decision making, but when, as in this case, the two possibilities yield radically different utilities for various treatment actions, one needs exact values in order to make rational decisions. It is still possible to avoid direct assessment of the prior probability of the "symptoms," by considering an exhaustive set of cases. For example, we can write equations for  $M$  and for  $\neg M$ :

$$P(M|S) = \frac{P(S|M)P(M)}{P(S)}$$

$$P(\neg M|S) = \frac{P(S|\neg M)P(\neg M)}{P(S)}$$

Adding these two equations, and using the fact that  $P(M|S) + P(\neg M|S) = 1$ , we obtain

$$P(S) = P(S|M)P(M) + P(S|\neg M)P(\neg M)$$

Substituting into the equation for  $P(M|S)$ , we have

$$P(M|S) = \frac{P(S|M)P(M)}{P(S|M)P(M) + P(S|\neg M)P(\neg M)}$$

#### NORMALIZATION

This process is called **normalization**, because it treats  $1/P(S)$  as a normalizing constant that allows the conditional terms to sum to 1. Thus, in return for assessing the conditional probability  $P(S|\neg M)$ , we can avoid assessing  $P(S)$  and still obtain exact probabilities from Bayes' rule. In the general, multivalued case, we obtain the following form for Bayes' rule:

$$\mathbf{P}(Y|X) = \alpha \mathbf{P}(X|Y)\mathbf{P}(Y)$$

where  $\alpha$  is the normalization constant needed to make the entries in the table  $\mathbf{P}(Y|X)$  sum to 1. The normal way to use normalization is to calculate the unnormalized values, and then scale them all so that they add to 1 (Exercise 14.7).

## Using Bayes' rule: Combining evidence

Suppose we have two conditional probabilities relating to cavities:

$$\begin{aligned} P(\text{Cavity}|\text{Toothache}) &= 0.8 \\ P(\text{Cavity}|\text{Catch}) &= 0.95 \end{aligned}$$

which might perhaps have been computed using Bayes' rule. What can a dentist conclude if her nasty steel probe catches in the aching tooth of a patient? If we knew the whole joint distribution, it would be easy to read off  $P(\text{Cavity}|\text{Toothache} \wedge \text{Catch})$ . Alternatively, we could use Bayes' rule to reformulate the problem:

$$P(\text{Cavity}|\text{Toothache} \wedge \text{Catch}) = \frac{P(\text{Toothache} \wedge \text{Catch}|\text{Cavity})P(\text{Cavity})}{P(\text{Toothache} \wedge \text{Catch})}$$

For this to work, we need to know the conditional probabilities of the pair *Toothache A Catch* given *Cavity*. Although it seems feasible to estimate conditional probabilities (given *Cavity*) for  $n$  different individual variables, it is a daunting task to come up with numbers for  $n^2$  pairs of variables. To make matters worse, a diagnosis may depend on dozens of variables, not just two. That means we need an exponential number of probability values to complete the diagnosis—we might as well go back to using the joint. This is what first led researchers away from probability theory toward approximate methods for evidence combination that, while giving incorrect answers, require fewer numbers to give any answer at all.

In many domains, however, the application of Bayes' rule can be simplified to a form that requires fewer probabilities in order to produce a result. The first step is to take a slightly different view of the process of incorporating multiple pieces of evidence. The process of **Bayesian updating** incorporates evidence one piece at a time, modifying the previously held belief in the unknown variable. Beginning with *Toothache*, we have (writing Bayes' rule in such a way as to reveal the updating process):

$$P(\text{Cavity}|\text{Toothache}) = P(\text{Cavity}) \frac{P(\text{Toothache}|\text{Cavity})}{P(\text{Toothache})}$$

When *Catch* is observed, we can apply Bayes' rule with *Toothache* as the constant conditioning context (see Exercise 14.5):

$$\begin{aligned} P(\text{Cavity}|\text{Toothache} \wedge \text{Catch}) &= P(\text{Cavity}|\text{Toothache}) \cdot \frac{P(\text{Catch}|\text{Toothache A Cavity})}{P(\text{Catch}|\text{Toothache})} \\ &= P(\text{Cavity}) \cdot \frac{P(\text{Toothache}|\text{Cavity})P(\text{Catch}|\text{Toothache A Cavity})}{P(\text{Toothache})P(\text{Catch}|\text{Toothache})} \end{aligned}$$

Thus, in Bayesian updating, as each new piece of evidence is observed, the belief in the unknown variable is multiplied by a factor that depends on the new evidence. Exercise 14.8 asks you to prove that this process is order-independent, as we would hope.

So far we are not out of the woods, because the multiplication factor depends not just on the new evidence, but also on the evidence already obtained. Finding a value for the numerator,  $P(\text{Catch}|\text{Toothache A Cavity})$ , is not necessarily any easier than finding a value for  $P(\text{Toothache A Catch}|\text{Cavity})$ . We will need to make a substantive assumption in order to simplify our expressions. The key observation, in the cavity case, is that the cavity is the *direct cause* of both the toothache and the probe catching in the tooth. Once we know the patient has a cavity, we do not expect the probability of the probe catching to depend on the presence of a toothache; similarly, the probe catching is not going to change the probability that the cavity is causing a toothache. Mathematically, these properties are written as

$$\begin{aligned} P(\text{Catch}|\text{Cavity} \wedge \text{Toothache}) &= P(\text{Catch}|\text{Cavity}) \\ P(\text{Toothache}|\text{Cavity} \wedge \text{Catch}) &= P(\text{Toothache}|\text{Cavity}) \end{aligned}$$

CONDITIONAL  
INDEPENDENCE

These equations express the **conditional independence** of *Toothache* and *Catch* given *Cavity*. Given conditional independence, we can simplify the equation for updating:

$$P(\text{Cavity}|\text{Toothache} \wedge \text{Catch}) = P(\text{Cavity}) \cdot \frac{P(\text{Toothache}|\text{Cavity})}{P(\text{Toothache})} \cdot \frac{P(\text{Catch}|\text{Cavity})}{P(\text{Catch}|\text{Toothache})}$$

There is still the term  $P(\text{Catch}|\text{Toothache})$ , which might seem to involve considering all pairs (triples, etc.) of symptoms, but in fact this term goes away. Notice that the product of the denominators is  $P(\text{Catch}|\text{Toothache})P(\text{Toothache})$ , or  $P(\text{Toothache A Catch})$ . We can eliminate this term by normalization, as before, provided we also assess  $P(\text{Toothache}|\neg\text{Cavity})$  and  $P(\text{Catch}|\neg\text{Cavity})$ . Thus, we are back where we were with a single piece of evidence: we just need to evaluate the prior for the cause, and the conditional probabilities of each of its effects.

We can also use conditional independence in the multivalued case. To say that  $X$  and  $Y$  are independent given  $Z$ , we write

$$\mathbf{P}(X|Y, Z) = \mathbf{P}(X|Z)$$

which represents a set of individual conditional independence statements. The corresponding simplification of Bayes' rule for multiple evidence is

$$P(Z|X, Y) = \alpha \mathbf{P}(Z)\mathbf{P}(X|Z)\mathbf{P}(Y|Z)$$

where  $\alpha$  is a normalization constant such that the entries in  $\mathbf{P}(Z|X, Y)$  sum to 1.

It is important to remember that this simplified form of Bayesian updating only works when the conditional independence relationships hold. Conditional independence information therefore is crucial to making probabilistic systems work effectively. In Chapter 15, we show how it can be represented and manipulated in a systematic fashion.

## 14.5 WHERE Do PROBABILITIES COME FROM?

FREQUENTIST

There has been endless debate over the source and status of probability numbers. The **frequentist** position is that the numbers can come only from *experiments*: if we test 100 people and find that 10 of them have a cavity, then we can say the probability of a cavity is approximately 0.1. A great deal of work has gone into making such statistical assessments reliable. The **objectivist** view is that probabilities are real aspects of the universe—propensities of objects to behave in certain ways—rather than being just descriptions of an observer's degree of belief. In this view, frequentist measurements are attempts to observe the real probability value. The **subjectivist** view describes probabilities as a way of characterizing an agent's beliefs, rather than having any external physical significance. This allows the doctor or analyst to make these numbers up, to say, "In my opinion, I expect the probability of a cavity to be about 0.1." Several more reliable techniques, such as the betting systems described earlier, have also been developed for eliciting probability assessments from humans.

In the end, even a strict frequentist position involves subjective analysis, so the difference probably has little practical importance. Consider the probability that the sun will still exist tomorrow (a question first raised by Hume's *Inquiry*). There are several ways to compute this:

- The probability is undefined, because there has never been an experiment that tested the existence of the sun *tomorrow*.
- The probability is 1, because in all the experiments that have been done (on past days) the sun has existed.
- The probability is  $1 - \epsilon$ , where  $\epsilon$  is the proportion of stars in the universe that go supernova and explode per day.
- The probability is  $(d + 1)/(d + 2)$ , where  $d$  is the number of days that the sun has existed so far. (This formula is due to Laplace.)
- The probability can be derived from the type, age, size, and temperature of the sun, even though we have never observed another star with those exact properties.

REFERENCE CLASS

The first three of these methods are frequentist, whereas the last two are subjective. But even if you prefer not to allow subjective methods, the choice of which of the first three experiments to use is a subjective choice known as the **reference class** problem. It is the same problem faced by the doctor who wants to know the chances that a patient has a particular disease. The doctor wants to consider other patients who are similar in important ways—age, symptoms, perhaps sex—and see what proportion of them had the disease. But if the doctor considered everything that is known about the patient—weight to the nearest gram, hair color, maternal grandmother's maiden name—the result would be that there are no other patients who are exactly the same, and thus no reference class from which to collect experimental data. This has been a vexing problem in the philosophy of science. Carnap (along with other philosophers) tried in vain to find a way of reducing theories to objective truth—to show how a series of experiments necessarily leads to one theory and not another. The approach we will take in the next chapter is to minimize the number of probabilities that need to be assessed, and to maximize the number of cases available for each assessment, by taking advantage of independence relationships in the domain.

## 14.6 SUMMARY

---

This chapter shows that probability is the right way to reason about uncertainty.

- Uncertainty arises because of both laziness and ignorance. It is inescapable in complex, dynamic, or inaccessible worlds.
- Uncertainty means that many of the simplifications that are possible with deductive inference are no longer valid.
- Probabilities express the agent's inability to reach a definite decision regarding the truth of a sentence, and summarize the agent's beliefs.
- Basic probability statements include **prior probabilities** and **conditional probabilities** over simple and complex propositions.
- The axioms of probability specify constraints on reasonable assignments of probabilities to propositions. An agent that violates the axioms will behave irrationally in some circumstances.
- **The joint probability distribution** specifies the probability of each complete assignment of values to random variables. It is usually far too large to create or use.
- **Bayes' rule** allows unknown probabilities to be computed from known, stable ones.
- In the general case, combining many pieces of evidence may require assessing a large number of conditional probabilities.
- **Conditional independence** brought about by direct causal relationships in the domain allows **Bayesian updating** to work effectively even with multiple pieces of evidence.

---

## BIBLIOGRAPHICAL AND HISTORICAL NOTES

Although games of chance date back at least to around 300 B.C., the mathematical analysis of odds and probability appears to be much later. Some work done by Mahaviracarya in India is dated to roughly the ninth century A.D. In Europe, the first attempts date only to the Italian Renaissance, beginning around 1500 A.D. The first significant systematic analyses were produced by Girolamo Cardano around 1565, but they remained unpublished until 1663. By that time, the discovery by Blaise Pascal (in correspondence with Pierre Fermat in 1654) of a systematic way of calculating probabilities had for the first time established probability as a widely and fruitfully studied mathematical discipline. The first published textbook on probability was *De Ratiociniis in Ludo Aleae* (Huygens, 1657). Pascal also introduced conditional probability, which is covered in Huygens's textbook. The Rev. Thomas Bayes (1702-1761) introduced the rule for reasoning about conditional probabilities that was named after him. It was published posthumously (Bayes, 1763). Kolmogorov (1950, first published in German in 1933) presented probability theory in a rigorously axiomatic framework for the first time. Rényi (1970) later gave an axiomatic presentation that took conditional probability, rather than absolute probability, as primitive.

Pascal used probability in ways that required both the objective interpretation, as a property of the world based on symmetry or relative frequency, and the subjective interpretation as degree of belief: the former in his analyses of probabilities in games of chance, the latter in the famous "Pascal's wager" argument about the possible existence of God. However, Pascal did not clearly realize the distinction between these two interpretations. The distinction was first drawn clearly by James Bernoulli (1654-1705).

Leibniz introduced the "classical" notion of probability as a proportion of enumerated, equally probable cases, which was also used by Bernoulli, although it was brought to prominence by Laplace (1749-1827). This notion is ambiguous between the frequency interpretation and the subjective interpretation. The cases can be thought to be equally probable either because of a natural, physical symmetry between them, or simply because we do not have any knowledge that would lead us to consider one more probable than another. The use of this latter, subjective consideration to justify assigning equal probabilities is known as the *principle of indifference* (Keynes, 1921).

The debate between objective and subjective interpretations of probability became sharper in the twentieth century. Kolmogorov (1963), R. A. Fisher (1922), and Richard von Mises(1928) were advocates of the relative frequency interpretation. Karl Popper's (1959, first published in German in 1934) "propensity" interpretation traces relative frequencies to an underlying physical symmetry. Frank Ramsey (1931), Bruno de Finetti (1937), R. T. Cox (1946), Leonard Savage (1954), and Richard Jeffrey (1983) interpreted probabilities as the degrees of belief of specific individuals. Their analyses of degree of belief were closely tied to utilities and to behavior, specifically to the willingness to place bets. Rudolf Carnap, following Leibniz and Laplace, offered a different kind of subjective interpretation of probability: not as any actual individual's degree of belief, but as the degree of belief that an idealized individual *should* have in a particular proposition  $p$  given a particular body of evidence  $E$ . Carnap attempted to go further than Leibniz or Laplace by making this notion of degree of **confirmation** mathematically precise, as a logical relation between  $p$  and  $E$ . The study of this relation was intended to constitute a mathematical discipline called **inductive logic**, analogous to ordinary deductive logic (Carnap, 1948; Carnap, 1950). Carnap was not able to extend his inductive logic much beyond the propositional case, and Putnam (1963) showed that some fundamental difficulties would prevent a strict extension to languages capable of expressing arithmetic.

CONFIRMATION

INDUCTIVE LOGIC

The question of reference classes is closely tied to the attempt to find an inductive logic. The approach of choosing the "most specific" reference class of sufficient size was formally proposed by Reichenbach (1949). Various attempts have been made to formulate more sophisticated policies in order to avoid some obvious fallacies that arise with Reichenbach's rule, notably by Henry Kyburg (1977; 1983), but such approaches remain somewhat *ad hoc*. More recent work by Bacchus, Grove, Halpern, and Koller (1992) extends Carnap's methods to first-order theories on finite domains, thereby avoiding many of the difficulties associated with reference classes.

Bayesian probabilistic reasoning has been used in AI since the 1960s, especially in medical diagnosis. It was used not only to make a diagnosis from available evidence, but also to select further questions and tests when available evidence was inconclusive (Gorry, 1968; Gorry *et al.*, 1973), using the theory of information value (Section 16.6). One system outperformed human experts in the diagnosis of acute abdominal illnesses (de Dombal *et al.*, 1974). These early Bayesian systems suffered from a number of problems, however. Because they lacked any

theoretical model of the conditions they were diagnosing, they were vulnerable to unrepresentative data occurring in situations for which only a small sample was available (de Dombal *et al.*, 1981). Even more fundamentally, because they lacked a concise formalism (such as the one to be described in Chapter 15) for representing and using conditional independence information, they depended on the acquisition, storage, and processing of enormous amounts of probabilistic data. De Dombal's system, for example, was built by gathering and analyzing enough clinical cases to provide meaningful data for every entry in a large joint probability table. Because of these difficulties, probabilistic methods for coping with uncertainty fell out of favor in AI from the 1970s to the mid-1980s. In Chapter 15, we will examine the alternative approaches that were taken and the reason for the resurgence of probabilistic methods in the late 1980s.

There are many good introductory textbooks on probability theory, including those by Chung (1979) and Ross (1988). Morris DeGroot (1989) offers a combined introduction to probability and statistics from a Bayesian standpoint, as well as a more advanced text (1970). Richard Hamming's (1991) textbook gives a mathematically sophisticated introduction to probability theory from the standpoint of a propensity interpretation based on physical symmetry. Hacking (1975) and Hald (1990) cover the early history of the concept of probability.

---

## EXERCISES

**14.1** Show from first principles that

$$P(A|B \wedge A) = 1$$

14.2 Consider the domain of dealing five-card poker hands from a standard deck of 52 cards, under the assumption that the dealer is fair.

- a. How many atomic events are there in the joint probability distribution (i.e., how many five-card hands are there)?
- b. What is the probability of each atomic event?
- c. What is the probability of being dealt a royal straight flush (the ace, king, queen, jack and ten of the same suit)?
- d. What is the probability of four of a kind?

14.3 After your yearly checkup, the doctor has bad news and good news. The bad news is that you tested positive for a serious disease, and that the test is 99% accurate (i.e., the probability of testing positive given that you have the disease is 0.99, as is the probability of testing negative given that you don't have the disease). The good news is that this is a rare disease, striking only one in 10,000 people. Why is it good news that the disease is rare? What are the chances that you actually have the disease?

14.4 Would it be rational for an agent to hold the three beliefs  $P(A) = 0.4$ ,  $P(B) = 0.3$ , and  $P(A \vee B) = 0.5$ ? If so, what range of probabilities would be rational for the agent to hold for  $A \wedge B$ ? Make up a table like the one in Figure 14.3 and show how it supports your argument

about rationality. Then draw another version of the table where  $P(A \vee B) = 0.7$ . Explain why it is rational to have this probability, even though the table shows one case that is a loss and three that just break even. (*Hint:* what is Agent 1 committed to about the probability of each of the four cases, especially the case that is a loss?)

**14.5** It is quite often useful to consider the effect of some specific propositions in the context of some general background evidence that remains fixed, rather than in the complete absence of information. The following questions ask you to prove more general versions of the product rule and Bayes' rule, with respect to some background evidence  $E$ :

- Prove the conditionalized version of the general product rule:

$$P(A, B|E) = P(A|B, E)P(B|E)$$

- Prove the conditionalized version of Bayes' rule:

$$P(A|B, C) = \frac{P(B|A, C)P(A|C)}{P(B|C)}$$

**14.6** Show that the statement

$$P(A, B|C) = P(A|C)P(B|C)$$

is equivalent to the statement

$$P(A|B, C) = P(A|C)$$

and also to

$$P(B|A, C) = P(B|C)$$

**14.7** In this exercise, you will complete the normalization calculation for the meningitis example. First, make up a suitable value for  $P(S|\neg M)$ , and use it to calculate unnormalized values for  $P(M|S)$  and  $P(\neg M|S)$  (i.e., ignoring the  $P(S)$  term in the Bayes' rule expression). Now normalize these values so that they add to 1.

**14.8** Show that the degree of belief after applying the Bayesian updating process is independent of the order in which the pieces of evidence arrive. That is, show that  $P(A|B, C) = P(A|C, B)$  using the Bayesian updating rule.

**14.9** This exercise investigates the way in which conditional independence relationships affect the amount of information needed for probabilistic calculations.

- Suppose we wish to calculate  $P(H|E_1, E_2)$ , and we have no conditional independence information. Which of the following sets of numbers are sufficient for the calculation?

- (i)  $P(E_1, E_2), P(H), P(E_1|H), P(E_2|H)$
- (ii)  $P(E_1, E_2), P(H), P(E_1, E_2|H)$
- (iii)  $P(H), P(E_1|H), P(E_2|H)$

- Suppose we know that  $P(E_1|H, E_2) = P(E_1|H)$  for all values of  $H, E_1, E_2$ . Now which of the above three sets are sufficient?

**14.10** Express the statement that  $X$  and  $Y$  are conditionally independent given  $Z$  as a constraint on the joint distribution entries for  $P(X, Y, Z)$ .

**14.11** (Adapted from Pearl (1988).) You are a witness of a night-time hit-and-run accident involving a taxi in Athens. All taxis in Athens are blue or green. You swear, under oath, that the taxi was blue. Extensive testing shows that under the dim lighting conditions, discrimination between blue and green is 75% reliable. Is it possible to calculate the most likely color for the taxi? (Hint: distinguish carefully between the proposition that the taxi *is* blue and the proposition that it *appears* blue.)

What now, given that 9 out of 10 Athenian taxis are green?

**14.12** (Adapted from Pearl (1988).) Three prisoners, A, B, and C, are locked in their cells. It is common knowledge that one of them will be executed the next day and the others pardoned. Only the governor knows which one will be executed. Prisoner A asks the guard a favor: "Please ask the governor who will be executed, and then take a message to one of my friends B and C to let him know that he will be pardoned in the morning." The guard agrees, and comes back later and tells A that he gave the pardon message to B.

What are As chances of being executed, given this information? (Answer this *mathematically*, not by energetic waving of hands.)

**14.13** This exercise concerns Bayesian updating in the meningitis example. Starting with a patient about whom we know nothing, show how the probability of having meningitis,  $P(M)$ , is updated after we find the patient has a stiff neck. Next, show how  $P(M)$  is updated again when we find the patient has a fever. (Say what probabilities you need to compute this, and make up values for them.)

**14.14** In previous chapters, we found the technique of **reification** useful in creating representations in first-order logic. For example, we handled change by reifying situations, and belief by reifying sentences. Suppose we try to do this for uncertain reasoning by reifying probabilities, thus embedding probability entirely *within* first-order logic. Which of the following are true?

- This would not work.
- This would work fine; in fact, it is just another way of describing probability theory.
- This would work fine; it would be an alternative to probability theory.

**14.15** Prove that the three axioms of probability are necessary for rational behavior in betting situations, as shown by de Finetti.

# 15

# PROBABILISTIC REASONING SYSTEMS

*In which we explain how to build reasoning systems that use network models to reason with uncertainty according to the laws of probability theory.*

Chapter 14 gave the syntax and semantics of probability theory. This chapter introduces an inference mechanism, thus giving us everything we need to build an uncertain-reasoning system.

The main advantage of probabilistic reasoning over logical reasoning is in allowing the agent to reach rational decisions even when there is not enough information to prove that any given action will work. We begin by showing how to capture uncertain knowledge in a natural and efficient way. We then show how probabilistic inference, although exponentially hard in the worst case, can be done efficiently in many practical situations. We conclude the chapter with a discussion of knowledge engineering techniques for building probabilistic reasoning systems, a case study of one successful system, and a survey of alternate approaches.

## 15.1 REPRESENTING KNOWLEDGE IN AN UNCERTAIN DOMAIN

In Chapter 14, we saw that the joint probability distribution can answer any question about the domain, but can become intractably large as the number of variables grows. Furthermore, specifying probabilities for atomic events is rather unnatural and may be very difficult unless a large amount of data is available from which to gather statistical estimates.

We also saw that, in the context of using Bayes' rule, conditional independence relationships among variables can simplify the computation of query results and greatly reduce the number of conditional probabilities that need to be specified. We use a data structure called a **belief network**<sup>1</sup> to represent the dependence between variables and to give a concise specification of the joint probability distribution. A belief network is a graph in which the following holds:

<sup>1</sup> This is the most common name, but there are many others, including **Bayesian network**, **probabilistic network**, **causal network**, and **knowledge map**. An extension of belief networks called a **decision network** or **influence diagram** will be covered in Chapter 16.

1. A set of random variables makes up the nodes of the network.
2. A set of directed links or arrows connects pairs of nodes. The intuitive meaning of an arrow from node  $X$  to node  $Y$  is that  $X$  has a *direct influence* on  $Y$ .
3. Each node has a conditional probability table that quantifies the effects that the parents have on the node. The parents of a node are all those nodes that have arrows pointing to it.
4. The graph has no directed cycles (hence is a directed, acyclic graph, or DAG).

It is usually easy for a domain expert to decide what direct conditional dependence relationships hold in the domain—much easier, in fact, than actually specifying the probabilities themselves. Once the topology of the belief network is specified, we need only specify conditional probabilities for the nodes that participate in direct dependencies, and use those to compute any other probability values.

Consider the following situation. You have a new burglar alarm installed at home. It is fairly reliable at detecting a burglary, but also responds on occasion to minor earthquakes. (This example is due to Judea Pearl, a resident of Los Angeles; hence the acute interest in earthquakes.) You also have two neighbors, John and Mary, who have promised to call you at work when they hear the alarm. John always calls when he hears the alarm, but sometimes confuses the telephone ringing with the alarm and calls then, too. Mary, on the other hand, likes rather loud music and sometimes misses the alarm altogether. Given the evidence of who has or has not called, we would like to estimate the probability of a burglary. This simple domain is described by the belief network in Figure 15.1.

The topology of the network can be thought of as an abstract knowledge base that holds in a wide variety of different settings, because it represents the general structure of the causal processes in the domain rather than any details of the population of individuals. In the case of the burglary network, the topology shows that burglary and earthquakes directly affect the probability of the alarm going off, but whether or not John and Mary call depends only on the alarm—the network thus represents our assumption that they do not perceive any burglaries directly, and they do not feel the minor earthquakes.

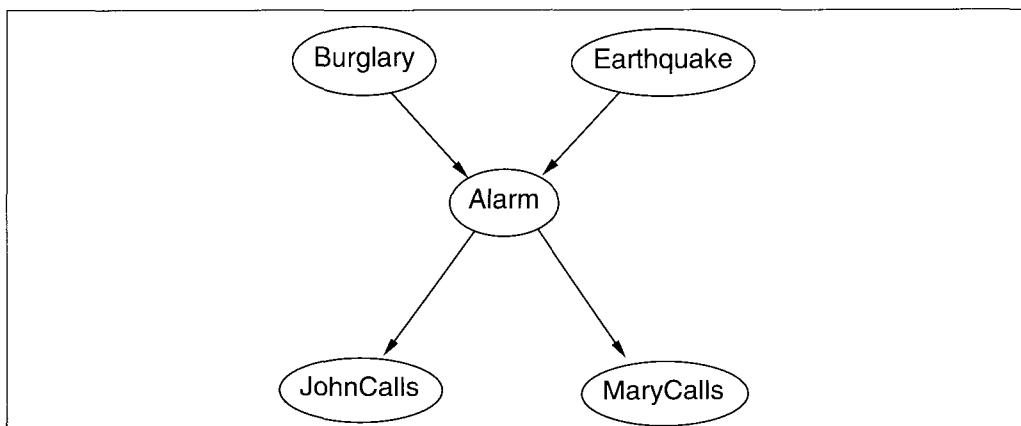


Figure 15.1 A typical belief network.

Notice that the network does not have nodes corresponding to Mary currently listening to loud music, or to the telephone ringing and confusing John. These factors are summarized in the uncertainty associated with the links from *Alarm* to *JohnCalls* and *MaryCalls*. This shows both laziness and ignorance in operation: it would be a lot of work to determine any reason why those factors would be more or less likely in any particular case, and we have no reasonable way to obtain the relevant information anyway. The probabilities actually summarize a *potentially infinite* set of possible circumstances in which the alarm might fail to go off (high humidity, power failure, dead battery, cut wires, dead mouse stuck inside bell, ...) or John or Mary might fail to call and report it (out to lunch, on vacation, temporarily deaf, passing helicopter, ...). In this way, a small agent can cope with a very large world, at least approximately. The degree of approximation can be improved if we introduce additional relevant information.

CONDITIONAL  
PROBABILITY TABLE  
  
CONDITIONING CASE

Once we have specified the topology, we need to specify the **conditional probability table** or CPT for each node. Each row in the table contains the conditional probability of each node value for a **conditioning case**. A conditioning case is just a possible combination of values for the parent nodes (a miniature atomic event, if you like). For example, the conditional probability table for the random variable *Alarm* might look like this:

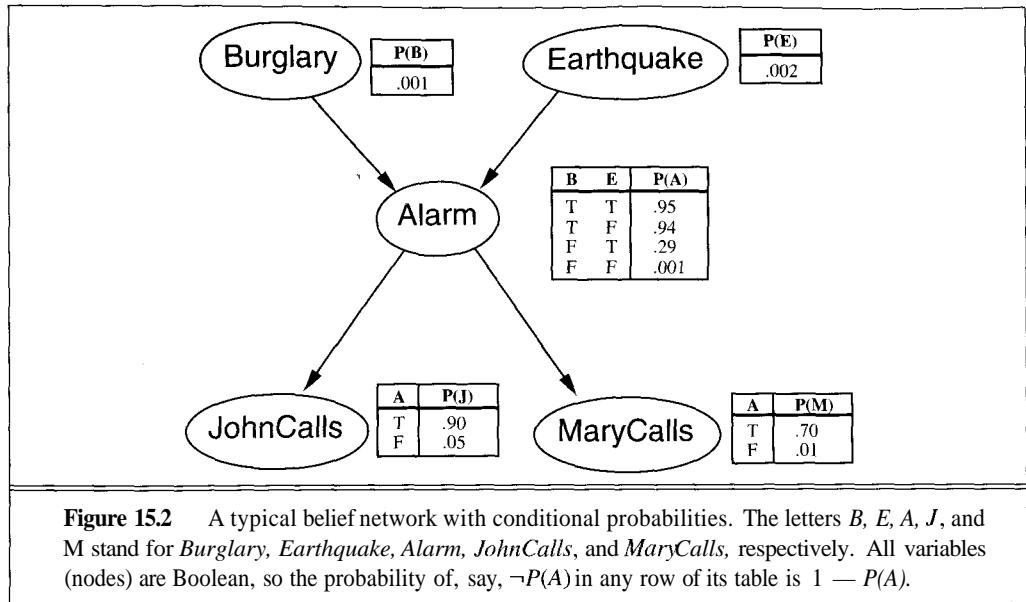
Burglary	Earthquake	$P(Alarm Burglary, Earthquake)$	
		True	False
True	True	0.950	0.050
True	False	0.950	0.050
False	True	0.290	0.710
False	False	0.001	0.999

Each row in a conditional probability table must sum to 1, because the entries represent an exhaustive set of cases for the variable. Hence only one of the two numbers in each row shown above is independently specifiable. In general, a table for a Boolean variable with  $n$  Boolean parents contains  $2^n$  independently specifiable probabilities. A node with no parents has only one row, representing the prior probabilities of each possible value of the variable.

The complete network for the burglary example is shown in Figure 15.2, where we show just the conditional probability for the *True* case of each variable.

## 15.2 THE SEMANTICS OF BELIEF NETWORKS

The previous section described what a network is, but not what it means. There are two ways in which one can understand the semantics of belief networks. The first is to see the network as a representation of the joint probability distribution. The second is to view it as an encoding of a collection of conditional independence statements. The two views are equivalent, but the first turns out to be helpful in understanding how to *construct* networks, whereas the second is helpful in designing inference procedures.



**Figure 15.2** A typical belief network with conditional probabilities. The letters  $B$ ,  $E$ ,  $A$ ,  $J$ , and  $M$  stand for *Burglary*, *Earthquake*, *Alarm*, *JohnCalls*, and *MaryCalls*, respectively. All variables (nodes) are Boolean, so the probability of, say,  $\neg P(A)$  in any row of its table is  $1 - P(A)$ .

## Representing the joint probability distribution

A belief network provides a complete description of the domain. Every entry in the joint probability distribution can be calculated from the information in the network. A generic entry in the joint is the probability of a conjunction of particular assignments to each variable, such as  $P(X_1 = x_1 \wedge \dots \wedge X_n = x_n)$ . We use the notation  $P(x_1, \dots, x_n)$  as an abbreviation for this. The value of this entry is given by the following formula:

$$P(x_1, \dots, x_n) = \prod_{i=1}^n P(x_i | \text{Parents}(X_i)) \quad (15.1)$$

Thus, each entry in the joint is represented by the product of the appropriate elements of the conditional probability tables (CPTs) in the belief network. The CPTs therefore provide a decomposed representation of the joint.

To illustrate this, we can calculate the probability of the event that the alarm has sounded but neither a burglary nor an earthquake has occurred, and both John and Mary call. We use single-letter names for the variables:

$$\begin{aligned} & P(J \wedge M \wedge \neg A \wedge \neg B \wedge \neg E) \\ &= P(J|A)P(M|A)P(\neg A|\neg B \wedge \neg E)P(\neg B)P(\neg E) \\ &= 0.90 \times 0.70 \times 0.001 \times 0.999 \times 0.998 = 0.00062 \end{aligned}$$

Section 14.3 explained that the joint distribution can be used to answer any query about the domain. If a belief network is a representation of the joint, then it too can be used to answer any query. Trivially, this can be done by first computing all the joint entries. We will see below that there are much better methods.

### A method for constructing belief networks

Equation (15.1) defines what a given belief network means. It does not, however, explain how to *construct* a belief network such that the resulting joint distribution is a good representation of a given domain. We will now show that Equation (15.1) implies certain conditional independence relationships that can be used to guide the knowledge engineer in constructing the topology of the network. First, we rewrite the joint in terms of a conditional probability using the definition of conditional probability:

$$P(x_1, \dots, x_n) = P(x_n|x_{n-1}, \dots, x_1)P(x_{n-1}, \dots, x_1)$$

Then we repeat this process, reducing each conjunctive probability to a conditional probability and a smaller conjunction. We end up with one big product:

$$\begin{aligned} P(x_1, \dots, x_n) &= P(x_n|x_{n-1}, \dots, x_1)P(x_{n-1}|x_{n-2}, \dots, x_1) \cdots P(x_2|x_1)P(x_1) \\ &= \prod_{i=1}^n P(x_i|x_{i-1}, \dots, x_1) \end{aligned}$$

Comparing this with Equation (15.1), we see that the specification of the joint is equivalent to the general assertion that

$$\mathbf{P}(X_i|X_{i-1}, \dots, X_1) = P(X_i|\text{Parents}(X_i)) \quad (15.2)$$

provided that  $\text{Parents}(X_i) \subseteq \{x_{i-1}, \dots, x_1\}$ . This last condition is easily satisfied by labelling the nodes in any order that is consistent with the partial order implicit in the graph structure.

What the preceding equation says is that the belief network is a correct representation of the domain only if each node is conditionally independent of its predecessors in the node ordering, given its parents. Hence, in order to construct a belief network with the correct structure for the domain, we need to choose parents for each node such that this property holds. Intuitively, the parents of node  $X_i$  should contain all those nodes in  $X_1, \dots, X_{i-1}$  that directly influence  $X_i$ . For example, suppose we have completed the network in Figure 15.1 except for the choice of parents for *MaryCalls*. *MaryCalls* is certainly influenced by whether or not there is a *Burglary* or an *Earthquake*, but it is not *directly* influenced. Intuitively, our knowledge of the domain tells us that these events only influence Mary's calling behavior through their effect on the alarm. Also, given the state of the alarm, whether or not John calls has no influence on Mary's calling. Formally speaking, we believe that the following conditional independence statement holds:

$$\mathbf{P}(\text{MaryCalls}|\text{JohnCalls}, \text{Alarm}, \text{Earthquake}, \text{Burglary}) = \mathbf{P}(\text{MaryCalls}|\text{Alarm})$$

The general procedure for incremental network construction is as follows:

1. Choose the set of relevant variables  $X_i$  that describe the domain.
2. Choose an ordering for the variables.
3. While there are variables left:
  - (a) Pick a variable  $X_i$  and add a node to the network for it.
  - (b) Set  $\text{Parents}(X_i)$  to some minimal set of nodes already in the net such that the conditional independence property (15.2) is satisfied.
  - (c) Define the conditional probability table for  $X_i$ .



Because each node is only connected to earlier nodes, this construction method guarantees that the network is acyclic. Another important property of belief networks is that they contain no redundant probability values, except perhaps for one entry in each row of each conditional probability table. This means that *it is impossible for the knowledge engineer or domain expert to create a belief network that violates the axioms of probability*. We will see examples of the application of the construction method in the next section.

LOCALLY  
STRUCTURED  
SPARSE

### Compactness and node ordering

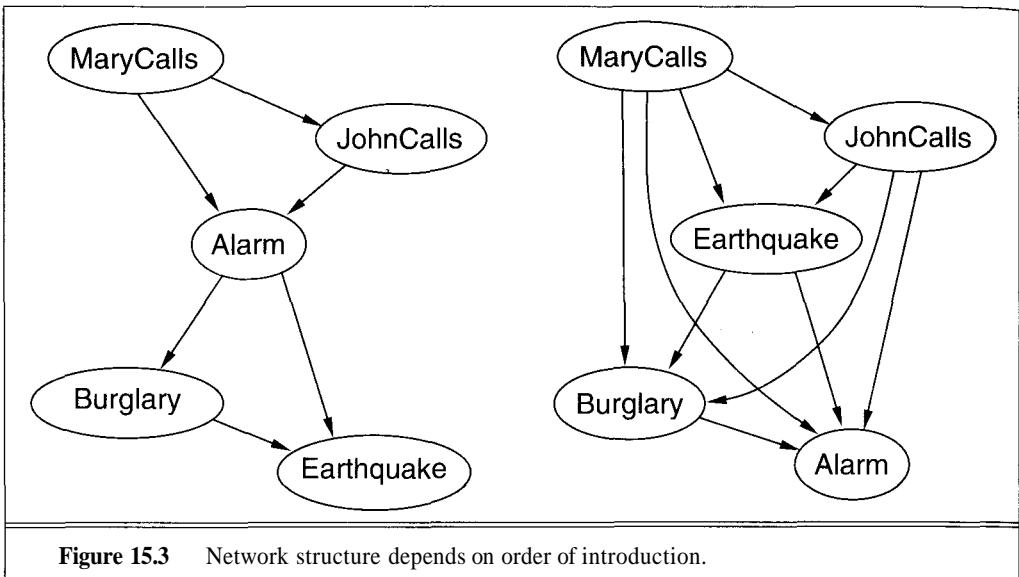
As well as being a complete and nonredundant representation of the domain, a belief network can often be far more *compact* than the full joint. This property is what makes it feasible to handle a large number of pieces of evidence without the exponential growth in conditional probability values that we saw in the discussion of Bayesian updating in Section 14.4.

The compactness of belief networks is an example of a very general property of **locally structured** (also called **sparse**) systems. In a locally structured system, each subcomponent interacts directly with only a bounded number of other components, regardless of the total number of components. Local structure is usually associated with linear rather than exponential growth in complexity. In the case of belief networks, it is reasonable to suppose that in most domains each random variable is directly influenced by at most  $k$  others, for some constant  $k$ . If we assume Boolean variables for simplicity, then the amount of information needed to specify the conditional probability table for a node will be at most  $2^k$  numbers, so the complete network can be specified by  $n2^k$  numbers. In contrast, the joint contains  $2^n$  numbers. To make this concrete, suppose we have 20 nodes ( $n = 20$ ) and each has at most 5 parents ( $k = 5$ ). Then the belief network requires 640 numbers, but the full joint requires over a million.

There are domains in which each variable can be influenced directly by all the others, so that the network is fully connected. Then specifying the conditional probability tables requires the same amount of information as specifying the joint. The reduction in information that occurs in practice comes about because real domains have a lot of structure, which networks are very good at capturing. In some domains, there will be slight dependencies that should strictly be included by adding a new link. But if these dependencies are very tenuous, then it may not be worth the additional complexity in the network for the small gain in accuracy. For example, one might object to our burglary network on the grounds that if there is an earthquake, then John and Mary would not call even if they heard the alarm, because they assume the earthquake is the cause. Whether to add the link from *Earthquake* to *JohnCalls* and *MaryCalls* (and thus enlarge the tables) depends on the importance of getting more accurate probabilities compared to the cost of specifying the extra information.

Even in a locally structured domain, constructing a locally structured belief network is not a trivial problem. We require not only that each variable is directly influenced by only a few others, but also that the network topology actually reflects those direct influences with the appropriate set of parents. Because of the way that the construction procedure works, the "direct influencers" will have to be added to the network first if they are to become parents of the node they influence. Therefore, *the correct order to add nodes is to add the "root causes" first, then the variables they influence*, and so on until we reach the "leaves," which have no direct causal influence on the other variables.





What happens if we happen to choose the wrong order? Let us consider the burglary example again. Suppose we decide to add the nodes in the order *MaryCalls, JohnCalls, Alarm, Burglary, Earthquake*. We get a somewhat more complicated network (Figure 15.3, left). The process goes as follows:

- Adding *MaryCalls*: no parents.
- Adding *JohnCalls*: if Mary calls, that probably means the alarm has gone off, which of course would make it more likely that John calls. Therefore, there is a dependence:

$$P(\text{JohnCalls}|\text{MaryCalls}) \neq P(\text{JohnCalls})$$

Hence, *JohnCalls* needs *MaryCalls* as a parent.

- Adding *Alarm*: clearly, if both call, it is more likely that the alarm has gone off than if just one or neither call, so we need both *MaryCalls* and *JohnCalls* as parents.
- Adding *Burglary*: if we know the alarm state, then the call (or lack of it) from John or Mary might tell us about whether our telephone is ringing or whether Mary's music is on loud, but it does not give us further information about a burglary. That is,

$$P(\text{Burglary}|\text{Alarm}, \text{JohnCalls}, \text{MaryCalls}) = P(\text{Burglary}|\text{Alarm})$$

Hence we need just *Alarm* as parent.

- Adding *Earthquake*: if the alarm is on, it is more likely that there has been an earthquake (because the alarm is an earthquake detector of sorts). But if we know there has been a burglary, then that accounts for the alarm and the probability of an earthquake would be only slightly above normal. Hence we need both *Alarm* and *Burglary* as parents:

$$P(\text{Earthquake}|\text{Burglary}, \text{Alarm}, \text{JohnCalls}, \text{MaryCalls}) = P(\text{Earthquake}|\text{Burglary}, \text{Alarm})$$

The resulting network has two more links than the original network in Figure 15.1, and requires three more probabilities to be specified. But the worst part is that some of the links represent

tenuous relationships that require difficult and unnatural probability judgments, such as assessing the probability of *Earthquake* given *Burglary* and *Alarm*. This phenomenon is quite general. If we try to build a diagnostic model with links from symptoms to causes (as from *MaryCalls* to *Alarm*, or *Alarm* to *Burglary*), we end up having to specify additional dependencies between otherwise independent causes, and often between separately occurring symptoms as well. *If we stick to a causal model, we end up having to specify fewer numbers, and the numbers will often be easier to come up with.* In the domain of medicine, for example, it has been shown by Tversky and Kahneman (1982) that expert physicians prefer to give probability judgments for causal rules rather than for diagnostic ones.

The right-hand side of Figure 15.3 shows a really bad ordering of nodes: *MaryCalls*, *JohnCalls*, *Earthquake*, *Burglary*, *Alarm*. This network requires 31 distinct probabilities to be specified—exactly the same as the full joint distribution. It is important to realize, however, that any of the three networks can represent *exactly the same joint distribution*. The last two versions simply fail to represent all the conditional independence relationships, and end up specifying a lot of unnecessary numbers instead.

### Representation of conditional probability tables

Even with a fairly small number of parents, a node's conditional probability table still requires a lot of numbers. Filling in the table would appear to require a good deal of time and also a lot of experience with all the possible conditioning cases. In fact, this is a worst-case scenario, where the relationship between the parents and the child is completely arbitrary. Usually, such relationships fall into one of several categories that have **canonical distributions**—that is, they fit some standard pattern. In such cases, the complete table can be specified by naming the pattern and perhaps supplying a few parameters.

The simplest example is provided by **deterministic nodes**. A deterministic node has its value specified exactly by the values of its parents, with no uncertainty. The relationship can be a logical one—for example, the relationship between parent nodes *Canadian*, *US*, *Mexican* and the child node *NorthAmerican* is simply that the child is the disjunction of the parents. The relationship can also be numerical—for example, if the parent nodes are the prices of a particular model of car at several dealers, and the child node is the price that a bargain hunter ends up paying, then the child node is the minimum of the parent values; or if the parent nodes are the inflows (rivers, runoff, precipitation) into a lake and the outflows (rivers, evaporation, seepage) from the lake and the child is the change in lake level, then the child is the difference between the inflow parents and the outflow parents.

Uncertain relationships can often be characterized by so-called "noisy" logical relationships. The standard example is the so-called **noisy-OR** relation, which is a generalization of the logical OR. In propositional logic, we might say *Fever* is true if and only if *Cold*, *Flu*, or *Malaria* is true. The noisy-OR model adds some uncertainty to this strict logical approach. The model makes three assumptions. First, it assumes that each cause has an independent chance of causing the effect. Second, it assumes that all the possible causes are listed. (This is not as strict as it seems, because we can always add a so-called **leak node** that covers "miscellaneous causes") Third, it assumes that whatever inhibits, say, *Cold* from causing a fever is independent of whatever inhibits *Flu* from causing a fever. These inhibitors are not represented as nodes but

CANONICAL DISTRIBUTIONS

DETERMINISTIC NODES

NOISY-OR

LEAK NODE

rather are summarized as "noise parameters." If  $P(\text{Fever}|\text{Cold}) = 0.4$ ,  $P(\text{Fever}|\text{Flu}) = 0.8$ , and  $P(\text{Fever}|\text{Malaria}) = 0.9$ , then the noise parameters are 0.6, 0.2, and 0.1, respectively. If no parent node is true, then the output node is false with 100% certainty. If exactly one parent is true, then the output is false with probability equal to the noise parameter for that node. In general, the probability that the output node is *False* is just the product of the noise parameters for all the input nodes that are true. For this example, we have the following:

<i>Cold</i>	<i>Flu</i>	<i>Malaria</i>	$P(\text{Fever})$	$P(\neg\text{Fever})$
F	F	F	0.0	1.0
F	F	T	0.9	0.1
F	T	F	0.8	0.2
F	T	T	0.98	$0.02 = 0.2 \times 0.1$
T	F	F	0.4	0.6
T	F	T	0.94	$0.06 = 0.6 \times 0.1$
T	T	F	0.88	$0.12 = 0.6 \times 0.2$
T	T	T	0.988	$0.012 = 0.6 \times 0.2 \times 0.1$

In general, noisy logical relationships in which a variable depends on  $k$  parents can be described using  $O(k)$  parameters instead of  $O(2^k)$  for the full conditional probability table. This makes assessment and learning much easier. For example, the CPSC network (Pradhan *et al.*, 1994) uses noisy-OR and noisy-MAX, and requires "only" 8,254 values instead of 133,931,430 for a network with full CPTs.

## Conditional independence relations in belief networks

The preceding analysis shows that a belief network expresses the conditional independence of a node and its predecessors, given its parents, and uses this independence to design a construction method for networks. If we want to design inference algorithms, however, we will need to know whether more general conditional independencies hold. If we are given a network, is it possible to "read off" whether a set of nodes  $X$  is independent of another set  $Y$ , given a set of evidence nodes  $E$ ? The answer is yes, and the method is provided by the notion of **direction-dependent separation** or **d-separation**.

First, we will say what d-separation is good for. *If every undirected path<sup>2</sup> from a node in  $X$  to a node in  $Y$  is d-separated by  $E$ , then  $X$  and  $Y$  are conditionally independent given  $E$ .* The definition of d-separation is somewhat complicated. We will need to appeal to it several times in constructing our inference algorithms. Once this is done, however, the process of constructing and using belief networks does not involve any uses of d-separation.

A set of nodes  $E$  d-separates two sets of nodes  $X$  and  $Y$  if every undirected path from a node in  $X$  to a node in  $Y$  is **blocked** given  $E$ . A path is blocked given a set of nodes  $E$  if there is a node  $Z$  on the path for which one of three conditions holds:

1.  $Z$  is in  $E$  and  $Z$  has one arrow on the path leading in and one arrow out.

<sup>2</sup> An undirected path is a path through the network that ignores the direction of the arrows.

D-SEPARATION

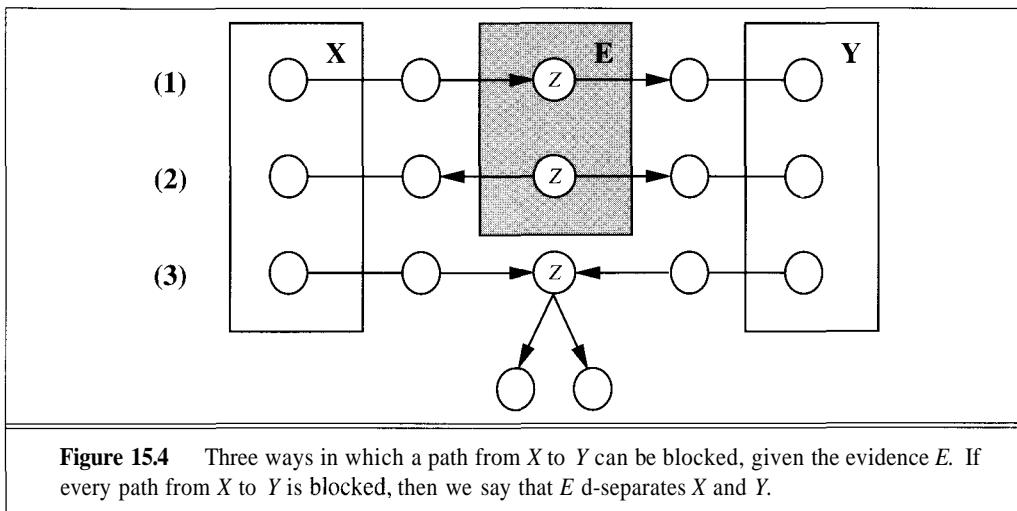


BLOCKED

2. Z is in E and Z has both path arrows leading out.
3. Neither Z nor any descendant of Z is in E, and both path arrows lead in to Z.

Figure 15.4 shows these three cases. The proof that d-separated nodes are conditionally independent is also complicated. We will use Figure 15.5 to give examples of the three cases:

1. Whether there is *Gas* in the car and whether the car *Radio* plays are independent given evidence about whether the *SparkPlugs* fire.
2. *Gas* and *Radio* are independent if it is known if the *Battery* works.
3. *Gas* and *Radio* are independent given no evidence at all. But they are dependent given evidence about whether the car *Starts*. For example, if the car does not start, then the radio playing is increased evidence that we are out of gas. *Gas* and *Radio* are also dependent given evidence about whether the car *Moves*, because that is enabled by the car starting.

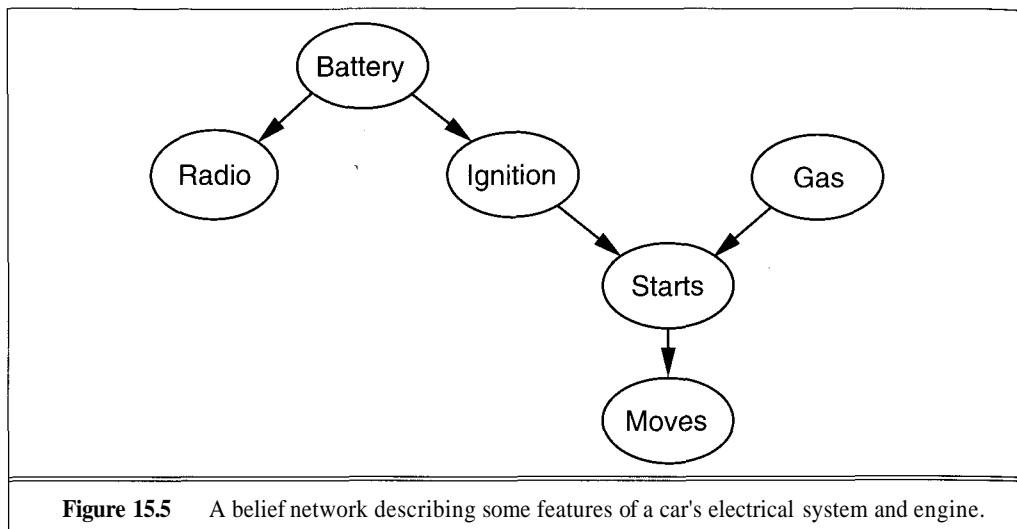


**Figure 15.4** Three ways in which a path from *X* to *Y* can be blocked, given the evidence *E*. If every path from *X* to *Y* is blocked, then we say that *E* d-separates *X* and *Y*.

## 15.3 INFERENCE IN BELIEF NETWORKS

The basic task for any probabilistic inference system is to compute the posterior probability distribution for a set of **query variables**, given exact values for some **evidence variables**. That is, the system computes  $\mathbf{P}(\text{Query}|\text{Evidence})$ . In the alarm example, *Burglary* is an obvious query variable, and *JohnCalls* and *MaryCalls* could serve as evidence variables. Of course, belief networks are flexible enough so that any node can serve as either a query or an evidence variable. There is nothing to stop us from asking  $\mathbf{P}(\text{Alarm}|\text{JohnCalls}, \text{Earthquake})$ , although it would be somewhat unusual. In general, *an agent gets values for evidence variables from its percepts (or from other reasoning), and asks about the possible values of other variables so that it can decide*





**Figure 15.5** A belief network describing some features of a car's electrical system and engine.

what action to take. The two functions we need are BELIEF-NET-TELL, for adding evidence to the network, and BELIEF-NET-ASK, for computing the posterior probability distribution for a given query variable.

### The nature of probabilistic inferences

Before plunging into the details of the inference algorithms, it is worthwhile to examine the kinds of things such algorithms can achieve. We will see that a single mechanism can account for a very wide variety of plausible inferences under uncertainty.

Consider the problem of computing  $P(\text{Burglary}|\text{JohnCalls})$ , the probability that there is a burglary given that John calls. This task is quite tricky for humans, and therefore for many reasoning systems that attempt to encode human judgment. The difficulty is not the complexity of the problem, but keeping the reasoning straight. An incorrect but all-too-common line of reasoning starts by observing that when the alarm goes off,  $\text{JohnCalls}$  will be true 90% of the time. The alarm is fairly accurate at reflecting burglaries, so  $P(\text{Burglary}|\text{JohnCalls})$  should also be about 0.9, or maybe 0.8 at worst. The problem is that this line of reasoning ignores the prior probability of John calling. Over the course of 1000 days, we expect one burglary, for which John is very likely to call. However, John also calls with probability 0.05 when there actually is no alarm—about 50 times over 1000 days. Thus, we expect to receive about 50 false alarms from John for every 1 burglary, so  $P(\text{Burglary}|\text{JohnCalls})$  is about 0.02. In fact, if we carry out the exact computation, we find that the true value is 0.016. It is less than our 0.02 estimate because the alarm is not perfect.

Now suppose that as soon as we get off the phone with John, Mary calls. We are now interested in incrementally updating our network to give  $P(\text{Burglary}|\text{JohnCalls} \wedge \text{MaryCalls})$ . Again, humans often overestimate this value; the correct answer is only 0.29. We can also determine that  $P(\text{Alarm}|\text{JohnCalls} \wedge \text{MaryCalls})$  is 0.76 and  $P(\text{Earthquake}|\text{JohnCalls} \wedge \text{MaryCalls})$  is 0.18.

In both of these problems, the reasoning is diagnostic. But belief networks are not limited to diagnostic reasoning and in fact can make four distinct kinds of inference:

DIAGNOSTIC INFERENCES

CAUSAL INFERENCES

INTERCAUSAL INFERENCES

EXPLAINING AWAY

MIXED INFERENCES

SENSITIVITY ANALYSIS

SINGLY CONNECTED

**0 Diagnostic inferences** (from effects to causes).

Given that *JohnCalls*, infer that  $P(\text{Burglary}|\text{JohnCalls}) = 0.016$ .

**0 Causal inferences** (from causes to effects).

Given *Burglary*,  $P(\text{JohnCalls}|\text{Burglary}) = 0.86$  and  $P(\text{MaryCalls}|\text{Burglary}) = 0.67$ .

**◊ Intercausal inferences** (between causes of a common effect).

Given *Alarm*, we have  $P(\text{Burglary}|\text{Alarm}) = 0.376$ . But if we add the evidence that *Earthquake* is true, then  $P(\text{Burglary}|\text{Alarm} \wedge \text{Earthquake})$  goes down to 0.003. Even though burglaries and earthquakes are independent, the presence of one makes the other less likely. This pattern of reasoning is also known as **explaining away**.<sup>3</sup>

**0 Mixed inferences** (combining two or more of the above).

Setting the effect *JohnCalls* to true and the cause *Earthquake* to false gives

$$P(\text{Alarm}|\text{JohnCalls} \wedge \neg\text{Earthquake}) = 0.03$$

This is a simultaneous use of diagnostic and causal inference. Also,

$$P(\text{Burglary}|\text{JohnCalls} \wedge \neg\text{Earthquake}) = 0.017$$

This is a combination of intercausal and diagnostic inference.

These four patterns are depicted in Figure 15.6. Besides calculating the belief in query variables given definite values for evidence variables, belief networks can also be used for the following:

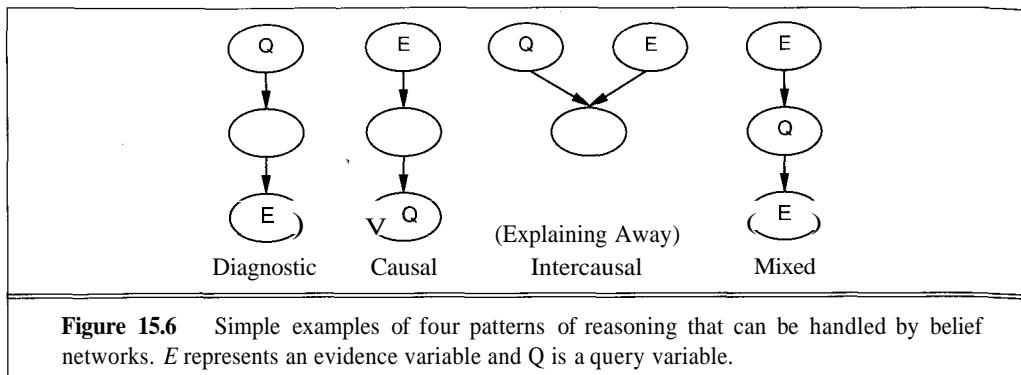
- Making decisions based on probabilities in the network and on the agent's utilities.
- Deciding which additional evidence variables should be observed in order to gain useful information.
- Performing **sensitivity analysis** to understand which aspects of the model have the greatest impact on the probabilities of the query variables (and therefore must be accurate).
- Explaining the results of probabilistic inference to the user.

These tasks are discussed further in Chapter 16. In this chapter, we focus on computing posterior probabilities of query variables. In Chapter 18, we show how belief networks can be learned from example cases.

## An algorithm for answering queries

In this section, we will derive an algorithm for BELIEF-NET-ASK. This is a rather technical section, and some of the mathematics and notation are unavoidably intricate. The algorithm itself, however, is very simple. Our version will work rather like the backward-chaining algorithm in Chapter 9, in that it begins at the query variable and chains along the paths from that node until it reaches evidence nodes. Because of the complications that may arise when two different paths converge on the same node, we will derive an algorithm that works only on **singly connected**

<sup>3</sup> To get this effect, we do not have to *know* that *Earthquake* is true, we just need some evidence for it. When one of the authors first moved to California, he lived in a rickety house that shook when large trucks went by. After one particularly large shake, he turned on the radio, heard the Carole King song / *Feel the Earth Move*, and considered this strong evidence that a large truck had not gone by.



## POLYTREES

networks, also known as **polytress**. In such networks, there is at most one undirected path between any two nodes in the network. Algorithms for general networks (Section 15.4) will use the polytree algorithms as their main subroutine.

Figure 15.7 shows a generic singly connected network. Node  $X$  has parents  $V = U_1 \dots U_m$ , and children  $Y = Y_1 \dots Y_n$ . For each child and parent we have drawn a box that contains all the node's descendants and ancestors (except for  $X$ ). The singly connected property means that all the boxes are disjoint and have no links connecting them. We are assuming that  $X$  is the query variable, and that there is some set of evidence variables  $E$ . The aim is to compute  $P(X|E)$ . (Obviously, if  $X$  is itself an evidence variable in  $E$ , then calculating  $P(X|E)$  is trivial. We will assume that  $X$  is not in  $E$ .)

In order to derive the algorithm, it will be helpful to be able to refer to different portions of the total evidence. The first distinction we will need is the following:

## CAUSAL SUPPORT

$EX$  is the the **causal support** for  $X$ —the evidence variables "above"  $X$  that are connected to  $X$  through its parents.

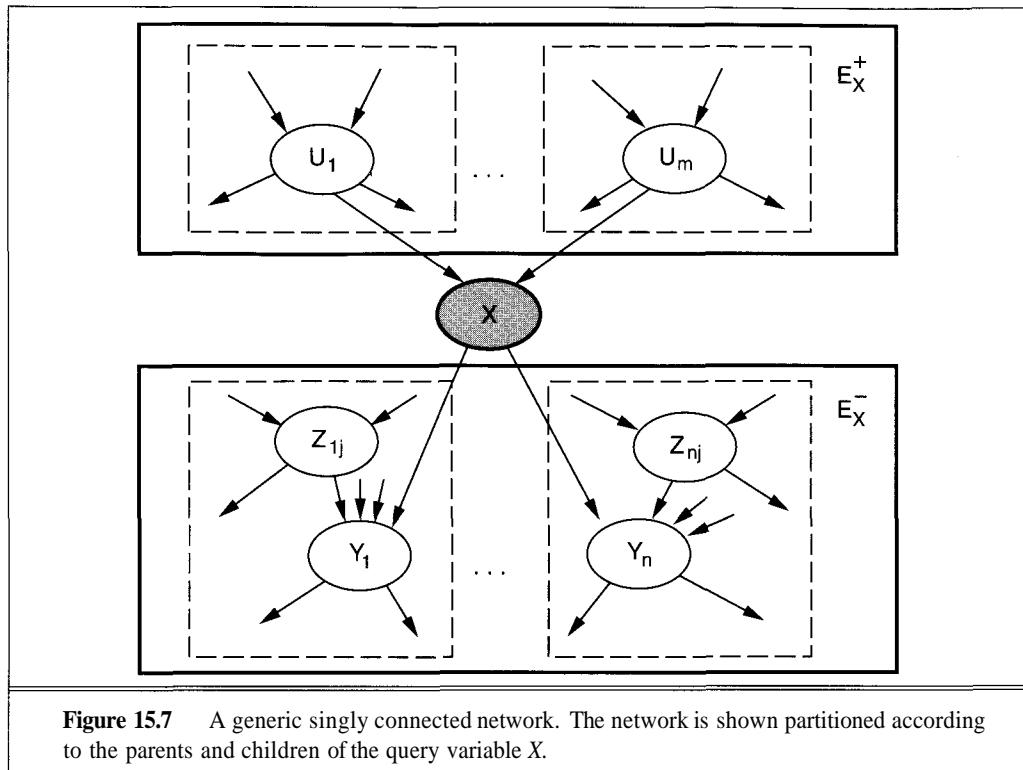
## EVIDENTIAL SUPPORT

$EX$  is the **evidential support** for  $X$ —the evidence variables "below"  $X$  that are connected to  $X$  through its children.

Sometimes we will need to exclude certain paths when considering evidence connected to a certain variable. For example, we will use  $E_{U_i \setminus X}$  to refer to all the evidence connected to node  $U_i$  *except* via the path from  $X$ . Similarly,  $E_{Y_i \setminus X}^+$  means all the evidence connected to  $Y_i$  through its parents *except* for  $X$ . Notice that the total evidence  $E$  can be written as  $E_X$  (all the evidence connected to  $X$ ) and as  $E_{X \setminus}$  (all the evidence connected to  $X$  with no exceptions).

Now we are ready to compute  $P(X|E)$ . The general strategy is roughly the following:

- Express  $P(X|E)$  in terms of the contributions of  $E_X^+$  and  $E_X^-$  •
- Compute the contribution of  $E_X^+$  by computing its effect on the parents of  $X$ , and then passing that effect on to  $X$ . Notice that computing the effect on each parent of  $X$  is a recursive instance of the problem of computing the effect on  $X$ .
- Compute the contribution of  $E_X^-$  by computing its effect on the children of  $X$ , and then passing that effect on to  $X$ . Notice that computing the effect on each child of  $X$  is a recursive instance of the problem of computing the effect on  $X$ .



**Figure 15.7** A generic singly connected network. The network is shown partitioned according to the parents and children of the query variable  $X$ .

Our derivation will work by applying Bayes' rule and other standard methods for manipulating probability expressions, until we have massaged the formulas into something that looks like a recursive instance of the original problems. Along the way, we will use simplifications sanctioned by the conditional independence relationships inherent in the network structure.

The total evidence  $E$  consists of the evidence above  $X$  and the evidence below  $X$ , since we are assuming that  $X$  itself is not in  $E$ . Hence, we have

$$\mathbf{P}(X|E) = \mathbf{P}(X|E_X^-, E_X^+)$$

To separate the contributions of  $E_X^+$  and  $E_X^-$ , we apply the conditionalized version of Bayes' rule (Equation (14.4)) keeping  $E_X^-$  as fixed background evidence:

$$\mathbf{P}(X|E_X^-, E_X^+) = \frac{\mathbf{P}(E_X^-|X, E_X^+) \mathbf{P}(X|E_X^+)}{\mathbf{P}(E_X^-|E_X^+)}$$

Because  $X$  d-separates  $E_X^+$  from  $E_X^-$  in the network, we can use conditional independence to simplify the first term in the numerator. Also, we can treat  $1/\mathbf{P}(E_X^-|E_X^+)$  as a normalizing constant, giving us

$$\mathbf{P}(X|E) = \alpha \mathbf{P}(E_X^-|X) \mathbf{P}(X|E_X^+)$$

So now we just need to compute the two terms  $\mathbf{P}(E_X^-|X)$  and  $\mathbf{P}(X|E_X^+)$ . The latter is easier, so we shall look at it first.

We compute  $\mathbf{P}(X|E_X^+)$  by considering all the possible configurations of the *parents* of  $X$ , and how likely they are given  $E_X^+$ . Given each configuration, we know the probability of  $X$  directly from the CPT; we then average those probabilities, weighted by the likelihood of each configuration. To say this formally, let  $U$  be the vector of parents  $U_1, \dots, U_m$ , and let  $u$  be an assignment of values to them.<sup>4</sup> Then we have

$$\mathbf{P}(X|E_X^+) = \sum_u \mathbf{P}(X|u, E_X^+) P(u|E_X^+)$$

Now  $U$  d-separates  $X$  from  $E_X^+$ , so we can simplify the first term to  $\mathbf{P}(X|u)$ . We can simplify the second term by noting that  $E_X^+$  d-separates each  $U_i$  from the others, and by remembering that the probability of a conjunction of independent variables is equal to the product of their individual probabilities. This gives us

$$\mathbf{P}(X|E_X^+) = \sum_u \mathbf{P}(X|u) \prod_i \mathbf{P}(u_i|E_X^+)$$

The final step is to simplify the last term of this equation by partitioning  $EX$  into  $E_{U_1 \setminus X}, \dots, E_{U_m \setminus X}$  (the separate boxes in Figure 15.7) and using the fact that  $E_{U_i \setminus X}$  d-separates  $U_i$  from all the other evidence in  $E_X^+$ . So that gives us

$$\mathbf{P}(X|E_X^+) = \sum_u \mathbf{P}(X|u) \prod_i \mathbf{P}(u_i|E_{U_i \setminus X})$$

which we can plug into our earlier equation to yield

$$\mathbf{P}(X|E) = \alpha \mathbf{P}(E_X^-|X) \sum_u \mathbf{P}(X|u) \prod_i \mathbf{P}(U_i|E_{U_i \setminus X}) \quad (15.3)$$

Finally, this is starting to look like an algorithm:  $\mathbf{P}(X|u)$  is a lookup in the conditional probability table of  $X$ , and  $\mathbf{P}(U_i|E_{U_i \setminus X})$  is a recursive instance of the original problem, which was to compute  $\mathbf{P}(X|E)$ —that is,  $\mathbf{P}(X|E_X^-)$ . Note that the set of evidence variables in the recursive call is a proper subset of those in the original call—a good sign that the algorithm will terminate.

Now we return to  $\mathbf{P}(E_X^-|X)$ , with an eye toward coming up with another recursive solution. In this case we average over the values of  $Y_i$ , the children of  $X$ , but we will also need to include  $Y_i$ 's parents. We let  $Z_i$  be the parents of  $Y_i$  other than  $X$ , and let  $z_i$  be an assignment of values to the parents. The derivation is similar to the previous one. First, because the evidence in each  $Y_i$  box is conditionally independent of the others given  $X$ , we get

$$\mathbf{P}(E_X^-|X) = \prod_i \mathbf{P}(E_{Y_i \setminus X}|X)$$

Averaging over  $Y_i$  and  $z_i$  yields

$$\mathbf{P}(E_X^-|X) = \prod_i \mathbf{E}_{y_i} \mathbf{E}_{z_i} \mathbf{P}(E_{Y_i \setminus X}|X, y_i, z_i) \mathbf{P}(y_i, z_i | X)$$

Breaking  $E_{Y_i \setminus X}$  into the two independent components  $E_{Y_i}^-$  and  $E_{Y_i \setminus X}^+$ :

$$\mathbf{P}(E_X^-|X) = \prod_i \mathbf{E}_{y_i} \mathbf{E}_{z_i} \mathbf{P}(E_{Y_i}^-|X, y_i, z_i) \mathbf{P}(E_{Y_i \setminus X}^+|X, y_i, z_i) \mathbf{P}(y_i, z_i | X)$$

<sup>4</sup> For example, if there are two Boolean parents,  $U_1$  and  $U_2$ , then  $u$  ranges over four possible assignments, of which  $[true, false]$  is one.

$E_{Y_i}^-$  is independent of  $X$  and  $\mathbf{z}_i$  given  $y_i$ , and  $E_{Y_i \setminus X}^+$  is independent of  $X$  and  $y_i$ . We can also pull a term with no  $\mathbf{z}_i$  out of the  $\mathbf{z}_i$  summation:

$$\mathbf{P}(E_X^- | X) = \prod_i \mathbf{E}_{y_i} P(E_{Y_i}^- | y_i) \sum_{\mathbf{z}_i} \mathbf{P}(E_{Y_i \setminus X}^+ | \mathbf{z}_i) \mathbf{P}(y_i, \mathbf{z}_i | X)$$

Apply Bayes' rule to  $\mathbf{P}(E_{Y_i \setminus X}^+ | \mathbf{z}_i)$ :

$$\mathbf{P}(E_X^- | X) = \prod_i \sum_{y_i} P(E_{Y_i}^- | y_i) \sum_{\mathbf{z}_i} \frac{P(\mathbf{z}_i | E_{Y_i \setminus X}^+) P(E_{Y_i \setminus X}^+)}{P(\mathbf{z}_i)} \mathbf{P}(y_i, \mathbf{z}_i | X)$$

Rewriting the conjunction  $Y_i, \mathbf{z}_i$ :

$$\mathbf{P}(E_X^- | X) = \prod_i \sum_{y_i} P(E_{Y_i}^- | y_i) \sum_{\mathbf{z}_i} \frac{\bar{r}(\mathbf{z}_i | E_{Y_i \setminus X}^+) P(E_{Y_i \setminus X}^+)}{P(\mathbf{z}_i)} \mathbf{P}(y_i | X, \mathbf{z}_i) \mathbf{P}(\mathbf{z}_i | X)$$

Now  $\mathbf{P}(\mathbf{z}_i | X) = P(\mathbf{z}_i)$ , because  $Z$  and  $X$  are d-separated, so we can cancel them out. We can also replace  $P(E_{Y_i \setminus X}^+)$  by a normalizing constant  $\beta_i$ :

$$\mathbf{P}(E_X^- | X) = \prod_i \mathbf{E}_{y_i} P(E_{Y_i}^- | y_i) \mathbf{E}_{\mathbf{z}_i} \beta_i P(\mathbf{z}_i | E_{Y_i \setminus X}^+) \mathbf{P}(y_i | X, \mathbf{z}_i)$$

Finally, the parents of  $Y_i$  (the  $Z_{ij}$ ) are independent of each other, so we can multiply them together, just as we did with the  $U_i$  parents previously. We also combine the  $\beta_i$  into one big normalizing constant  $\beta$ :

$$\mathbf{P}(E_X^- | X) = \beta \prod_i \sum_{y_i} P(E_{Y_i}^- | y_i) \sum_{\mathbf{z}_i} \mathbf{P}(y_i | X, \mathbf{z}_i) \prod_j P(z_{ij} | E_{Z_{ij} \setminus Y_i}) \quad (15.4)$$

Notice that each of the terms in the final expression is easily evaluated:

- $P(E_{Y_i}^- | yj)$  is a recursive instance of  $\mathbf{P}(E_X^- | X)$ .
- $P(y_i | X, \mathbf{z}_i)$  is a conditional probability table entry for  $Y_i$ .
- $P(z_{ij} | E_{Z_{ij} \setminus Y_i})$  is a recursive instance of the  $\mathbf{P}(X | E)$  calculation—that is,  $\mathbf{P}(X | E_X \setminus Y_i)$

It is now a simple matter to turn all this into an algorithm. We will need two basic routines. SUPPORT-EXCEPT( $X, V$ ) computes  $\mathbf{P}(X | E_{X \setminus V})$ , using a slight generalization of Equation (15.3) to handle the "except" variable  $V$ . EVIDENCE-EXCEPT( $X, V$ ) computes  $P(E_X^- | V | X)$ , using a generalization of Equation (15.4). The algorithms are shown in Figure 15.8.

The computation involves recursive calls that spread out from  $X$  along all paths in the network. The recursion terminates on evidence nodes, root nodes (which have no parents), and leaf nodes (which have no children). Each recursive call excludes the node from which it was called, so each node in the tree is covered only once. Hence the algorithm is linear in the number of nodes in the network. Remember that this only works because the network is a *polytree*. If there were more than one path between a pair of nodes, then our recursions would either count the same evidence more than once or they would fail to terminate.

We have chosen to present a "backward-chaining" algorithm because it is the simplest algorithm for polytrees. One drawback is that it computes the probability distribution for just one variable. If we wanted the posterior distributions for all the non-evidence variables, we would have to run the algorithm once for each. This would give a quadratic runtime, and

**function** BELIEF-NET-ASK( $X$ ) **returns** a probability distribution over the values of  $X$   
**inputs:**  $X$ , a random variable

SUPPORT-EXCEPT( $X$ , null)

---

**function** SUPPORT-EXCEPT( $X, V$ ) **returns**  $\mathbf{P}(X|E_{X \setminus V})$

if EVIDENCE?( $X$ ) **then return** observed point distribution for  $X$

**else**

calculate  $\mathbf{P}(E_{X \setminus V}^- | X) = \text{EVIDENCE-EXCEPT}(X, V)$   
 $U = \text{PARENTS}[X]$

if  $U$  is empty

**then return**  $\alpha \mathbf{P}(E_{X \setminus V}^- | X) \mathbf{P}(X)$

**else**

**for each**  $U_i$  **in**  $U$

calculate and store  $\mathbf{P}(U_i | E_{U_i \setminus X}) = \text{SUPPORT-EXCEPT}(U_i, X)$

**return**  $\beta \mathbf{P}(E_{X \setminus V}^- | X) \sum_{\mathbf{u}} \mathbf{P}(X | \mathbf{u}) \prod_i \mathbf{P}(U_i | E_{U_i \setminus X})$

---

function EVIDENCE-EXCEPT( $X, V$ ) **returns**  $\mathbf{P}(E_{X \setminus V}^- | X)$

$Y = \text{CHILDREN}[X] - V$

if  $Y$  is empty

**then return** a uniform distribution

**else**

**for each**  $Y_i$  **in**  $Y$  **do**

calculate  $\mathbf{P}(E_{Y_i}^- | Y_i) = \text{EVIDENCE-EXCEPT}(Y_i, \text{null})$

$Z_i = \text{PARENTS}[Y_i] - X$

**for each**  $Z_{ij}$  **in**  $Z_i$

calculate  $\mathbf{P}(Z_{ij}^- | E_{Z_{ij} \setminus Y_i}) = \text{SUPPORT-EXCEPT}(Z_{ij}, Y_i)$

**return**  $\gamma \prod_i \sum_{Y_i} P(E_{Y_i}^- | Y_i) \sum_{\mathbf{z}_i} \mathbf{P}(Y_i | X, \mathbf{z}_i) \prod_j P(z_{ij}^- | E_{Z_{ij} \setminus Y_i})$

---

**Figure 15.8** A backward-chaining algorithm for solving probabilistic queries on a polytree. To simplify the presentation, we have assumed that the network is fixed and already primed with evidence, and that evidence variables satisfy the predicate EVIDENCE?. The probabilities  $\mathbf{P}(X|\mathbf{U})$ , where  $\mathbf{U}$  denotes the parents of  $X$ , are available from the CPT for  $X$ . Calculating the expressions  $\alpha \dots$  and  $\beta \dots$  is done by normalization.

would involve many repeated calculations. A better way to arrange things is to “memoize” the computations by forward-chaining from the evidence variables. Given careful bookkeeping, the entire computation can be done in linear time. It is interesting to note that the forward-chaining version can be viewed as consisting of “messages” being “propagated” through the network. This leads to a simple implementation on parallel computers, and an intriguing analogy to message propagation among neurons in the brain (see Chapter 19).

## 15.4 INFERENCE IN MULTIPLY CONNECTED BELIEF NETWORKS

MULTIPLY CONNECTED

A **multiply connected** graph is one in which two nodes are connected by more than one path. One way this happens is when there are two or more possible causes for some variable, and the causes share a common ancestor. Alternatively, one can think of multiply connected networks as representing situations in which one variable can influence another through more than one causal mechanism. For example, Figure 15.9 shows a situation in which whether it is cloudy has a causal link to whether it rains, and also a causal link to whether the lawn sprinklers are turned on (because a gardener who observes clouds is less likely to turn the sprinklers on). Both rain and sprinklers have an effect on whether the grass gets wet.

There are three basic classes of algorithms for evaluating multiply connected networks, each with its own areas of applicability:

CLUSTERING

0 **Clustering** methods transform the network into a probabilistically equivalent (but topologically different) polytree by merging offending nodes.

CONDITIONING

0 **Conditioning** methods do the transformation by instantiating variables to definite values, and then evaluating a polytree for each possible instantiation.

STOCHASTIC SIMULATION

◇ **Stochastic simulation** methods use the network to generate a large number of concrete models of the domain that are consistent with the network distribution. They give an approximation of the exact evaluation.

In the general case, exact inference in belief networks is known to be NP-hard. It is fairly straightforward to prove this, because a general belief network can represent any propositional logic problem (if all the probabilities are 1 or 0) and propositional logic problems are known to be NP-complete. For very large networks, approximation using stochastic simulation is currently the method of choice. The problem of approximating the posterior probabilities to within an arbitrary tolerance is itself NP-hard, but for events that are not too unlikely the calculations are usually feasible.

MEGANODE

### Clustering methods

One way of evaluating the network in Figure 15.9 is to transform it into a polytree by combining the *Sprinkler* and *Rain* node into a **meganode** called *Sprinkler+Rain*, as shown in Figure 15.10. The two Boolean nodes are replaced by a meganode that takes on four possible values: *TT*, *TF*, *FT*, and *FF*. The meganode has only one parent, the Boolean variable *Cloudy*, so there are two conditioning cases. Once the network has been converted to a polytree, a linear-time algorithm can be applied to answer queries. Queries on variables that have been clustered can be answered by averaging over the values of the other variables in the cluster.

Although clustering makes it possible to use a linear-time algorithm, the NP-hardness of the problem does not go away. In the worst case, the size of the network increases exponentially, because the conditional probability tables for the clusters involve the cross-product of the domains of the variables. In Figure 15.10, there are six independently specifiable numbers in *Sprinkler+Rain*, as opposed to four total in *Sprinkler* and *Rain*. (One of the numbers in each row

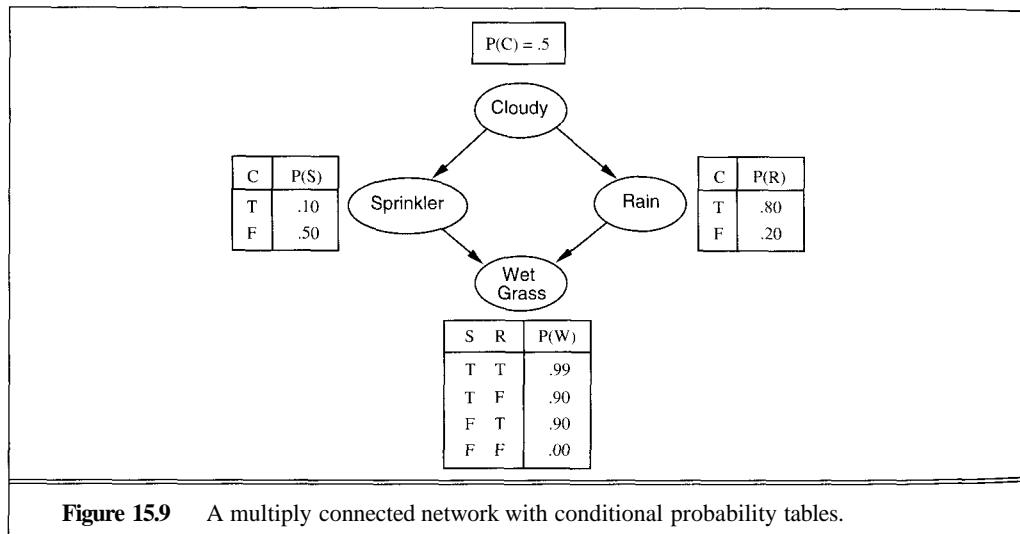
is not independent, because the row must sum to 1. In Figure 15.9, we dropped one of the two columns, but here we show all four.)

The tricky part about clustering is choosing the right meganodes. There are several ways to make this choice, but all of them ultimately produce meganodes with large probability tables. Despite this problem, clustering methods are currently the most effective approach for exact evaluation of multiply connected networks.

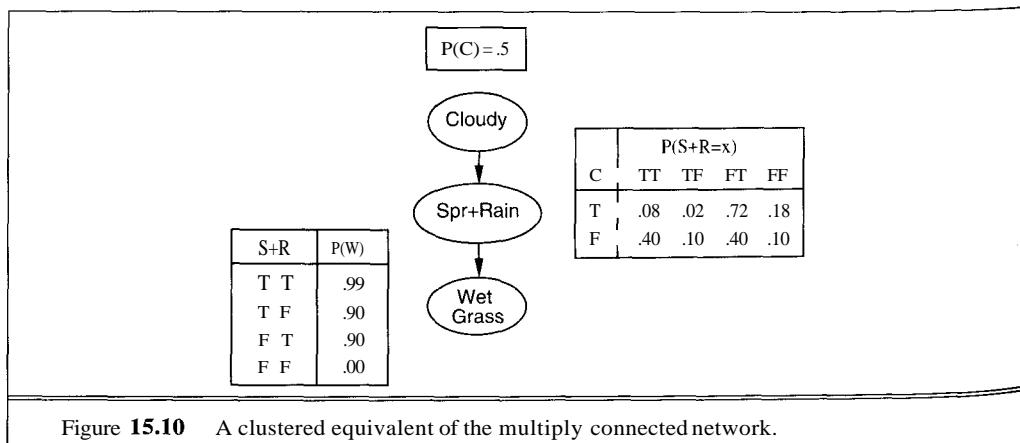
### Cutset conditioning methods

CUTSET  
CONDITIONING

The **cutset conditioning** method takes the opposite approach. Instead of transforming the network into one complex polytree, this method transforms the network into several simpler



**Figure 15.9** A multiply connected network with conditional probability tables.



**Figure 15.10** A clustered equivalent of the multiply connected network.

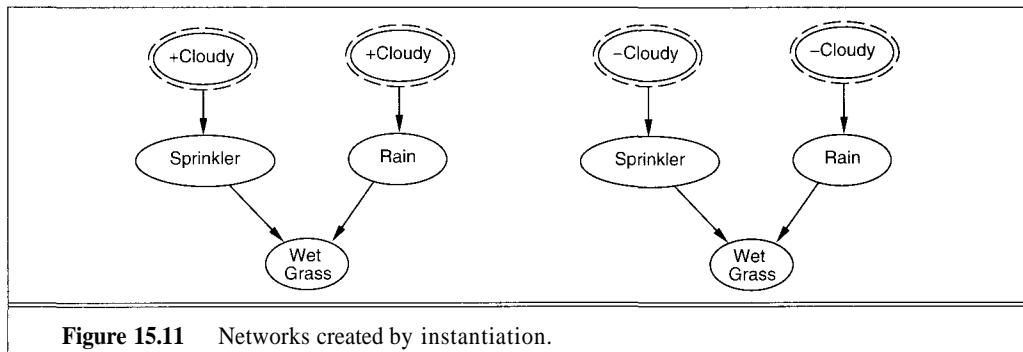
polytrees. Each simple network has one or more variables *instantiated* to a definite value.  $P(X|E)$  is computed as a weighted average over the values computed by each polytree.

CUTSET

A set of variables that can be instantiated to yield polytrees is called a **cutset**. In Figure 15.11, the cutset is just  $\{\text{Cloudy}\}$ , and because it is a Boolean variable, there are just two resulting polytrees. In general, the number of resulting polytrees is exponential in the size of the cutset, so we want to find a small cutset if possible.

BOUNDED CUTSET CONDITIONING

Cutset conditioning can be approximated by evaluating only some of the resulting polytrees. The error in the approximation is bounded by the total probability weight of the polytrees not yet evaluated. The obvious approach is to evaluate the most likely polytrees first. For example, if we need an answer accurate to within 0.1, we evaluate trees in decreasing order of likelihood until the total probability exceeds 0.9. This technique is called **bounded cutset conditioning**, and is useful in systems that need to make approximately correct decisions quickly.



**Figure 15.11** Networks created by instantiation.

## Stochastic simulation methods

LOGIC SAMPLING

In the stochastic simulation method known as **logic sampling**, we run repeated simulations of the world described by the belief network, and estimate the probability we are interested in by counting the frequencies with which relevant events occur. Each round of the simulation starts by randomly choosing a value for each root node of the network, weighting the choice by the prior probabilities. If the prior  $P(\text{Cloudy}) = 0.5$ , then we would pick  $\text{Cloudy} = \text{True}$  half the time, and  $\text{Cloudy} = \text{False}$  half the time. Whatever value is chosen, we can then choose values randomly for *Sprinkler* and *Rain*, using the conditional probabilities of those nodes given the known value of *Cloudy*. Finally, we do the same for *WetGrass*, and the first round is done.

To estimate  $P(\text{WetGrass}|\text{Cloudy})$  (or in general  $P(X|E)$ ), we repeat the process many times, and then compute the ratio of the number of runs where *WetGrass* and *Cloudy* are true to the number of runs where just *Cloudy* is true. This will always converge to the right value, although it may take many runs.

The main problem is when we are interested in some assignment of values to  $E$  that rarely occurs. For example, suppose we wanted to know

$$P(\text{WetGrass}|\text{Sprinkler A Rain})$$

Because *Sprinkler A Rain* is rare in the world, most of the simulation rounds would end up with different values for these evidence variables, and we would have to discard those runs. The fraction of useful runs decreases exponentially with the number of evidence variables.

We can get around this problem with an approach called **likelihood weighting**. The idea is that every time we reach an evidence variable, instead of randomly choosing a value (according to the conditional probabilities), we take the given value for the evidence variable, but use the conditional probabilities to see how likely that is. For example, to compute  $P(\text{WetGrass}|\text{Rain})$  we would do the following:

1. Choose a value for *Cloudy* with prior  $P(\text{Cloudy}) = 0.5$ . Assume we choose *Cloudy* = *False*.
2. Choose a value for *Sprinkler*. We see that  $P(\text{Sprinkler} \neg \text{Cloudy}) \approx 0.5$ , so we randomly choose a value given that distribution. Assume we choose *Sprinkler* = *True*.
3. Look at *Rain*. This is an evidence variable that has been set to *True*, so we look at the table to see that  $P(\text{Rain}|\neg\text{Cloudy}) = 0.2$ . This run therefore counts as 0.2 of a complete run.
4. Look at *WetGrass*. Choose randomly with  $P(\text{WetGrass}|\text{Sprinkler A Rain}) = 0.99$ ; assume we choose *WetGrass* = *True*.
5. We now have completed a run with likelihood 0.2 that says *Wetgrass* = *True* given *Rain*. The next run will result in a different likelihood, and (possibly) a different value for *WetGrass*. We continue until we have accumulated enough runs, and then add up the evidence for each value, weighted by the likelihood score.

Likelihood weighting usually converges considerably faster than logic sampling, and can handle very large networks. In the CPSC project (Pradhan *et al.*, 1994), for example, a belief network has been constructed for internal medicine that contains 448 nodes, 906 links and 8,254 conditional probability values. (The front cover shows a small portion of the network.) Likelihood weighting typically obtains accurate values in around 35 minutes on this network.

The main difficulty with likelihood weighting, and indeed with any stochastic sampling method, is that it takes a long time to reach accurate probabilities for unlikely events. In general, the runtime necessary to reach a given level of accuracy is inversely proportional to the probability of the event. Events such as the meltdown of a nuclear reactor on a particular day are *extremely* unlikely, but there is a very big difference between  $10^{-5}$  and  $10^{-10}$ , so we still need to get accurate values. Researchers are currently working to find ways around this problem.

## 15.5 KNOWLEDGE ENGINEERING FOR UNCERTAIN REASONING

The approach to knowledge engineering for probabilistic reasoning systems is very much like the approach for logical reasoning systems outlined in Section 8.2:

- *Decide what to talk about.* This remains a difficult step. It is important to decide which factors will be modelled, and which will just be summarized by probability statements. In an expert system of all dental knowledge, we will certainly want to talk about toothaches, gum disease, and cavities. We may want to know if a patient's parents have a history of gum disease, but we probably do not need to talk about the patient's third cousins. Once we

have an initial set of factors, we can extend the model by asking, "What directly influences this factor?" and "What does this factor directly influence?"

- *Decide on a vocabulary of random variables.* Determine the variables you want to use, and what values they can take on. Sometimes it is useful to quantize a continuous-valued variable into discrete ranges.
- *Encode general knowledge about the dependence between variables.* Here there is a qualitative part, where we say what variables are dependent on what others, and a quantitative part, where we specify probability values. The values can come either from the knowledge engineer's (or expert's) subjective experience, or from measurements of frequencies in a database of past experiences, or from some combination of the two.
- *Encode a description of the specific problem instance.* For example, we say that the particular patient we are interested in is a 34-year-old female with moderate to severe pain in the lower jaw.
- *Pose queries to the inference procedure and get answers.* The most common query is to ask for the value of some hypothesis variable. For example, given the patient's symptoms, what is the probability of gum disease, or of any other disorder. It is also common to use sensitivity analysis to determine how robust the answers are with respect to perturbations in the conditional probability table values.

## Case study: The Pathfinder system

PATHFINDER is a diagnostic expert system for lymph-node diseases, built by members of the Stanford Medical Computer Science program during the 1980s (see Heckerman (1991) for a discussion). The system deals with over 60 diseases and over 100 disease findings (symptoms and test results). Four versions of the system have been built, and the history is instructive because it shows a trend toward increasing sophistication in reasoning with uncertainty.

PATHFINDER I was a rule-based system written with the logical metareasoning system MRS. It did not do any uncertain reasoning.

PATHFINDER II experimented with several methods for uncertain reasoning, including certainty factors and the Dempster-Shafer theory (see Section 15.6). The results showed that a simplified Bayesian model (in which all disease findings were assumed to be independent) outperformed the other methods. One interesting result of the experiment was that 10% of cases were diagnosed incorrectly because the expert had given a probability of zero to an unlikely but possible event.

PATHFINDER III used the same simplified Bayesian model, but with a reassessment of the probabilities using a different protocol and paying attention to low-probability events.

PATHFINDER IV used a belief network to represent the dependencies that could not be handled in the simplified Bayesian model. The author of the system sat down with an expert physician and followed the knowledge engineering approach described earlier. Deciding on a vocabulary took 8 hours, devising the topology of the network took 35 hours, and making the 14,000 probability assessments took another 40 hours. The physician reportedly found it easy to think in terms of probabilities on causal links, and a concept called **similarity networks** made it easier for the expert to assess a large number of probabilities. The network constructed in this

process covers one of the 32 major areas of pathology. The plan is to cover all of pathology through consultations with leading experts in each area.

An evaluation of PATHFINDER III and IV used 53 actual cases of patients who were referred to a lymph-node specialist. As referrals, these cases were probably of above-average difficulty. In a blind evaluation, expert analysis of the diagnoses showed that PATHFINDER III scored an average 7.9 out of 10, and PATHFINDER IV scored 8.9, significantly better. The difference amounts to saving one life every thousand cases or so. A recent comparison showed that Pathfinder is now outperforming the experts who were consulted during its creation—those experts being some of the world's leading pathologists.

## 15.6 OTHER APPROACHES TO UNCERTAIN REASONING

---

Other sciences (e.g., physics, genetics, economics) have long favored probability as a model for uncertainty. Pierre Laplace said in 1819 that "Probability theory is nothing but common sense reduced to calculation." James Maxwell said in 1850 that "the true logic for this world is the calculus of Probabilities, which takes account of the magnitude of the probability which is, or ought to be, in a reasonable man's mind." Stephen Jay Gould (1994) claimed that "misunderstanding of probability may be the greatest of all general impediments to scientific literacy."

Given this long tradition, it is perhaps surprising that AI has considered many alternatives to probability. The earliest expert systems of the 1970s ignored uncertainty and used strict logical reasoning, but soon it became clear that this was impractical for most real-world domains. The next generation of expert systems (especially in medical domains) used probabilistic techniques. Initial results were promising, but they did not scale up because of the exponential number of probabilities required in the full joint distribution. (Belief net algorithms were not known then.) As a result, probabilistic approaches fell out of favor from roughly 1975 to 1988, and a variety of alternatives were tried for a variety of reasons:

- One common view is that probability theory is essentially numerical, whereas human judgmental reasoning is more "qualitative." Certainly, we are not consciously aware of doing numerical calculations of degrees of belief. (On the other hand, it might be that we have some kind of numerical degrees of belief encoded directly in strengths of connections and activations in our neurons. In that case, the difficulty of conscious access to those strengths is only to be expected.) One should also note that qualitative reasoning mechanisms can be built directly on top of probability theory, so that the "no numbers" argument against probability has little force. Nonetheless, some qualitative schemes have a good deal of appeal in their own right. One of the most well-studied is **default reasoning**, which treats conclusions not as "believed to a certain degree," but as "believed until a better reason is found to believe something else."
- **Rule-based** approaches to uncertainty also have been tried. Such approaches hope to build on the success of logical rule-based systems, but add a sort of "fudge factor" to each rule to accommodate uncertainty. These methods were developed in the mid-1970s, and formed the basis for a large number of expert systems in medicine and other areas.

- One area that we have not addressed so far is the question of **ignorance**, as opposed to uncertainty. Consider flipping a coin. If we know the coin to be fair, then a probability of 0.5 for heads is reasonable. If we know the coin is biased, but we do not know which way, then 0.5 is the only reasonable probability. Obviously, the two cases are different, yet probability seems not to distinguish them. The **Dempster-Shafer theory** uses **interval-valued** degrees of belief to represent an agent's knowledge of the probability of a proposition. Other methods using second-order probabilities are also discussed.
- Probability makes the same ontological commitment as logic: that events are true or false in the world, even if the agent is uncertain as to which is the case. Researchers in **fuzzy logic** have proposed an ontology that allows vagueness: that an event can be "sort of" true. Vagueness and uncertainty are in fact orthogonal issues, as we will see.

The following sections treat each of these approaches in slightly more depth. We will not provide detailed technical material, but we provide references for further study.

## Default reasoning

Commonsense reasoning is often said to involve "jumping to conclusions." For example, when one sees a car parked on the street, one would normally be willing to accept that it has four wheels even though only three are visible. (If you feel that the existence of the fourth wheel is dubious, consider also the question as to whether the three visible wheels are real or merely cardboard facsimiles.) Probability theory can certainly provide a conclusion that the fourth wheel exists with high probability. On the other hand, introspection suggests that the possibility of the car not having four wheels does not even arise *unless some new evidence presents itself*. Thus, it seems that the four-wheel conclusion is reached *by default*, in the absence of any reason to doubt it. If new evidence arrives—for example, if one sees the owner carrying a wheel and notices that the car is jacked up—then the conclusion can be retracted. This kind of reasoning is said to exhibit **nonmonotonicity**, because the set of beliefs does not grow monotonically over time as new evidence arrives. First-order logic, on the other hand, exhibits strict **monotonicity**.

NONMONOTONICITY  
MONOTONICITY  
DEFAULT LOGIC  
NONMONOTONIC LOGIC  
CIRCUMSCRIPTION

Reasoning schemes such as **default logic** (Reiter, 1980), **nonmonotonic logic** (McDermott and Doyle, 1980) and **circumscription** (McCarthy, 1980) are designed to handle reasoning with default rules and retraction of beliefs. Although the technical details of these systems are quite different, they share a number of problematic issues that arise with default reasoning:

- What is the *semantic* status of default rules? If "Cars have four wheels" is false, what does it mean to have it in one's knowledge base? What is a good set of default rules to have? Without a good answer to these questions, default reasoning systems will be nonmodular, and it will be hard to develop a good knowledge engineering methodology.
- What happens when the evidence matches the premises of two default rules with conflicting conclusions? We saw examples of this in the discussion of multiple inheritance in Section 10.6. In some schemes, one can express priorities between rules so that one rule takes precedence. **Specificity preference** is a commonly used form of priority in which a special-case rule takes precedence over the general case. For example, "Three-wheeled cars have three wheels" takes precedence over "Cars have four wheels."

SPECIFICITY  
PREFERENCE

- Sometimes a system may draw a number of conclusions on the basis of a belief that is later retracted. How can a system keep track of which conclusions need to be retracted as a result? Conclusions that have multiple justifications, only some of which have been abandoned, should be retained; whereas those with no remaining justifications should be dropped. These problems have been addressed by **truth maintenance systems**, which are discussed in Section 10.8.
- How can beliefs that have default status be used to make decisions? This is probably the hardest issue for default reasoning. Decisions often involve trade-offs, and one therefore needs to compare the *strength* of belief in the outcomes of different actions. In cases where the same kinds of decisions are being made repeatedly, it is possible to interpret default rules as "threshold probability" statements. For example, the default rule "My brakes are always OK" really means "The probability that my brakes are OK, given no other information, is sufficiently high that the optimal decision is for me to drive without checking them." When the decision context changes—for example, when one is driving a heavily laden truck down a steep mountain road—the default rule suddenly becomes inappropriate, even though there is no new evidence to suggest that the brakes are faulty.

To date, no default reasoning system has successfully addressed all of these issues. Furthermore, most systems are formally undecidable, and very slow in practice. There have been several attempts to subsume default reasoning in a probabilistic system, using the idea that a default rule is basically a conditional probability of  $1 - \epsilon$ . For reasons already mentioned, such an approach is likely to require a full integration of decision making before it fully captures the desirable features of default reasoning.

## Rule-based methods for uncertain reasoning

In addition to monotonicity, logical reasoning systems have three other important properties that probabilistic reasoners lack:

LOCALITY

**0 Locality:** In logical systems, whenever we have a rule of the form  $A \Rightarrow B$ , we can conclude  $B$  given evidence  $A$ , *without worrying about any other rules*. In probabilistic systems, we need to consider *all* of the available evidence.

DETACHMENT

**◊ Detachment:** Once a logical proof is found for a proposition  $B$ , the proposition can be used regardless of how it was derived. That is, it can be **detached** from its justification. In dealing with probabilities, on the other hand, the source of the evidence for a belief is important for subsequent reasoning.

TRUTH-FUNCTIONALITY

**0 Truth-functionality:** In logic, the truth of complex sentences can be computed from the truth of the components. Probability combination does not work this way, except under strong independence assumptions.

These properties confer obvious computational advantages. There have been several attempts to devise uncertain reasoning schemes by attaching degrees of belief to propositions and rules in what is essentially a logical system. This means treating degree of belief as a generalized truth value, in order to retain truth-functionality. That is, each proposition is assigned a degree of belief, and the degree of belief in, say,  $A \vee B$  is a function of the belief in  $A$  and the belief in  $B$ .



The bad news for truth-functional systems is that the properties of *locality*, *detachment*, and *truth-functionality* are simply not appropriate for uncertain reasoning. Let us look at truth-functionality first. Let  $H_1$  be the event of a coin coming up heads on a fair flip, let  $T_1$  be the event of the coin coming up tails on that same flip, and let  $H_2$  be the event of the coin coming up heads on a second flip. Clearly, all three events have the same probability, 0.5, and so a truth-functional system must assign the same belief to the conjunction of any two of them. But we can see that the probability of the conjunction depends on the events themselves, and not just on their probabilities:

$P(A)$	$P(B)$	$P(A \vee B)$
$P(H_1) = 0.5$	$P(H_1) = 0.5$	$P(H_1 \vee H_1) = 0.50$
	$P(T_1) = 0.5$	$P(H_1 \vee T_1) = 1.00$
	$P(H_2) = 0.5$	$P(H_1 \vee H_2) = 0.75$

It gets worse when we chain evidence together. Truth-functional systems have **rules** of the form  $A \mapsto B$ , which allow us to compute the belief in  $B$  as a function of the belief in the rule and the belief in  $A$ . Both forward- and backward-chaining systems can be devised. The belief in the rule is assumed to be constant, and is usually specified by the knowledge engineer, for example,  $A \mapsto_{0.9} B$ .

Consider the wet-grass situation from Section 15.4. If we wanted to be able to do both causal and diagnostic reasoning, we would need the two rules:

$$Rain \mapsto WetGrass$$

and

$$WetGrass \mapsto Rain$$

If we are not careful, these two rules will act in a feedback loop so that evidence for *Rain* increases the belief in *WetGrass*, which in turn increases the belief in *Rain* even more. Clearly, uncertain reasoning systems need to keep track of the paths along which evidence is propagated.

Intercausal reasoning (or explaining away) is also tricky. Consider what happens when we have the two rules:

$$Sprinkler \mapsto WetGrass$$

and

$$WetGrass \mapsto Rain$$

Suppose we see that the sprinkler is on. Chaining forward through our rules, this increases the belief that the grass will be wet, which in turn increases the belief that it is raining. But this is ridiculous: the fact that the sprinkler is on explains away the wet grass, and should *reduce* the belief in rain. In a truth-functional system, the transitively derived rule  $Sprinkler \mapsto Rain$  is unavoidable.

Given these difficulties, how is it possible that truth-functional systems were ever considered useful? The answer lies in restricting the tasks required of them, and in carefully engineering the rule base so that undesirable interactions do not occur. The most famous example of a truth-functional system for uncertain reasoning is the **certainty factors** model, which was developed for the MYCIN medical diagnosis program and widely used in expert systems of the late 1970s and 1980s. Almost all uses of certainty factors involved rule sets that were either purely diagnostic (as in MYCIN) or purely causal. Furthermore, evidence was only entered at the "roots" of the rule set, and most rule sets were singly connected. Heckerman (1986) has shown that under these circumstances, a minor variation on certainty-factor inference was exactly equivalent to Bayesian

inference on polytrees. In other circumstances, certainty factors could yield disastrously incorrect degrees of belief through overcounting of evidence. Details of the method can be found under the "Certainty Factors" entry in *Encyclopedia of AI*, but the use of certainty factors is no longer recommended (even by one of its inventors—see the foreword to Heckerman (1991)).

## Representing ignorance: Dempster–Shafer theory

DEMPSTER-SHAFER

BELIEF FUNCTION

The **Dempster-Shafer** theory is designed to deal with the distinction between **uncertainty and ignorance**. Rather than computing the probability of a proposition, it computes the probability that the evidence supports the proposition. This measure of belief is called a **belief function**, written  $Bel(X)$ .

We return to coin flipping for an example of belief functions. Suppose a shady character comes up to you and offers to bet you \$10 that his coin will come up heads on the next flip. Given that the coin may or may not be fair, what belief should you ascribe to the event of it coming up heads? Dempster-Shafer theory says that because you have no evidence either way, you have to say that the belief  $Bel(Heads) = 0$ , and also that  $Bel(\neg Heads) = 0$ . This makes Dempster-Shafer reasoning systems skeptical in a way that has some intuitive appeal. Now suppose you have an expert at your disposal who testifies with 90% certainty that the coin is fair (i.e., he is 90% sure that  $P(Heads) = 0.5$ ). Then Dempster-Shafer theory gives  $Bel(Heads) = 0.9 \times 0.5 = 0.45$  and likewise  $Bel(\neg Heads) = 0.45$ . There is still a 0.1 "gap" that is not accounted for by the evidence. "Dempster's rule" (Dempster, 1968) shows how to combine evidence to give new values for  $Bel$ , and Shafer's work extends this into a complete computational model.

As with default reasoning, there is a problem in connecting beliefs to actions. With probabilities, decision theory says that if  $P(Heads) = P(\neg Heads) - 0.5$  then (assuming that winning \$10 and losing \$10 are considered equal opposites) the reasoner will be indifferent between the action of accepting and declining the bet. A Dempster-Shafer reasoner has  $Bel(\neg Heads) - 0$ , and thus no reason to accept the bet, but then it also has  $Bel(Heads) - 0$ , and thus no reason to decline it. Thus, it seems that the Dempster-Shafer reasoner comes to the same conclusion about how to act in this case. Unfortunately, Dempster-Shafer theory allows no definite decision in many other cases where probabilistic inference does yield a specific choice. In fact, the notion of utility in the Dempster-Shafer model is not yet well-understood, partly because the semantics of  $Bel$  is not defined precisely with respect to decision making.

One interpretation of Dempster-Shafer theory is that it defines a probability interval—the interval for *Heads* is  $[0, 1]$  before our expert testimony, and  $[0.45, 0.55]$  after. The width of the interval can be a good aid in deciding when we need to acquire more evidence: it can tell you that the expert's testimony will help you if you do not know whether the coin is fair, but will not help you if you have already determined that the coin is fair. In the Bayesian approach, this kind of reasoning can be done easily by examining how much one's belief would change if one were to acquire more evidence. For example, knowing whether the coin is fair would have a significant impact on the belief that it will come up heads. A Bayesian probability therefore has an "implicit" uncertainty associated with the various possible changes that it might undergo as a result of future observations.