The McGraw·Hill Companies

ARTIFICIAL INTELLIGENCE

Third Edition

Elaine Rich
Kevin Knight
Shivashankar B Nair

Mc Graw Hill Education ORIGINAL

Copyrighted material

Tata McGraw-Hill

Copyright © 2009 by Tata McGraw-Hill Publishing Company Limited.

First reprint
DLLYYDDXRCYAR

No part of this publication may be reproduced or distributed in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise or stored in a database or retrieval system without the prior written permission of the publishers. The program listings (if any) may be entered, stored and executed in a computer system, but they may not be reproduced for publication.

This edition can be exported from India only by the publishers,
Tata McGraw-Hill Publishing Company Limited.

ISBN-13: 978-0-07-008770-5
ISBN-10: 0-07-008770-9

Managing Director: *Ajay Shukla*

General Manager: Publishing—SEM & Tech Ed: *Vibha Mahajan*
Sponsoring Editor: *Shalini Jha*
Jr. Sponsoring Editor: *Nalanjan Chakravarty*
Sr. Copy Editor: *Dipika Dey*
Sr. Production Manager: *P L Pandita*

General Manager: Marketing—Higher Education & School: *Michael J Cruz*
Product Manager: SEM & Tech Ed: *Biju Ganesan*

Controller—Production: *Rajender P Ghansela*
Asst. General Manager—Production: *B L Dogra*

Information contained in this work has been obtained by Tata McGraw-Hill, from sources believed to be reliable. However, neither Tata McGraw-Hill nor its authors guarantee the accuracy or completeness of any information published herein, and neither Tata McGraw-Hill nor its authors shall be responsible for any errors, omissions, or damages arising out of use of this information. This work is published with the understanding that Tata McGraw-Hill and its authors are supplying information but are not attempting to render engineering or other professional services. If such services are required, the assistance of an appropriate professional should be sought.

Published by Tata McGraw-Hill Publishing Company Limited,
7 West Patel Nagar, New Delhi 110 008, Typeset at Bukprint India, B-180A, Guru Nanak Pura, Laxmi Nagar, Delhi-110 092 and printed at Gopsons Papers Ltd., Noida 201 301

Cover: Gopsons Papers Ltd.

The McGraw·Hill Companies

Copyrighted material

Contents

Preface to the Third Edition *xiii*
Preface to the Second Edition *xvii*

PART I: PROBLEMS AND SEARCH

1. What is Artificial Intelligence? 1

1.1 The AI Problems *4*
1.2 The Underlying Assumption *6*
1.3 What is an AI Technique? *7*
1.4 The Level of the Model *18*
1.5 Criteria for Success *20*
1.6 Some General References *21*
1.7 One Final Word and Beyond *22*
Exercises *24*

2. Problems, Problem Spaces, and Search 25

2.1 Defining the Problem as a State Space Search *25*
2.2 Production Systems *30*
2.3 Problem Characteristics *36*
2.4 Production System Characteristics *43*
2.5 Issues in the Design of Search Programs *45*
2.6 Additional Problems *47*
Summary *48*
Exercises *48*

3. Heuristic Search Techniques 50

3.1 Generate-and-Test *50*
3.2 Hill Climbing *52*
3.3 Best-first Search *57*
3.4 Problem Reduction *64*
3.5 Constraint Satisfaction *68*
3.6 Means-ends Analysis *72*
Summary *74*
Exercises *75*

PART II: KNOWLEDGE REPRESENTATION

4. Knowledge Representation Issues 79

4.1 Representations and Mappings *79*
4.2 Approaches to Knowledge Representation *82*

Copyrighted material

4.3 Issues in Knowledge Representation *86*
4.4 The Frame Problem *96*
Summary *97*

5. Using Predicate Logic **98**

5.1 Representing Simple Facts in Logic *99*
5.2 Representing Instance and *ISA* Relationships *103*
5.3 Computable Functions and Predicates *105*
5.4 Resolution *108*
5.5 Natural Deduction *124*
Summary *125*
Exercises *126*

6. Representing Knowledge Using Rules **129**

6.1 Procedural Versus Declarative Knowledge *129*
6.2 Logic Programming *131*
6.3 Forward Versus Backward Reasoning *134*
6.4 Matching *138*
6.5 Control Knowledge *142*
Summary *145*
Exercises *145*

7. Symbolic Reasoning Under Uncertainty **147**

7.1 Introduction to Nonmonotonic Reasoning *147*
7.2 Logics for Nonmonotonic Reasoning *150*
7.3 Implementation Issues *157*
7.4 Augmenting a Problem-solver *158*
7.5 Implementation: Depth-first Search *160*
7.6 Implementation: Breadth-first Search *166*
Summary *169*
Exercises *170*

8. Statistical Reasoning **172**

8.1 Probability and Bayes' Theorem *172*
8.2 Certainty Factors and Rule-based Systems *174*
8.3 Bayesian Networks *179*
8.4 Dempster-Shafer Theory *181*
8.5 Fuzzy Logic *184*
Summary *185*
Exercises *186*

9. Weak Slot-and-Filler Structures **188**

9.1 Semantic Nets *188*
9.2 Frames *193*
Exercises *205*

Copyrighted material

10. Strong Slot-and-Filler Structures **207**

10.1 Conceptual Dependency *207*
10.2 Scripts *212*
10.3 CYC *216*
Exercises *220*

11. Knowledge Representation Summary **222**

11.1 Syntactic-semantic Spectrum of Representation *222*
11.2 Logic and Slot-and-filler Structures *224*
11.3 Other Representational Techniques *225*
11.4 Summary of the Role of Knowledge *227*
Exercises *227*

PART III: ADVANCED TOPICS

12. Game Playing **231**

12.1 Overview *231*
12.2 The Minimax Search Procedure *233*
12.3 Adding Alpha-beta Cutoffs *236*
12.4 Additional Refinements *240*
12.5 Iterative Deepening *242*
12.6 References on Specific Games *244*
Exercises *246*

13. Planning **247**

13.1 Overview *247*
13.2 An Example Domain: The Blocks World *250*
13.3 Components of a Planning System *250*
13.4 Goal Stack Planning *255*
13.5 Nonlinear Planning Using Constraint Posting *262*
13.6 Hierarchical Planning *268*
13.7 Reactive Systems *269*
13.8 Other Planning Techniques *269*
Exercises *270*

14. Understanding **272**

14.1 What is Understanding? *272*
14.2 What Makes Understanding Hard? *273*
14.3 Understanding as Constraint Satisfaction *278*
Summary *283*
Exercises *284*

15. Natural Language Processing **285**

15.1 Introduction *286*
15.2 Syntactic Processing *291*

Copyrighted material

15.3 Semantic Analysis *300*
15.4 Discourse and Pragmatic Processing *313*
15.5 Statistical Natural Language Processing *321*
15.6 Spell Checking *325*
Summary *329*
Exercises *331*

16. Parallel and Distributed AI **333**

16.1 Psychological Modeling *333*
16.2 Parallelism in Reasoning Systems *334*
16.3 Distributed Reasoning Systems *336*
Summary *346*
Exercises *346*

17. Learning **347**

17.1 What is Learning? *347*
17.2 Rote Learning *348*
17.3 Learning by Taking Advice *349*
17.4 Learning in Problem-solving *351*
17.5 Learning from Examples: Induction *355*
17.6 Explanation-based Learning *364*
17.7 Discovery *367*
17.8 Analogy *371*
17.9 Formal Learning Theory *372*
17.10 Neural Net Learning and Genetic Learning *373*
Summary *374*
Exercises *375*

18. Connectionist Models **376**

18.1 Introduction: Hopfield Networks *377*
18.2 Learning in Neural Networks *379*
18.3 Applications of Neural Networks *396*
18.4 Recurrent Networks *399*
18.5 Distributed Representations *400*
18.6 Connectionist AI and Symbolic AI *403*
Exercises *405*

19. Common Sense **408**

19.1 Qualitative Physics *409*
19.2 Common Sense Ontologies *411*
19.3 Memory Organization *417*
19.4 Case-based Reasoning *419*
Exercises *421*

Copyrighted material

20. Expert Systems **422**

20.1 Representing and Using Domain Knowledge *422*
20.2 Expert System Shells *424*
20.3 Explanation *425*
20.4 Knowledge Acquisition *427*
Summary *429*
Exercises *430*

21. Perception and Action **431**

21.1 Real-time Search *433*
21.2 Perception *434*
21.3 Action *438*
21.4 Robot Architectures *441*
Summary *443*
Exercises *443*

22. Fuzzy Logic Systems **445**

22.1 Introduction *445*
22.2 Crisp Sets *445*
22.3 Fuzzy Sets *446*
22.4 Some Fuzzy Terminology *446*
22.5 Fuzzy Logic Control *447*
22.6 Sugeno Style of Fuzzy Inference Processing *453*
22.7 Fuzzy Hedges *454*
22.8 α Cut Threshold *454*
22.9 Neuro Fuzzy Systems *455*
22.10 Points to Note *455*
Exercises *456*

23. Genetic Algorithms: Copying Nature's Approaches **457**

23.1 A Peek into the Biological World *457*
23.2 Genetic Algorithms (GAs) *458*
23.3 Significance of the Genetic Operators *470*
23.4 Termination Parameters *471*
23.5 Niching and Speciation *471*
23.6 Evolving Neural Networks *472*
23.7 Theoretical Grounding *474*
23.8 Ant Algorithms *476*
23.9 Points to Ponder *477*
Exercises *478*

24. Artificial Immune Systems **479**

24.1 Introduction *479*
24.2 The Phenomenon of Immunity *479*
24.3 Immunity and Infection *480*

Copyrighted material

24.4 The Innate Immune System—The First Line of Defence *480*
24.5 The Adaptive Immune System—The Second Line of Defence *481*
24.6 Recognition *483*
24.7 Clonal Selection *484*
24.7 Learning *485*
24.8 Immune Network Theory *485*
24.9 Mapping Immune Systems to Practical Applications *486*
24.10 Other Applications *493*
24.11 Points to Ponder *493*
Exercises *493*
Glossary of Biological Terms Used *494*

25. Prolog—The Natural Language of Artificial Intelligence **496**

25.1 Introduction *496*
25.2 Converting English to Prolog Facts and Rules *496*
25.3 Goals *497*
25.4 Prolog Terminology *499*
25.5 Variables *499*
25.6 Control Structures *500*
25.7 Arithmetic Operators *500*
25.8 Matching in Prolog *502*
25.9 Backtracking *503*
25.10 Cuts *505*
25.11 Recursion *506*
25.12 Lists *508*
25.13 Dynamic Databases *512*
25.14 Input/Output and Streams *515*
25.15 Some Aspects Specific to LPA Prolog *516*
Summary *528*
Exercises *528*

26. Conclusion **529**

26.1 Components of an AI Program *529*
26.2 AI Skeptics—An Open Argument *529*
Exercises *530*

References **533**
Author Index **553**
Subject Index **558**

Copyrighted material

PREFACE TO THE THIRD EDITION

With the Internet and the World Wide Web penetrating all walks of life, the field of Artificial Intelligence (AI) has of late, seen a resurgence and an upward trend in the last decade. The earlier edition did cater well to the needs of a basic course in AI and still does. But the need to add areas which have slowly and unknowingly glued on to the field of AI has forced the birth of this new edition. While almost all the contents of the previous edition have been retained, this book has been augmented by specific chapters describing the newer areas that have found a variety of uses in a gamut of domains. It is envisaged that these areas, coupled with the classical AI models should definitely motivate the reader to generate novel ideas for AI-based application scenarios.

Four new chapters find their way into this new edition, namely,

1. Fuzzy Logic Systems
2. Genetic Algorithms: Copying Nature's Approaches
3. Artificial Immune Systems
4. PROLOG—The Natural Language of Artificial Intelligence

Chapter 22 entitled Fuzzy Logic Systems, highlights aspects of the conventional crisp sets and introduces the concept of fuzzy sets. It describes the design methodology of a fuzzy air cooler through a real-world application. Some relevant aspects of how memberships can be tweaked (Fuzzy Hedges) are also described. A fuzzy system augmented with neural technology to achieve neuro-fuzzy control is explained in this chapter.

Chapter 23 deals with Nature's own method of optimization and is thus entitled Genetic Algorithms: Copying Nature's Approaches. It elucidates the use of Genetic Algorithms (GA) in optimization and learning. A typical case of optimization with constraints using the example of allocating employees based on their skills has been described. As a tail-ender to this section, the reader is made aware of the ways to juggle the algorithm so as to avoid getting trapped in a local optimum. Further, the use of GAs to evolve neural networks is also explained. The chapter also discusses the Schema theorem—the basis for this algorithm. Though not directly related to GAs, the chapter concludes by igniting the reader's interest on Ant algorithms.

Chapter 24 discusses a relatively new and emerging area of bio-inspired systems—the Artificial Immune System (AIS). It may be worth mentioning that the field by itself, to date, has seen only seven international conferences; the last one (ICARIS 2008) was held at Phuket, Thailand in August 2008. The chapter initially takes the reader on a tour of the general aspects of immunology. Since most of the readers are new to this field, this part has been emphasised in as technical a way as was possible. Two of the major theories—The Clonal Selection and the Network or Idiotypic theory—in immunology have been described. Aspects on how one can go about modelling an Artificial Immune System based on its natural counterpart have also been explained. Two diverse real-world applications conclude the chapter. A special glossary of terms is appended to ease the comprehension of specific biological terms used therein.

Chapter 25 deals with PROLOG—The Natural Language of Artificial Intelligence. The chapter has been included to assist the reader in daring to implement AI-based applications. The previous edition did not give much thought to a language and assumed that the reader would write programs to see how well the content of the book works in the real world. Implementing AI to comprehend its feasibility in the real world is of utmost importance. History stands witness to the initial failures, thanks to the notion that AI-based techniques once coded could be easily bolted onto existing general-purpose hardware. Robotics is one domain where such failures were prominently seen. Resources in terms of computing speed, memory, time and speed of the

Copyrighted material

actuating device—all play a vital role in embedding AI into the real world. This chapter is thus aimed at instilling confidence and motivating the reader (student) to implement what he has learnt from the book. It describes the use of Prolog and its syntax using typical programs. The new world is the world of intelligent agents. This chapter describes LPA Prolog's Chimera Agent Engine and its use to realise a multi-agent scenario. Readers can now use the AI methodologies discussed in this book to tweak and make these agent programs intelligent.

Apart from the addition of these four chapters, some changes have been made to a few existing ones of the older edition.

Chapter 1 now talks of Stevan Harnad's point of *Symbol Grounding*—trying to put down the concept of a symbol in the *Physical Symbol System Hypothesis*. Some arguments are provided to enlighten the user that the hypothesis does not seem to be the ultimate. **Chapter 15** on Natural Language Processing has now two sections, one describing Statistical Approaches to Language Processing and another on Spell Checking. These sections help the reader to think beyond just grammar-based language-processing systems and explore techniques which seem more akin to how human beings learn a language. **Chapter 18** on Connectionist Models has an additional modest section on Kohonen Networks.

The new edition caters broadly to undergraduate students in Computer Science who are new to the concept of AI. AI novices are encouraged to read Parts I and II and also **Chapter 25** to initially aid them commence experimentation in the laboratory. Postgraduate students will especially benefit from Part III and the new chapters added therein. Laboratory implementations of these could be worked out either in Prolog or some other familiar language. AI is not a field confined to Computer Science alone; its applications find their way into a variety of domains. Thus, those engaged in interdisciplinary studies will find possibilities of new approaches to a wide variety of research-oriented programmes in this edition.

The **Online Learning Center** for this book includes a wealth of supplements for learning and teaching.

For the Student

- Source Code for the GA program described in Chapter 23
- Source Code for PROLOG programs in Chapter 25

For the Instructor

- PowerPoint Slides
- Solutions to the Exercises

ACKNOWLEDGEMENTS

While the acknowledgements made by the authors of the previous edition stand for all purposes, I venture to add some of mine for the additions made.

The making of this revised edition has been a slightly long-drawn affair, thanks to so many eventful and uneventful situations I went through during the last three years. I must thank the publishers, McGraw-Hill Education India, for having been extremely understanding and allowing me all the time to complete this edition.

From the technical perspective, I thank the previous authors Elaine Rich and Kevin Wright for their valuable comments in the making of this book. Pradip K Das of my department has been kind to spare his valuable time, both in-flights and out, to help me with his invaluable statistical and logical inputs. There have been long-drawn discussions, especially on fuzzy logic and statistical language processing, for which I sincerely thank him. The GA program described for skill allocation is adapted from an AI project assignment carried out by my ex-student Dipti Gagneja, currently pursuing her doctoral degree in Computational Biology and Bioinformatics at the University of Southern California. She has, at my behest, spent a considerable amount of time at the Indian Institute of Technology Guwahati, to see that the related figures and graphs portray the significance of the problem. I thank her too for this. I also owe my gratitude to a number of my students who have contributed in some way or the other in the making of this edition.

Copyrighted material

Had my late mother been with me for a little longer, this book would have been published way back in 2006. A retired professor in English, she had promised to edit and review this edition. Hers, as also my father's spirits, have yet finally coaxed me into finishing this book. I thank my sisters for having encouraged and supported me during bouts of uncertainty.

My wife, Priya, and son, Jayprakash, have and continue to be my driving forces and thus figure in some *logical* form in this book. While the former has always found time to crack what I have written from the point of view of a third party, the latter has been the source of my AI-based ideas! Thanking them would thus never be enough. I also thank my in-laws for all the assistance they provided in the making of this book.

Shivashankar B Nair

A note of acknowledgement is due to the following reviewers who have contributed to the shaping of the book by providing their valuable suggestions.

R C Joshi	Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, Roorkee
Kamlesh Dutta	Department of Computer Science, National Institute of Technology, Hamirpur
R P Arora	Department of Computer Science and Engineering, Dehradoon Institute of Technology, Dehradoon
J P Singh	Department of Information Technology, Academy of Technology, Hooghly
D R Desai	Department of Computer Science and Engineering, Modern Engineering College, Pune
Kishore Bhoyar	Department of Computer Technology/IT, Yeshwantrao Chavan College of Engineering, Nagpur
L M R J Lobo	Department of Computer Science and Engineering, Walchand Institute of Technology, Sangli
S Fatima	Department of Computer Science and Engineering, College of Engineering, Osmania University, Hyderabad
S V Gangashetty	Language Technology Research Center, Indian Institute of Information Technology, Hyderabad
Kamalini Martin	Department of Electrical Sciences Karunya Institute of Technology and Sciences, Coimbatore

Copyrighted material

PREFACE TO THE SECOND EDITION

In the years since the first edition of this book appeared, Artificial Intelligence (AI) has grown from a small-scale laboratory science into a technological and industrial success. We now possess an arsenal of techniques for creating computer programs that control manufacturing processes, diagnose computer faults and human diseases, design computers, do insurance underwriting, play grandmaster-level chess, and so on. Basic research in AI has expanded enormously during this period. For the student, extracting theoretical and practical knowledge from such a large body of scientific knowledge is a daunting task. The goal of the first edition of this book was to provide a readable introduction to the problems and techniques of AI. In this edition, we have tried to achieve the same goal for the expanded field that AI has become. In particular, we have tried to present both the theoretical foundations of AI and an indication of the ways that current techniques can be used in application programs.

As a result of this effort, the book has grown. It is probably no longer possible to cover everything in a single semester. Because of this, we have structured the book so that an instructor can choose from a variety of paths through the chapters. The book is divided into three parts:

Part I. Problems and Search.
Part II. Knowledge Representation.
Part III. Advanced Topics.

Part I introduces AI by examining the nature of the difficult problems that AI seeks to solve. It then develops the theory and practice of heuristic search, providing detailed algorithms for standard search methods, including best-first search, hill climbing, simulated annealing, means-ends analysis, and constraint satisfaction.

The last thirty years of AI have demonstrated that intelligence requires more than the ability to reason. It also requires a great deal of knowledge about the world. So Part II explores a variety of methods for encoding knowledge in computer systems. These methods include predicate logic, production rules, semantic networks, frames, and scripts. There are also chapters on both symbolic and numeric techniques for reasoning under uncertainty. In addition, we present some very specific frameworks in which particular commitments to a set of representational primitives are made.

Parts I and II should be covered in any basic course in AI. They provide the foundation for the advanced topics and applications that are presented in Part III. While the chapters in Parts I and II should be covered in order since they build on each other, the chapters in Part III are, for the most part, independent and can be covered in almost any combination, depending on the goals of a particular course. The topics that are covered include: game playing, planning, understanding, natural language processing (which depends on the understanding chapter), parallel and distributed AI (which depends on planning and natural language), learning, connectionist models, common sense, expert systems, and perception and action.

To use this book effectively, students should have some background in both computer science and mathematics. As computer science background, they should have experience programming and they should feel comfortable with the material in an undergraduate data structures course. They should be familiar with the use of recursion as a program control structure. And they should be able to do simple analyses of the time complexity of algorithms. As mathematical background, students should have the equivalent of an undergraduate coarse in logic, including predicate logic with quantifiers and the basic notion of a decision procedure.

Copyrighted material

This book contains, spread throughout it, many references to the AI research literature. These references are important for two reasons. First, they make it possible for the student to pursue individual topics in greater depth than is possible within the space restrictions of this book. This is the common reason for including references in a survey text. The second reason that these references have been included is more specific to the content of this book. AI is a relatively new discipline. In many areas of the field there is still pot complete agreement on how things should be done. The references to the source literature guarantee that students have access not just to one approach, but to as many as possible of those whose eventual success still needs to be determined by further research, both theoretical and empirical.

Since the ultimate goal of AI is the construction of programs that solve hard problems, no study of AI is complete without some experience writing programs. Most AI programs are written in LISP, PROLOG, or some specialized AI shell. Recently though, as AI has spread out into the mainstream computing world, AI programs are being written in a wide variety of programming languages. The algorithms presented in this book are described in sufficient detail to enable students to exploit them in their programs, but they are not expressed in code. This book should probably be supplemented with a good book on whatever language is being used for programming in the course.

This book would not have happened without the help of many people. The content of the manuscript has been greatly improved by the comments of Srinivas Akella, Jim Blevins, Clay Bridges, R. Martin Chavez, Alan Cline, Adam Farquar, Anwar Ghuloum, Yolanda Gil, R. V. Guha, Lucy Hadden, Ajay Jain, Craig Knoblock, John Laird, Clifford Mercer, Michael Newton, Charles Petrie, Robert Rich, Steve Shafer, Reid Simmons, Herbert Simon, Munindar Singh, Milind Tambe, David Touretzky, Manuela Veloso, David Wroblewski, and Marco Zagha.

Special thanks to Yolanda Gil and Alan Cline for help above and beyond. Yolanda kept the project going under desperate circumstances, and Alan spent innumerable hours designing the cover and bringing it into the world. We hank them for these things and much, much more.

Linda Mitchell helped us put together many draft editions along the way. Some of those drafts were used in actual courses, where students found innumerable bugs for us. We would like to thank them as well as their instructors, Tom Mitchell and Jean Scholtz. Thanks also to Don Speray for his help in producing the cover.

David Shapiro and Joe'Murphy deserve credit for superb editing, and for keeping us on schedule.

We would also like to thank Nicole Vecchi for her wisdom and patience in the world of high resolution printing. Thanks to David Long and Lily Mummert for pointing us to the right fonts.

Thanks to the following reviewers for their comments: Yigal Arens, University of Southern California; Jaime Carbonell, Carnegie Mellon University; Charles Dyer, University of Wisconsin, Madison; George Ernst, Case Western Reserve University; Pat Langley, University of California, Irvine; Brian Schunck, University of Michigan; and James Slagle, University of Minnesota.

Carnegie Mellon University and MCC provided us the environment in which we could write and produce this book. We would like to thank our colleagues, particularly Jim Barnett and Masaru Tomita, for putting up with us while we were writing this book instead of doing the other things we were supposed to be doing.

Elaine Rich
Kevin Knight

Copyrighted material

PART I

PROBLEMS AND SEARCH

Copyrighted material

Copyrighted material

CHAPTER

1

WHAT IS ARTIFICIAL INTELLIGENCE?

There are three kinds of intelligence: one kind understands things for itself, the other appreciates what others can understand, the third understands neither for itself nor through others. This first kind is excellent, the second good, and the third kind useless.

—Niccolo Machiavelli
(1469–1527), Italian diplomat, political philosopher,
musician, poet and playwright

What exactly is artificial intelligence? Although most attempts to define complex and widely used terms precisely are exercises in futility, it is useful to draw at least an approximate boundary around the concept to provide a perspective on the discussion that follows. To do this, we propose the following by no means universally accepted definition. *Artificial intelligence* (AI) is the study of how to make computers do things which, at the moment, people do better. This definition is, of course, somewhat ephemeral because of its reference to the current state of computer science. And it fails to include some areas of potentially very large impact, namely problems that cannot now be solved well by either computers or people. But it provides a good outline of what constitutes artificial intelligence, and it avoids the philosophical issues that dominate attempts to define the meaning of either *artificial* or *intelligence*. Interestingly, though, it suggests a similarity with philosophy at the same time it is avoiding it. Philosophy has always been the study of those branches of knowledge that were so poorly understood that they had not yet become separate disciplines in their own right. As fields such as mathematics or physics became more advanced, they broke off from philosophy. Perhaps if AI succeeds it can reduce itself to the empty set. As on date this has not happened. There are signs which seem to suggest that the newer off-shoots of AI together with their real world applications are gradually overshadowing it. As AI migrates to the real world we do not seem to be satisfied with just a computer playing a chess game. Instead we wish a robot would sit opposite to us as an opponent, visualize the real board and make the right moves in this physical world. Such notions seem to push the definitions of AI to a greater extent. As we read on, there will be always that lurking feeling that the definitions propounded so far are not adequate. Only what we finally achieve in the future will help us propound an apt definition for AI! The feeling of intelligence is a mirage, if you achieve it, it ceases to make you feel so. As somebody has aptly put it – AI is *Artificial Intelligence* till it is achieved; after which the acronym reduces to *Already Implemented*.

Copyrighted material

One must also appreciate the fact that comprehending the concept of AI also aids us in understanding how natural intelligence works. Though a complete comprehension of its working may remain a mirage, the very attempt will definitely assist in unfolding mysteries one by one.

1.1 THE AI PROBLEMS

What then are some of the problems contained within AI? Much of the early work in the field focused on formal tasks, such as game playing and theorem proving. Samuel wrote a checkers-playing program that not only played games with opponents but also used its experience at those games to improve its later performance. Chess also received a good deal of attention. The Logic Theorist was an early attempt to prove mathematical theorems. It was able to prove several theorems from the first chapter of Whitehead and Russell's *Principia Mathematica*. Gelernter's theorem prover explored another area of mathematics: geometry. Game playing and theorem proving share the property that people who do them well are considered to be displaying intelligence. Despite this, it appeared initially that computers could perform well at those tasks simply by being fast at exploring a large number of solution paths and then selecting the best one. It was thought that this process required very little knowledge and could therefore be programmed easily. As we will see later, this assumption turned out to be false since no computer is fast enough to overcome the combinatorial explosion generated by most problems.

Another early foray into AI focused on the sort of problem solving that we do every day when we decide how to get to work in the morning, often called *commonsense reasoning*. It includes reasoning about physical objects and their relationships to each other (e.g., an object can be in only one place at a time), as well as reasoning about actions and their consequences (e.g., if you let go of something, it will fall to the floor and maybe break). To investigate this sort of reasoning, Newell, Shaw, and Simon built the General Problem Solver (GPS), which they applied to several commonsense tasks as well as to the problem of performing symbolic manipulations of logical expressions. Again, no attempt was made to create a program with a large amount of knowledge about a particular problem domain. Only simple tasks were selected.

As AI research progressed and techniques for handling larger amounts of world knowledge were developed, some progress was made on the tasks just described and new tasks could reasonably be attempted. These include perception (vision and speech), natural language understanding, and problem solving in specialized domains such as medical diagnosis and chemical analysis.

Perception of the world around us is crucial to our survival. Animals with much less intelligence than people are capable of more sophisticated visual perception than are current machines. Perceptual tasks are difficult because they involve analog (rather than digital) signals; the signals are typically very noisy and usually a large number of things (some of which may be partially obscuring others) must be perceived at once. The problems of perception are discussed in greater detail in Chapter 21.

The ability to use language to communicate a wide variety of ideas is perhaps the most important thing that separates humans from the other animals. The problem of understanding spoken language is a perceptual problem and is hard to solve for the reasons just discussed. But suppose we simplify the problem by restricting it to written language. This problem, usually referred to as *natural language understanding*, is still extremely difficult. In order to understand sentences about a topic, it is necessary to know not only a lot about the language itself (its vocabulary and grammar) but also a good deal about the topic so that unstated assumptions can be recognized. We discuss this problem again later in this chapter and then in more detail in Chapter 15.

In addition to these mundane tasks, many people can also perform one or maybe more specialized tasks in which carefully acquired expertise is necessary. Examples of such tasks include engineering design, scientific discovery, medical diagnosis, and financial planning. Programs that can solve problems in these domains also fall under the aegis of artificial intelligence. Figure 1.1 lists some of the tasks that are the targets of work in AI.

Copyrighted material

A person who knows how to perform tasks from several of the categories shown in the figure learns the necessary skills in a standard order. First, perceptual, linguistic, and commonsense skills are learned. Later (and of course for some people, never) expert skills such as engineering, medicine, or finance are acquired. It might seem to make sense then that the earlier skills are easier and thus more amenable to computerized duplication than are the later, more specialized ones. For this reason, much of the initial AI work was concentrated in those early areas. But it turns out that this naive assumption is not right. Although expert skills require knowledge that many of us do not have, they often require much less knowledge than do the more mundane skills and that knowledge is usually easier to represent and deal with inside programs.

Mundane Tasks

- Perception
  - Vision
  - Speech
- Natural language
  - Understanding
  - Generation
  - Translation
- Commonsense reasoning
- Robot control

Formal Tasks

- Games
  - Chess
  - Backgammon
  - Checkers -Go
- Mathematics
  - Geometry
  - Logic
  - Integral calculus
  - Proving properties of programs

Expert Tasks

- Engineering
  - Design
  - Fault finding
  - Manufacturing planning
- Scientific analysis
- Medical diagnosis
- Financial analysis

Fig. 1.1 *Some of the Task Domains of Artificial Intelligence*

As a result, the problem areas where AI is now flourishing most as a practical discipline (as opposed to a purely research one) are primarily the domains that require only specialized expertise without the assistance of commonsense knowledge. There are now thousands of programs called *expert systems* in day-to-day operation throughout all areas of industry and government. Each of these systems attempts to solve part, or perhaps all, of a practical, significant problem that previously required scarce human expertise. In Chapter 20 we examine several of these systems and explore techniques for constructing them.

Copyrighted material

Before embarking on a study of specific AI problems and solution techniques, it is important at least to discuss, if not to answer, the following four questions:

1. What are our underlying assumptions about intelligence?
2. What kinds of techniques will be useful for solving AI problems?
3. At what level of detail, if at all, are we trying to model human intelligence?
4. How will we know when we have succeeded in building an intelligent program?

The next four sections of this chapter address these questions. Following that is a survey of some AI books that may be of interest and a summary of the chapter.

1.2 THE UNDERLYING ASSUMPTION

At the heart of research in artificial intelligence lies what Newell and Simon [1976] call the *physical symbol system hypothesis.* They define a physical symbol system as follows:

> A physical symbol system consists of a set of entities, called symbols, which are physical patterns that can occur as components of another type of entity called an expression (or symbol structure). Thus, a symbol structure is composed of a number of instances (or tokens) of symbols related in some physical way (such as one token being next to another). At any instant of time the system will contain a collection of these symbol structures. Besides these structures, the system also contains a collection of processes that operate on expressions to produce other expressions: processes of creation, modification, reproduction and destruction. A physical symbol system is a machine that produces through time an evolving collection of symbol structures. Such a system exists in a world of objects wider than just these symbolic expressions themselves.

They then state the hypothesis as

> *The Physical Symbol System Hypothesis.* A physical symbol system has the necessary and sufficient means for general intelligent action.

This hypothesis is only a hypothesis. There appears to be no way to prove or disprove it on logical grounds. So it must be subjected to empirical validation. We may find that it is false. We may find that the bulk of the evidence says that it is true. But the only way to determine its truth is by experimentation.

Computers provide the perfect medium for this experimentation since they can be programmed to simulate any physical symbol system we like. This ability of computers to serve as arbitrary symbol manipulators was noticed very early in the history of computing. Lady Lovelace made the following observation about Babbage's proposed Analytical Engine in 1842.

> The operating mechanism can even be thrown into action independently of any object to operate upon (although of course no result could then be developed). Again, it might act upon other things besides numbers, were objects found whose mutual fundamental relations could be expressed by those of the abstract science of operations, and which should be also susceptible of adaptations to the action of the operating notation and mechanism of the engine. Supposing, for instance, that the fundamental relations of pitched sounds in the science of harmony and of musical composition were susceptible of such expression and adaptations, the engine might compose elaborate and scientific pieces of music of any degree of complexity or extent. [Lovelace, 1961]

As it has become increasingly easy to build computing machines, so it has become increasingly possible to conduct empirical investigations of the physical symbol system hypothesis. In each such investigation, a particular task that might be regarded as requiring intelligence is selected. A program to perform the task is proposed and then tested. Although we have not been completely successful at creating programs that perform

Copyrighted material

all the selected tasks, most scientists believe that many of the problems that have been encountered will ultimately prove to be surmountable by more sophisticated programs than we have yet produced.

Evidence in support of the physical symbol system hypothesis has come not only from areas such as game playing, where one might most expect to find it, but also from areas such as visual perception, where it is more tempting to suspect the influence of subsymbolic processes. However, subsymbolic models (for example, neural networks) are beginning to challenge symbolic ones at such low-level tasks. Such models are discussed in Chapter 18. Whether certain subsymbolic models conflict with the physical symbol system hypothesis is a topic still under debate (e.g., Smolensky [1988]). And it is important to note that even the success of subsymbolic systems is not necessarily evidence against the hypothesis. It is often possible to accomplish a task in more than one way.

One interesting attempt to reduce a particularly human activity, the understanding of jokes, to a process of symbol manipulation is provided in the book *Mathematics and Humor* [Paulos, 1980]. It is, of course, possible that the hypothesis will turn out to be only partially true. Perhaps physical symbol systems will prove able to model some aspects of human intelligence and not others. Only time and effort will tell.

The importance of the physical symbol system hypothesis is twofold. It is a significant theory of the nature of human intelligence and so is of great interest to psychologists. It also forms the basis of the belief that it is possible to build programs that can perform intelligent tasks now performed by people. Our major concern here is with the latter of these implications, although as we will soon see, the two issues are not unrelated.

1.3 WHAT IS AN AI TECHNIQUE?

Artificial intelligence problems span a very broad spectrum. They appear to have very little in common except that they are hard. Are there any techniques that are appropriate for the solution of a variety of these problems? The answer to this question is yes, there are. What, then, if anything, can we say about those techniques besides the fact that they manipulate symbols? How could we tell if those techniques might be useful in solving other problems, perhaps ones not traditionally regarded as AI tasks? The rest of this book is an attempt to answer those questions in detail. But before we begin examining closely the individual techniques, it is enlightening to take a broad look at them to see what properties they ought to possess.

One of the few hard and fast results to come out of the first three decades of AI research is that *intelligence requires knowledge*. To compensate for its one overpowering asset, indispensability, knowledge possesses some less desirable properties, including:

- It is voluminous.
- It is hard to characterize accurately.
- It is constantly changing.
- It differs from data by being organized in a way that corresponds to the ways it will be used.

So where does this leave us in our attempt to define AI techniques? We are forced to conclude that an AI technique is a method that exploits knowledge that should be represented in such a way that:

- The knowledge captures generalizations. In other words, it is not necessary to represent separately each individual situation. Instead, situations that share important properties are grouped together. If knowledge does not have this property, inordinate amounts of memory and updating will be required. So we usually call something without this property "data" rather than knowledge.
- It can be understood by people who must provide it. Although for many programs, the bulk of the data can be acquired automatically (for example, by taking readings from a variety of instruments), in many AI domains, most of the knowledge a program has must ultimately be provided by people in terms they understand.

Copyrighted material

- It can easily be modified to correct errors and to reflect changes in the world and in our world view.
- It can be used in a great many situations even if it is not totally accurate or complete.
- It can be used to help overcome its own sheer bulk by helping to narrow the range of possibilities that must usually be considered.

Although AI techniques must be designed in keeping with these constraints imposed by AI problems, there is some degree of independence between problems and problem-solving techniques. It is possible to solve AI problems without using AI techniques (although, as we suggested above, those solutions are not likely to be very good). And it is possible to apply AI techniques to the solution of non-AI problems. This is likely to be a good thing to do for problems that possess many of the same characteristics as do AI problems. In order to try to characterize AI techniques in as problem-independent a way as possible, let's look at two very different problems and a series of approaches for solving each of them.

1.3.1 Tic-Tac-Toe

In this section, we present a series of three programs to play tic-tac-toe. The programs in this series increase in:

- Their complexity
- Their use of generalizations
- The clarity of their knowledge
- The extensibility of their approach. Thus, they move toward being representations of what we call AI techniques.

Program 1

Data Structures

Board — A nine-element vector representing the board, where the elements of the vector correspond to the board positions as follows:

1	2	3
4	5	6
7	8	9

An element contains the value 0 if the corresponding square is blank, 1 if it is filled with an X, or 2 if it is filled with an O.

Movetable — A large vector of 19,683 elements (3^9), each element of which is a nine-element vector. The contents of this vector are chosen specifically to allow the algorithm to work.

The Algorithm

To make a move, do the following:

1. View the vector Board as a ternary (base three) number. Convert it to a decimal number.
2. Use the number computed in step 1 as an index into Movetable and access the vector stored there.
3. The vector selected in step 2 represents the way the board will look after the move that should be made. So set Board equal to that vector.

Comments

This program is very efficient in terms of time. And, in theory, it could play an optimal game of tic-tac-toe. But it has several disadvantages:

Copyrighted material

- It takes a lot of space to store the table that specifies the correct move to make from each board position.
- Someone will have to do a lot of work specifying all the entries in the movetable.
- It is very unlikely that all the required movetable entries can be determined and entered without any errors.
- If we want to extend the game, say to three dimensions, we would have to start from scratch, and in fact this technique would no longer work at all, since 3^{27} board positions would have to be stored, thus overwhelming present computer memories.

The technique embodied in this program does not appear to meet any of our requirements for a good AI technique. Let's see if we can do better.

Program 2

Data Structures

Board	A nine-element vector representing the board, as described for Program 1. But instead of using the numbers 0,1, or 2 in each element, we store 2 (indicating blank), 3 (indicating X), or 5 (indicating O).
Turn	An integer indicating which move of the game is about to be played; 1 indicates the first move, 9 the last.

The Algorithm

The main algorithm uses three subprocedures:

Make 2	Returns 5 if the center square of the board is blank, that is, if Board[5] = 2. Otherwise, this function returns any blank noncorner square (2, 4, 6, or 8).
Posswin(*p*)	Returns 0 if player *p* cannot win on his next move; otherwise, it returns the number of the square that constitutes a winning move. This function will enable the program both to win and to block the opponent's win. Posswin operates by checking, one at a time, each of the rows, columns, and diagonals. Because of the way values are numbered, it can test an entire row (column or diagonal) to see if it is a possible win by multiplying the values of its squares together. If the product is 18 ($3 \times 3 \times 2$), then X can win. If the product is 50 ($5 \times 5 \times 2$), then O can win. If we find a winning row, we determine which element is blank, and return the number of that square.
Go(*n*)	Makes a move in square *n*. This procedure sets Board[*n*] to 3 if Turn is odd, or 5 if Turn is even. It also increments Turn by one.

The algorithm has a built-in strategy for each move it may have to make. It makes the odd-numbered moves if it is playing X, the even-numbered moves if it is playing O. The strategy for each turn is as follows:

Turn=1	Go(1) (upper left corner).
Turn=2	If Board[5] is blank, Go(5), else Go(1).
Turn=3	If Board[9] is blank, Go(9), else Go(3).
Turn=4	If Posswin(X) is not 0, then Go(Posswin(X)) [i.e., block opponent's win], else Go(Make2).
Turn=5	If Posswin(X) is not 0 then Go(Posswin(X)) [i.e., win] else if Posswin(O) is not 0, then Go(Posswin(O)) [i.e., block win], else if Board[7] is blank, then Go(7), else Go(3). [Here the program is trying to make a fork.]

Copyrighted material

Turn=6	If Posswin(O) is not 0 then Go (Posswin(O)), else if Posswin(X) is not 0, then Go(Posswin(X)), else Go(Make2).
Turn=7	If Posswin(X) is not 0 then Go(Posswin(X)), else if Posswin(O) is not 0, then Go(Posswin(O)), else go anywhere that is blank.
Turn=8	If Posswin(O) is not 0 then Go(Posswin(O)), else if Posswin(X) is not 0, then Go(Posswin(X»), else go anywhere that is blank.
Turn=9	Same as Turn=7.

Comments

This program is not quite as efficient in terms of time as the first one since it has to check several conditions before making each move. But it is a lot more efficient in terms of space. It is also a lot easier to understand the program's strategy or to change the strategy if desired. But the total strategy has still been figured out in advance by the programmer. Any bugs in the programmer's tic-tac-toe playing skill will show up in the program's play. And we still cannot generalize any of the program's knowledge to a different domain, such as three-dimensional tic-tac-toe.

Program 2′

This program is identical to Program 2 except for one change in the representation of the board. We again represent the board as a nine-element vector, but this time we assign board positions to vector elements as follows:

8	3	4
1	5	9
6	7	2

Notice that this numbering of the board produces a magic square: all the rows, columns, and diagonals sum up to 15. This means that we can simplify the process of checking for a possible win. In addition to marking the board as moves are made, we keep a list, for each player, of the squares in which he or she has played. To check for a possible win for one player, we consider each pair of squares owned by that player and compute the difference between 15 and the sum of the two squares. If this difference is not positive or if it is greater than 9, then the original two squares were not collinear and so can be ignored. Otherwise, if the square representing the difference is blank, a move there will produce a win. Since no player can have more than four squares at a time, there will be many fewer squares examined using this scheme than there were using the more straightforward approach of Program 2. This shows how the choice of representation can have a major impact on the efficiency of a problem-solving program.

Comments

This comparison raises an interesting question about the relationship between the way people solve problems and the way computers do. Why do people find the row-scan approach easier while the number-counting approach is more efficient for a computer? We do not know enough about how people work to answer that question completely. One part of the answer is that people are parallel processors and can look at several parts of the board at once, whereas the conventional computer must look at the squares one at a time. Sometimes an investigation of how people solve problems sheds great light on how computers should do so. At other times, the differences in the hardware of the two seem so great that different strategies seem best. As we learn more about problem solving both by people and by machines, we may know better whether the same representations and algorithms are best for both people and machines. We will discuss this question further in Section 1.4.

Copyrighted material

Program 3

Data Structures

BoardPosition A structure containing a nine-element vector representing the board, a list of board positions that could result from the next move, and a number representing an estimate of how likely the board position is to lead to an ultimate win for the player to move.

The Algorithm

To decide on the next move, look ahead at the board positions that result from each possible move. Decide which position is best (as described below), make the move that leads to that position, and assign the rating of that best move to the current position.

To decide which of a set of board positions is best, do the following for each of them:

1. See if it is a win. If so, call it the best by giving it the highest possible rating.
2. Otherwise, consider all the moves the opponent could make next. See which of them is worst for us (by recursively calling this procedure). Assume the opponent will make that move. Whatever rating that move has, assign it to the node we are considering.
3. The best node is then the one with the highest rating.

This algorithm will look ahead at various sequences of moves in order to find a sequence that leads to a win. It attempts to maximize the likelihood of winning, while assuming that the opponent will try to minimize that likelihood. This algorithm is called the *minimax procedure*, and it is discussed in detail in Chapter 12.

Comments

This program will require much more time than either of the others since it must search a tree representing all possible move sequences before making each move. But it is superior to the other programs in one very big way: It could be extended to handle games more complicated than tic-tac-toe, for which the exhaustive enumeration approach of the other programs would completely fall apart. It can also be augmented by a variety of specific kinds of knowledge about games and how to play them. For example, instead of considering all possible next moves, it might consider only a subset of them that are determined, by some simple algorithm, to be reasonable. And, instead of following each series of moves until one player wins, it could search for a limited time and evaluate the merit of each resulting board position using some static function.

Program 3 is an example of the use of an AI technique. For very small problems, it is less efficient than a variety of more direct methods. However, it can be used in situations where those methods would fail.

1.3.2 Question Answering

In this section we look at a series of programs that read in English text and then answer questions, also stated in English, about that text. This task differs from the last one in that it is more difficult now to state formally and precisely what our problem is and what constitutes correct solutions to it. For example, suppose that the input text were just the single sentence

Russia massed troops on the Czech border.

Then either of the following question-answering dialogues might occur (and in fact did occur with the POLITICS program [Carbonell, 1980]):

Copyrighted material

Dialogue 1

Q: Why did Russia do this?
A: Because Russia thought that it could take political control of Czechoslovakia by sending troops.
Q: What should the United States do?
A: The United States should intervene militarily.

Dialogue 2

Q: Why did Russia do this?
A: Because Russia wanted to increase its political influence over Czechoslovakia.
Q: What should the United States do?
A: The United States should denounce the Russian action in the United Nations.

In the POLITICS program, answers were constructed by considering both the input text and a separate model of the beliefs and actions of various political entities, including Russia. When the model is changed, as it was between these two dialogues, the system's answers also change. In this example, the first dialogue was produced when POLITICS was given a model that was intended to correspond to the beliefs of a typical American conservative (circa 1977). The second dialogue occurred when POLITICS was given a model that was intended to correspond to the beliefs of a typical American liberal (of the same vintage).

The general point here is that defining what it means to produce a *correct* answer to a question may be very hard. Usually, question-answering programs define what it means to be an answer by the procedure that is used to compute the answer. Then their authors appeal to other people to agree that the answers found by the program "make sense" and so to confirm the model of question answering defined in the program. This is not completely satisfactory, but no better way of defining the problem has yet been found. For lack of a better method, we will do the same here and illustrate three definitions of question answering, each with a corresponding program that implements the definition.

In order to be able to compare the three programs, we illustrate all of them using the following text:

> Mary went shopping for a new coat. She found a red one she really liked. When she got it home, she discovered that it went perfectly with her favorite dress.

We will also attempt to answer each of the following questions with each program:

Q1: What did Mary go shopping for?
Q2: What did Mary find that she liked?
Q3: Did Mary buy anything?

Program 1

This program attempts to answer questions using the literal input text. It simply matches text fragments in the questions against the input text.

Data Structures

QuestionPatterns A set of templates that match common question forms and produce patterns to be used to match against inputs. Templates and patterns (which we call *text patterns)* are paired so that if a template matches successfully against an input question then its associated text

Copyrighted material

	patterns are used to try to find appropriate answers in the text. For example, if the template "Who did *x y*" matches an input question, then the text pattern "*x y z*" is matched against the input text and the value of *z* is given as the answer to the question.
Text	The input text stored simply as a long character string.
Question	The current question also stored as a character string.

The Algorithm

To answer a question, do the following:

1. Compare each element of QuestionPatterns against the Question and use all those that match successfully to generate a set of text patterns.
2. Pass each of these patterns through a substitution process that generates alternative forms of verbs so that, for example, "go" in a question might match "went" in the text. This step generates a new, expanded set of text patterns.
3. Apply each of these text patterns to Text, and collect all the resulting answers.
4. Reply with the set of answers just collected.

Examples

Q1: The template "What did *x v*" matches this question and generates the text pattern "Mary go shopping for *z*." After the pattern-substitution step, this pattern is expanded to a set of patterns including "Mary goes shopping for *z*," and "Mary went shopping for *z*." The latter pattern matches the input text; the program, using a convention that variables match the longest possible string up to a sentence delimiter (such as a period), assigns *z* the value, "a new coat," which is given as the answer.

Q2: Unless the template set is very large, allowing for the insertion of the object of "find" between it and the modifying phrase "that she liked," the insertion of the word "really" in the text, and the substitution of "she" for "Mary," this question is hot answerable. If all of these variations are accounted for and the question can be answered, then the response is "a red one."

Q3: Since no answer to this question is contained in the text, no answer will be found.

Comments

This approach is clearly inadequate to answer the kinds of questions people could answer after reading a simple text. Even its ability to answer the most direct questions is delicately dependent on the exact form in which questions are stated and on the variations that were anticipated in the design of the templates and the pattern substitutions that the system uses. In fact, the sheer inadequacy of this program to perform the task may make you wonder how such an approach could even be proposed. This program is substantially farther away from being useful than was the initial program we looked at for tic-tac-toe. Is this just a strawman designed to make some other technique look good in comparison? In a way, yes, but it is worth mentioning that the approach that this program uses, namely matching patterns, performing simple text substitutions, and then forming answers using-straightforward combinations of canned text and sentence fragments located by the matcher, is the same approach that is used in one of the most famous "AI" programs ever written—ELIZA, which we discuss in Section 6.4.3. But, as you read the rest of this sequence of programs, it should become clear that what we mean by the term "artificial intelligence" does not include programs such as this except by a substantial stretching of definitions.

Program 2

This program first converts the input text into a structured internal form that attempts to capture the meaning of the sentences. It also converts questions into that form. It finds answers by matching structured forms against each other.

Copyrighted material

Data Structures

EnglishKnow — A description of the words, grammar, and appropriate semantic interpretations of a large enough subset of English to account for the input texts that the system will see. This knowledge of English is used both to map input sentences into an internal, meaning-oriented form and to map from such internal forms back into English. The former process is used when English text is being read; the latter is used to generate English answers from the meaning-oriented form that constitutes the program's knowledge base.

InputText — The input text in character form.

StructuredText — A structured representation of the content of the input text. This structure attempts to capture the essential knowledge contained in the text, independently of the exact way that the knowledge was stated in English. Some things that were not explicit in the English text, such as the referents of pronouns, have been made explicit in this form. Representing knowledge such as this is an important issue in the design of almost all AI programs. Existing programs exploit a variety of frameworks for doing this. There are three important families of such *knowledge representation* systems: production rules (of the form "if *x* then *y*"), slot-and-filler structures, and statements in mathematical logic. We discuss all of these methods later in substantial detail, and we look at key questions that need to be answered in order to choose a method for a particular program'. For now though, we just pick one arbitrarily. The one we've chosen is a slot-and-filler structure. For example, the sentence "She found a red one she really liked," might be represented as shown in Fig. 1.2. Actually, this is a simplified description of the contents of the sentence. Notice that it is not very explicit about temporal relationships (for example, events are just marked as past tense) nor have we made any real attempt to represent the meaning of the qualifier "really." It should, however, illustrate-the basic form that representations such as this take. One of the key ideas in this sort of representation is that entities in the representation derive their meaning from their connections to other entities. In the figure, only the entities defined by the sentence are shown. But other entities, corresponding to concepts that the program knew about before it read this sentence, also exist in the representation and can be re- ferred to within these new structures. In this example, for instance, we refer to the entities *Mary, Coat* (the general concept of a coat of which *Thing1* is a specific instance), *Liking* (the general concept of liking), and *Finding* (the general concept of finding).

Event 2	
instance :	*Finding*
tense:	*Past*
agent :	*Mary*
object:	*Thing1*
Thing1	
instance:	*Coat*
color:	*Red*
Event2	
instance:	*Liking*
tense :	*Past*
modifier:	*Much*
object:	*Thing1*

Fig. 1.2 *A Structured Representation of a Sentence*

Copyrighted material

InputQuestion	The input question in character form.
StructQuestion	A structured representation of the content of the user's question. The structure is the same as the one used to represent the content of the input text.

The Algorithm

Convert the InputText into structured form using the knowledge contained in EnglishKnow. This may require considering several different potential structures, for a variety of reasons, including the fact that English words can be ambiguous, English grammatical structures can be ambiguous, and pronouns may have several possible antecedents. Then, to answer a question, do the following:

1. Convert the question to structured form, again using the knowledge contained in EnglishKnow. Use some special marker in the structure to indicate the part of the structure that should be returned as the answer. This marker will often correspond to the occurrence of a question word (like "who" or "what") in the sentence. The exact way in which this marking gets done depends on the form chosen for representing StructuredText. If a slot-and-filler structure, such as ours, is used, a special marker can be placed in one or more slots. If a logical system is used, however, markers will appear as variables in the logical formulas that represent the question.
2. Match this structured form against StructuredText.
3. Return as the answer those parts of the text that match the requested segment of the question.

Examples

Ql: This question is answered straightforwardly with, "a new coat".
Q2: This one also is answered successfully with, "a red coat".
Q3: This one, though, cannot be answered, since there is no direct response to it in the text.

Comments

This approach is substantially more meaning (knowledge)-based than that of the first program and so is more effective. It can answer most questions to which replies are contained in the text, and it is much less brittle than the first program with respect to the exact forms of the text and the questions. As we expect, based on our experience with the pattern recognition and tic-tac-toe programs, the price we pay for this increased flexibility is time spent searching the various knowledge bases (i.e., EnglishKnow, StructuredText).

One word of warning is appropriate here. The problem of producing a knowledge base for English that is powerful enough to handle a wide range of English inputs is very difficult. It is discussed at greater length in Chapter 15. In addition, it is now recognized that knowledge of English alone is not adequate in general to enable a program to build the kind of structured representation shown here. Additional knowledge about the world with which the text deals is often required to support lexical and syntactic disambiguation and the correct assignment of antecedents to pronouns, among other things. For example, in the text

> Mary walked up to the salesperson. She asked where the toy department was.

it is not possible to determine what the word "she" refers to without knowledge about the roles of customers and sales people in stores. To see this, contrast the correct antecedent of 'she" in that text with the correct antecedent for the first occurrence of "she" in the following example:

> Mary walked up to the sales person. She asked her if she needed any help.

In the simple case illustrated in our coat-buying example, it is possible to derive correct answers to our first two questions without any additional knowledge about stores or coats, and the fact that some such additional information may be necessary to support question answering has already been illustrated by the failure of this

Copyrighted material

program to find an answer to question 3. Thus we see that although extracting a structured representation of the meaning of the input text is an improvement over the meaning-free approach of Program 1, it is by no means sufficient in general. So we need to look at an even more sophisticated (i.e., knowledge-rich) approach, which is what we do next.

Program 3

This program converts the input text into a structured form that contains the meanings of the sentences in the text, and then it combines that form with other structured forms that describe prior knowledge about the objects and situations involved in the text. It answers questions using this augmented knowledge structure.

Data Structures

WorldModel — A structured representation of background world knowledge. This structure contains knowledge about objects, actions and situations that are described in the input text. This structure is used to construct IntegratedText from the input text. For example, Figure 1.3 shows an example of a structure that represents the system's knowledge about shopping. This kind of stored knowledge about stereotypical events is called a *script* and is discussed in more detail in Section 10.2. The notation used here differs from the one normally used in the literature for the sake of simplicity. The prime notation describes an object of the same type as the unprimed symbol that may or may not refer to the identical object. In the case of our text, for example, M is a coat and M' is a red coat. Branches in the figure describe alternative paths through the script.

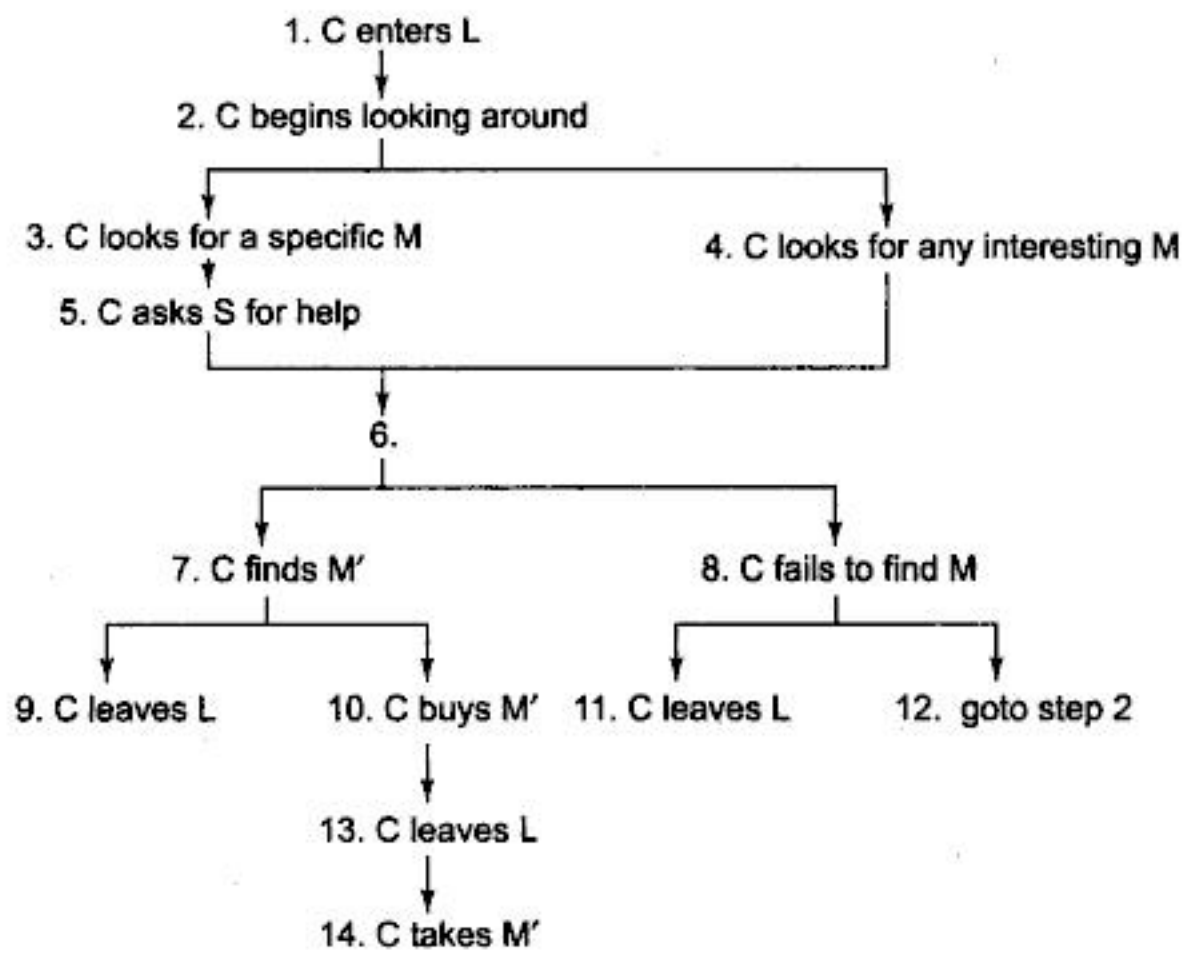


Fig. 1.3 *A Shopping Script*

EnglishKnow — Same as in Program 2.
InputText — The input text in character form.

Copyrighted material

IntegratedText	A structured representation of the knowledge contained in the in- put text (similar to the structured description of Program 2) but combined now with other background, related knowledge.
InputQuestion	The input question in character form.
StructQuestion	A structured representation of the question.

The Algorithm

Convert the InputText into structured form using both the knowledge contained in EnglishKnow and that contained in WorldModel. The number of possible structures will usually be greater now than it was in Program 2 because so much more knowledge is being used. Sometimes, though, it may be possible to consider fewer possibilities by using the additional knowledge to filter the alternatives.

Shopping Script:
roles: C (customer), S (salesperson)
props: M (merchandise), D (dollars)
location: L (a store)

To answer a question, do the following:

1. Convert the question to structured form as in Program 2 but use WorldModel iT necessary to resolve any ambiguities that may arise.
2. Match this structured form against IntegratedText.
3. Return as the answer those parts of the text that match the requested segment of the question.

Examples

Ql: Same as Program 2.

Q2: Same as Program 2.

Q3: Now this question can be answered. The shopping script is instantiated for this text, and because of the last sentence, the path through step 14 of the script is the one that is used in forming the representation of this text. When the script is instantiated M' is bound to the structure representing the red coat (because the script says that M' is what gets taken home and the text says that a red coat is what got taken home). After the script has been instantiated, IntegratedText contains several events.that are taken from the script but that are not described in the original text, including the event "Mary buys a red coat" (from step 10 of the script). Thus, using the integrated text as the basis for question answering allows the program to respond "She bought a red coat."

Comments

This program is more powerful than either of the first two because it exploits more knowledge. Thus, like the final program in each of the other two sequences we have examined, it is exploiting what we call AI techniques. But, again, a few caveats are in order. Even the techniques we have exploited in this program are not adequate for complete English question answering. The most important thing that is missing from this program is a general reasoning (inference) mechanism to be used when the requested answer is not contained explicitly even in IntegratedText, but that answer does follow logically from the knowledge that is there. For example, given the text

> Saturday morning Mary went shopping. Her brother tried to call her then, but he couldn't get hold of her.

it should be possible to answer the question

Copyrighted material

Why couldn't Mary's brother reach her?

with the reply

Because she wasn't home.

But to do so requires knowing that one cannot be at two places at once and then using that fact to conclude that Mary could not have been home because she was shopping instead. Thus, although we avoided the inference problem temporarily by building IntegratedText, which had some obvious inferences built into it, we cannot avoid it forever. It is simply not practical to anticipate all legitimate inferences. In later chapters, we look at ways of providing a general inference mechanism that could be used to support a program such as the last one in this series.

This limitation does not contradict the main point of this example though. In fact, it is additional evidence for that point, namely, an effective question-answering procedure must be one based soundly on knowledge and the computational use of that knowledge. The purpose of AI techniques is to support this effective use of knowledge.

With the advent of the Internet and the vast amount of knowledge in the ever increasing websites and associated pages, came the Web based Question Answering Systems. Try for instance the START natural language question answering system (http://start.csail.mit.edu/). You will find that both the questions – *What is the capital of India?* and *Is Delhi the capital of India?* yield the same answers, viz. *New Delhi is the capital of India.* On the contrary the question — *Are there wolves in Korea?* yields *I don't know if there are wolves in Korea.* which looks quite natural.

1.3.3 Conclusion

We have just examined two series of programs to solve two very different problems. In each series, the final program exemplifies what we mean by an AI technique. These two programs are slower to execute than the earlier ones in their respective series, but they illustrate three important AI techniques:

- Search—Provides a way of solving problems for which no more direct approach is available as well as a framework into which any direct techniques that are available can be embedded..
- Use of Knowledge—Provides a way of solving complex problems by exploiting the structures of the objects that are involved.
- Abstraction—Provides a way of separating important features and variations from the many unimportant ones that would otherwise overwhelm any process.

For the solution of hard problems, programs that exploit these techniques have several advantages over those that do not. They are much less fragile; they will not be thrown off completely by a small perturbation in their input. People can easily understand what the program's knowledge is. And these techniques can work for large problems where more direct methods break down.

We have still not given a precise definition of an AI technique. It is probably not possible to do so. But we have given some examples of what one is and what one is not. Throughout the rest of this book, we talk in great detail about what one is The definition should then become a bit clearer, or less necessary.

1.4 THE LEVEL OF THE MODEL

Before we set out to do something, it is a good idea to decide exactly what we are trying to do. So we must ask ourselves, "What is our goal in trying to produce programs that do the intelligent things that people do?" Are we trying to produce programs that do the tasks the same way people do? Or, are we attempting to produce

Copyrighted material

programs that simply do the tasks in whatever way appears easiest? There have been AI projects motivated by each of these goals.

Efforts to build programs that perform tasks the way people do can be divided into two classes. Programs in the first class attempt to solve problems that do not really fit our definition of an AI task. They are problems that a computer could easily solve, although that easy solution would exploit mechanisms that do not seem to be available to people. A classical example of this class of program is the Elementary Perceiver and Memorizer (EPAM) [Feigenbaum, 1963], which memorized associated pairs of nonsense syllables. Memorizing pairs of nonsense syllables is easy for a computer. Simply input them. To retrieve a response syllable given its associated stimulus one, the computer just scans for the stimulus syllable and responds with the one stored next to it. But this task is hard for people. EPAM simulated one way people might perform the task. It built a discrimination net through which it could find images of the syllables it had seen. It also stored, with each stimulus image, a cue that it could later pass through the discrimination net to try to find the correct response image. But it stored as a cue only as much information about the response syllable as was necessary to avoid ambiguity at the time the association was stored. This might be just the first letter, for example: But, of course, as the discrimination net grew and more syllables were added, an old cue might no longer be sufficient to identify a response syllable uniquely. Thus EPAM, like people, sometimes "forgot" previously learned responses. Many people regard programs in this first class to be uninteresting, and to some extent they are probably right. These programs can, however, be useful tools for psychologists who want to test theories of human performance.

The second class of programs that attempt to model human performance are those that do things that fall more clearly within our definition of AI tasks; they do things that are not trivial for the computer. There are several reasons one might want to model human performance at these sorts of tasks:

1. To test psychological theories of human performance. One example of a program that was written for this reason is PARRY [Colby, 1975], which exploited a model of human paranoid behavior to simulate the conversational behavior of a paranoid person. The model was good enough that when several psychologists were given the opportunity to converse with the program via a terminal, they diagnosed its behavior as paranoid.
2. To enable computers to understand human reasoning. For example, for a computer to be able to read a newspaper story and then answer a question, such as "Why did the terrorists kill the hostages?" its program must be able to simulate the reasoning processes of people.
3. To enable people to understand computer reasoning. In many circumstances, people are reluctant to rely on the output of a computer unless they can understand how the machine arrived at its result. If the computer's reasoning process is similar to that of people, then producing an acceptable explanation is much easier.
4. To exploit what knowledge we can glean from people. Since people are the best-known performers of most of the tasks with which we are dealing, it makes a lot of sense to look to them for clues as to how to proceed.

This last motivation is probably the most pervasive of the four. It motivated several very early systems that attempted to produce intelligent behavior by imitating people at the level of individual neurons. For examples of this, see the early theoretical work of McCulloch and Pitts [1943], the work on perceptrons, originally developed by Frank Rosenblatt but best described in *Perceptrons* [Minsky and Papert, 1969] and *Design for a Brain* [Ashby, 1952]. It proved impossible, however, to produce even minimally intelligent behavior with such simple devices. One reason was that there were severe theoretical limitations to the particular neural, net architecture that was being used. More recently, several new neural net architectures have been proposed. These structures are not subject to the same theoretical limitations as were perceptrons. These new architectures are loosely called *connectionist,* and they have been used as a basis for several learning and problem-solving programs. We have more to say about them in Chapter 18. Also, we must consider that while human brains are

Copyrighted material

highly parallel devices, most current computing systems are essentially serial engines. A highly successful parallel technique may be computationally intractable on a serial computer. But recently, partly because of the existence of the new family of parallel cognitive models, as well as because of the general promise of parallel computing, there is now substantial interest in the design of massively parallel machines to support AI programs.

Human cognitive theories have also influenced AI to look for higher-level (i.e., far above the neuron level) theories that do not require massive parallelism for their implementation. An early example of this approach can be seen in GPS, which are discussed in more detail in Section 3.6. This same approach can also be seen in much current work in natural language understanding. The failure of straightforward syntactic parsing mechanisms to make much of a dent in the problem of interpreting English sentences has led many people who are interested in natural language understanding by machine to look seriously for inspiration at what little we know about how people interpret language. And when people who are trying to build programs to analyze pictures discover that a filter function they have developed is very similar to what we think people use, they take heart that perhaps they are on the right track.

As you can see, this last motivation pervades a great many areas of AI-research. In fact, it, in conjunction.with the other motivations we mentioned, tends to make the distinction between the goal of simulating human performance and the goal of building an intelligent program any way we can seem much less different than they at first appeared. In either case, what we really need is a good model of the processes involved in intelligent reasoning. The field of *cognitive science*, in which psychologists, linguists, and computer scientists all work together, has as its goal the discovery of such a model. For a good survey of the variety of approaches contained within the field, see Norman [1981], Anderson [1985], and Gardner [1985].

1.5 CRITERIA FOR SUCCESS

One of the most important questions to answer in any scientific or engineering research project is "How will we know if we have succeeded?" Artificial intelligence is no exception. How will we know if we have constructed a machine that is intelligent? That question is at least as hard as the unanswerable question "What is intelligence?" But can we do anything to measure our progress?

In 1950, Alan Turing proposed the following method for determining whether a machine can think. His method has since become known as the *Turing Test*. To conduct this test, we need two people and the machine to be evaluated. One person plays the role of the interrogator, who is in a separate room from the computer and the other person. The interrogator can ask questions of either the person or the computer by typing questions and receiving typed responses. However, the interrogator knows them only as A and B and aims to determine which is the person and which is the machine. The goal of the machine is to fool the interrogator into believing that it is the person. If the machine succeeds at this, then we will conclude that the machine can think. The machine is allowed to do whatever it can to fool the interrogator. So, for example, if asked the question "How much is 12,324 times 73,981?" it could wait several minutes and then respond with the wrong answer [Turing, 1963].

The more serious issue, though, is the amount of knowledge that a machine would need to pass the Turing test. Turing gives the following example of the sort of dialogue a machine would have to be capable of:

Interrogator:	In the first line of your sonnet which reads "Shall I compare thee to a summer's day," would not "a spring day" do as well or better?
A:	It wouldn't scan.
Interrogator:	How about "a winter's day." That would scan all right.
A:	Yes, but nobody wants to be compared to a winter's day.
Interrogator:	Would you say Mr. Pickwick reminded you of Christmas?
A:	In a way.

Copyrighted material

Interrogator: Yet Christmas is a winter's day, and I do not think Mr. Pickwick would mind the comparison.

A: I don't think you're serious. By a winter's day one means a typical winter's day, rather than a special one like Christmas.

It will be a long time before a computer passes the Turing test. Some people believe none ever will. But suppose we are willing to settle for less than a complete imitation of a person. Can we measure the achievement of AI in more restricted domains?

Often the answer to this question is yes. Sometimes it is possible to get a fairly precise measure of the achievement of a program. For example, a program can acquire a chess rating in the same way as a human player. The rating is based on the ratings of players whom the program can beat. Already programs have acquired chess ratings higher than the vast majority of human players. For other problem domains, a less precise measure of a program's achievement is possible. For example, DENDRAL is a program that analyzes organic compounds to determine their structure. It is hard to gel a precise measure of DENDRAL's level of achievement compared to human chemists, but it has produced analyses that have been published as original research results. Thus it is certainly performing competently.

In other technical domains, it is possible to compare the time it takes for a program to complete a task to the time required by a person to do the same thing. For example, there are several programs in use by computer companies to configure particular systems to customers' needs (of which the pioneer was a program called R1). These programs typically require minutes to perform tasks that previously required hours of a skilled engineer's time. Such programs are usually evaluated by looking at the bottom line— whether they save (or make) money.

For many everyday tasks, though, it may be even harder to measure a program's performance. Suppose, for example, we ask a program to paraphrase a newspaper story. For problems such as this, the best test is usually just whether the program responded in a way that a person could have.

If our goal in writing a program is to simulate human performance at a task, then the measure of success is the extent to which the program's behavior corresponds to that performance, as measured by various kinds of experiments and protocol analyses. In this we do not simply want a program that does as well as possible. We want one that fails when people do. Various techniques developed by psychologists for comparing individuals and for testing models can be used to do this analysis.

We are forced to conclude that the question of whether a machine has intelligence or can think is too nebulous to answer precisely. But it is often possible to construct a computer program that meets some performance standard for a particular task. That does not mean that the program does the task in the best possible way. It means only that we understand at least one way of doing at least part of a task. When we set out to design an AI program, we should attempt to specify as well as possible the criteria for success for that particular program functioning in its restricted domain. For the moment, that is the best we can do.

1.6 SOME GENERAL REFERENCES

There are a great many sources of information about artificial intelligence. First, some survey books: The broadest are the multi-volume *Handbook of Artificial Intelligence* [Barr *et al.*. 1981] and *Encyclopedia of Artificial Intelligence* [Shapiro and Eckroth, 1987], both of which contain articles on each of the major topics in the field. Four other books that provide good overviews of the field are *Artificial Intelligence* [Winston, 1984], *Introduction to Artificial Intelligence* [Charniak and McDermott, 1985], *Logical Foundations of Artificial Intelligence* [Genesereth and Nilsson, 1987], and *The Elements of Artificial Intelligence* [Tanimoto, 1987]. Of more restricted scope is *Principles of Artificial Intelligence* [Nilsson, 1980], which contains a formal treatment of some general-purpose AI techniques.

Copyrighted material

The history of research in artificial intelligence is a fascinating story, related by Pamela McCordiick [1979] in her book *Machines Who Think*. Because almost all of what we call AI has been developed over the last 30 years, McCorduck was able to conduct her research for the book by actually interviewing almost all of the people whose work was influential in forming the field.

Most of the work conducted in AI has been originally reported in journal articles, conference proceedings, or technical reports. But some of the most interesting of these papers have later appeared in special collections published as books. *Computers and Thought* [Feigenbaum and Feldman, 1963] is a very early collection of this sort. Later ones include Simon and Siklossy [1972], Schank and Colby [1973], Bobrow and Collins [1975], Waterman and Hayes-Roth [1978], Findler [1979], Webber and Nilsson [1981], Halpern [1986], Shrobe [1988], and several others that are mentioned in later chapters in connection with specific topics.For newer AI paradigms the book *Fundamentals of the New Artificial Intelligence* [Toshinori Munakata, 1998] is a good one.

The major journal of AI research is called simply *Artificial Intelligence*. In addition, *Cognitive Science* is devoted to papers dealing with the overlapping areas of psychology, linguistics, and artificial intelligence. *AI Magazine* is a more ephemeral, less technical magazine that is published by the American Association for Artificial Intelligence (AAAI). *IEEE Expert, IEEE Transactions on Systems, Man and Cybernetics, IEEE Transactions on Neural Networks* and several other journals publish papers on a broad spectrum of AI application domains.

Since 1969, there has been a major AI conference, the International Joint Conference on Artificial Intelligence (IJCAI), held every two years. The proceedings of these conferences give a good picture of the work that was taking place at the time. The other important AI conference, held three out of every four years starting in 1980, is sponsored by the AAAI, and its proceedings, too, are published.

In addition to these general references, there exists a whole array of papers and books describing individual AI projects. Rather than trying to list them all here, they are referred to as appropriate throughout the rest of this book.

1.7 ONE FINAL WORD AND BEYOND

What conclusions can we draw from this hurried introduction to the major questions of AI? The problems are varied, interesting, and hard. If we solve them, we will have useful programs and perhaps a better understanding of human thought. We should do the best we can to set criteria so that we can tell if we have solved the problems, and then we must try to do so.

How actually to go about solving these problems is the topic for the rest of this book. We need methods to help us solve AI's serious dilemma:

1. An AI system must contain a lot of knowledge if it is to handle anything but trivial toy problems.
2. But as the amount of knowledge grows, it becomes harder to access the appropriate things when needed, so more knowledge must be added to help. But now there is even more knowledge to manage, so more must be added, and so forth.

Our goal in AI is to construct working programs that solve the problems we are interested in. Throughout most of this book we focus on the design of representation mechanisms and algorithms that can be used by programs to solve the problems. We do not spend much time discussing the programming process required to turn these designs into working programs. In theory, it does not matter how this process is carried out, in what language it is done, or on what machine the product is run. In practice, of course, it is often much easier to produce a program using one system rather than another. Specifically, AI programs are easiest to build using languages that have been designed to support symbolic rather than primarily numeric computation.

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

CHAPTER

2

PROBLEMS, PROBLEM SPACES, AND SEARCH

It's not that I'm so smart, it's just that I stay with problems longer.

—**Albert Einstein**
(1879 –1955), German-born theoretical physicist

In the last chapter, we gave a brief description of the kinds of problems with which AI is typically concerned, as well as a couple of examples of the techniques it offers to solve those problems. To build a system to solve a particular problem, we need to do four things:

1. Define the problem precisely. This definition must include precise specifications of what the initial situation (s) will be as well as what final situations constitute acceptable solutions to the problem.
2. Analyze the problem. A few very important features can have an immense impact on the appropriateness of various possible techniques for solving the problem.
3. Isolate and represent the task knowledge that is necessary to solve the problem.
4. Choose the best problem-solving technique(s) and apply it (them) to the particular problem.

In this chapter and the next, we discuss the first two and the last of these issues. Then, in the chapters in Part II, we focus on the issue of knowledge representation.

2.1 DEFINING THE PROBLEM AS A STATE SPACE SEARCH

Suppose we start with the problem statement "Play chess". Although there are a lot of people to whom we could say that and reasonably expect that they will do as we intended, as our request now stands it is a very incomplete statement of the problem we want solved. To build a program that could "Play chess," we would first have to specify the starting position of the chess board, the rules that define the legal moves, and the board positions that represent a win for one side or the other. In addition, we must make explicit the previously implicit goal of not only playing a legal game of chess **but** also winning the game, if possible.

For the problem "Play chess," it is fairly easy to provide a formal and complete problem description. The starting position can be described as an 8×8 array where each position contains a symbol standing for the appropriate piece in the official chess opening position. We can define as our goal any board position in which the opponent does not have a legal move and his or her king is under attack. The legal moves provide the way of getting from the initial state to a goal state. They can be described easily as a set of rules consisting of two parts: a left side that serves as a pattern to be matched against the current board position and a right side that

Copyrighted material

describes the change to be made to the board position to reflect the move. There are several ways in which these rules can be written. For example, we could write a rule such as that shown in Fig. 2.1.

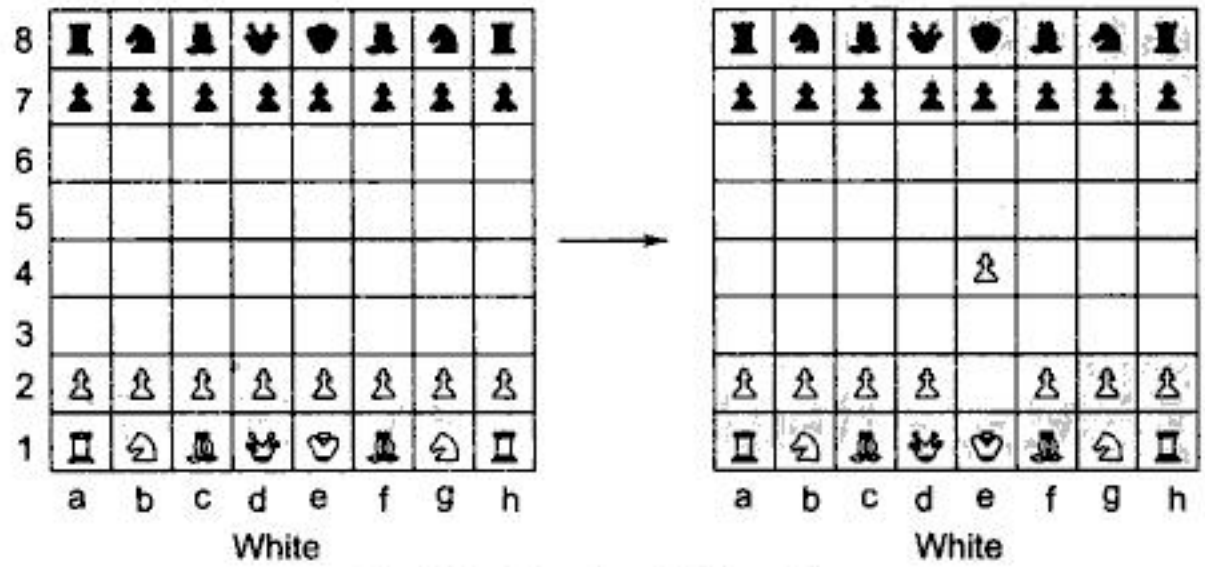


Fig. 2.1 *One Legal Chess Move*

However, if we write rules like the one above, we have to write a very large number of them since there has to be a separate rule for each of the roughly 10^{120} possible board positions. Using so many rules poses two serious practical difficulties:

- No person could ever supply a complete set of such rules. It would take too long and could certainly not be done without mistakes.
- No program could easily handle all those rules. Although a hashing scheme could be used to find the relevant rules for each move fairly quickly, just storing that many rules poses serious difficulties.

In order to minimize such problems, we should look for a way to write the rules describing the legal moves in as general a way as possible. To do this, it is useful to introduce some convenient notation for describing patterns and substitutions. For example, the rule described in Fig. 2.1, as well as many like it, could be written as shown in Fig. 2.2.[1] In general, the more succinctly we can describe the rules we need, the less work we will have to do to provide them and the more efficient the program that uses them can be.

White pawn at Square(file e, rank 2) AND Square(file e, rank 3) is empty AND Square(file e, rank 4) is empty	→	move pawn from Square(file e, rank 2) to Square(file e, rank 4)

Fig. 2.2 *Another Way to Describe Chess Moves*

We have just defined the problem of playing chess as a problem of moving around in a *slate space,* where each state corresponds to a legal position of the board. We can then play chess by starting at an initial state, using a set of rules to move from one state to another, and attempting to end up in one of a set of final states. This state space representation seems natural for chess because the set of states, which corresponds to the set of board positions, is artificial and well-organized. This same kind of representation is also useful for naturally occurring, less well-structured problems, although it may be necessary to use more complex structures than a

[1] To be completely accurate, this rule should include a check for pinned pieces, which have been ignored here.

Copyrighted material

matrix to describe an individual state. The state space representation forms the basis of most of the AI methods we discuss here. Its structure corresponds to the structure of problem solving in two important ways:

- It allows for a formal definition of a problem as the need to convert some given situation into some desired situation using a set of permissible operations.
- It permits us to define the process of solving a particular problem as a combination of known techniques (each represented as a rule defining a single step in the space) and search, the general technique of exploring the space to try to find some path from the current state to a goal state. Search is a very important process in the solution of hard problems for which no more direct techniques are available.

In order to show the generality of the state space representation, we use it to describe a problem very different from that of chess.

A Water Jug Problem: You are given two jugs, a 4-gallon one and a 3-gallon one. Neither has any measuring markers on it. There is a pump that can be used to fill the jugs with water. How can you get exactly 2 gallons of water into the 4-gallon jug?

The state space for this problem can be described as the set of ordered pairs of integers (x, y), such that $x = 0, 1, 2, 3$, or 4 and $y = 0, 1, 2$, or 3; x represents the number of gallons of water in the 4-gallon jug, and y represents the quantity of water in the 3-gallon jug. The start state is (0, 0). The goal state is $(2, n)$ for any value of n (since the problem does not specify how many gallons need to be in the 3-gallon jug).

The operators[2] to be used to solve the problem can be described as shown in Fig. 2.3. As in the chess problem, they are represented as rules whose left sides are matched against the current state and whose right sides describe the new state that results from applying the rule. Notice that in order to describe the operators completely, it was necessary to make explicit some assumptions not mentioned in the problem statement. We have assumed that we can fill a jug from the pump, that we can pour water out of a jug onto the ground, that we can pour water from one jug to another, and that there are no other measuring devices available. Additional assumptions such as these are almost always required when converting from a typical problem statement given in English to a formal representation of the problem suitable for use by a program.

To solve the water jug problem, all we need, in addition to the problem description given above, is a control structure that loops through a simple cycle in which some rule whose left side matches the current state is chosen, the appropriate change to the state is made as described in the corresponding right side, and the resulting state is checked to see if it corresponds to a goal state. As long as it does not, the cycle continues. Clearly the speed with which the problem gets solved depends on the mechanism that is used to select the next operation to be performed. In Chapter 3, we discuss several ways of making that selection.

For the water jug problem, as with many others, there are several sequences of operators that solve the problem. One such sequence is shown in Fig. 2.4. Often, a problem contains the explicit or implied statement that the shortest (or cheapest) such sequence be found. If present, this requirement will have a significant effect on the choice of an appropriate mechanism to guide the search for a solution. We discuss this issue in Section 2.3.4.

Several issues that often arise in converting an informal problem statement into a formal problem description are illustrated by this sample water jug problem. The first of these issues concerns the role of the conditions that occur in the left sides of the rules. All but one of the rules shown in Fig. 2.3 contain conditions that must be satisfied before the operator described by the rule can be applied. For example, the first rule says, "If the 4-gallon jug is not already full, fill it." This rule could, however, have been written as, "Fill the 4-gallon jug," since it is physically possible to fill the jug even if it is already full. It is stupid to do so since no change in the problem state results, but it is possible. By encoding in the left sides of the rules constraints that are not strictly necessary but that restrict the application of the rules to states in which the rules are most likely to lead to a solution, we can generally increase the efficiency of the problem-solving program that uses the rules.

[2] The word "operator" refers to some representation of an action. An operator usually includes information about what must be true in the world before the action can take place, and how the world is changed by the action.

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

heuristic is a technique that improves the efficiency of a search process, possibly by sacrificing claims of completeness. Heuristics are like tour guides. They are good to the extent that they point in generally interesting directions; they are bad to the extent that they may miss points of interest to particular individuals. Some heuristics help to guide a search process without sacrificing any claims to completeness that the process might previously have had. Others (in fact, many of the best ones) may occasionally cause an excellent path to be overlooked. But, on an average, they improve the quality of the paths that are explored. Using good heuristics, we can hope to get good (though possibly nonoptimal) solutions to hard problems, such as the traveling salesman, in less than exponential time. There are some good general-purpose heuristics that are useful in a wide variety of problem domains. In addition, it is possible to construct special-purpose heuristics that exploit domain-specific knowledge to solve particular problems.

One example of a good general-purpose heuristic that is useful for a variety of combinatorial problems is the *nearest neighbor heuristic,* which works by selecting the locally superior alternative at each step. Applying it to the traveling salesman problem, we produce the following procedure:

1. Arbitrarily select a starting city.
2. To select the next city, look at all cities not yet visited, and select the one closest to the current city. Go to it next.
3. Repeat step 2 until all cities have been visited.

This procedure executes in time proportional to N^2, a significant improvement over $N!$, and it is possible to prove an upper bound on the error it incurs. For general-purpose heuristics, such as nearest neighbor, it is often possible to prove such error bounds, which provides reassurance that one is not paying too high a price in accuracy for speed.

In many AI problems, however, it is not possible to produce such reassuring bounds. This is true for two reasons:

- For real world problems, it is often hard to measure precisely the value of a particular solution. Although the length of a trip to several cities is a precise notion, the appropriateness of a particular response to such questions as "Why has inflation increased?" is much less so.
- For real world problems, it is often useful to introduce heuristics based on relatively unstructured knowledge. It is often impossible to define this knowledge in such a way that a mathematical analysis of its effect on the search process can be performed.

There are many heuristics that, although they are not as general as the nearest neighbor heuristic, are nevertheless useful in a wide variety of domains. For example, consider the task of discovering interesting ideas in some specified area. The following heuristic [Lenat, 1983b] is often useful:

If there is an interesting function of two arguments $f(x, y)$, look at what happens if the two arguments are identical.

In the domain of mathematics, this heuristic leads to the discovery of *squaring iff* is the multiplication function, and it leads to the discovery of an *identity* function if f is the function of set union. In less formal domains, this same heuristic leads to the discovery of *introspection* if f is the function contemplate or it leads to the notion of *suicide iff* is the function kill.

Without heuristics, we would become hopelessly ensnarled in a combinatorial ex-plosion. This alone might be a sufficient argument in favor of their use. But there are other arguments as well:

- Rarely do we actually need the optimum solution; a good approximation will usually serve very well. In fact, there is some evidence that people, when they solve problems, are not optimizers but rather are *satisficers* [Simon, 1981]. In other words, they seek any solution that satisfies some set of requirements, and as soon as they find one they quit. A good example of this is the search for a parking space. Most people stop as soon as they find a fairly good space, even if there might be a slightly better space up ahead.

Copyrighted material

- Although the approximations produced by heuristics may not be very good in the worst case, worst cases rarely arise in the real world. For example, although many graphs are not separable (or even nearly so) and thus cannot be considered as a set of small problems rather than one large one, a lot of graphs describing the real world are.[6]
- Trying to understand why a heuristic works, or why it doesn't work, often leads to a deeper understanding of the problem.

One of the best descriptions of the importance of heuristics in solving interesting problems is *How to Solve It* [Polya, 1957]. Although the focus of the book is the solution of mathematical problems, many of the techniques it describes are more generally applicable. For example, given a problem to solve, look for a similar problem you have solved before. Ask whether you can use either the solution of that problem or the method that was used to obtain the solution to help solve the new problem. Polya's work serves as an excellent guide for people who want to become better problem solvers. Unfortunately, it is not a panacea for AI for a couple of reasons. One is that it relies on human abilities that we must first understand well enough to build into a program. For example, many of the problems Polya discusses are geometric ones in which once an appropriate picture is drawn, the answer can be seen immediately. But to exploit such techniques in programs, we must develop a good way of representing and manipulating descriptions of those Fig.s. Another is that the rules are very general.

They have extremely underspecified left sides, so it is hard to use them to guide a search—too many of them are applicable at once. Many of the rules are really only useful for looking back and rationalizing a solution after it has been found. In essence, the problem is that Polya's rules have not been operationalized.

Nevertheless, Polya was several steps ahead of AI. A comment he made in the preface to the first printing (1944) of the book is interesting in this respect:

The following pages are written somewhat concisely, but as simply as possible, and are based on a long and serious study of methods of solution. This sort of study, called *heuristic* by some writers, is not in fashion nowadays but has a long past and, perhaps, some future.

There are two major ways in which domain-specific, heuristic knowledge can be incorporated into a rule-based search procedure:

- In the rules themselves. For example, the rules for a chess-playing system might describe not simply the set of legal moves but rather a set of "sensible" moves, as determined by the rule writer.
- As a heuristic function that evaluates individual problem states and determines how desirable they are.

A *heuristic function* is a function that maps from problem state descriptions to measures of desirability, usually represented as numbers. Which aspects of the problem state are considered, how those aspects are evaluated, and the weights given to individual aspects are chosen in such a way that the value of the heuristic function at a given node in the search process gives as good an estimate as possible of whether that node is on the desired path to a solution.

Well-designed heuristic functions can play an important part in efficiently guiding a search process toward a solution. Sometimes very simple heuristic functions can provide a fairly good estimate of whether a path is any good or not. In other situations, more complex heuristic functions should be employed. Fig. 2.8 shows some simple heuristic functions for a few problems. Notice that sometimes a high value of the heuristic function indicates a relatively good position (as shown for chess and tic-tac- toe), while at other times a low value indicates an advantageous situation (as shown for the traveling salesman). It does not matter, in general, which way the function is stated. The program that uses the values of the function can attempt to minimize it or to maximize it as appropriate.

[6] For arguments in support of this, see Simon [1981].

Copyrighted material

Chess	the material advantage of our side over the opponent
Traveling Salesman	the sum of the distances so far
Tic-Tac-Toe	1 for each row in which we could win and in which we already have one piece plus 2 for each such row in which we have two pieces

Fig. 2.8 *Some Simple Heuristic Functions*

The purpose of a heuristic function is to guide the search process in the most profitable direction by suggesting which path to follow first when more than one is available. The more accurately the heuristic function estimates the true merits of each node in the search tree (or graph), the more direct the solution process. In the extreme, the heuristic function would be so good that essentially no search would be required. The system would move directly to a solution. But for many problems, the cost of computing the value of such a function would outweigh the effort saved in the search process, After all, it would be possible to compute a perfect heuristic function by doing a complete search from the node in question and determining whether it leads to a good solution. In general, there is a trade-off between the cost of evaluating a heuristic function and the savings in search time that the function provides.

In the previous section, the solutions to AI problems were described as centering on a search process. From the discussion in this section, it should be clear that it can more precisely be described as a process of heuristic search. Some heuristics will be used to define the control structure that guides the application of rules in the search process. Others, as we shall see, will be encoded in the rules themselves. In both cases, they will represent either general or specific world knowledge that makes the solution of hard problems feasible. This leads to another way that one could define artificial intelligence: the study of techniques for solving exponentially hard problems in polynomial time by exploiting knowledge about the problem domain.

2.3 PROBLEM CHARACTERISTICS

Heuristic search is a very general method applicable to a large class of problems. It encompasses a variety of specific techniques, each of which is particularly effective for a small class of problems. In order to choose the most appropriate method (or combination of methods) for a particular problem, it is necessary to analyze the problem along several key dimensions:

- Is the problem decomposable into a set of (nearly) independent smaller or easier subproblems?
- Can solution steps be ignored or at least undone if they prove unwise?
- Is the problem's universe predictable?
- Is a good solution to the problem obvious without comparison to all other possible solutions?
- Is the desired solution a state of the world or a path to a state?
- Is a large amount of knowledge absolutely required to solve the problem, or is knowledge important only to constrain the search?
- Can a computer that is simply given the problem return the solution, or will the solution of the problem require interaction between the computer and a person?

In the rest of this section, we examine each of these questions in greater detail. Notice that some of these questions involve not just the statement of the problem itself but also characteristics of the solution that is desired and the circumstances under which the solution must take place.

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The significance of these categories of production systems lies in the relationship between the categories and appropriate implementation strategies. But before discussing that relationship, it may be helpful to make the meanings of the definitions clearer by showing how they relate to specific problems.

Thus we arrive at the second question above, which asked whether there is an interesting relationship between classes of production systems and classes of problems. For any solvable problem, there exist an infinite number of production systems that describe ways to find solutions. Some will be more natural or efficient than others. Any problem that can be solved by any production system can be solved by a commutative one (our most restricted class), but the commutative one may be so unwieldy as to be practically useless. It may use individual states to represent entire sequences of applications of rules of a simpler, noncommutative system. So in a formal sense, there is no relationship between kinds of problems and kinds of production systems since all problems can be solved by all kinds of systems. But in a practical sense, there definitely is such a relationship between kinds of problems and the kinds of systems that lend themselves naturally to describing those problems. To see this, let us look at a few examples. Fig. 2.17 shows the four categories of production systems produced by the two dichotomies, monotonic versus nonmonotonic and partially commutative versus

	Monotonic	Nonmonotonic
Partially commutative	Theorem proving	Robot navigation
Not partially commutative	Chemical synthesis	Bridge

Fig. 2.17 *The Four Categories of Production Systems*

nonpartially commutative, along with some problems that can naturally be solved by each type of system. The upper left corner represents commutative systems.

Partially commutative, monotonic production systems are useful for solving ignorable problems. This is not surprising since the definitions of the two are essentially the same. But recall that ignorable problems are those for which a *natural* formulation leads to solution steps that can be ignored. Such a natural formulation will then be a partially commutative, monotonic system. Problems that involve creating new things rather than changing old ones are generally ignorable. Theorem proving, as we have described it, is one example of such a creative process. Making deductions from some known facts is a similar creative process. Both of those processes can easily be implemented with a partially commutative, monotonic system.

Partially commutative, monotonic production systems are important from an implementation standpoint because they can be implemented without the ability to backtrack to previous states when it is discovered that an incorrect path has been followed. Although it is often useful to implement such systems with backtracking in order to guarantee a systematic search, the actual database representing the problem state need not be restored. This often results in a considerable increase in efficiency, particularly because, since the database will never have to be restored, it is not necessary to keep track of where in the search process every change was made.

We have now discussed partially commutative production systems that are also monotonic. They are good for problems where things do not change; new things get created. Nonmonotonic, partially commutative systems, on the other hand, are useful for problems in which changes occur but can be reversed and in which order of operations is not critical. This is usually the case in physical manipulation problems, such as robot navigation on a flat plane. Suppose that a robot has the following operators: go north (N), go east (E), go south (S), and go west (W). To reach its goal, it does not matter whether the robot executes N-N-E or N-E-N.

Copyrighted material

Depending on how the operators are chosen, the 8-Puzzle and the blocks world problem can also be considered partially commutative.

Both types of partially commutative production systems are significant from an implementation point of view because they tend to lead to many duplications of individual states during the search process. This is discussed further in Section 2.5.

Production systems that are not partially commutative are useful for many problems in which irreversible changes occur. For example, consider the problem of determining a process to produce a desired chemical compound. The operators available include such things as "Add chemical x to the pot" or "Change the temperature to t degrees." These operators may cause irreversible changes to the potion being brewed. The order in which they are performed can be very important in determining the final output. It is possible that if x is added to y, a stable compound will be formed, so later addition of z will have no effect; if z is added to y, however, a different stable compound may be formed, so later addition of x will have no effect. Nonpartially commutative production systems are less likely to produce the same node many times in the search process. When dealing with ones that describe irreversible processes, it is particularly important to make correct decisions the first time, although if the universe is predictable, planning can be used to make that less important.

2.5 ISSUES IN THE DESIGN OF SEARCH PROGRAMS

Every search process can be viewed as a traversal of a tree structure in which each node represents a problem state and each arc represents a relationship between the states represented by the nodes it connects. For example, Fig. 2.18 shows part of a search tree for a water jug problem. The arcs have not been labeled in the Fig., but they correspond to particular water-pouring operations. The search process must find a path or paths through the tree that connect an initial state with one or more final states. The tree that must be searched could, in principle, be constructed in its entirety from the rules that define allowable moves in the problem space. But, in practice, most of it never is. It is too large and most of it need never be explored. Instead of first building the tree *explicitly* and then searching it, most search programs represent the tree *implicitly* in the rules and generate explicitly only those parts that they decide to explore. Throughout our discussion of search methods, it is important to keep in mind this distinction between implicit search trees and the explicit partial search trees that are actually constructed by the search program.

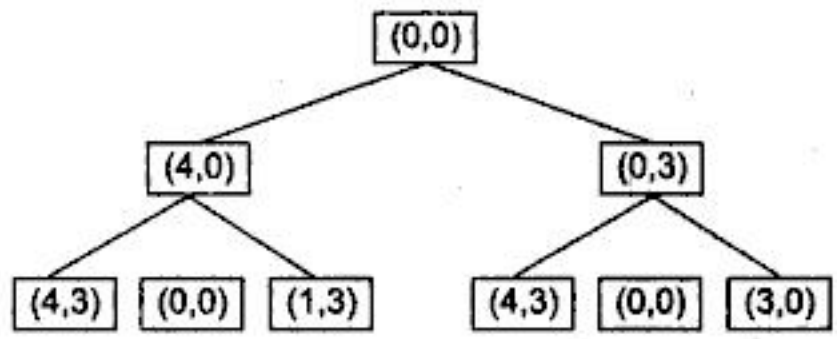


Fig. 2.18 *A Search Tree for the Water Jug Problem*

In the next chapter, we present a family of general-purpose search techniques. But before doing so we need to mention some important issues that arise in all of them:

- The direction in which to conduct the search (*forward* versus *backward* reasoning). We can search forward through the state space from the start state to a goal state, or we can search backward from the goal.
- How to select applicable rules *(matching)*. Production systems typically spend most of their time looking for rules to apply, so it is critical to have efficient procedures for matching rules against states.
- How to represent each node of the search process (the *knowledge representation problem* and the *frame problem)*. For problems like chess, a node can be fully represented by a simple array. In more complex problem solving, however, it is inefficient and/or impossible to represent all of the facts in the world and to determine all of the side effects an action may have.

We discuss the knowledge representation and frame problems further in Chapter 4. We investigate matching and forward versus backward reasoning when we return to production systems in Chapter 6.

Copyrighted material

One other issue we should consider at this point is that of search trees versus search graphs. As mentioned above, we can think of production rules as generating nodes in a search tree. Each node can be expanded in turn, generating a set of successors. This process continues until a node representing a solution is found. Implementing such a procedure requires little bookkeeping. However, this process often results in the same node being generated as part of several paths and so being processed more than once. This happens because the search space may really be an arbitrary directed graph rather than a tree.

For example, in the tree shown in Fig. 2.18, the node (4,3), representing 4-gallons of water in one jug and 3 gallons in the other, can be generated either by first filling the 4-gallon jug and then the 3-gallon one or by filling them in the opposite order. Since the order does not matter, continuing to process both these nodes would be redundant. This example also illustrates another problem that often arises when the search process operates as a tree walk. On the third level, the node (0, 0) appears. (In fact, it appears twice.) But this is the same as the top node of the tree, which has already been expanded. Those two paths have not gotten Us anywhere. So we would like to eliminate them and continue only along the other branches.

The waste of effort that arises when the same node is generated more than once can be avoided at the price of additional bookkeeping. Instead of traversing a search tree, we traverse a directed graph. This graph differs from a tree in that several paths may come together at a node. The graph corresponding to the tree of Fig. 2.18 is shown in Fig. 2.19.

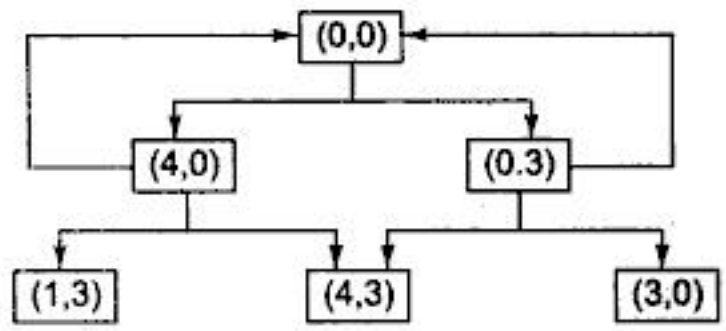


Fig. 2.19 *A Search Graph for the Water Jug Problem*

Any tree search procedure that keeps track of all the nodes that have been generated so far can be converted to a graph search procedure by modifying the action performed each time a node is generated. Notice that of the two systematic search procedures we have discussed so far, this requirement that nodes be kept track of is met by breadth-first search but not by depth-first search. But, of course, depth-first search could be modified, at the expense of additional storage, to retain in memory nodes that have been expanded and then backed-up over. Since all nodes are saved in the search graph, we must use the following algorithm instead of simply adding a new node to the graph.

Algorithm: Check Duplicate Nodes

1. Examine the set of nodes that have been created so far to see if the new node already exists.
2. If it does not-simply add it to the graph just as for a tree.
3. If it does already exist, then do the following:
   (a) Set the node that is being expanded to point to the already existing-node corresponding to its successor rather than to the new one. The new one can simply be thrown away.
   (b) If you are keeping track of the best (shortest or otherwise least-cost) path to each node, then check to see if the new path is better or worse than the old one. If worse, do nothing. If better, record the new path as the correct path to use to get to the node and propagate the corresponding change in cost down through successor nodes as necessary.

One problem that may arise here is that cycles may be introduced into the search graph. A *cycle* is a path through the graph in which a given node appears more than once. For example, the graph of Fig. 2.19 contains two cycles of length two. One includes the nodes (0, 0) and (4, 0); the other includes the nodes (0, 0) and (0, 3). Whenever there is a cycle, there can be paths of arbitrary length. Thus it may become more difficult to show that a graph traversal algorithm is guaranteed to terminate.

Treating the search process as a graph search rather than as a tree search reduces the amount of effort that is spent exploring essentially the same path several times. But it requires additional effort each time a node is

Copyrighted material

generated to see if it has been generated before. Whether this effort is justified depends on the particular problem. If it is very likely that the same node will be generated in several different ways, then it is more worthwhile to use a graph procedure than if such duplication will happen only rarely.

Graph search procedures are especially useful for dealing with partially commutative production systems in which a given set of operations will produce the same result regardless of the order in which the operations are applied. A systematic search procedure will try many of the permutations of these operators and so will generate the same node many times. This is exactly what happened in the water jug example shown above.

2.6 ADDITIONAL PROBLEMS

Several specific problems have been discussed throughout this chapter. Other problems have not yet been mentioned, but are common throughout the AI literature. Some have become such classics that no AI book could be complete without them, so we present them in this section. A useful exercise, at this point, would be to evaluate each of them in light of the seven problem characteristics we have just discussed.

A brief justification is perhaps required before this parade of toy problems is presented. Artificial intelligence is not merely a science of toy problems and microworlds (such as the blocks world). Many of the techniques that have been developed for these problems have become the core of systems that solve very nontoy problems. So think about these problems not as defining the scope of AI but rather as providing a core from which much more has developed.

The Missionaries and Cannibals Problem

Three missionaries and three cannibals find themselves on one side of a river. They have agreed that they would all like to get to the other side. But the missionaries are not sure what else the cannibals have agreed to. So the missionaries want to manage the trip across the river in such a way that the number of missionaries on either side of the river is never less than the number of cannibals who are on the same side. The only boat available holds only two people at a time. How can everyone get across the river without the missionaries risking being eaten?

The Tower of Hanoi

Somewhere near Hanoi there is a monastery whose monks devote their lives to a very important task. In their courtyard are three tall posts. On these posts is a set of sixty-four disks, each with a hole in the center and each of a different radius. When the monastery was established, all of the disks were on one of the posts, each disk resting on the one just larger than it. The monks' task is to move all of the disks to one of the other pegs. Only one disk may be moved at a time, and all the other disks must be on one of the pegs. In addition, at no time during the process may a disk be placed on top of a smaller disk. The third peg can, of course, be used as a temporary resting place for the disks. What is the quickest way for the monks to accomplish their mission?

Even the best solution to this problem will take the monks a very long time. This is fortunate, since legend has it that the world will end when they have finished.

The Monkey and Bananas Problem

A hungry monkey finds himself in a room in which a bunch of bananas is hanging from the ceiling. The monkey, unfortunately, cannot reach the bananas. However, in the room there are also a chair and a stick. The ceiling is just the right height so that a monkey standing on a chair could knock the bananas down with the stick. The monkey knows how to move around, carry other things around, reach for the bananas, and wave a stick in the air. What is the best sequence of actions for the monkey to take to acquire lunch?

Copyrighted material

SEND	DONALD	CROSS
+MORE	+GERALD	+ROADS
.....	.....	.....
MONEY	ROBERT	DANGER

Fig. 2.20 *Some Cryptarithmetic Problems*

Cryptarithmetic

Consider an arithmetic problem represented in letters, as shown iff the examples in Fig. 2.20. Assign a decimal digit to each of the letters in such a way that the answer to the problem is correct. If the same letter occurs more than once, it must be assigned the same digit each time. No two different letters may be assigned the same digit.

People's strategies for solving cryptarithmetic problems have been, studied intensively by Newell and Simon [1972].

SUMMARY

In this chapter, we have discussed the first two steps that must be taken toward the design of a program to solve a particular problem:

1. Define the problem precisely. Specify the problem space, the operators for moving within the space, and the starting and goal state(s).
2. Analyze the problem to determine where it falls with respect to seven important issues.

The last two steps for developing a program to solve that problem are, of course:

3. Identify and represent the knowledge required by the task.
4. Choose one or more techniques for problem solving, and apply those techniques to the problem.

Several general-purpose, problem-solving techniques are presented in the next chapter, and several of them have already been alluded to in the discussion of the problem characteristics in this chapter. The relationships between problem characteristics and specific techniques should become even clearer as we go on. Then, in Part II, we discuss the issue of how domain knowledge is to be represented.

EXERCISES

1. In this chapter, the following problems were mentioned:
   - Chess
   - 8-puzzle
   - Missionaries and cannibals
   - Monkey and bananas
   - Bridge
   - Water jug
   - Traveling salesman
   - Tower of Hanoi
   - Cryptarithmetic

   Analyze each of them with respect to the seven problem characteristics discussed in Section 2.3.
2. Before we can solve a problem using state space search, we must define an appropriate state space. For each of the problems mentioned above for which it was not done in the text, find a good state space representation.
3. Describe how the branch-and-bound technique could be used to find the shortest solution to a water jug problem.

Copyrighted material

4. For each of the following types of problems, try to describe a good heuristic function:
   (a) Blocks world
   (b) Theorem proving
   (c) Missionaries and cannibals
5. Give an example of a problem for which breadth-first search would work better than depth-first search. Give an example of a problem for which depth-first search would work better than breadth-first search.
6. Write an algorithm to perform breadth-first search of a problem *graph*. Make sure your algorithm works properly when a single node is generated at more than one level in the graph.
7. Try to construct an algorithm for solving blocks world problems, such as the one in Fig. 2.10. Do not cheat by looking ahead to Chapter 13.

Copyrighted material

CHAPTER 3

HEURISTIC SEARCH TECHNIQUES

Failure is the opportunity to begin again more intelligently.

—**Moshe Arens**
(1925-), Israeli politician

In the last chapter, we saw that many of the problems that fall within the purview of artificial intelligence are too complex to be solved by direct techniques; rather they must be attacked by appropriate search methods armed with whatever direct techniques are available to guide the search. In this chapter, a framework for describing search methods is provided and several general-purpose search techniques are discussed. These methods are all varieties of heuristic search. They can be described independently of any particular task or problem domain. But when applied to particular problems, their efficacy is highly dependent on the way they exploit domain-specific knowledge since in and of themselves they are unable to overcome the combinatorial explosion to which search processes are so vulnerable. For this reason, these techniques are often called *weak methods*. Although a realization of the limited effectiveness of these weak methods to solve hard problems by themselves has been an important result that emerged from the last three decades of AI research, these techniques continue to provide the framework into which domain-specific knowledge can be placed, either by hand or as a result of automatic learning. Thus they continue to form the core of most AI systems. We have already discussed two very basic search strategies:

- Depth-first search
- Breadth-first search

In the rest of this chapter, we present some others:

- Generate-and-test
- Hill climbing
- Best-first search
- Problem reduction
- Constraint satisfaction
- Means-ends analysis

3.1 GENERATE-AND-TEST

The generate-and-test strategy is the simplest of all the approaches we discuss. It consists of the following steps:

Algorithm: Generate-and-Test

1. Generate a possible solution. For some problems, this means generating a particular point in the problem space. For others, it means generating a path from a start state.

Copyrighted material

2. Test to see if this is actually a solution by comparing the chosen point or the endpoint of the chosen path to the set of acceptable goal states.
3. If a solution has been found, quit. Otherwise, return to step 1.

If the generation of possible solutions is done systematically, then this procedure will find a solution eventually, if one exists. Unfortunately, if the problem space is very large, "eventually" may be a very long time.

The generate-and-test algorithm is a depth-first search procedure since complete solutions must be generated before they can be tested. In its most systematic form, it is simply an exhaustive search of the problem space. Generate-and-test can, of course, also operate by generating solutions randomly, but then there is no guarantee that a solution will ever be found. In this form, it is also known as the British Museum algorithm, a reference to a method for finding an object in the British Museum by wandering randomly.[1] Between these two extremes lies a practical middle ground in which the search process proceeds systematically, but some paths are not considered because they seem unlikely to lead to a solution. This evaluation is performed by a heuristic function, as described in Section 2.2.2.

The most straightforward way to implement systematic generate-and-test is as a depth-first search tree with backtracking. If some intermediate states are likely to appear often in the tree, however, it may be better to modify that procedure, as described above, to traverse a graph rather than a tree.

For simple problems, exhaustive generate-and-test is often a reasonable technique. For example, consider the puzzle that consists of four six-sided cubes, with each side of each cube painted one of four colors. A solution to the puzzle consists of an arrangement of the cubes in a row such that on all four sides of the row one block face of each color is showing. This problem can be solved by a person (who is a much slower processor for this sort of thing than even a very cheap computer) in several minutes by systematically and exhaustively trying all possibilities. It can be solved even more quickly using a heuristic generate-and-test procedure. A quick glance at the four blocks reveals that there are more, say, red faces than there are of other colors. Thus when placing a block with several red faces, it would be a good idea to use as few of them as possible as outside faces. As many of them as possible should be placed to abut the next block. Using this heuristic, many configurations need never be explored and a solution can be found quite quickly.

Unfortunately, for problems much harder than this, even heuristic generate-and-test, all by itself, is not a very effective technique. But when combined with other techniques to restrict the space in which to search even further, the technique can be very effective.

For example, one early example of a successful AI program is DENDRAL [Lindsay *et al.*, 1980], which infers the structure of organic compounds using mass spectrogram and nuclear magnetic resonance (NMR) data. It uses a strategy called *plan-generate-test* in which a planning process that uses constraint-satisfaction techniques (see Section 3.5) creates lists of recommended and contraindicated substructures. The generate-and-test procedure then uses those lists so that it can explore only a fairly limited set of structures. Constrained in this way, the generate-and-test procedure has proved highly effective.

This combination of planning, using one problem-solving method (in this case, constraint satisfaction) with the use of the plan by another problem-solving method, generate-and-test, is an excellent example of the way techniques can be combined to overcome the limitations that each possesses individually. A major weakness of planning is that it often produces somewhat inaccurate solutions since there is no feedback from the world. But by using it only to produce pieces of solutions that will then be exploited in the generate-and-test process, the lack of detailed accuracy becomes unimportant. And, at the same time, the combinatorial problems that arise in simple generate-and-test are avoided by judicious reference to the plans.

[1] Or, as another story goes, if a sufficient number of monkeys were placed in front of a set of typewriters and left alone long enough, then they would eventually produce all of the works of Shakespeare.

Copyrighted material

3.2 HILL CLIMBING

Hill climbing is a variant of generate-and-test in which feedback from the test procedure is used to help the generator decide which direction to move in the search space. In a pure generate-and-test procedure, the test function responds with only a yes or no. But if the test function is augmented with a heuristic function[2] that provides an estimate of how close a given state is to a goal state, the generate procedure can exploit it as shown in the procedure below. This is particularly nice because often the computation of the heuristic function can be done at almost no cost at the same time that the test for a solution is being performed. Hill climbing is often used when a good heuristic function is available for evaluating states but when no other useful knowledge is available. For example, suppose you are in an unfamiliar city without a map and you want to get downtown. You simply aim for the tall buildings. The heuristic function is just distance between the current location and the location of the tall buildings and the desirable states are those in which this distance is minimized.

Recall from Section 2.3.4 that one way to characterize problems is according to their answer to the question, "Is a good solution absolute or relative?" Absolute solutions exist whenever it is possible to recognize a goal state just by examining it. Getting downtown is an example of such a problem. For these problems, hill climbing can terminate whenever a goal state is reached. Only relative solutions exist, however, for maximization (or minimization) problems, such as the traveling salesman problem. In these problems, there is no *a priori* goal state. For problems of this sort, it makes sense to terminate hill climibing when there is no reasonable alternative state to move to.

3.2.1 Simple Hill Climbing

The simplest way to implement hill climbing is as follows.

Algorithm: Simple Hill Climbing

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise, continue with the initial state as the current state.
2. Loop until a solution is found or until there are no new operators left to be applied in the current state:
   (a) Select an operator that has not yet been applied to the current state and apply it to produce a new state.
   (b) Evaluate the new state.
      (i) If it is a goal state, then return it and quit.
      (ii) If it is not a goal state but it is better than the current state, then make it the current state.
      (iii) If it is not better than the current state, then continue in the loop.

The key difference between this algorithm and the one we gave for generate-and-test is the use of an evaluation function as a way to inject task-specific knowledge into the control process. It is the use of such knowledge that makes this and the other methods discussed in the rest of this chapter *heuristic* search methods, and it is that same knowledge that gives these methods their power to solve some otherwise intractable problems.

Notice that in this algorithm, we have asked the relatively vague question, "Is one state *better* than another?" For the algorithm to work, a precise definition of *better* must be provided. In some cases, it means a higher value of the heuristic function. In others, it means a lower value. It does not matter which, as long as a particular hill-climbing program is consistent in its interpretation.

To see how hill climbing works, let's return to the puzzle of the four colored blocks. To solve the problem, we first need to define a heuristic function that describes how close a particular configuration is to being a solution. One such function is simply the sum of the number of different colors on each of the four sides. A solution to the puzzle will have a value of 16. Next we need to define a set of rules that describe ways of transforming one configuration into another. Actually, one rule will suffice. It says simply pick a block and

[2] What we are calling the heuristic function is sometimes also called the *objective function*, particularly in the literature of mathematical optimization.

Copyrighted material

rotate it 90 degrees in any direction. Having provided these definitions, the next step is to generate a starting configuration. This can either be done at random or with the aid of the heuristic function described in the last section. Now hill climbing can begin. We generate a new state by selecting a block and rotating it. If the resulting state is better, then we keep it. If not, we return to the previous state and try a different perturbation.

3.2.2 Steepest-Ascent Hill Climbing

A useful variation on simple hill climbing considers all the moves from the current state and selects the best one as the next state. This method is called *steepest-ascent hill climbing* or *gradient search*. Notice that this contrasts with the basic method in which the first state that is better than the current state is selected. The algorithm works as follows.

Algorithm: Steepest-Ascent Hill Climbing

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise, continue with the initial state as the current state.
2. Loop until a solution is found or until a complete iteration produces no change to current state:
   (a) Let *SUCC* be a state such that any possible successor of the current state will be better than *SUCC*.
   (b) For each operator that applies to the current state do:
      (i) Apply the operator and generate a new state.
      (ii) Evaluate the new state. If it is a goal state, then return it and quit. If not, compare it to *SUCC*. If it is better, then set *SUCC* to this state. If it is not better, leave *SUCC* alone.
   (c) If the *SUCC* is better than current state, then set current state to *SUCC*.

To apply steepest-ascent hill climbing to the colored blocks problem, we must consider all perturbations of the initial state and choose the best. For this problem, this is difficult since there are so many possible moves. There is a trade-off between the time required to select a move (usually longer for steepest-ascent hill climbing) and the number of moves required to get to a solution (usually longer for basic hill climbing) that must be considered when deciding which method will work better for a particular problem.

Both basic and steepest-ascent hill climbing may fail to find a solution. Either algorithm may terminate not by finding a goal state but by getting to a state from which no better states can be generated. This will happen if the program has reached either a local maximum, a plateau, or a ridge.

A *local maximum* is a state that is better than all its neighbors but is not better than some other states farther away. At a local maximum, all moves appear to make things worse. Local maxima are particularly frustrating because they often occur almost within sight of a solution. In this case, they are called *foothills*.

A plateau is a flat area of the search space in which a whole set of neighboring states have the same value. On a plateau, it is not possible to determine the best direction in which to move by making local comparisons.

A *ridge* is a special kind of local maximum. It is an area of the search space that is higher than surrounding areas and that itself has a slope (which one would like to climb). But the orientation of the high region, compared to the set of available moves and the directions in which they move, makes it impossible to traverse a ridge by single moves.

There are some ways of dealing with these problems, although these methods are by no means guaranteed:

- Backtrack to some earlier node and try going in a different direction. This is particularly reasonable if at that node there was another direction that looked as promising or almost as promising as the one that was chosen earlier. To implement this strategy, maintain a list of paths almost taken and go back to one of them if the path that was taken leads to a dead end. This is a fairly good way of dealing with local maxima.
- Make a big jump in some direction to try to get to a new section of the search space. This is a particularly good way of dealing with plateaus. If the only rules available describe single small steps, apply them several times in the same direction.
- Apply two or more rules before doing the test. This corresponds to moving in several directions at once. This is a particularly good strategy for dealing with ridges.

Copyrighted material

Even with these first-aid measures, hill climbing is not always very effective. It is particularly unsuited to problems where the value of the heuristic function drops off suddenly as you move away from a solution. This is often the case whenever any sort of threshold effect is present. Hill climbing is a local method, by which we mean that it decides what to do next by looking only at the "immediate" consequences of its choice rather than by exhaustively exploring all the consequences. It shares with other local methods, such as the nearest neighbor heuristic described in Section 2.2.2, the advantage of being less combinatorially explosive than comparable global methods. But it also shares with other local methods a lack of a guarantee that it will be effective. Although it is true that the hill-climbing procedure itself looks only one move ahead and not any farther, that examination may in fact exploit an arbitrary amount of global information if that information is encoded in the heuristic function. Consider the blocks world problem shown in Fig. 3.1. Assume the same operators (i.e., pick up one block and put it on the table; pick up one block and put it on another one) that were used in Section 2.3.1. Suppose we use the following heuristic function:

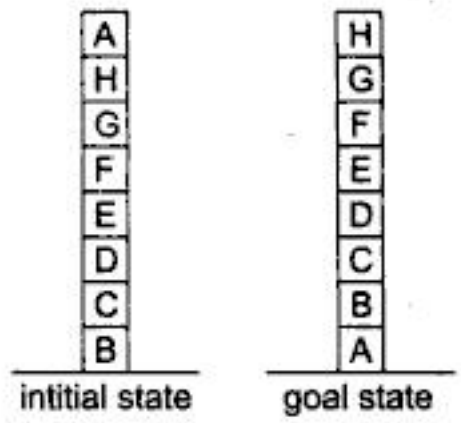


Fig. 3.1 *A Hill-Climbing Problem*

> **Local:** Add one point for every block that is resting on the thing it is supposed to be resting on. Subtract one point for every block that is sitting on the wrong thing.

Using this function, the goal state has a score of 8. The initial state has a score of 4 (since it gets one point added for blocks C, D, E, F, G, and H and one point subtracted for blocks A and B). There is only one move from the initial state, namely to move block A to the table. That produces a state with a score of 6 (since now A's position causes a point to be added rather than subtracted). The hill-climbing procedure will accept that move. From the new state, there are three possible moves, leading to the three states shown in Fig. 3.2. These states have the scores: (*a*) 4, (*b*) 4, and (*c*) 4. Hill climbing will halt because all these states have lower scores than the current state. The process has reached a local maximum that is not the global maximum. The problem is that by purely local examination of support structures, the current state appears to be better than any of its successors because more blocks rest on the correct objects. To solve this problem, it is necessary to disassemble a good local structure (the stack B through H) because it is in the wrong global context.

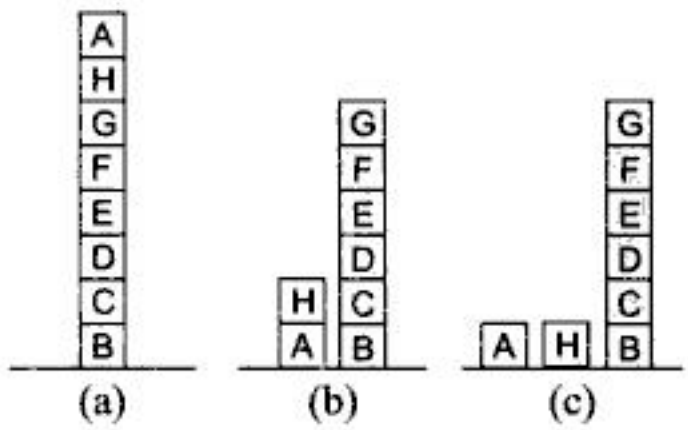


Fig. 3.2 *Three Possible Moves*

We could blame hill climbing itself for this failure to look far enough ahead to find a solution. But we could also blame the heuristic function and try to modify it. Suppose we try the following heuristic function in place of the first one:

> **Global:** For each block that has the correct support structure (i.e., the complete structure underneath it is exactly as it should be), add one point for every block in the support structure. For each block that has an incorrect support structure, subtract one point for every block in the existing support structure.

Using this function, the goal state has the score 28 (1 for B, 2 for C, etc.). The initial state has the score —28. Moving A to the table yields a state with a score of —21 since A no longer has seven wrong blocks under it. The three states that can be produced next now have the following scores: (*a*) –28, (*b*) –16, and (*c*) –15. This time, steepest-ascent hill climbing will choose move (*c*), which is the correct one. This new heuristic function captures the two key aspects of this problem: incorrect structures are bad and should be taken apart;

Copyrighted material

and correct structures are good and should be built up. As a result, the same hill climbing procedure that failed with the earlier heuristic function now works perfectly.

Unfortunately, it is not always possible to construct such a perfect heuristic function. For example, consider again the problem of driving downtown. The perfect heuristic function would need to have knowledge about one-way and dead-end streets, which, in the case of a strange city, is not always available. And even if perfect knowledge is, in principle, available, it may not be computationally tractable to use. As an extreme example, imagine a heuristic function that computes a value for a state by invoking its own problem-solving procedure to look ahead from the state it is given to find a solution. It then knows the exact cost of finding that solution and can return that cost as its value. A heuristic function that does this converts the local hill-climbing procedure into a global method by embedding a global method within it. But now the computational advantages of a local method have been lost. Thus it is still true that hill climbing can be very inefficient in a large, rough problem space. But it is often useful when combined with other methods that get it started in the right general neighborhood.

3.2.3 Simulated Annealing

Simulated annealing is a variation of hill climbing in which, at the beginning of the process, some downhill moves may be made. The idea is to do enough exploration of the whole space early on so that the final solution is relatively insensitive to the starting state. This should lower the chances of getting caught at a local maximum, a plateau, or a ridge.

In order to be compatible with standard usage in discussions of simulated annealing, we make two notational changes for the duration of this section. We use the term *objective function* in place of the term *heuristic function.*

And we attempt to *minimize* rather than maximize the value of the objective function. Thus we actually describe a process of valley descending rather than hill climbing.

Simulated annealing [Kirkpatrick *et al.*, 1983] as a computational process is patterned after the physical process *of annealing,* in which physical substances such as metals are melted (i.e., raised to high energy levels) and then gradually cooled until some solid state is reached. The goal of this process is to produce a minimal-energy final state. Thus this process is one of valley descending in which the objective function is the energy level. Physical substances usually move from higher energy configurations to lower ones, so the valley descending occurs naturally. But there is some probability that a transition to a higher energy state will occur. This probability is given by the function

$$p = e^{-\Delta E/kT}$$

where A £ is the positive change in the energy level T is the temperature, and k is Boltzmann's constant. Thus, in the physical valley descending that occurs during annealing, the probability of a large uphill move is lower than the probability of a small one. Also, the probability that an uphill move will be made decreases as the temperature decreases. Thus such moves are more likely during the beginning of the process when the temperature is high, and they become less likely at the end as the temperature becomes lower. One way to characterize this process is that downhill moves are allowed anytime. Large upward moves may occur early on, but as the process progresses, only relatively small upward moves are allowed until finally the process converges to a local minimum configuration.

The rate at which the system is cooled is called the *annealing schedule.* Physical annealing processes are very sensitive to the annealing schedule. If cooling occurs too rapidly, stable regions of high energy will form. In other words, a local but not global minimum is reached. If, however, a slower schedule is used, a uniform crystalline structure, which corresponds to a global minimum, is more likely to develop. But, if the schedule is too slow, time is wasted. At high temperatures, where essentially random motion is allowed, nothing useful happens. At low temperatures a lot of time may be wasted after the final structure has already been formed. The optimal annealing schedule for each particular annealing problem must usually be discovered empirically.

Copyrighted material

These properties of physical annealing can be used to define an analogous process of simulated annealing, which can be used (although not always effectively) whenever simple hill climbing can be used. In this analogous process, ΔE is generalized so that it represents not specifically the change in energy but more generally, the change in the value of the objective function, whatever it is. The analogy for kT is slightly less straightforward. In the physical process, temperature is a well-defined notion, measured in standard units. The variable k describes the correspondence between the units of temperature and the units of energy. Since, in the analogous process, the units for both E and T are artificial, it makes sense to incorporate k into T, selecting values for T that produce desirable behavior on the part of the algorithm. Thus we use the revised probability formula

$$p' = e^{-\Delta E/T}$$

But we still need to choose a schedule of values for T (which we still call temperature). We discuss this briefly below after we present the simulated annealing algorithm.

The algorithm for simulated annealing is only slightly different from the simple hill-climbing procedure. The three differences are:

- The annealing schedule must be maintained.
- Moves to worse states may be accepted.
- It is a good idea to maintain, in addition to the current state, the best state found so far. Then, if the final state is worse than that earlier state (because of bad luck in accepting moves to worse states), the earlier state is still available.

Algorithm: Simulated Annealing

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise, continue with the initial state as the current state.
2. Initialize *BEST-SO-FAR* to the current state.
3. Initialize T according to the annealing schedule.
4. Loop until a solution is found or until there are no new operators left to be applied in the current state.
   (a) Select an operator that has not yet been applied to the current state and apply it to produce a new state.
   (b) Evaluate the new state. Compute
   $$\Delta E = (\text{value of current}) - (\text{value of new state})$$
   - If the new state is a goal state, then return it and quit.
   - If it is not a goal state but is better than the current state, then make it the current state. Also set *BEST-SO-FAR* to this new state.
   - If it is not better than the current state, then make it the current state with probability p' as defined above. This step is usually implemented by invoking a random number generator to produce a number in the range [0,1]. If that number is less than p', then the move is accepted. Otherwise, do nothing.

   (c) Revise T as necessary according to the annealing schedule.
5. Return *BEST-SO-FAR*, as the answer.

To implement this revised algorithm, it is necessary to select an annealing schedule, which has three components. The first is the initial value to be used for temperature. The second is the criteria that will be used to decide when the temperature of the system should be reduced. The third is the amount by which the temperature will be reduced each time it is changed. There may also be a fourth component of the schedule, namely, when to quit. Simulated annealing is often used to solve problems in which the number of moves from a given state is very

Copyrighted material

large (such as the number of permutations that can be made to a proposed traveling salesman route). For such problems, it may not make sense to try all possible moves. Instead, it may be useful to exploit some criterion involving the number of moves that have been tried since an improvement was found.

Experiments that have been done with simulated annealing on a variety of problems suggest that the best way to select an annealing schedule is by trying several and observing the effect on both the quality of the solution that is found and the rate at which the process converges. To begin to get a feel for how to come up with a schedule, the first thing to notice is that as T approaches zero, the probability of accepting a move to a worse state goes to zero and simulated annealing becomes identical to simple hill climbing. The second thing to notice is that what really matters in computing the probability of accepting a move is the ratio $\Delta E/T$. Thus it is important that values of T be scaled so that this ratio is meaningful. For example, T could be initialized to a value such that, for an average ΔE, p' would be 0.5.

Chapter 18 returns to simulated annealing in the context of neural networks.

3.3 BEST-FIRST SEARCH

Until now, we have really only discussed two systematic control strategies, breadth-first search and depth-first search (of several varieties). In this section, we discuss a new method, best-first search, which is a way of combining the advantages of both depth-first and breadth-first search into a single method.

3.3.1 OR Graphs

Depth-first search is good because it allows a solution to be found without all competing branches having to be expanded. Breadth-first search is good because it does not get trapped on dead-end paths. One way of combining the two is to follow a single path at a time, but switch paths whenever some competing path looks more promising than the current one does.

At each step of the best-first search process, we select the most promising of the nodes we have generated so far. This is done by applying an appropriate heuristic function to each of them. We then expand the chosen node by using the rules to generate its successors. If one of them is a solution, we can quit. If not, all those new nodes are added to the set of nodes generated so far. Again the most promising node is selected and the process continues. Usually what happens is that a bit of depth-first searching occurs as the most promising branch is explored. But eventually, if a solution is not found, that branch will start to look less promising than one of the top-level branches that had been ignored. At that point, the now more promising, previously ignored branch will be explored. But the old branch is not forgotten.. Its last node remains in the set of generated but unexpanded nodes. The search can return to it whenever all the others get bad enough that it is again the most promising path.

Figure 3.3 shows the beginning of a best-first search procedure. Initially, there is only one node, so it will be expanded. Doing so generates three new nodes. The heuristic function, which, in this example, is an estimate of the cost of getting to a solution from a given node, is applied to each of these new nodes. Since node D is the most promising, it is expanded next, producing two successor nodes, E and F. But then the heuristic function is applied to them. Now another path, that going through node B, looks more promising, so it is pursued, generating nodes G and H. But again when these new nodes are evaluated they look less promising than another path, so attention is returned to the path through D to E. E is then expanded, yielding nodes I and J. At the next step, J will be expanded, since it is the most promising. This process can continue until a solution is found.

Notice that this procedure is very similar to the procedure for steepest-ascent hill climbing, with two exceptions. In hill climbing, one move is selected and all the others are rejected, never to be reconsidered. This produces the straightline behavior that is characteristic of hill climbing. In best-first search, one move is selected, but the others are kept around so that they can be revisited later if the selected path becomes less

Copyrighted material

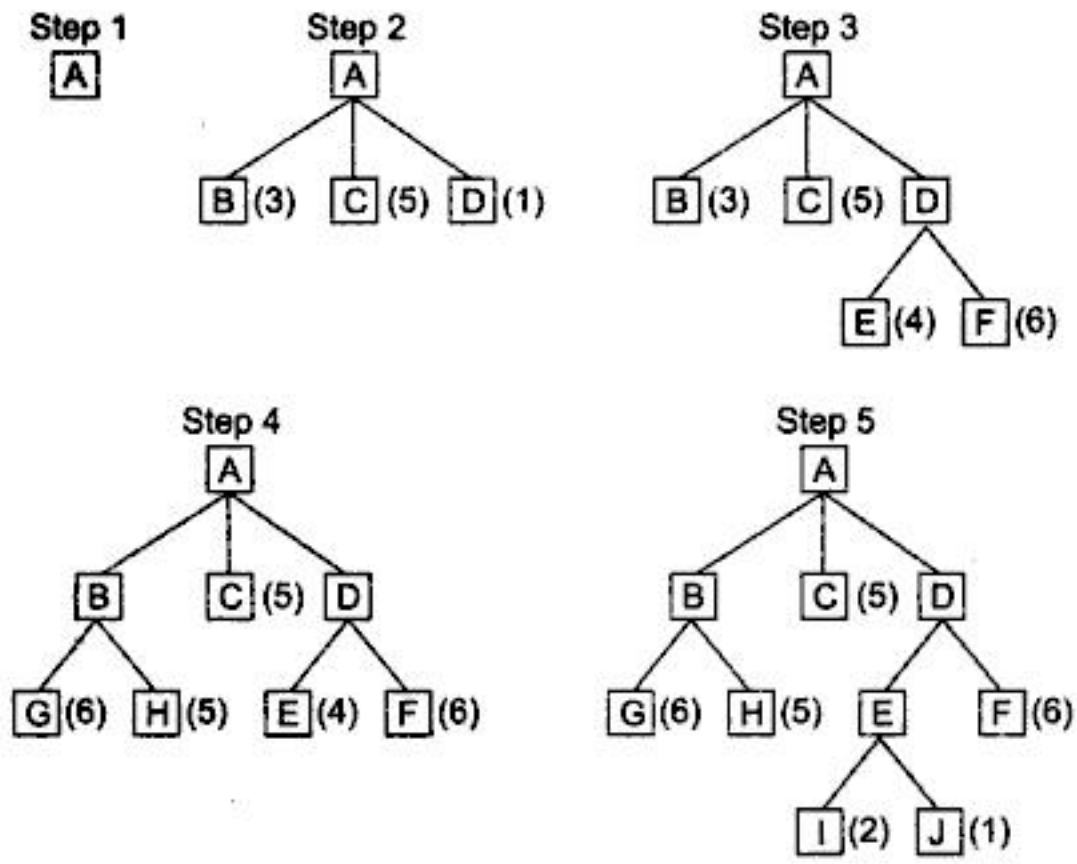


Fig. 3.3 *A Best-First Search*

promising.[3] Further, the best available state is selected in best-first search, even if that state has a value that is lower than the value of the state that was just explored. This contrasts with hill climbing, which will stop if there are no successor states with better values than the current state.

Although the example shown above illustrates a best-first search of a tree, it is sometimes important to search a graph instead so that duplicate paths will not be pursued. An algorithm to do this will operate by searching a directed graph in which each node represents a point in the problem space. Each node will contain, in addition to a description of the problem state it represents, an indication of how promising it is, a parent link that points back to the best node from which it came, and a list of the nodes that were generated from it. The parent link will make it possible to recover the path to the goal once the goal is found. The list of successors will make it possible, if a better path is found to an already existing node, to propagate the improvement down to its successors. We will call a graph of this sort an *OR graph*, since each of its branches represents an alternative problem-solving path.

To implement such a graph-search procedure, we will need to use two lists of nodes:

- *OPEN* — nodes that have been generated and have had the heuristic function applied to them but which have not yet been examined (i.e., had their successors generated). *OPEN* is actually a priority queue in which the elements with the highest priority are those with the most promising value of the heuristic function. Standard techniques for manipulating priority queues can be used to manipulate the list.
- *CLOSED* — nodes that have already been examined. We need to keep these nodes in memory if we want to search a graph rather than a tree, since whenever a new node is generated, we need to check whether it has been generated before.

We will also need a heuristic function that estimates the merits of each node we generate. This will enable the algorithm to search more promising paths first. Call this function f'(to indicate that it is an approximation to a

[3] In a variation of best-first search, called *beam search*, only the n most promising states are kept for future consideration. This procedure is more efficient with respect to memory but introduces the possibility of missing a solution altogether by pruning the search tree too early.

Copyrighted material

function/that gives the true evaluation of the node). For many applications, it is convenient to define this function as the sum of two components that we call g and h'. The function g is a measure of the cost of getting from the initial state to the current node. Note that g is not an estimate of anything; it is known to be the exact sum of the costs of applying each of the rules that were applied along the best path to the node. The function h' is an estimate of the additional cost of getting from the current node to a goal state. This is the place where knowledge about the problem domain is exploited. The combined function f', then, represents an estimate of the cost of getting from the initial state to a goal state along the path that generated the current node. If more than one path generated the node, then the algorithm will record the best one. Note that because g and h' must be added, it is important that h' be a measure of the cost of getting from the node to a solution (i.e., good nodes get low values; bad nodes get high values) rather than a measure of the goodness of a node (i.e., good nodes get high values). But that is easy to arrange with judicious placement of minus signs. It is also important that g be nonnegative. If this is not true, then paths that traverse cycles in the graph will appear to get better as they get longer.

The actual operation of the algorithm is very simple. It proceeds in steps, expanding one node at each step, until it generates a node that corresponds to a goal state. At each step, it picks the most promising of the nodes that have so far been generated but not expanded. It generates the successors of the chosen node, applies the heuristic function to them, and adds them to the list of open nodes, after checking to see if any of them have been generated before. By doing this check, we can guarantee that each node only appears once in the graph, although many nodes may point to it as a successor. Then the next step begins.

This process can be summarized as follows.

Algorithm: Best-First Search

1. Start with *OPEN* containing just the initial state.
2. Until a goal is found or there are no nodes left on *OPEN* do:
   (a) Pick them best node on *OPEN*.
   (b) Generate its successors.
   (c) For each successor do:
      (i) If it has not been generated before, evaluate it, add it to *OPEN*, and record its parent.
      (ii) If it has been generated before, change the parent if this new path is better than the previous one. In that case, update the cost of getting to this node and to any successors that this node may already, have.

The basic idea of this algorithm is simple. Unfortunately, it is rarely the case that graph traversal algorithms are simple to write correctly. And it is even rarer that it is simple to guarantee the correctness of such algorithms. In the section that follows, we describe this algorithm in more detail as an example of the design and analysis of a graph-search program.

3.3.2 The A* Algorithm

The best-first search algorithm that was just presented is a simplification of an algorithm called A*, which was first presented by Hart *et al.* [1968; 1972]. This algorithm uses the same f', g, and h' functions, as well as the lists *OPEN* and *CLOSED*, that we have already described.

Algorithm: A*

1. Start with *OPEN* containing only the initial node. Set that node's g value to 0, its h' value to whatever it is, and its f' value to $h' + 0$, or h'. Set *CLOSED* to the empty list.
2. Until a goal node is found, repeat the following procedure: If there are no nodes on *OPEN*, report failure. Otherwise, pick the node on *OPEN* with the lowest f' value. Call it *BESTNODE*. Remove it from *OPEN*. Place it on *CLOSED*. See if *BESTNODE* is a goal node. If so, exit and report a solution (either *BESTNODE* if all we want is the node or the path that has been created between the initial state

Copyrighted material

and *BESTNODE* if we are interested in the path). Otherwise, generate the successors of *BESTNODE* but do not set *BESTNODE* to point to them yet. (First we need to see if any of them have already been generated.) For each such *SUCCESSOR*, do the following:

(a) Set *SUCCESSOR* to point back to *BESTNODE*. These backwards links will make it possible to recover the path once a solution is found.

(b) Compute $g(SUCCESSOR) = g(BESTNODE)$ + the cost of getting from *BESTNODE* to *SUCCESSOR*.

(c) See if *SUCCESSOR* is the same as any node on *OPEN* (i.e., it has already been generated but not processed). If so, call that node *OLD*. Since this node already exists in the graph, we can throw *SUCCESSOR* away and add *OLD* to the list of *BESTNODE's* successors. Now we must decide whether *OLD's* parent link should be reset to point to *BESTNODE*. It should be if the path we have just found to *SUCCESSOR* is cheaper than the current best path to *OLD* (since *SUCCESSOR* and *OLD* are really the same node). So see whether it is cheaper to get to *OLD* via its current parent or to *SUCCESSOR* via *BESTNODE* by comparing their g values. If *OLD* is cheaper (or just as cheap), then we need do nothing. If *SUCCESSOR* is cheaper, then reset *OLD's* parent link to point to *BESTNODE*, record the new cheaper path in $g(OLD)$, and update f'(OLD).

(d) If *SUCCESSOR* was not on *OPEN*, see if it is on *CLOSED*. If so, call the node on *CLOSED OLD* and add *OLD* to the list of *BESTNODE's*, successors. Check to see if the new path or the old path is better just as in step *2(c)*, and set the parent link-and g and f' values appropriately. If we have just found a better path to *OLD*, we must propagate the improvement to *OLD's* successors. This is a bit tricky. *OLD* points to its successors. Each successor in turn points to its successors, and so forth, until each branch terminates with a node that either is still on *OPEN* or has no successors. So to propagate the new cost downward, do a depth-first traversal of the tree starting at *OLD*, changing each node's g value (and thus also its f' value), terminating each branch when you reach either a node with no successors or a node to which an equivalent or better path has already been found.[4] This condition is easy to check for. Each node's parent link points back to its best known parent. As we propagate down to a node, see if its parent points to the node we are coming from. If so, continue the propagation. If not, then its g value already reflects the better path of which it is part. So the propagation may stop here. But it is possible that with the new value of g being propagated downward, the path we are following may become better than the path through the current parent. So compare the two. If the path through the current parent is still better, stop the propagation. If the path we are propagating through is now better, reset the parent and continue propagation.

(e) If *SUCCESSOR* was not already on either *OPEN* or *CLOSED*, then put it on *OPEN*, and add it to the list of *BESTNODE's* successors. Compute $f'(.SUCCESSOR) = g(SUCCESSOR) + h'(SUCCESSOR)$.

Several interesting observations can be made about this algorithm. The first concerns the role of the g function. It lets us choose which node to expand next on the basis not only of how good the node itself looks (as measured by h'), but also on the basis of how good the path to the node was. By incorporating g into f', we will not always choose as our next node to expand the node that appears to be closest to the goal. This is useful if we care about the path we find. If, on the other hand, we only care about getting to a solution somehow, we can define g always to be 0, thus always choosing the node that seems closest to a goal. If we want to find a path involving the fewest number of steps, then we set the cost of going from a node to its successor as a constant, usually 1. If, on the other hand, we want to find the cheapest path and some operators cost more than others, then we set the

[4] This second check guarantees that the algorithm will terminate even if there are cycles in the graph. If there is a cycle, then the second time that a given node is visited, the path will be no better than the first time and so propagation will stop.

Copyrighted material

cost of going from one node to another to reflect those costs. Thus the A* algorithm can be used whether we are interested in finding a minimal-cost overall path or simply any path as quickly as possible.

The second observation involves h', the estimator of h, the distance of a node to the goal. If h' is a perfect estimator of h, then A* will converge immediately to the goal with no search. The better hi is, the closer we will get to that direct approach. If, on the other hand, the value of h' is always 0, the search will be controlled by g. If the value of g is also 0, the search strategy will be random. If the value of g is always 1, the search will be breadth first. All nodes on one level will have lower g values, and thus lower f' values than will all nodes on the next level. What if, on the other hand, h' is neither perfect nor 0? Can we say anything interesting about the behavior of the search? The answer is yes if we can guarantee that h' never overestimates h. In that case, the A* algorithm is guaranteed to find an optimal (as determined by g) path to a goal, if one exists. This can easily be seen from a few examples.[5]

Consider the situation shown in Fig. 3.4. Assume that the cost of all arcs is 1. Initially, all nodes except A are on *OPEN* (although the Fig. shows the situation two steps later, after B and E have been expanded). For each node, f' is indicated as the sum of h' and g. In this example, node B has the lowest f', 4, so it is expanded first. Suppose it has only one successor E, which also appears to be three moves away from a goal. Now f'(E) is 5, the same as f'(C). Suppose we resolve this in favor of the path we are currently following. Then we will expand E next. Suppose it too has a single successor F, also judged to be three moves from a goal. We are clearly using up moves and making no progress. But f'(F) = 6, which is greater than f'(C). So we will expand C next. Thus we see that by underestimating $h'(B)$ we have wasted some effort. But eventually we discover that B was farther away than we thought and we go back and try another path.

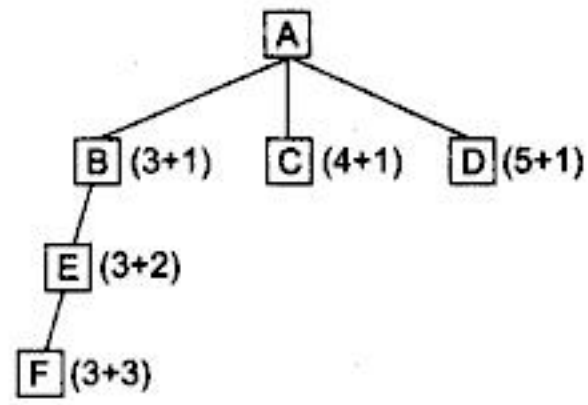


Fig. 3.4 *h′ Underestimates h*

Now consider the situation shown in Fig. 3.5. Again we expand B on the first step. On the second step we again expand E. At the next step we expand F, and finally we generate G, for a solution path of length 4. But suppose there is a direct path from D to a solution, giving a path of length 2. We will never find it. By overestimating $h'(D)$ we make D look so bad that we may find some other, worse solution without ever expanding D. In general, if h' might overestimate h, we cannot be guaranteed of finding the cheapest path solution unless we expand the entire graph until all paths are longer than the best solution. An interesting question is, "Of what practical significance is the theorem that if h' never overestimates h then A* is admissible?" The answer is, "almost none," because, for most real problems, the only way to guarantee that hi never overestimates h is to set it to zero. But then we are back to breadth-first search, which is admissible but not efficient. But there is a corollary to this theorem that is very useful. We can state it loosely as follows:

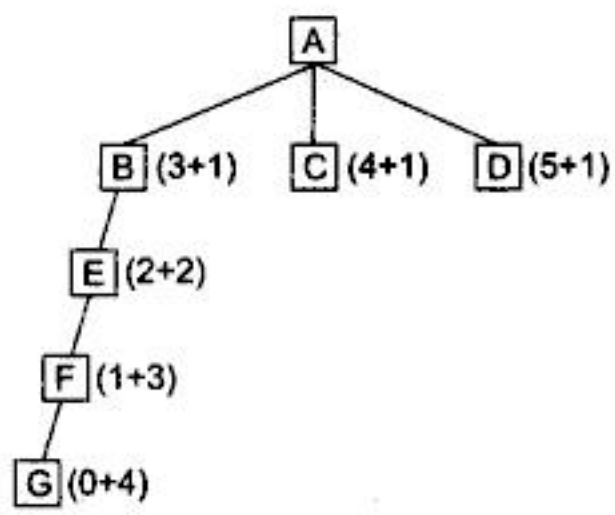


Fig. 3.5 *h′ Overestimates h*

> **Graceful Decay of Admissibility:** If h' rarely overestimates h by more than δ, then the A* algorithm will rarely find a solution whose cost is more than δ greater than the cost of the optimal solution.

The formalization and proof of this corollary will be left as an exercise.

The third observation we can make about the A* algorithm has to do with the relationship between trees and graphs. The algorithm was stated in its most general form as it applies to graphs. It can, of course, be

[5] A search algorithm that is guaranteed to find an optimal path to a goal, if one exists, is called *admissible* [Nilsson, 1980].

Copyrighted material

simplified to apply to trees by not bothering to check whether a new node is already on *OPEN* or *CLOSED*. This makes it faster to generate nodes but may result in the same search being conducted many times if nodes are often duplicated.

Under certain conditions, the A* algorithm can be shown to be optimal in that it generates the fewest nodes in the process of finding a solution to a problem. Under other conditions it is not optimal. For formal discussions of these conditions, see Gelperin [1977] and Martelli [1977].

3.3.3 Agendas

In our discussion of best-first search in OR graphs, we assumed that we could evaluate multiple paths to the same node independently of each other. For example, in the water jug problem, it makes no difference to the evaluation of the merit of the position (4, 3) that there are at least two separate paths by which it could be reached. This is not true, however, in all situations, e.g., especially when there is no single, simple heuristic function that measures the distance between a given node and a goal.

Consider, for example, the task faced by the mathematics discovery program AM, written by Lenat [1977; 1982]. AM was given a small set of starting facts about number theory and a set of operators it could use to develop new ideas. These operators included such things as "Find examples of a concept you already know." AM's goal was to generate new "interesting" mathematical concepts. It succeeded in discovering such things as prime numbers and Goldbach's conjecture.

Armed solely with its basic operators, AM would have been able to create a great many new concepts, most of which would have been worthless. It needed a way to decide intelligently which rules to apply. For this it was provided with a set of heuristic rules that said such things as "The extreme cases of any concept are likely to be interesting." "Interest" was then used as the measure of merit of individual tasks that the system could perform. The system operated by selecting at each cycle the most interesting task, doing it, and possibly generating new tasks in the process. This corresponds to the selection of the most promising node in the best-first search procedure. But in AM's situation the fact that several paths recommend the same task does matter. Each contributes a reason why the task would lead to an interesting result. The more such reasons there are, the more likely it is that the task really would lead to something good. So we need a way to record proposed tasks along with the reasons they have been proposed. AM used a task agenda. An *agenda* is a list of tasks a system could perform. Associated with each task there are usually two things: a list of reasons why the task is being proposed (often called *justifications*) and a rating representing the overall weight of evidence suggesting that the task would be useful.

An agenda-driven system uses the following procedure.

Algorithm: Agenda-Driven Search

1. Do until a goal state is reached or the agenda is empty:
   (a) Choose the most promising task from the agenda. Notice that this task can be represented in any desired form. It can be thought of as an explicit statement of what to do next or simply as an indication of the next node to be expanded.
   (b) Execute the task by devoting to it the number of resources determined by its importance. The important resources to consider are time and space. Executing the task will probably generate additional tasks (successor nodes). For each of them, do the following:
      (i) See if it is already on the agenda. If so, then see if this same reason for doing it is already on its list of justifications. If so, ignore this current evidence. If this justification was not already present, add it to the list. If the task was not on the agenda, insert it.
      (ii) Compute the new task's rating, combining the evidence from all its justifications. Not all justifications need have equal weight. It is often useful to associate with each justification a measure of how strong a reason it is. These measures are then combined at this step to produce an overall rating for the task.

Copyrighted material

One important question that arises in agenda-driven systems is how to find the most promising task on each cycle. One way to do this is simple. Maintain the agenda sorted by rating. When a new task is created, insert it into the agenda in its proper place. When a task has its justifications changed, recompute its rating and move it to the correct place in the list. But this method causes a great deal of time to be spent keeping the agenda in perfect order. Much of this time is wasted since we do not need perfect order. We only need to know the proper first element. The following modified strategy may occasionally cause a task other than the best to be executed, but it is significantly cheaper than the perfect method. When a task is proposed, or a new justification is added to an existing task, compute the new rating and compare it against the top few (e.g., five or ten) elements on the agenda. If it is better, insert the node into its proper position at the top of the list. Otherwise, leave it where it is or simply insert it at the end of the agenda. At the beginning of each cycle, choose the first task on the agenda. In addition, once in a while, go through the agenda and reorder it properly.

An agenda-driven control structure is also useful if some tasks (or nodes) provide negative evidence about the merits of other tasks (or nodes). This can be represented by justifications with negative weightings. If these negative weightings are used, it may be important to check not only for the possibility of moving a task to the head of the agenda but also of moving a top task to the bottom if new, negative justifications appear. But this is easy to do.

As you can see, the agenda mechanism provides a good way of focusing the attention of a complex system in the areas suggested by the greatest number of positive indicators. But the overhead for each task executed may be fairly high. This raises the question of the proper grain size for the division of the entire problem-solving process into individual tasks. Suppose each task is very small. Then we will never do even a very small thing unless it really is the best thing to do. But we will spend a large percentage of our total effort on figuring out what to do next. If, on the other hand, the size of an individual task is very large, then some effort may be spent finishing one task when there are more promising ones that could be done. But a smaller percentage of the total time will be spent on the overhead of figuring out what to do. The exact choice of task size for a particular system depends on the extent to which doing one small thing really means that a set of other small things is likely to be very good to do too. It often requires some experimentation to get right.

There are some problem domains for which an agenda mechanism is inappropriate. The agenda mechanism assumes that if there is good reason to do something now, then there will also be the same good reason to do something later unless something better comes along in the interim. But this is not always the case, particularly for systems that are interacting with people. The following dialogue would not be acceptable to most people:

Person:	I don't want to read any more about China. Give me something else.
Computer:	OK. What else are you interested in?
Person:	How about Italy? I think I'd find Italy fascinating.
Computer:	What things about Italy are you interested in reading about?
Person:	I think I'd like to start with its history.
Computer:	Why don't you want to read any more about China?

It would have been fine to have tried to find out why the person was no longer interested in China right after he or she mentioned it. The computer chose instead to try to find a new area of positive interest, also a very reasonable thing to do. But in conversations, the fact that something is reasonable now does not mean that it will continue to be so after the conversation has proceeded for a while. So it is not a good idea simply to put possible statements on an agenda, wait until a later lull, and then pop out with them. More precisely, agendas are a good way to implement monotonic production systems (in the sense of Section 2.4) and a poor way to implement nonmonotonic ones.

Despite these difficulties, agenda-driven control structures are very useful. They provide an excellent way of integrating information from a variety of sources into one program since each source simply adds tasks and

Copyrighted material

justifications to the agenda. As AI programs become more complex and their knowledge bases grow, this becomes a particularly significant advantage.

3.4 PROBLEM REDUCTION

So far, we have considered search strategies for OR graphs through which we want to find a single, path to a goal. Such structures represent the fact that we will know how to get from a node to a goal state if we can discover how to get from that node to a goal state along any one of the branches leaving it.

3.4.1 AND-OR Graphs

Another kind of structure, the AND-OR graph (or tree), is useful for representing the solution of problems that can be solved by decomposing them into a set of smaller problems, all of which must then be solved. This decomposition, or reduction, generates arcs that we call AND arcs. One AND arc may point to any number of successor nodes, all of which must be solved in order for the arc to point to a solution. Just as in an OR graph, several arcs may emerge from a single node, indicating a variety of ways in which the briginal problem might be solved. This is why the structure is called not simply an AND graph but rather an AND-OR graph. An example of an AND-OR graph (which also happens to be an AND-OR tree) is given in Fig. 3.6. AND arcs are indicated with a line connecting all the components.

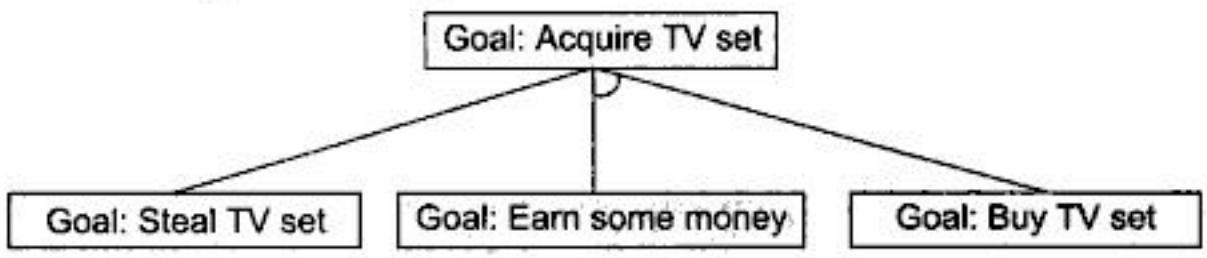


Fig. 3.6 *A Simple AND-OR Graph*

In order to find solutions in an AND-OR graph, we need an algorithm similar to best-first search but with the ability to handle the AND arcs appropriately. This algorithm should find a path from the starting node of the graph to a set of nodes representing solution states. Notice that it may be necessary to get to more than one solution state since each arm of an AND arc must lead to its own solution node.

To see why our best-first search algorithm is not adequate for searching AND-OR graphs, consider Fig. 3.7(a). The top node, A, has been expanded, producing two arcs, one leading to B and one leading to C and D. The numbers at each node represent the value of f' at that node. We assume, for simplicity, that every operation has a uniform cost, so each arc with a single successor has a cost of 1 and each AND arc with multiple successors has a cost of 1 for each of its components. If we look just at the nodes and choose for expansion the one with the lowest f' value, we must select C. But using the information now available, it would be better to explore the path going through B since to use C we must also use D, for a total cost of 9 (C + D + 2) compared to the cost of 6 that we get by going through B. The problem is that the choice of which node to expand next must depend not only on

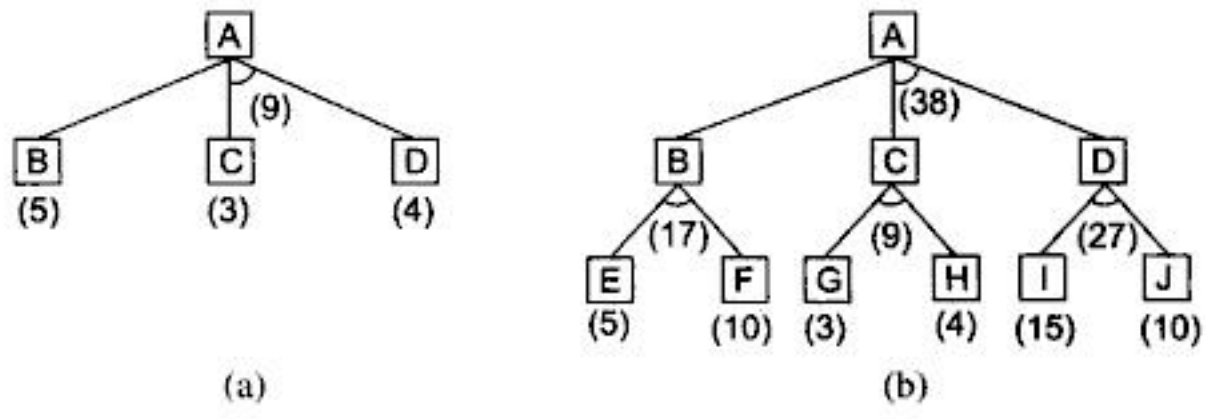


Fig. 3.7 *AND-OR Graphs*

Copyrighted material

the f' value of that node but also on whether that node is part of the current best path from the initial node. The tree shown in Fig. 3.7(b) makes this even clearer. The most promising single node is G with an f' value of 3. It is even part of the most promising arc G-H, with a total cost of 9. But that arc is not part of the current best path since to use it we must also use the arc I-J, with a cost of 27. The path from A, through B, to E and F is better, with a total cost of 18. So we should not expand G next; rather we should examine either E or F.

In order to describe an algorithm for searching an AND-OR graph, we need to exploit a value that we call *FUTILITY*. If the estimated cost of a solution becomes greater than the value of *FUTILITY*, then we abandon the search. *FUTILITY* should be chosen to correspond to a threshold such that any solution with a cosflibove it is too expensive to be practical, even if it could ever be found. Now we can state the algorithm.

Algorithm: Problem Reduction

1. Initialize the graph to the starting node.
2. Loop until the starting node is labeled *SOLVED* or until its cost goes above *FUTILITY*:
   (a) Traverse the graph, starting at the initial node and following the current best path, and accumulate the set of nodes that are on that path and have not yet been expanded or labeled as solved.
   (b) Pick one of these unexpanded nodes and expand it. If there are no successors, assign *FUTILITY* as the value of this node. Otherwise, add its successors to the graph and for each of them compute f' (use only h' and ignore g, for reasons we discuss below). If of any node is 0, mark that node as *SOLVED*.
   (c) Change the f' estimate of the newly expanded node to reflect the new information provided by its successors. Propagate this change backward through the graph. If any node contains a successor arc whose descendants are all solved, label the node itself as *SOLVED*. At each node that is visited while going up the graph, decide which of its successor arcs is the most promising and mark it as part of the current best path. This may cause the current best path to change. This propagation of revised cost estimates back up the tree was not necessary in the best-first search algorithm because only unexpanded nodes were examined. But now expanded nodes must be reexamined so that the best current path can be selected. Thus it is important that their f' values be the best estimates available.

This process is illustrated in Fig. 3.8. At step 1, A is the only node, so it is at the end of the current best path. It is expanded, yielding nodes B, C, and D. The arc to D is labeled as the most promising one emerging from A, since it costs 6 compared to B and C, which costs 9. (Marked arcs are indicated in the Fig.s by arrows.) In step 2, node D) is chosen for expansion. This process produces one new arc, the AND arc to E and F, with a combined cost estimate of 10. So we update the f' value of D to 10. Going back one more level, we see that this makes the AND arc B-C better than the arc to D, so it is labeled as the current best path. At step 3, we traverse that arc from A and discover the unexpanded nodes B and C. If we are going to find a solution along this path, we will have to expand both B and C eventually, so let's choose to explore B first. This generates two new arcs, the ones to G and to H. Propagating their f' values backward, we update f' of B to 6 (since that is the best we think we can do, which we can achieve by going through G). This requires updating the cost of the AND arc B-C to 12 (6 + 4 + 2). After doing that, the arc to D is again the better path from A, so we record that as the current best path and either node E or node F will be chosen for expansion at step 4. This process continues until either a solution is found or all paths have led to dead ends, indicating that there is no solution.

In addition to the difference discussed above, there is a second important way in which an algorithm for searching an AND-OR graph must differ from one for searching an OR graph. This difference, too, arises from the fact that individual paths from node to node cannot be considered independently of the paths through other nodes connected to the original ones by AND arcs. In the best-first search algorithm, the desired path

Copyrighted material

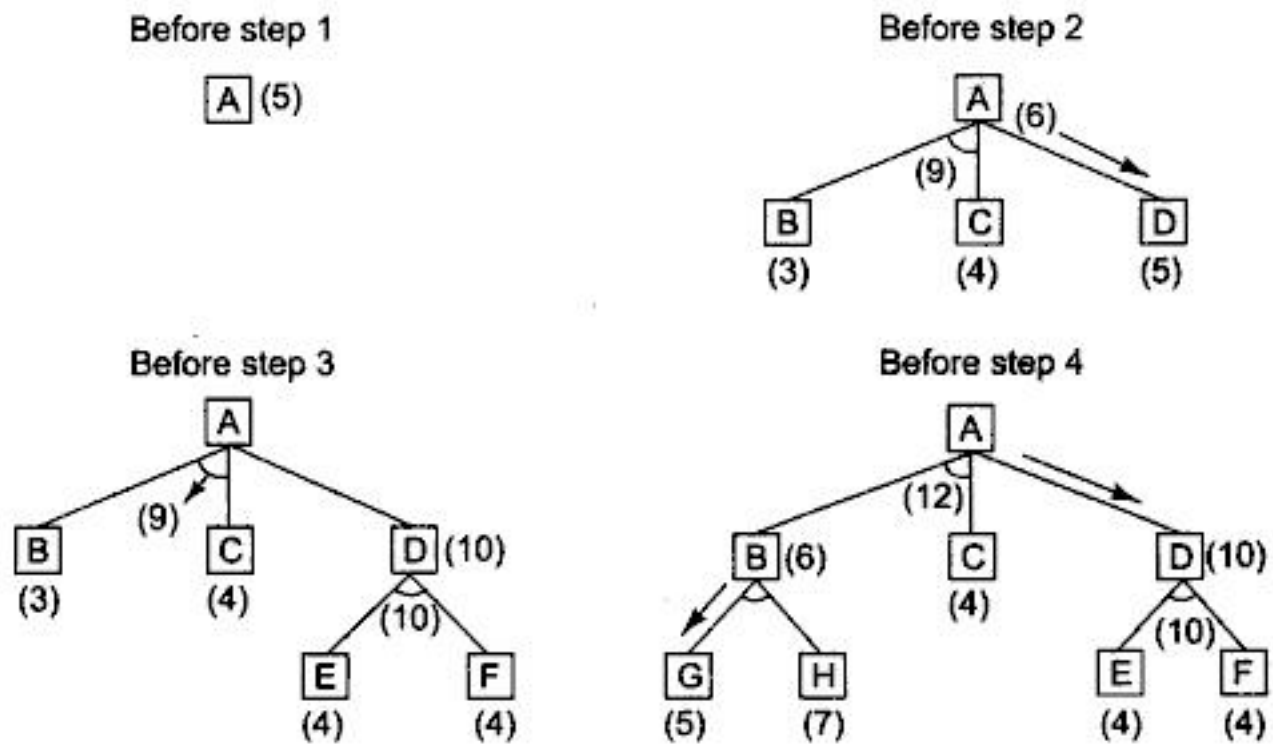


Fig. 3.8 *The Operation of Problem Reduction*

from one node to another was always the one with the lowest cost. But this is not always the case when searching an AND-OR graph.

Consider the example shown in Fig. 3.9(a). The nodes were generated in alphabetical order. Now suppose that node J is expanded at the next step and that one of its successors is node E, producing the graph shown in Fig. 3.9(b). This new path to E is longer than the previous path to E going through C. But since the path through C will only lead to a solution if there is also a solution to D, which we know there is not, the path through J is better.

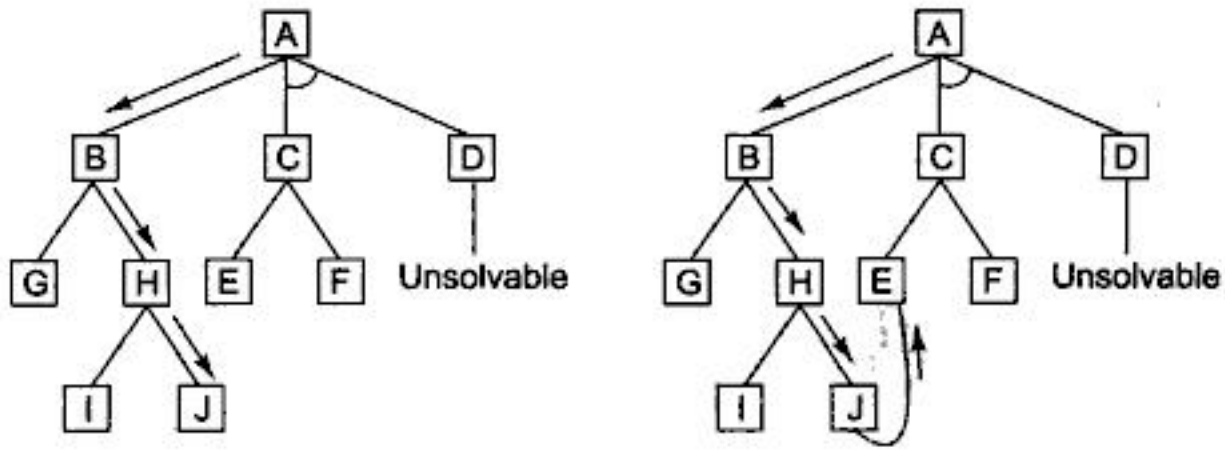


Fig. 3.9 *A Longer Path May Be Better*

There is one important limitation of the algorithm we have just described. It fails to take into account any interaction between subgoals. A simple example of this failure is shown in Fig. 3.10. Assuming that both node C and node E ultimately lead to a solution, our algorithm will report a complete solution that includes both of them. The AND-OR graph states that for A to be solved, both C and D must be solved. But then the algorithm considers the solution of D as a completely separate process from the solution of C. Looking just at the alternatives from D, E is the best path. But it turns out that C is necessary anyway, so it would be better also to use it to satisfy D. But since our algorithm does not consider such interactions, it will find a nonoptimal path. In Chapter 13, problem-solving methods that can consider interactions among subgoals are presented.

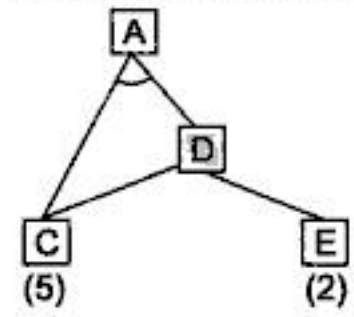


Fig. 3.10 *Interacting Subgoals*

Copyrighted material

3.4.2 The AO* Algorithm

The problem reduction algorithm we just described is a simplification of an algorithm described in Martelli and Montanari [1973], Martelli and Montanari [1978], and Nilsson [1980]. Nilsson calls it the AO* algorithm, the name we assume.

Rather than the two lists, *OPEN* and *CLOSED*, that were used in the A* algorithm, the AO* algorithm will use a single structure *GRAPH*, representing the part of the search graph that has been explicitly generated so far. Each node in the graph will point both down to its immediate successors and up to its immediate predecessors. Each node in the graph will also have associated with it an h' value, an estimate of the cost of a path from itself to a set of solution nodes. We. will not store g (the cost of getting from the start node to the current node) as we did in the A* algorithm. It is not possible to compute a single such value since there may be many paths to the same state. And such a value is not necessary because of the top-down traversing of the best-known path, which guarantees that only nodes that are on the best path will ever be considered for expansion. So h' will serve as the estimate of goodness of a node.

Algorithm: AO*

1. Let *GRAPH* consist only of the node representing the initial state. (Call this node *INIT.)* Compute $h'(INIT)$
2. Until *INIT* is labeled *SOLVED* or until *INIT's* h' value becomes greater than *FUTILITY*, repeat the following procedure:
   (a) Trace the labeled arcs from *INIT* and select for expansion one of the as yet unexpanded nodes that occurs on this path. Call the selected node *NODE*.
   (b) Generate the successors of *NODE*. If there are none, then assign *FUTILITY* as the h' value of *NODE*. This is equivalent to saying that *NODE* is not solvable. If there are successors, then for each one (called *SUCCESSOR)* that is not also an ancestor of *NODE* do the following:
      (i) Add *SUCCESSOR* to *GRAPH*.
      (ii) If *SUCCESSOR* is a terminal node, label it *SOLVED* and assign it an h' value of 0.
      (iii) If *SUCCESSOR* is not a terminal node, compute its h' value.
   (c) Propagate the newly discovered information up the graph by doing the following: Let S be a set of nodes that have been labeled *SOLVED* or whose h' values have been changed and so need to have values propagated back to their parents. Initialize 5 to *NODE*. Until S is empty, repeat the, following procedure:
      (i) If possible, select from S a node none of whose descendants in *GRAPH* occurs in S. If there is no such node, select any node from S. Call this node *CURRENT*, and remove it from S.
      (ii) Compute the cost of each of the arcs emerging from *CURRENT*. The cost of each arc is equal to the sum of the h' values of each of the nodes at the end of the arc plus whatever the cost of the arc itself is. Assign as *CURRENT'S* new h' value the minimum of the costs just computed for the arcs emerging from it.
      (iii) Mark the best path out of *CURRENT* by marking the arc that had the minimum cost as computed in the previous step.
      (iv) Mark *CURRENT SOLVED* if all of the nodes connected to it through the new labeled arc have been labeled *SOLVED*.
      (v) If *CURRENT* has been labeled *SOLVED* or if the cost of *CURRENT* was just changed, then its new status must be propagated back up the graph. So add all of the ancestors of *CURRENT* to S.

It is worth noticing a couple of points about the operation of this algorithm. In step $2(c)v$, the ancestors of anode whose cost was altered are added to the set of nodes whose costs must also be revised. As stated, the algorithm will insert all the node's ancestors' into the set, which may result in the propagation of the cost

Copyrighted material

change back up through a large number of paths that are already known not to be very good. For example, in Fig. 3.11, it is clear that the path through C will always be better than the path through B, so work expended on the path through B is wasted. But if the cost of E is revised and that change is not propagated up through B as well as through C, B may appear to be better. For example, if, as a result of expanding node E, we update its cost to 10, then the cost of C will be updated to 11. If this is all that is done, then when A is examined, the path through B will have a cost of only 11 compared to 12 for the path through C, and it will be labeled erroneously as the most promising path. In this example, the mistake might be detected at the next step, during which D will be expanded. If its cost changes and is propagated back to B, B's cost will be recomputed and the new cost of E will be used. Then the new cost of B will propagate back to A. At that point, the path through C will again be better. All that happened was that some time was wasted in expanding D. But if the node whose cost has changed is farther down in the search graph, the error may never be detected. An example of this is shown in Fig. 3.12(a). If the cost of G is revised as shown in Fig. 3.12(b) and if it is not immediately propagated back to E, then the change will never be recorded and a nonoptimal solution through B may be discovered.

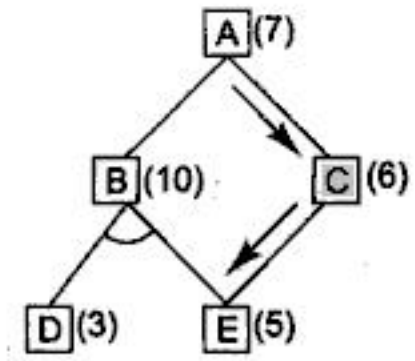


Fig. 3.11 *An Unnecessary Backward Propagation*

A second point concerns the termination of the backward cost propagation of step 2(*c*). Because *GRAPH* may contain cycles, there is no guarantee that this process will terminate simply because it reaches the "top" of the graph. It turns out that the process can be guaranteed to terminate for a different reason, though. One of the exercises at the end of this chapter explores why.

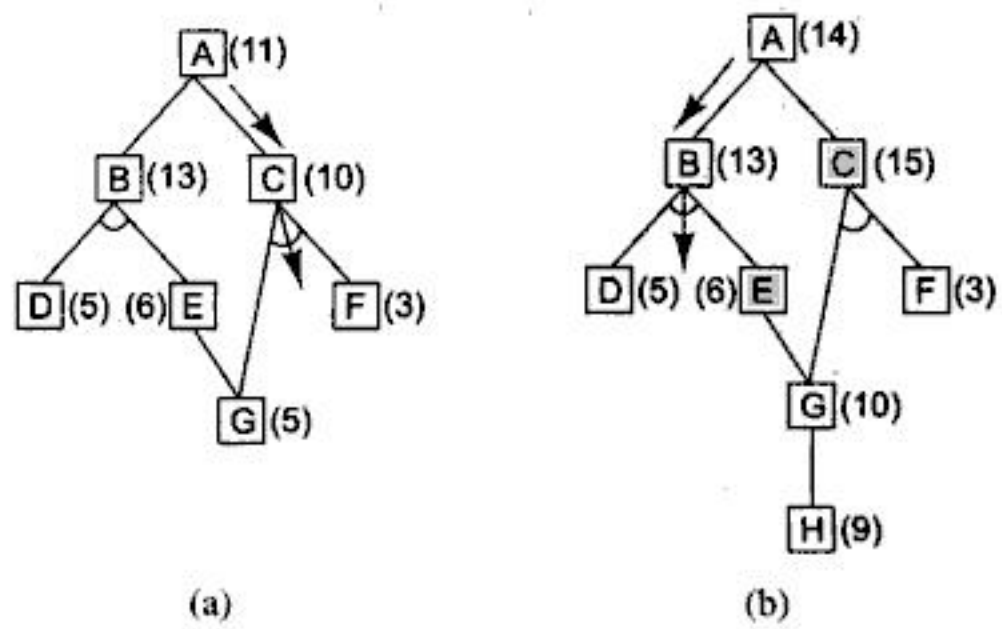


Fig. 3.12 *A Necessary Backward Propagation*

3.5 CONSTRAINT SATISFACTION

Many problems in AI can be viewed as problems of *constraint satisfaction* in which the goal is to discover some problem state that satisfies a given set of constraints. Examples of this sort of problem include cryptarithmetic puzzles (as described in Section 2.6) and many real-world perceptual labeling problems. Design tasks can also be viewed as constraint-satisfaction problems in which a design must be created within fixed limits on time, cost and materials.

By viewing a problem as one of constraint satisfaction, it is often possible to reduce substantially the amount of search that is required as compared with a method that attempts to form partial solutions directly by choosing specific values for components of the eventual solution. For example, a straightforward search procedure to solve a cryptarithmetic problem might operate in a state space of partial solutions in which letters are assigned particular numbers as their valuer. A depth-first control scheme could then follow a path of assignments until either a solution or an inconsistency is discovered. In contrast, a constraint satisfaction approach to solving this problem avoids making guesses on particular assignments of numbers to letters until it has to. Instead, the initial set of constraints, which says that each number may correspond to only one letter and that the sums of the digits must be as they are given in the problem, is first augmented to include restrictions that can be inferred from the rules of arithmetic. Then, although guessing may still be required, the number of allowable guesses is reduced and so the degree of search is curtailed.

Copyrighted material

Constraint satisfaction is a search procedure that operates in a space of constraint sets. The initial state contains the constraints that are originally given in the problem description. A Goal State is any state that has been constrained "enough," where "enough" must be defined for each problem. For example, for cryptarithmetic, enough means that each letter has been assigned a unique numeric value.

Constraint satisfaction is a two-step process. First, constraints are discovered and propagated as far as possible throughout the system. Then, if there is still not a solution, search begins. A guess about something is made and added as a new constraint. Propagation can then occur with this new constraint, and so forth.

The first step, propagation, arises from the fact that there are usually dependencies among the constraints. These dependencies occur because many constraints involve more than one object and many objects participate in more than one constraint. So, for example, assume we start with one constraint, N = E + 1. Then, if we added the constraint N = 3, we could propagate that to get a stronger constraint on E. namely that E = 2. Constraint propagation also arises from the presence of inference rules that allow additional constraints to be inferred from the ones that are given. Constraint propagation terminates for one of two reasons. First, a contradiction may be detected. If this happens, then there is no solution consistent with all the known constraints. If the contradiction involves only those constraints that were given as part of the problem specification (as opposed to ones that were guessed during problem solving), then no solution exists. The second possible reason for termination is that the propagation has run out of steam and there are no further changes that can be made on the basis of current knowledge. If this happens and a solution has not yet been adequately specified, then search is necessary to get the process moving again.

At this point, the second step begins. Some hypothesis about a way to strengthen the constraints must be made. In the case of the cryptarithmetic problem, for example, this usually means guessing a particular value for some letter. Once this has been done, constraint propagation can begin again from this new state. If a solution is found, it can be reported. If still more guesses are required, they can be made. If a contradiction is detected, then backtracking can be used to try a different guess and proceed with it. We can state this procedure more precisely as follows:

Algorithm: Constraint Satisfaction

1. Propagate available constraints. To do this, first set *OPEN* to the set of all objects that must have values assigned to them in a complete solution. Then do until an inconsistency is detected or until *OPEN* is empty:
   (a) Select an object *OB* from *OPEN*. Strengthen as much as possible the set of constraints that apply to *OB*.
   (b) If this set is different from the set that was assigned the last time *OB* was examined or if this is the first time *OB* has been examined, then add to *OPEN* all objects that share any constraints with *OB*.
   (c) Remove *OB* from *OPEN*.
2. If the union of the constraints discovered above defines a solution, then quit and report the solution.
3. If the union of the constraints discovered above defines a contradiction, then return failure.
4. If neither of the above occurs, then it is necessary to make a guess at something in order to proceed. To do this, loop until a solution is found or all possible solutions have been eliminated:
   (a) Select an object whose value is not yet determined and select a way of strengthening the constraints on that object.
   (b) Recursively invoke constraint satisfaction with the current set of constraints augmented by the strengthening constraint just selected.

This algorithm has been stated as generally as possible. To apply it in a particular problem domain requires the use of two kinds of rules: rules that define the way constraints may validly be propagated and rules that suggest guesses when guesses are necessary. It is worth noting, though, that in some problem domains guessing

Copyrighted material

may not be required. For example, the Waltz algorithm for propagating line labels in a picture, which is described in Chapter 14, is a version\of this constraint satisfaction algorithm with the guessing step eliminated. In general, the more powerful the rules for propagating constraints, the less need there is for guessing.

To see how this algorithm works, consider the cryptarithmetic problem shown in Fig. 3.13. The goal state is a problem state in which all letters have been assigned a digit in such a way that all the initial constraints are satisfied.

Problem:

```
  SEND
+ MORE
 ......
 MONEY
```

Initial State:

No two letters have the same value.
The sums of the digits must be as shown in the problem.

Fig. 3.13 *A Cryptarithmetic Problem*

The solution process proceeds in cycles. At each cycle, two significant things are done (corresponding to steps 1 and 4 of this algorithm):

1. Constraints are propagated by using rules that correspond to the properties of arithmetic.
2. A value is guessed for some letter whose value is not yet determined.

In the first step, it does not usually matter a great deal what order the propagation is done in, since all available propagations will be performed before the step ends. In the second step, though, the order in which guesses are tried may have a substantial impact on the degree of search that is necessary. A few useful heuristics can help to select the best guess to try first. For example, if there is a letter that has only two possible values and another with six possible values, there is a better chance of guessing right on the first than on the second. Another useful heuristic is that if there is a letter that participates in many constraints then it is a good idea to prefer it to a letter that participates in a few. A guess on such a highly constrained letter will usually lead quickly either to a contradiction (if it is wrong) or to the generation of many additional constraints (if it is right). A guess on a less constrained letter, on the other hand, provides less information. The result of the first few cycles of processing this example is shown in Fig. 3.14. Since constraints never disappear at lower levels, only the ones being added are shown for each level. It will not be much harder for the problem solver to access the constraints as a set of lists than as one long list, and this approach is efficient both in terms of storage space and the ease of backtracking. Another reasonable approach for this problem would be to store all the constraints in one central database and also to record at each node the changes that must be undone during backtracking. Cl, C2, C3, and C4 indicate the carry bits out of the columns, numbering from the right.

Initially, rules for propagating constraints generate the following additional constraints:

- M = 1, since two single-digit numbers plus a carry cannot total more than 19.
- S = 8 or 9, since S + M + C3 > 9 (to generate the carry) and M = 1, S + 1 + C3 > 9, so S + C3 > 8 and C3 is at most 1.
- O = 0, since S + M(1) + C3 (<= 1) must be at least 10 to generate a carry and it can be at most 11. But M is already 1, so O must be 0.
- N = E or E + 1, depending on the value of C2. But N cannot have the same value as E. So N = E + 1 and C2 is 1.

Copyrighted material

- In order for C2 to be 1, the sum of N + R + Cl must be greater than 9, so N + R must be greater than 8.
- N + R cannot be greater than 18, even with a carry in, so E cannot be 9.

At this point, let us assume that no more constraints can be generated. Then, to make progress from here, we must guess. Suppose E is assigned the value 2. (We chose to guess a value for E because it occurs three times and thus interacts highly with the other letters.) Now the next cycle begins.

The constraint propagator now observes that:

- N = 3, since N = E + 1.
- R = 8 or 9, since R + N (3) + Cl (1 or 0) = 2 or 12. But since N is already 3, the sum of these nonnegative numbers cannot be less than 3. Thus R + 3 + (0 or 1) = 12 and R = 8 or 9.
- 2 + D = Y or 2 + D = 10 + Y, from the sum in the rightmost column.

Again, assuming no further constraints can be generated, a guess is required. Suppose C1 is chosen to guess a value for. If we try the value 1, then we eventually reach dead ends, as shown in the Fig.. When this happens, the process will backtrack and try Cl = 0.

A couple of observations are worth making on this process. Notice that all that is required of the constraint propagation rules is that they do not infer spurious constraints. They do not have to infer all legal ones. For example, we could have reasoned through to the result that Cl equals 0. We could have done so by observing that for Cl to be 1, the following must hold: 2 + D = 10 + Y. For this to be the case, D would have to be 8 or 9. But both S and R must be either 8 or 9 and three letters cannot share two values. So Cl cannot be 1. If we had realized this initially, some search could have been avoided. But since the constraint propagation rules we used were not that sophisticated, it took some search. Whether the search route takes more or less actual time than does the constraint propagation route depends on how long it takes to perform the reasoning required for constraint propagation.

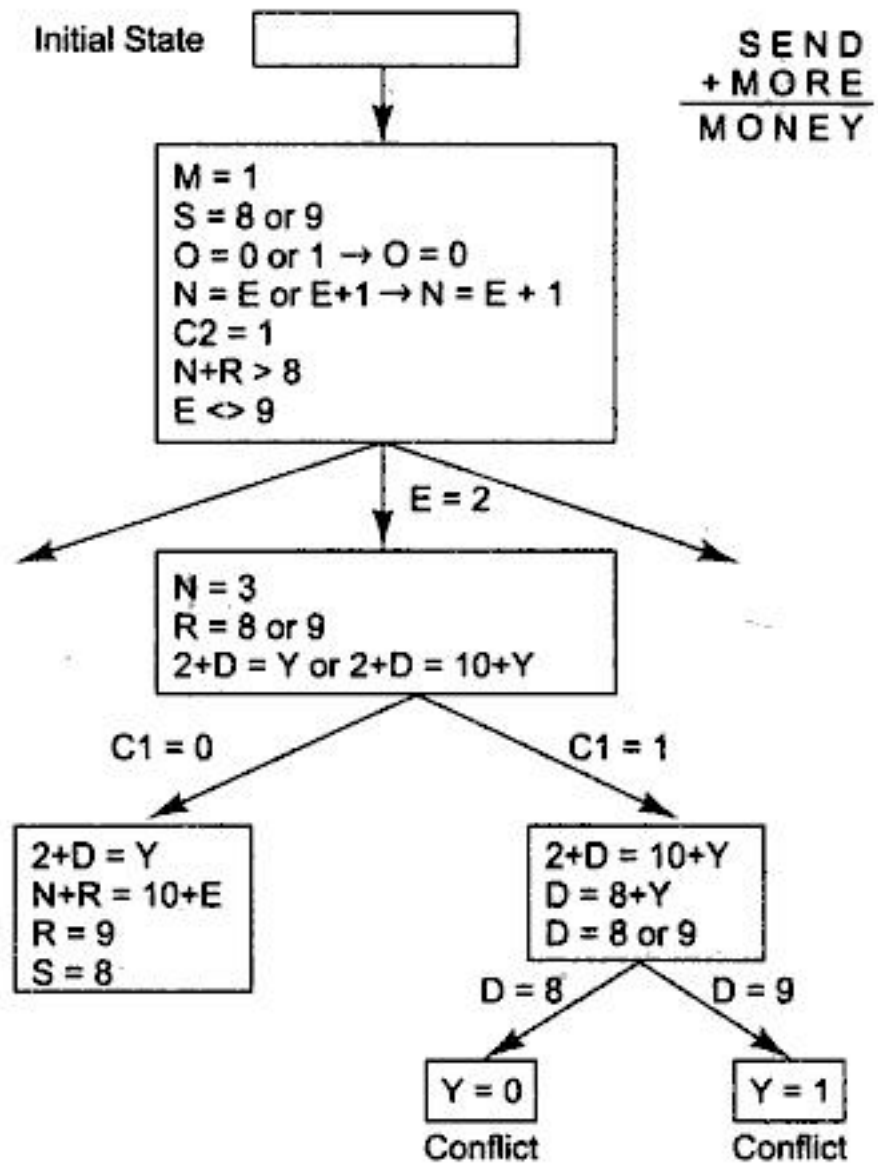


Fig. 3.14 *Solving a Cryptarithmetic Problem*

A second thing to notice is that there are often two kinds of constraints. The first kind is simple; they just list possible values for a single object. The second kind is more complex; they describe relationships between or among objects. Both kinds of constraints play the same role in the constraint satisfaction process, and in the cryptarithmetic example they were treated identically. For some problems, however, it may be useful to represent the two kinds of constraints differently. The simple, value-listing constraints are always dynamic, and so must always be represented explicitly in each problem state. The more complicated, relationship-expressing constraints are dynamic in the cryptarithmetic domain since they are different for each cryptarithmetic problem. But in many other domains they are static. For example, in the Waltz line labeling algorithm, the only binary constraints arise from the nature of the physical world, in which surfaces can meet in only a fixed number of possible ways. These ways are the same for all pictures that that algorithm may see. Whenever thc binary constraints are static, it may be computationally efficient not to represent them explicitly in the state description but rather to encode them in the algorithm directly. When this is done, the only things that get propagated are possible values. But the essential algorithm is the same in both cases.

Copyrighted material

So far, we have described a fairly simple algorithm for constraint satisfaction in which chronological backtracking is used when guessing leads to an inconsistent set of constraints. An alternative is to use a more sophisticated scheme in which the specific cause of the inconsistency is identified and only constraints that depend on that culprit are undone. Others, even though they may have been generated after the culprit, are left alone if they are independent of the problem and its cause. This approach is called dependency-directed backtracking (DDB). It is described in detail in Section 7.5.1.

3.6 MEANS-ENDS ANALYSIS

So far, we have presented a collection of search strategies that can reason either forward or backward, but for a given problem, one direction or the other must be chosen. Often, however, a mixture of the two directions is appropriate. Such a mixed strategy would make it possible to solve the major parts of a problem first and then go back and solve the small problems that arise in "gluing" the big pieces together. A technique known as *means-ends analysis* allows us to do that.

The means-ends analysis process centers around the detection of differences between the current state and the goal state. Once such a difference is isolated, an operator that can reduce the difference must be found. But perhaps that operator cannot be applied to the current state. So we set up a subproblem of getting to a state in which it can be applied. The kind of backward chaining in which operators are selected and then subgoals are set up to establish the preconditions of the operators is called *operator subgoaling*. But maybe the operator does not produce exactly the goal state we want. Then we have a second subproblem of getting from the state it does produce to the goal. But if the difference was chosen correctly and if the operator is really effective at reducing the difference, then the two subproblems should be easier to solve than the original problem. The means-ends analysis process can then be applied recursively. In order to focus the system's attention on the big problems first, the differences can be assigned priority levels. Differences of higher priority can then be considered before lower priority ones.

The first AI program to exploit means-ends analysis was the General Problem Solver (GPS) [Newell and Simon, 1963; Ernst and Newell, 1969]. Its design was motivated by the observation that people often use this technique when they solve problems. But GPS provides a good example of the fuzziness of the boundary between building programs that simulate what people do and building programs that simply solve a problem any way they can.

Just like the other problem-solving techniques we have discussed, means-ends analysis relies on a set of rules that can transform one problem state into another. These rules are usually not represented with complete state descriptions on each side. Instead, they are represented as a left side that describes the conditions that must be met for the rule to be applicable (these conditions are called the rule's *preconditions} and* a right side that describes those aspects of the problem state that will be changed by the application of the rule. A separate data structure called a *difference table* indexes the rules by the differences that they can be used to reduce.

Consider a simple household robot domain. The available operators are shown in Fig. 3.15, along with their preconditions and results. Figure 3.16 shows the difference table that describes when each of the operators is appropriate. Notice that sometimes there may be more than one operator that can reduce a given difference and that a given operator may be able to reduce more than one difference.

Suppose that the robot in this domain were given the problem of moving a desk with two things on it from one room to another. The objects on top must also be moved. The main difference between the start state and the goal state would be the location of the desk. To reduce this difference, either PUSH or CARRY could be chosen. If CARRY is chosen first, its preconditions must be met. This results in two more differences that must be reduced: the location of the robot and the size of the desk. The location of the robot can be handled by applying WALK, but there are no operators than can change the size of an object (since we did not include

Copyrighted material

Operator	*Preconditions*	*Results*
PUSH(obj, loc)	at(robot, obj)^ large(obj)^ clear(obj)^ armempty	at(obj, loc)^ at(robot, loc)
CARRY(obj, loc)	at(robot, obj)^ small(obj)	at(obj, loc)^ at(robot, loc)
WALK(loc)	none	at(robot, loc)
PICKUP(obj)	at(robot, obj)	holding(obj)
PUTDOWN(obj)	holding(obj)	¬holding(obj)
PLACE(obj1, obj2)	at(robot, obj2)^ holding(obj1)	on(obj1, obj2)

Fig. 3.15 *The Robot's Operators*

	Push	Carry	Walk	Pickup	Putdown	Place
Move object	*	*				
Move robot			*			
Clear object				*		
Get object on object						*
Get arm empty					*	*
Be holding object				*		

Fig. 3.16 *A Difference Table*

SAW-APART). So this path leads to a dead-end. Following the other branch, we attempt to apply PUSH. Figure 3.17 shows the problem solver's progress at this point. It has found a way of doing something useful. But it is not yet in a position to do that thing. And the thing does not get it quite to the goal state. So now the differences between A and B and between C and D must be reduced.

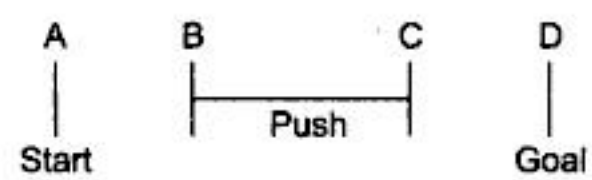


Fig. 3.17 *The Progress of the Means-Ends Analysis Method*

PUSH has four preconditions, two of which produce differences between the start and the goal states: the robot must be at the desk, and the desk must be clear. Since the desk is already large, and the robot's arm is empty, those two preconditions can be ignored. The robot can be brought to the correct location by using WALK. And the surface of the desk can be cleared by two uses of PICKUP. But after one PICKUP, an attempt to do the second results in another difference—the arm must be empty. PUTDOWN can be used to reduce that difference.

Once PUSH is performed, the problem state is close to the goal state, but not quite. The objects must be placed back on the desk. PLACE will put them there. But it cannot be applied immediately. Another difference must be eliminated, since the robot must be holding the objects. The progress of the problem solver at this point is shown in Fig. 3.18.

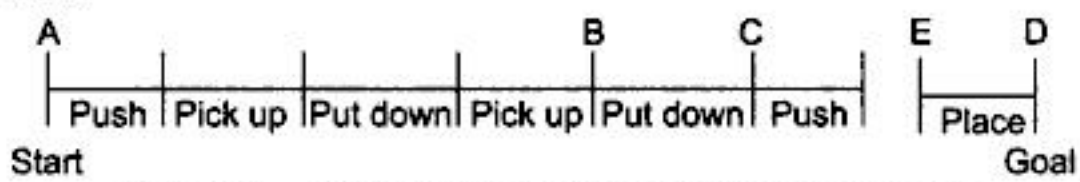


Fig. 3.18 *More Progress of the Means-Ends Method*

Copyrighted material

The final difference between C and E can be reduced by using WALK to get the robot back to the objects, followed by PICKUP and CARRY.

The process we have just illustrated (which we call MEA for short) can be summarized as follows:

Algorithm: Means-Ends Analysis (CURRENT, GOAL)

1. Compare *CURRENT* to *GOAL*. If there are no differences between them then return.
2. Otherwise, select the most important difference and reduce it by doing the following until success or failure is signaled:
   (a) Select an as yet untried operator *O* that is applicable to the current difference. If there are no such operators, then signal failure.
   (b) Attempt to apply *O* to *CURRENT*. Generate descriptions of two states: *O-START*, a state in which *O*'s preconditions are satisfied and *O-RESULT*, the state that would result if *O* were applied in *O-START*.
   (c) If
   (FIRST-PART ← MEA(CURRENT, O-START))
   and
   (LAST-PART ← MEMO-RESULT, GOAL))
   are successful, *then* signal success and return the result of concatenating
   FIRST-PART, O, and *LAST-PART*.

Many of the details of this process have been omitted in this discussion. In particular, the order in which differences are considered can be critical. It is important that significant differences be reduced before less critical ones. If this is not done, a great deal of effort may be wasted on situations that take care of themselves once the main parts of the problem are solved.

The simple process we have described is usually not adequate for solving complex problems. The number of permutations of differences may get too large. Working on one difference may interfere with the plan for reducing another. And in complex worlds, the required difference tables would be immense. In Chapter 13 we look at some ways in which the basic means-ends analysis approach can be extended to tackle some of these problems.

SUMMARY

In Chapter 2, we listed four steps that must be taken to design a program to solve an AI problem. The first two steps were:

1. Define the problem precisely. Specify the problem space, the operators for moving within the space, and the starting and goal state(s).
2. Analyze the problem to determine where it falls with respect to seven important issues.

The other two steps were to isolate and represent the task knowledge required, and to choose problem solving techniques and apply them to the problem. In this chapter, we began our discussion of the last step of this process by presenting some general-purpose, problem-solving methods. There are several important ways in which these algorithms differ, including:

- What the states in the search space(s) represent. Sometimes the states represent complete potential solutions (as in hill climbing). Sometimes they represent solutions that are partially specified (as in constraint satisfaction).

Copyrighted material

- How, at each stage of the search process, a state is selected for expansion.
- How operators to be applied to that node are selected.
- Whether an optimal solution can be guaranteed.
- Whether a given state may end up being considered more than once.
- How many state descriptions must be maintained throughout the Search process.
- Under what circumstances should a particular search path be abandoned.

In the chapters that follow, we talk about ways that knowledge about task domains can be encoded in problem-solving programs and we discuss techniques for combining problem-solving techniques with knowledge to solve several important classes of problems.

EXERCISES

1. When would best-first search be worse than simple breadth-first search?
2. Suppose we have a problem that we intend to solve using a heuristic best-first search procedure. We need to decide whether to implement it as a tree search or as a graph search. Suppose that we know that, on the average, each distinct node will be generated N times during the search process. We also know that if we use a graph, it will take, on the average, the same amount of time to check a node to see if it has already been generated as it takes to process M nodes if no checking is done. How can we decide whether to use a tree or a graph? In addition to the parameters N and M, what other assumptions must be made?
3. Consider trying to solve the 8-puzzle using hill climbing. Can you find a heuristic function that makes this work? Make sure it works on the following example:

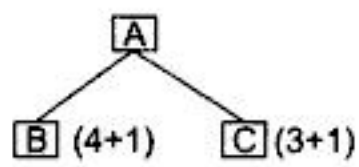
Start

1	2	3
8	5	6
4	7	

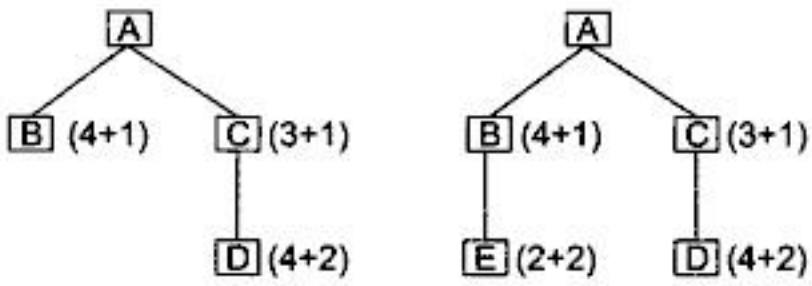
Goal

1	2	3
4	5	6
7	8	

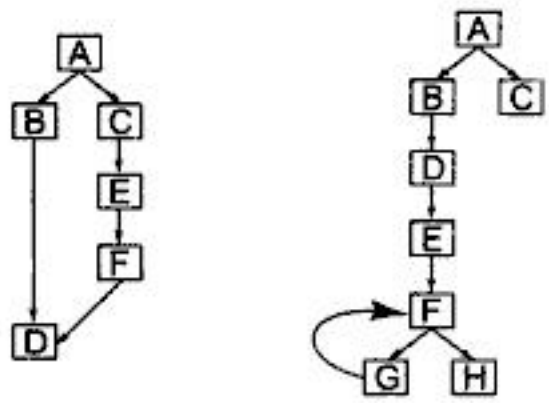
4. Describe the behavior of a revised version of the steepest ascent hill climbing algorithm in which step 2(c) is replaced by "set current state to best successor."
5. Suppose that the first step of the operation of the best-first search algorithm results in the following situation ($a + b$ means that the value of h' at a node is a and the value of g is b):



The second and third steps then result in the following sequence of situations:



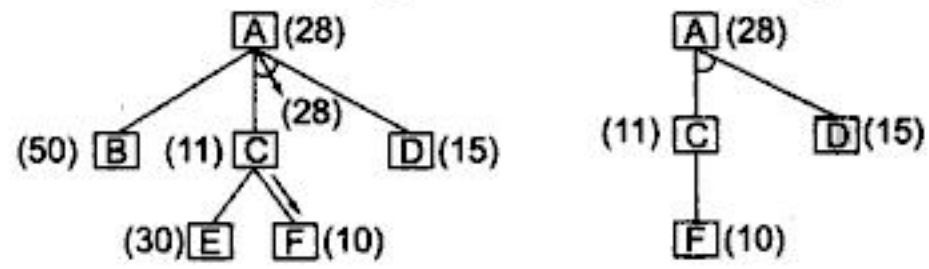
   (a) What node will be expanded at the next step?
   (b) Can we guarantee that the best solution will be found?
6. Why must the A* algorithm work properly on graphs containing cycles? Cycles could be prevented if when a new path is generated to an existing node, that path were simply thrown away if it is no better than the existing recorded one. If g is nonnegative, a cyclic path can never be better than the same path with the cycle omitted. For example, consider the first graph shown below, in which the nodes were generated in alphabetical order. The fact that node D is a successor of node F could simply not be recorded since the path through node F is longer than the one through node B. This same reasoning would also prevent us from

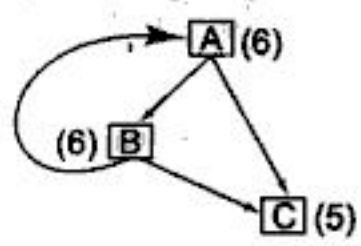


Copyrighted material

recording node E as a successor of node F, if such was the case. But what would happen in the situation shown in the second graph below if the path from node G to node F were not recorded and, at the next step, it were discovered that node G is a successor of node C?

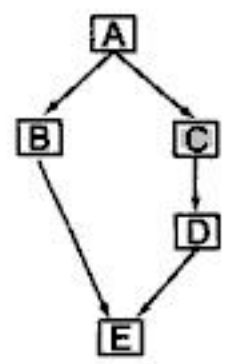
7. Formalize the Graceful Decay of Admissibility Corollary and prove that it is true of the A* algorithm.
8. In step 2(*a*) of the AO* algorithm, a random state at the end of the current best path is chosen for expansion. But there are heuristics that can be used to influence this choice. For example, it may make sense to choose the state whose current cost estimate is the lowest. The argument for this is that for such nodes, only a few steps are required before either a solution is found or a revised cost estimate is produced. With nodes whose current cost estimate is large, on the other hand, many steps may be required before any new information is obtained. How would the algorithm have to be changed to implement this state-selection heuristic?
9. The backward cost propagation step 2(*c*) of the AO* algorithm must be guaranteed to terminate even on graphs containing cycles. How can we guarantee that it does? To help answer this question, consider what happens for the following two graphs, assuming in each case that node F is expanded next and that its only successor is A:





Also consider what happens in the following graph if the cost of node C is changed to 3:

10. The AO* algorithm, in step 2(*c*)i, requires that a node with no descendants in *S* be selected from *S*, if possible. How should the manipulation of *S* be implemented so that such a node can be chosen efficiently? Make sure that your technique works correctly on the following graph, if the cost of node E is changed:



11. Consider again the AO* algorithm. Under what circumstances will it happen that there are nodes in 5 but there are no nodes in *S* that have no descendants also in S?
12. Trace the constraint satisfaction procedure solving the following cryptarithmetic problem:

```
  CROSS
+ ROADS
.........
 DANGER
```

13. The constraint satisfaction procedure we have described performs depth-first search whenever some kind of search is necessary. But depth-first is not the only way to conduct such a search (although it is perhaps the simplest).
    (a) Rewrite the constraint satisfaction procedure to use breadth-first search.
    (b) Rewrite the constraint satisfaction procedure to use best-first search.
14. Show how means-ends analysis could be used to solve the problem of getting from one place to another. Assume that the available operators are walk, drive, take the bus, take a cab, and fly.
15. Imagine a robot trying to move from one place in a city to another. It has complete knowledge of the connecting roads in the city. As it moves the road condition keep changing. If the robot is to reach its destination within a prescribed time, suggest an algorithm for the same. (Hint: Split the road map into a set of connected nodes and imagine that the costs of moving from one node to the other change based on some time-dependent conditions).

Copyrighted material

PART II

KNOWLEDGE REPRESENTATION

Copyrighted material

CHAPTER

4

KNOWLEDGE REPRESENTATION ISSUES

In general we are least aware of what our minds do best.

—**Marvin Minsky**
(1927-), American cognitive scientist

In Chapter 1, we discussed the role that knowledge plays in AI systems. In succeeding chapters up until now, though, we have paid little attention to knowledge and its importance as we instead focused on basic frameworks for building search-based problem-solving programs. These methods are sufficiently general that we have been able to discuss them without reference to how the knowledge they need is to be represented. For example, in discussing the best-first search algorithm, we hid all the references to domain-specific knowledge in the generation of successors and the computation of the h' function. Although these methods are useful and form the skeleton of many of the methods we are about to discuss, their problem-solving power is limited precisely because of their generality. As we look in more detail at ways of representing knowledge, it becomes clear that particular knowledge representation models allow for more specific, more powerful problem-solving mechanisms that operate on them. In this part of the book, we return to the topic of knowledge and examine specific techniques that can be used for representing and manipulating knowledge within programs.

4.1 REPRESENTATIONS AND MAPPINGS

In order to solve the complex problems encountered in artificial intelligence, one needs both a large amount of knowledge and some mechanisms for manipulating that knowledge to create solutions to new problems. A variety of ways of representing knowledge (facts) have been exploited in AI programs. But before we can talk about them individually, we must consider the following point that pertains to all discussions of representation, namely that we are dealing with two different kinds of entities:

- Facts: truths in some relevant world. These are the things we want to represent.
- Representations of facts in some chosen formalism. These are the things we will actually be able to manipulate.

One way to think of structuring these entities is as two levels:

- The *knowledge level*, at which facts (including each agent's behaviors and current goals) are described.

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

that every dog has at least one tail. On the other hand, the former could represent either the fact that every dog has at least one tail or the fact that each dog has several tails. The latter may represent either the fact that every dog has at least one tail or the fact that there is a tail that every dog has. As we will see shortly, when we try to convert English sentences into some other representation, such as logical propositions, we must first decide what facts the sentences represent and then convert those facts into the new representation.

The starred links of Fig. 4.1 are key components of the design of any knowledge-based program. To see why, we need to understand the role that the internal representation of a fact plays in a program. What an AI program does is to manipulate the internal representations of the facts it is given. This manipulation should result in new structures that can also be interpreted as internal representations of facts. More precisely, these structures should be the internal representations of facts that correspond to the answer to the problem described by the starting set of facts.

Sometimes, a good representation makes the operation of a reasoning program not only correct but trivial. A well-known example of this occurs in the context of the mutilated checker board problem, which can be stated as follows:

> **The Mutilated Checker board Problem.** Consider a normal checker board from which two squares, in opposite corners, have been removed. The task is to cover all the remaining squares exactly with dominoes, each of which covers two squares. No overlapping, either of dominoes on top of each other or of dominoes over the boundary of the mutilated board are allowed. Can this task be done?

One way to solve this problem is to try to enumerate, exhaustively, all possible tilings to see if one works. But suppose one wants to be more clever. Figure 4.2 shows three ways in which the mutilated checker board could be represented (to a person). The first representation does not directly suggest the answer to the problem. The second may; the third does, when combined with the single additional fact that each domino must cover exactly one white square and one black square. Even for human problem solvers a representation shift may make an enormous difference in problem-solving effectiveness. Recall that we saw a slightly less dramatic version of this phenomenon with respect to a problem-solving program in Section 1.3.1, where we considered two different ways of representing a tic-tac-toe board, one of which was as a magic square.

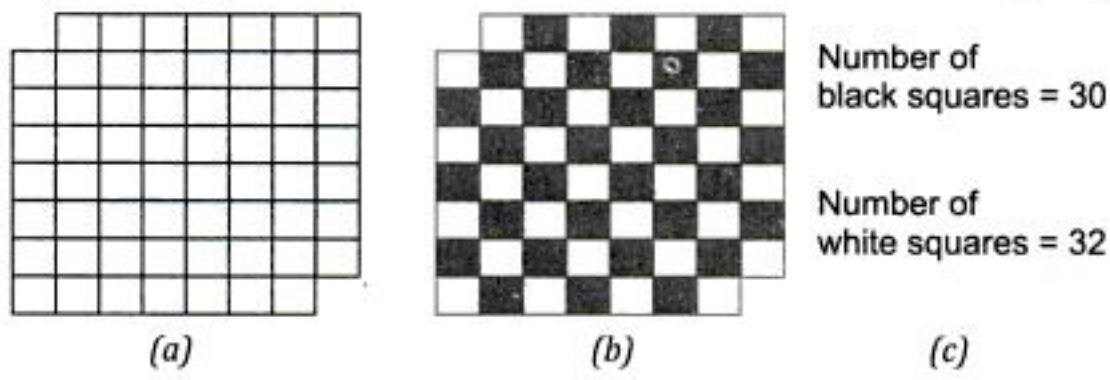


Fig. 4.2 *Three Representations of a Mutilated Checker board*

Figure 4.3 shows an expanded view of the starred part of Fig. 4.1 The dotted line across the top represents the abstract reasoning process that a program is intended to model. The solid line across the bottom represents the concrete reasoning process that a particular program performs. This program successfully models the abstract process to the extent that, when the backward representation mapping is applied to the program's output, the appropriate final facts are actually generated. If either the program's operation or one of the representation mappings is not faithful to the problem that is being modeled, then the final facts will probably not be the desired ones. The key role that is played by the nature of the representation mapping is apparent from this figure. If no good mapping can be defined for a problem, then no matter how good the program to solve the problem is, it will not be able to produce answers that correspond to real answers to the problem.

Copyrighted material

It is interesting to note that Fig. 4.3 looks very much like the sort of figure that might appear in a general programming book as a description of the relationship between an abstract data type (such as a set) and a concrete implementation of that type (e.g., as a linked list of elements). There are some differences, though, between this figure and the formulation usually used in programming texts (such as Aho *et al.* [1983]). For example, in data type design it is expected that the mapping that we are calling the backward representation mapping is a function (i.e., every representation corresponds to only one fact) and that it is onto (i.e., there is at least one representation for every fact). Unfortunately, in many AI domains, it may not be possible to come up with such a representation mapping, and we may have to live with one that gives less ideal results. But the main idea of what we are doing is the same as what programmers always do, namely to find concrete implementations of abstract concepts.

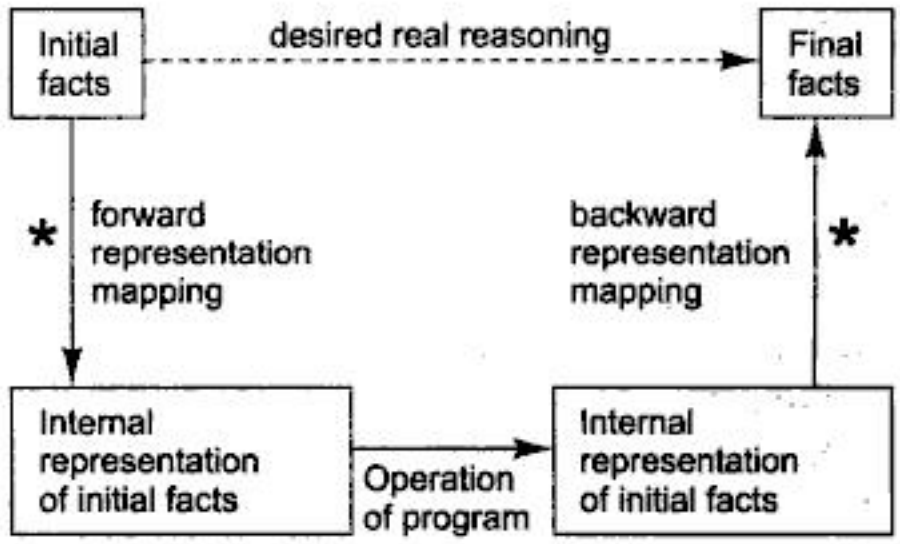


Fig. 4.3 *Representation of Facts*

4.2 APPROACHES TO KNOWLEDGE REPRESENTATION

A good system for the representation of knowledge in a particular domain should possess the following four properties:

Representational Adequacy — the ability to represent all of the kinds of knowledge that are needed in that domain.

- Inferential Adequacy — the ability to manipulate the representational structures in such a way as to derive new structures corresponding to new knowledge inferred from old.
- Inferential Efficiency — the ability to incorporate into the knowledge structure additional information that can be used to focus the attention of the inference mecha- nisms in the most promising directions.
- Acquisitional Efficiency — the ability to acquire new information easily. The simplest case involves direct insertion, by a person, of new knowledge into the database. Ideally, the program itself would be able to control knowledge acquisition.

Unfortunately, no single system that optimizes all of the capabilities for all kinds of knowledge has yet been found. As a result, multiple techniques for knowledge representation exist. Many programs rely on more than one technique. In the chapters that follow, the most important of these techniques are described in detail. But in this section, we provide a simple, example-based introduction to the important ideas.

Simple Relational Knowledge

The simplest way to represent declarative facts is as a set of relations of the same sort used in database systems. Figure 4.4 shows an example of such a relational system.

Player	Height	Weight	Bats-Throws
Hank Aaron	6-0	180	Right-Right
Willie Mays	5-10	170	Right-Right
Babe Ruth	6-2	215	Left-Left
Ted Williams	6-3	205	Left-Right
player_info('hank aaron', '6-0', 180,right-right).			

Fig. 4.4 *Simple Relational Knowledge and a sample fact in Prolog*

Copyrighted material

The reason that this representation is simple is that standing alone it provides very weak inferential capabilities But knowledge represented in this form may serve as the input to more powerful inference engines. For example, given just the facts of Fig. 4.4, it is not possible even to answer the simple question, "Who is the heaviest player?" But if a procedure for finding the heaviest player is provided, then these facts will enable the procedure to compute an answer. If, instead, we are provided with a set of rules for deciding which hitter to put up against a given pitcher (based on right- and left-handedness, say), then this same relation can provide at least some of the information required by those rules.

Providing support for relational knowledge is what database systems are designed to do. Thus we do not need to discuss this kind of knowledge representation structure further here. The practical issues that arise in linking a database system that provides this kind of support to a knowledge representation system that provides some of the other capabilities that we are about to discuss have already been solved in several commercial products.

Inheritable Knowledge

The relational knowledge of Fig. 4.4 corresponds to a set of attributes and associated values that together describe the objects of the knowledge base. Knowledge about objects, their attributes, and their values need not be as simple as that shown in our example. In particular, it is possible to augment the basic representation with inference mechanisms that operate on the structure of the representation. For this to be effective, the structure must be designed to correspond to the inference mechanisms that are desired. One of the most useful forms of inference is *property inheritance,* in which elements of specific classes inherit attributes and values from more general classes in which they are included.

In order to support property inheritance, objects must be organized into classes and classes must be arranged in a generalization hierarchy. Figure 4.5 shows some additional baseball knowledge inserted into a structure that is so arranged. Lines represent attributes. Boxed nodes represent objects and values of attributes of objects. These values can also be viewed as objects with attributes and values, and so on. The arrows on the lines point from an object to its value along the corresponding attribute line, The structure shown in the figure is a *slot-and-filler structure.* It may also be called a *semantic network* or a collection *of frames.* In the latter case, each individual frame represents the collection of attributes and values associated with a particular node. Figure 4.6 shows the node for baseball player displayed as a frame.

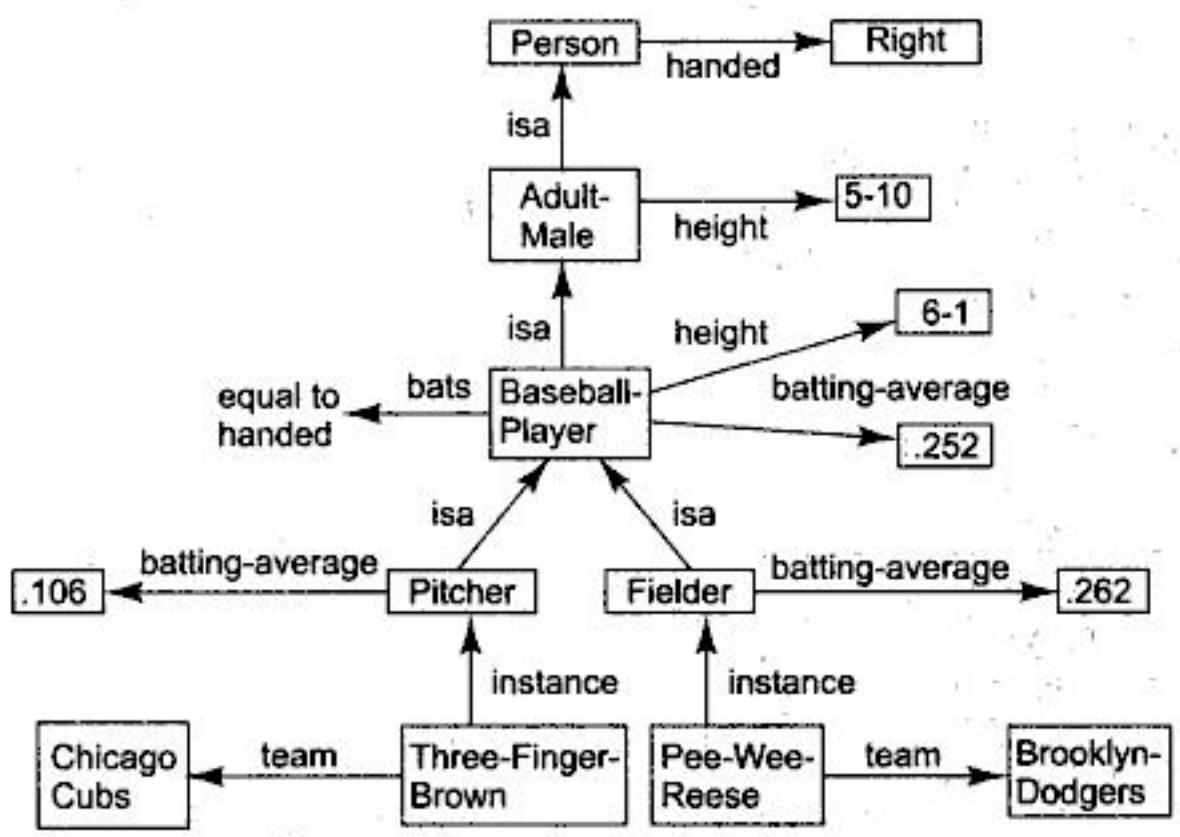


Figure 4.5 *Inheritable Knowledge*

Copyrighted material

Baseball-Player	
isa:	*Adult-Male*
bats:	(EQUAL *handed)*
height:	6-1
batting-average:	.252

Fig. 4.6 *Viewing a Node as a Frame*

Do not be put off by the confusion in terminology here. There is so much flexibility in the way that this (and the other structures described in this section) can be used to solve particular representation problems that it is difficult to reserve precise words for particular representations. Usually the use of the term *frame system* implies somewhat more structure on the attributes and the inference mechanisms that are available to apply to them than does the term *semantic network.*

In Chapter 9 we discuss structures such as these in substantial detail. But to get an idea of how these structures support inference using the knowledge they contain, we discuss them briefly here. All of the objects and most of the attributes shown in this example have been chosen to correspond to the baseball domain, and they have no general significance. The two exceptions to this are the attribute *isa,* which is being used to show class inclusion, and the attribute *instance,* which is being used to show class membership. These two specific (and generally useful) attributes provide the basis for property inheritance as an inference technique. Using this technique, the knowledge base can support retrieval both of facts that have been explicitly stored and of facts that can be derived from those that are explicitly stored.

An idealized form of the property inheritance algorithm can be stated as follows:

Algorithm: Property Inheritance

To retrieve a value *V* for attribute A of an instance object *O:*

1. Find *O* in the knowledge base.
2. If there is a value there for the attribute A, report that value.
3. Otherwise, see if there is a value for the attribute *instance.* If not, then fail.
4. Otherwise, move to the node corresponding to that value and look for a value for the attribute *A.* If one is found, report it.
5. Otherwise, do until there is no value for the *isa* attribute or until an answer is found:
   (a) Get the value of the *isa* attribute and move to that node.
   (b) See if there is a value for the attribute *A.* If there is, report it.

This procedure is simplistic. It does not say what we should do if there is more than one value of the *instance* or *isa* attribute. But it does describe the basic mechanism of inheritance. We can apply this procedure to our example knowledge base to derive answers to the following queries:

- *team(Pee-Wee-Reese) = Brooklyn-Dodgers.* This attribute had a value stored explicitly in the knowledge base.
- *batting-average(Three-Finger Brown)* = .106. Since there is no value for batting average stored explicitly for Three Finger Brown, we follow the *instance* attribute to *Pitcher* and extract the value stored there. Now we observe one of the critical characteristics of property inheritance, namely that it may produce default values that are not guaranteed to be correct but that represent "best guesses" in the face of a lack of more precise information. In fact, in 1906, Brown's batting average was .204.
- *height(Pee-Wee-Reese)* = 6-1. This represents another default inference. Notice here that because we get to it first, the more specific fact about the height of baseball players overrides a more general fact about the height of adult males.

Copyrighted material

- *bats(Three-Finger-Brown) = Right.* To get a value for the attribute *bats* required going up the *isa* hierarchy to the class *Baseball-Player.* But what we found there was not a value but a rule for computing a value. This rule required another value (that for *handed)* as input. So the entire process must be begun again recursively to find a value for *handed.* This time, it is necessary to go all the way up to *Person* to discover that the default value for handedness for people is *Right.* Now the rule for *bats* can be applied, producing the result *Right.* In this case, that turns out to be wrong, since Brown is a switch hitter (i.e., he can hit both left-and right-handed).

Inferential Knowledge

Property inheritance is a powerful form of inference, but it is not the only useful form. Sometimes all the power of traditional logic (and sometimes even more than that) is necessary to describe the inferences that are needed. Figure 4.7 shows two examples of the use of first-order predicate logic to represent additional knowledge about baseball.

$$\forall x : Ball(x) \wedge Fly(x) \wedge Fair(x) \wedge Infield\text{-}Catchable\ (x) \wedge$$
$$Occupied\text{-}Base(First) \wedge Occupied\text{-}Base(Second) \wedge (Outs < 2) \wedge$$
$$\neg[Line\text{-}Drive(x) \vee Attempted\text{-}Bt,(x)]$$
$$\rightarrow Infield\text{-}Fly(x)$$

$$\forall x,y : Batter(x) \wedge batted(x, y) \wedge Infield\text{-}Fly(y) \rightarrow \mathrm{Out}(x)$$

Fig. 4.7 *Inferential Knowledge*

Of course, this knowledge is useless unless there is also an inference procedure *that can exploit* it *(just as* the default knowledge in the previous example would have been useless without our algorithm for moving through the knowledge structure). The required inference procedure now is one that implements the standard logical rules of inference. There are many such procedures, some of which reason forward from given facts to conclusions, others of which reason backward from desired conclusions to given facts. One of the most commonly used of these procedures is *resolution,* which exploits a proof by contradiction strategy. Resolution is described in detail in Chapter 5.

Recall that we hinted at the need for something besides stored primitive values with the *bats* attribute of our previous example. Logic provides a powerful structure in which to describe relationships among values. It is often useful to combine this, or some other powerful description language, with an *isa* hierarchy. In general, in fact, all of the techniques we are describing here should not be regarded as complete and incompatible ways of representing knowledge. Instead, they should be viewed as building blocks of a complete representational system.

Procedural Knowledge

So far, our examples of baseball knowledge have concentrated on relatively static, declarative facts. But another, equally useful, kind of knowledge is operational, or procedural knowledge, that specifies what to do when. Procedural knowledge can be represented in programs in many ways. The most common way is simply as code (in some programming language such as LISP) for doing something. The machine uses the knowledge when it executes the code to perform a task. Unfortunately, this way of representing procedural knowledge gets low scores with respect to the properties of inferential adequacy (because it is very difficult to write a program that can reason about another program's behavior) and acquisitional efficiency (because the process of updating and debugging large pieces of code becomes unwieldy).

As an extreme example, compare the representation of the way to compute the value of *bats* shown in Fig. 4.6 to one in LISP shown in Fig. 4.8. Although the LISP one will work given a particular way of storing attributes and values in a list, it does not lend itself to being reasoned about in the same straightforward way

Copyrighted material

as the representation of Fig. 4.6 does. The LISP representation is slightly more powerful since it makes explicit use of the name of the node whose value for *handed* is to be found. But if this matters, the simpler representation can be augmented to do this as well.

```
Baseball-Player
    isa:            Adult-Male
    bats:           (lambda (x)
                        (prog ()
                            L1
                          (cond ((caddr x) (return (caddr x)))
                                (t (setq x (eval (cadr x)))
                                   (cond (x (go L1))
                                         (t (return nil)))))))
    height:         6-1
    batting-average: .252
```

Fig. 4.8 *Using LISP Code to Define a Value*

Because of this difficulty in reasoning with LISP, attempts have been made to find other ways of representing procedural knowledge so that it can relatively easily be manipulated both by other programs and by people.

The most commonly used technique for representing procedural knowledge in AI programs is the use of production rules. Figure 4.9 shows an example of a production rule that represents a piece of operational knowledge typically possessed by a baseball player.

Production rules, particularly ones that are augmented with information on how they are to be used, are more procedural than are the other representation methods discussed in this chapter. But making a clean distinction between declarative and procedural knowledge is difficult. Although at an intuitive level such a distinction makes some sense, at a formal level it disappears, as discussed in Section 6.1. In fact, as you can see, the structure of the declarative knowledge of Fig. 4.7 is not substantially different from that of the operational knowledge of Fig. 4.9. The important difference is in how the knowledge is used by the procedures that manipulate it.

```
If:     ninth inning, and
        score is close, and
        less than 2 outs, and
        first base is vacant, and
        batter is better hitter than next batter,
Then:   walk the batter.
```

Fig. 4.9 *Procedural Knowledge as Rules*

4.3 ISSUES IN KNOWLEDGE REPRESENTATION

Before embarking on a discussion of specific mechanisms that have been used to represent various kinds of real-world knowledge, we need briefly to discuss several issues that cut across all of them:

- Are any attributes of objects so basic that they occur in almost every problem domain? If there are, we need to make sure that they are handled appropriately in each of the mechanisms we propose. If such attributes exist, what are they?
- Are there any important relationships that exist among attributes of objects?

Copyrighted material

- At what level should knowledge be represented? Is there a good set of *primitives* into which all knowledge can be broken down? Is it helpful to use such primitives?
- How should sets of objects be represented?
- Given a large amount of knowledge stored in a database, how can relevant parts be accessed when they are needed?

We will talk about each of these questions briefly in the next five sections.

4.3.1 Important Attributes

There are two attributes that are of very general significance, and we have already seen their use: *instance* and *isa*. These attributes are important because they support property inheritance. They are called a variety of things in AI systems, but the names do not matter. What does matter is that they represent class membership and class inclusion and that class inclusion is transitive. In slot-and-filler systems, such as those described in Chapters 9 and 10, these attributes are usually represented explicitly in a way much like that shown in Fig. 4.5 and 4.6. In logic-based systems, these relationships may be represented this way or they may be represented implicitly by a set of predicates describing particular classes. See Section 5.2 for some examples of this.

4.3.2 Relationships among Attributes

The attributes that we use to describe objects are themselves entities that we represent. What properties do they have independent of the specific knowledge they encode? There are four such properties that deserve mention here:

- Inverses
- Existence in an *isa* hierarchy
- Techniques for reasoning about values
- Single-valued attributes

Inverses

Entities in the world are related to each other in many different ways. But as soon as we decide to describe those relationships as attributes, we commit to a perspective in which we focus on one object and look for binary relationships between it and others. Attributes are those relationships. So, for example, in Fig. 4.5, we used the attributes *instance, isa* and *team*. Each of these was shown in the figure with a directed arrow, originating at the object that was being described and terminating at the object representing the value of the specified attribute. But we could equally well have focused on the object representing the value. If we do that, then there is still a relationship between the two entities, although it is a different one since the original relationship was not symmetric (although some relationships, such as *sibling*, are). In many cases, it is important to represent this other view of relationships. There are two good ways to do this.

The first is to represent both relationships in a single representation that ignores focus. Logical representations are usually interpreted as doing this. For example, the assertion:

team(Pee-Wee-Reese, Brooklyn-Dodgers)

can equally easily be interpreted as a statement about Pee Wee Reese or about the Brooklyn Dodgers. How it is actually used depends on the other assertions that a system contains.

The second approach is to use attributes that focus on a single entity but to use them in pairs, one the inverse of the other. In this approach, we would represent the team information with two attributes:

- one associated with Pee Wee Reese:
  team = Brooklyn-Dodgers

Copyrighted material

- one associated with Brooklyn Dodgers:
  team-members = Pee-Wee-Reese,...

This is the approach that is taken in semantic net and frame-based systems. When it is used, it is usually accompanied by a knowledge acquisition tool that guarantees the consistency of inverse slots by forcing them to be declared and then checking each time a value is added to one attribute that the corresponding value is added to the inverse.

An Isa Hierarchy of Attributes

Just as there are classes of objects and specialized subsets of those classes, there are attributes and specializations of attributes. Consider, for example, the attribute *height*. It is actually a specialization of the more general attribute *physical-size* which is, in turn, a specialization of *physical-attribute*. These generalization-specialization relationships are important for attributes for the same reason that they are important for other concepts—they support inheritance. In the case of attributes, they support inheriting information about such things as constraints on the values that the attribute can have and mechanisms for computing those values.

Techniques for Reasoning about Values

Sometimes values of attributes are specified explicitly when a knowledge base is created. We saw several examples of that in the baseball example of Fig. 4.5. But often the reasoning system must reason about values it has not been given explicitly. Several kinds of information can play a role in this reasoning, including:

- Information about the type of the value. For example, the value of *height* must be a number measured in a unit of length.
- Constraints on the value, often stated in terms of related entities. For example, the age of a person cannot be greater than the age of either of that person's parents.
- Rules for computing the value when it is needed. We showed an example of such a rule in Fig. 4.5 for the *bats* attribute. These rules are called *backward* rules. Such rules have also been called *if-needed rules*.
- Rules that describe actions that should be taken if a value ever becomes known. These rules are called *forward* rules, or sometimes *if-added rules*.

We discuss forward and backward rules again in Chapter 6, in the context of rulebased knowledge representation.

Single-Valued Attributes

A specific but very useful kind of attribute is one that is guaranteed to take a unique value. For example, a baseball player can, at any one time, have only a single height and be a member of only one team. If there is already a value present for one of these attributes and a different value is asserted, then one of two things has happened. Either a change has occurred in the world or there is now a contradiction in the knowledge base that needs to be resolved. Knowledge-representation systems have taken several different approaches to providing support for single-valued attributes, including:

- Introduce an explicit notation for temporal interval. If two different values are ever asserted for the same temporal interval, signal a contradiction automatically.
- Assume that the only temporal interval that is of interest is now. So if a new value is asserted, replace the old value.
- Provide no explicit support. Logic-based systems are in this category. But in these systems, knowledge-base builders can add axioms that state that if an attribute has one value then it is known not to have all other values.

Copyrighted material

4.3.3 Choosing the Granularity of Representation

Regardless of the particular representation formalism we choose, it is necessary to answer the question "At what level of detail should the world be represented?" Another way this question is often phrased is "What should be our primitives?" Should there be a small number of low-level ones or should there be a larger number covering a range of granularities? A brief example illustrates the problem. Suppose we are interested in the following fact:

John spotted Sue.

We could represent this as[1]

spotted(agent(John),
object(Sue))

Such a representation would make it easy to answer questions such as:

Who spotted Sue?

But now suppose we want to know:

Did John see Sue?

The obvious answer is "yes," but given only the one fact we have, we cannot discover that answer. We could, of course, add other facts, such as

$spotted(x, y) \rightarrow saw(x, y)$

We could then infer the answer to the question.

An alternative solution to this problem is to represent the fact that spotting is really a special type of seeing explicitly in the representation of the fact. We might write something such as

saw(agent(John),
object(Sue),
timespan(briefly))

In this representation, we have broken the idea of *spotting* apart into more primitive concepts *of seeing* and *timespan*. Using this representation, the fact that John saw Sue is immediately accessible. But the fact that he spotted her is more difficult to get to.

The major advantage of converting all statements into a representation in terms of a small set of primitives is that the rules that are used to derive inferences from that knowledge need be written only in terms of the primitives rather than in terms of the many ways in which the knowledge may originally have appeared. Thus what is really being argued for is simply some sort of canonical form. Several AI programs, including those described by Schank and Abelson [1977] and Wilks [1972], are based on knowledge bases described in terms of a small number of low-level primitives.

[1] The arguments *agent* and *object* are usually called *cases*. They represent roles involved in the event. This semantic way of analyzing sentences contrasts with the probably more familiar syntactic approach in which sentences have a surface subject, direct object, indirect object, and so forth. We will discuss case grammar [Fillmore, 1968] and its use in natural language understanding in Section 15.3.2. For the moment, you can safely assume that the cases mean what their names suggest.

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Yet, for many purposes, this detailed primitive representation may be unnecessary. Both in understanding language and in interpreting the world that we see, many things appear that later turn out to be irrelevant. For the sake of efficiency, it may be desirable to store these things at a very high level and then to analyze in detail only those inputs that appear to be important.

A third problem with the use of low-level primitives is that in many domains, it is not at all clear what the primitives should be. And even in domains in which there may be an obvious set of primitives, there may not be enough information present in each use of the high-level constructs to enable them to be converted into their primitive components. When this is true, there is no way to avoid representing facts at a variety of granularities.

The classical example of this sort of situation is provided by kinship terminology [Lindsay, 1963]. There exists at least one obvious set of primitives: mother, father, son, daughter, and possibly brother and sister. But now suppose we are told that Mary is Sue's cousin. An attempt to describe the cousin relationship in terms of the primitives could produce any of the following interpretations:

- Mary = *daughter(brother(mother(Sue)))*
- Mary = *daughter(sister(mother(Sue)))*
- Mary = *daughter(brother(father(Sue)))*
- Mary = *daughter(sister(father(Sue)))*

If we do not already know that Mary is female, then of course there are four more possibilities as well. Since in general we may have no way of choosing among these representations, we have no choice but to represent the fact using the nonprimitive relation *cousin*.

The other way to solve this problem is to change our primitives. We could use the set: *parent, child, sibling, male,* and *female*. Then the fact that Mary is Sue's cousin could be represented as

Mary = *child(sibling(parent(Sue)))*

But now the primitives incorporate some generalizations that may or may not be appropriate. The main point to be learned from this example is that even in very simple domains, the correct set of primitives is not obvious.

In less well-structured domains, even more problems arise. For example, given just the fact

John broke the window.

a program would not be able to decide if John's actions consisted of the primitive sequence:

1. Pick up a hard object.
2. Hurl the object through the window.

or the sequence:

1. Pick up a hard object.
2. Hold onto the object while causing it to crash into the window.

or the single action:

1. Cause hand (or foot) to move fast and crash into the window.

or the single action:

1. Shut the window so hard that the glass breaks.

Copyrighted material

As these examples have shown, the problem of choosing the correct granularity of epresentation for a particular body of knowledge is not easy. Clearly, the lower the level we choose, the less inference required to reason with it in some cases, but the more conference required to create the representation from English and the more room it takes to store, since many inferences will be represented many times. The answer for any particular task domain must come to a large extent from the domain itself—to what use is the knowledge to be put?

One way of looking at the question of whether there exists a good set of low-level primitives is that it is a question of the existence of a unique representation. Does there exist a single, canonical way in which large bodies of knowledge can be represented independently of how they were originally stated? Another, closely related, uniqueness question asks whether individual objects can be represented uniquely and independently of how they are described. This issue is raised in the following quotation from Quine [1961] and discussed in Woods 11975]:

> The phrase *Evening Star* names a certain large physical object of spherical form, which is hurtling through space some scores of millions of miles from here. The phrase *Morning Star* names the same thing, as was probably first established by some observant Babylonian. But the two phrases cannot be regarded as having the same meaning; otherwise that Babylonian could have dispensed with his observations and contented himself with reflecting on the meaning of his words. The meanings, then, being different from one another, must be other than the named object, which is one and the same in both cases.

In order for a program to be able to reason as did the Babylonian, it must be able to handle several distinct representations that turn out to stand for the same object.

We discuss the question of the correct granularity of representation, as well as issues involving redundant storage of information, throughout the next several chapters, particularly in the section on conceptual dependency, since that theory explicitly proposes that a small set of low-level primitives should be used for representing actions.

4.3.4 Representing Sets of Objects

It is important to be able to represent sets of objects for several reasons. One is that there are some properties that are true of sets that are not true of the individual members of a set. As examples, consider the assertions that are being made in the sentences "There are more sheep than people in Australia" and "English speakers can be found all over the world." The only way to represent the facts described in these sentences is to attach assertions to the sets representing people, sheep, and English speakers, since, for example, no single English speaker can be found all over the world. The other reason that it is important to be able to represent sets of objects is that if a property is true of all (or even most) elements of a set, then it is more efficient to associate it once with the set rather than to associate it explicitly with every element of the set. We have already looked at ways of doing that, both in logical representations through the use of the universal quantifier and in slot-and-filler structures, where we used nodes to represent sets and inheritance to propagate set-level assertions down to individuals. As we consider ways to represent sets, we will want to consider both of these uses of set-level representations. We will also need to remember that the two uses must be kept distinct. Thus if we assert something like *large(Elephant)*, it must be clear whether we are asserting some property of the set itself (i.e., that the set of elephants is large) or some property that holds for individual elements of the set (i.e., that anything that is an elephant is large).

There are three obvious ways in which sets may be represented. The simplest is just by a name. This is essentially what we did in Section 4.2 when we used the node named *Baseball-Player* in our semantic net and when we used predicates such as *Ball* and *Batter* in our logical representation. This simple representation does make it possible to associate predicates with sets. But it does not, by itself, provide any information about the set it represents. It does not, for example, tell how to determine whether a particular object is a member of the set or not.

Copyrighted material

There are two ways to state a definition of a set and its elements. The first is to list the members. Such a specification is called an *extensional* definition. The second is to provide a rule that, when a particular object is evaluated, returns true or false depending on whether the object is in the set or not. Such a rule is called an *intensional* definition. For example, an extensional description of the set of our sun's planets on which people live is *{Earth}*. An intensional description is

$$\{x : sun\text{-}planet(x) \wedge human\text{-}inhabited(x)\}$$

For simple sets, it may not matter, except possibly with respect to efficiency concerns, which representation is used. But the two kinds of representations can function differently in some cases.

One way in which extensional and intensional representations differ is that they do not necessarily correspond one-to-one with each other. For example, the extensionally defined set *{Earth}* has many intensional definitions in addition to the one we just gave. Others include:

$$\{x : sun\text{-}planet\{x) \ ?\ nth\text{-}farthest\text{-}fmm\text{-}sun(x, 3)\}$$
$$\{x : sun\text{-}planet(x) \ ?\ nth\text{-}biggest(x, 5)\}$$

Thus, while it is trivial to determine whether two sets are identical if extensional descriptions are used, it may be very difficult to do so using intensional descriptions.

Intensional representations have two important properties that extensional ones lack, however. The first is that they can be used to describe infinite sets and sets not all of whose elements are explicitly known. Thus we can describe intensionally such sets as prime numbers (of which there are infinitely many) or kings of England (even though we do not know who all of them are or even how many of them there have been). The second thing we can do with intensional descriptions is to allow them to depend on parameters that can change, such as time or spatial location. If we do that, then the actual set that is represented by the description will change as a function of the value of those parameters. To see the effect of this, consider the sentence, "The president of the United States used to be a Democrat," uttered when the current president is a Republican. This sentence can mean two things. The first is that the specific person who is now president was once a Democrat. This meaning can be captured straightforwardly with an extensional representation of "the president of the United States." We just specify the individual. But there is a second meaning, namely that there was once someone who was the president and who was a Democrat. To represent the meaning of "the president of the United States" given this interpretation requires an intensional description that depends on time. Thus we might write *president(t)*, where *president* is some function that maps instances of time onto instances of people, namely U.S. presidents.

4.3.5 Finding the Right Structures as Needed

Recall that in Chapter 2, we briefly touched on the problem of matching rules against state descriptions during the problem-solving process. This same issue now rears its head with respect to locating appropriate knowledge structures that have been stored in memory.

For example, suppose we have a script (a description of a class of events in terms of contexts, participants, and subevents) that describes the typical sequence of events in a restaurant.[3] This script would enable us to take a text such as

> John went to Steak and Ale last night. He ordered a large rare steak, paid his bill, and left.

and answer "yes" to the question

[3] We discuss such a script in detail in Chapter 10.

Copyrighted material

Did John eat dinner last night?

Notice that nowhere in the story was John's eating anything mentioned explicitly. But the fact that when one goes to a restaurant one eats will be contained in the restaurant script. If we know in advance to use the restaurant script, then we can answer the question easily. But in order to be able to reason about a variety of things, a system must have many scripts for everything from going to work to sailing around the world. How will it select the appropriate one each time? For example, nowhere in our story was the word "restaurant" mentioned.

In fact, in order to have access to the right structure for describing a particular situation, it is necessary to solve all of the following problems.[4]

- How to perform an initial selection of the most appropriate structure.
- How to fill in appropriate details from the current situation.
- How to find a better structure if the one chosen initially turns out not to be appropriate.
- What to do if none of the available structures is appropriate.
- When to create and remember a new structure.

There is no good, general purpose method for solving all these problems. Some knowledge-representation techniques solve some of them. In this section we survey some solutions to two of these problems: how to select an initial structure to consider and how to find a better structure if that one turns out not to be a good match.

Selecting an Initial Structure

Selecting candidate knowledge structures to match a particular problem-solving situation is a hard problem; there are several ways in which it can be done. Three important approaches are the following:

- Index the structures directly by the significant English words that can be used to describe them. For example, let each verb have associated with it a structure that describes its meaning. This is the approach taken in conceptual dependency theory, discussed in Chapter 10. Even for selecting simple structures, such as those representing the meanings of individual words, though, this approach may not be adequate, since many words may have several distinct meanings. For example, the word "fly" has a different meaning in each of the following sentences:
  - John flew to New York. (He rode in a plane from one place to another.)
  - John flew a kite. (He held a kite that was up in the air.)
  - John flew down the street. (He moved very rapidly.)
  - John flew into a rage. (An idiom)

  Another problem with this approach is that it is only useful when there is an English description of the problem to be solved.
- Consider each major concept as a pointer to all of the structures (such as scripts) in which it might be involved. This may produce several sets of prospective structures. For example, the concept *Steak* might point to two scripts, one for restaurant and one for supermarket. The concept *Bill* might point to a restaurant and a shopping script. Take the intersection of those sets to get the structure(s), preferably precisely one, that involves all the content words. Given the pointers just described and the story about John's trip to Steak and Ale, the restaurant script would be evoked. One important problem with this method is that if the problem description contains any even slightly extraneous concepts, then the intersection of their associated structures will be empty. This might occur if we had said, for example, "John rode his bicycle to Steak and Ale last night." Another problem is that it may require a great deal of computation to compute all of the possibility sets and then to intersect them. However, if computing such sets and intersecting them could be done in parallel, then the time required to produce an answer

[4] This list is taken from Minsky [1975].

Copyrighted material

would be reasonable even if the total number of computations is large. For an exploration of this parallel approach to clue intersection, see Fahlman [1979].

- Locate one major clue in the problem description and use it to select an initial structure. As other clues appear, use them to refine the initial selection or to make a completely new one if necessary. For a discussion of this approach, see Charniak [1978]. The major problem with this method is that in some situations there is not an easily identifiable major clue. A second problem is that it is necessary to anticipate which clues are going to be important and which are not. But the relative importance of clues can change dramatically from one situation to another. For example, in many contexts, the color of the objects involved is not important. But if we are told "The light turned red," then the color of the light is the most important feature to consider.

None of these proposals seems to be the complete answer to the problem. It often turns out, unfortunately, that the more complex the knowledge structures are, the harder it is to tell when a particular one is appropriate.

Revising the Choice When Necessary

Once we find a candidate knowledge structure, we must attempt to do a detailed match of it to the problem at hand. Depending on the representation we are using, the details of the matching process will vary. It may require variables to be bound to objects. It may require attributes to have their values compared. In any case, if values that satisfy the required restrictions as imposed by the knowledge structure can be found, they are put into the appropriate places in the structure. If no appropriate values can be found, then a new structure must be selected. The way in which the attempt to instantiate this first structure failed may provide useful cues as to which one to try next. If, on the other hand, appropriate values can be found, then the current structure can be taken to be appropriate for describing the current situation. But, of course, that situation may change. Then information about what happened (for example, we walked around the room we were looking at) may be useful in selecting a new structure to describe the revised situation.

As was suggested above, the process of instantiating a structure in a particular situation often does not proceed smoothly. When the process runs into a snag, though, it is often not necessary to abandon the effort and start over. Rather, there are a variety of things that can be done:

- Select the fragments of the current structure that do correspond to the situation and match them against candidate alternatives. Choose the best match. If the current structure was at all close to being appropriate, much of the work that has been done to build substructures to fit into it will be preserved.
- Make an excuse for the current structure's failure and continue to use it. For example, a proposed chair with only three legs might simply be broken. Or there might be another object in front of it which occludes one leg. Part of the structure should contain information about the features for which it is acceptable to make excuses. Also, there are general heuristics, such as the fact that a structure is more likely to be appropriate if a desired feature is missing (perhaps because it is hidden from view) than if an inappropriate feature is present. For example, a person with one leg is more plausible than a person with a tail.
- Refer to specific stored links between structures to suggest new directions in which to explore. An example of this sort of linking among a set of frames is shown in the similarity network shown in Fig. 4.11.[5]
- If the knowledge structures are stored in an *isa* hierarchy, traverse upward in it until a structure is found that is sufficiently general that it does not conflict with the evidence. Either use this structure if it is specific enough to provide the required knowledge or consider creating a new structure just below the matching one.

[5] This example is taken from Minsky [1975].

Copyrighted material

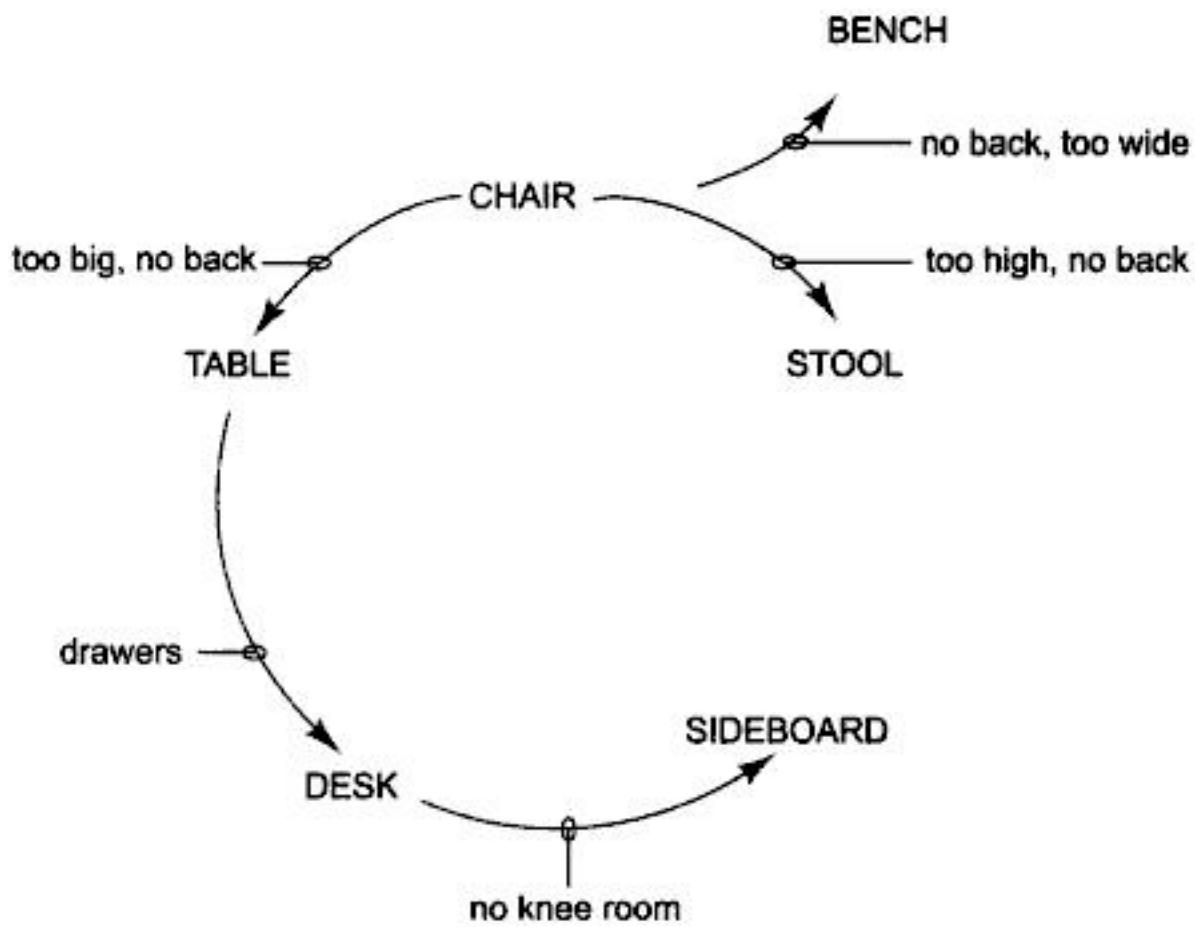


Fig. 4.11 *A Similarity Net*

4.4 THE FRAME PROBLEM

So far in this chapter, we have seen several methods for representing knowledge that would allow us to form complex state descriptions for a search program. Another issue concerns how to represent efficiently *sequences* of problem states that arise from a search process. For complex ill-structured problems, this can be a serious matter.

Consider the world of a household robot. There are many objects and relationships in the world, and a state description must somehow include facts like *on(Plant12, Table34)*, *under(Table34, Window13)*, and *in(Table34, Room 15)*. One strategy is to store each state description as a list of such facts. But what happens during the problem-solving process if each of those descriptions is very long? Most of the facts will not change from one state to another, yet each fact will be represented once at every node, and we will quickly run out of memory. Furthermore, we will spend the majority of our time creating these nodes and copying these facts—most of which do not change often—from one node to another. For example, in the robot world, we could spend a lot of time recording *above(Ceiling, Floor)* at every node. All of this is, of course, in addition to the real problem of figuring out which facts *should* be different at each node.

This whole problem of representing the facts that change as well as those that do not is known as *the frame problem* [McCarthy and Hayes, 1969]. In some domains, the only hard part is representing all the facts. In others, though, figuring out which ones change is nontrivial. For example, in the robot world, there might be a table with a plant on it under the window. Suppose we move the table to the center of the room. We must also infer that the plant is now in the center of the room too but that the window is not.

To support this kind of reasoning, some systems make use of an explicit set of axioms called *frame axioms*, which describe all the things that do not change when a particular operator is applied in state n to produce state $n + 1$. (The things that do change must be mentioned as part of the operator itself.) Thus, in the robot domain, we might write axioms such as

$$color(x,y, s_1) \wedge move(x, s_1,S_2) \rightarrow color(x,y, S_2)$$

Copyrighted material

which can be read as, "If x has color y in state s_1 and the operation of moving x is applied in state s_1 to produce state s_2, then the color of x in s_2 is still y." Unfortunately, in any complex domain, a huge number of these axioms becomes necessary. An alternative approach is to make the assumption that the only things that change are the things that must. By "must" here we mean that the change is either required explicitly by the axioms that describe the operator or that it follows logically from some change that is asserted explicitly. This idea of *circumscribing* the set of unusual things is a very powerful one; it can be used as a partial solution to the frame problem and as a way of reasoning with incomplete knowledge. We return to it in Chapter 7.

But now let us return briefly to the problem of representing a changing problem state. We could do it by simply starting with a description of the initial state and then making changes to that description as indicated by the rules we apply. This solves the problem of the wasted space and time involved in copying the information for each node. And it works fine until the first time the search has to backtrack. Then, unless all the changes that were made can simply be ignored (as they could be if, for example, they were simply additions of new theorems), we are faced with the problem of backing up to some earlier node. But how do we know what changes in the problem state description need to be undone? For example, what do we have to change to undo the effect of moving the table to the center of the room? There are two ways this problem can be solved:

- Do not modify the initial state description at all. At each node, store an indication of the specific changes that should be made at this node. Whenever it is necessary to refer to the description of the current problem state, look at the initial state description and also look back through all the nodes on the path from the start state to the current state. This is what we did in our solution to the cryptarithmetic problem in Section 3.5. This approach makes backtracking very easy, but it makes referring to the state description fairly complex.
- Modify the initial state description as appropriate, but also record at each node an indication of what to do to undo the move should it ever be necessary to backtrack through the node. Then, whenever it is necessary to backtrack, check each node along the way and perform the indicated operations on the state description.

Sometimes, even these solutions are not enough. We might want to remember, for example, in the robot world, that before the table was moved, it was under the window and after being moved, it was in the center of the room. This can be handled by adding to the representation of each fact a specific indication of the time at which that fact was true. This indication is called a *state variable*. But to apply the same technique to a real-world problem, we need, for example, separate facts to indicate all the times at which the Statue of Liberty is in New York.

There is no simple answer either to the question of knowledge representation or to the frame problem. Each of them is discussed in greater depth later in the context of specific problems. But it is important to keep these questions in mind when considering search strategies, since the representation of knowledge and the search process depend heavily on each other.

SUMMARY

The purpose of this chapter has been to outline the need for knowledge in reasoning programs and to survey issues that must be addressed in the design of a good knowledge representation structure. Of course, we have not covered everything. In the chapters that follow, we describe some specific representations and look at their relative strengths and weaknesses.

The following collections all contain further discussions of the fundamental issues in knowledge representation, along with specific techniques to address these issues: Bobrow [1975], Winograd [1978], Brachman and Levesque [1985], and Halpern [1986]. For especially clear discussions of specific issues on the topic of knowledge representation and use see Woods [1975] and Brachman [1985].

Copyrighted material

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

AUTHOR INDEX

A

Abelson 89, 208, 213
ACM Computing Surveys 331
Ades and Steedman 292
Agre 269
Aho 82
AIS, ICARIS 2007 331
Alain Colmerauer 496
Alan Turing 20, 231
Allen 300, 330
Allen and Perrault 319
Amsterdam 373
Anantharaman 227, 242, 245
Anderson 20, 30
Angluin 373
Ashby 19, 376
Atkin 242

B

Baker 437
Ballard 377, 397, 435, 436
Balzer 342
Baroness Orczy 322
Barr 21
Baudet 240
Beal 241
Benson 232
Berliner 227, 232, 242, 245
Bledsoe 125
Block 376
Bobrow 22, 97, 205
Boddy 433
Bond and Gasser 346
Bowden 231
Boyer 126
Brachman 97, 205, 223, 224
Brady 434, 443
Bratko 131
Brazil 331
Bridgeland and Huhns 346
Brooks 269, 442
Brost 440
Brown 369, 410, 435, 436
Brownston 30, 137, 145
Bruce 305
Buchanan 174, 177, 424, 430
Burton 302

C

Carbonell 11, 212, 372, 432
Chandrasekaran 43
Chandy 130
Chang 98, 112
Chapman 262, 267, 269
Charniak 21, 95
Chart Parsers Winograd 295
Chavez 181
Cheeseman 174
Cheng and Juang 336
Chomsky 292
Church and Patil 331

Copyrighted material

Clark 133, 186
Clark and Gregory 335
Claude Shannon 231
Clocksin 131
Colby 19, 22
Collins 22
Corkill 342
Cottrell 397, 398
Craig 434
Criscuolo 153
Cullingford 215, 316, 330

D

Damerau 326
Davis 109, 143, 425
de Champeaux 136
de Jong 374
de Kleer 166, 170, 410, 411
Dean 433
Dempster 181
Dijkstra 130
Dowty 330
Doyle 151, 162
Duda 173, 423
Dwork 336
Dyer 319, 420

E

Ebeling 227, 245
Eckroth 21
Eds. Leandro N. de Castro 331
Elman 400
Englemore and Morgan 344
Erman 342
Ernst 72
Eshelman 427
Etherington 151

F

Fahlman 95, 336, 377, 397, 400
Feigenbaum 19, 22
Feldman 22, 377
Fernando J. Von Zuben 331
Fikes 252, 268, 270, 352
Fillmore 305
Findler 22
Forbus 410
Forgy 135, 140
Fox 346
Frank Rosenblatt 19

G

Gabriel and McCarthy 335
Gale 186
Gardner 20
Gasser and Huhns 346
Gasser 344
Gelernter 98
Gelperin 62
Genesereth 21, 151, 340
Georgeff 339, 443
Ginsberg 150
Goldberg 374
Goldstein 205
Gordon and Lakoff 320
Gorman 398
GPSG Gazdar 292
Greenblatt 244
Greiner 205, 218
Grice 320
Grosz 330
Guarded Horn Clauses [Ueda, 1985] 335
Guha 205, 216, 408

H

Halpern 22, 97, 186
Halstead 335
Hammond 270, 420
Hanks 170
Hanna 369
Harmon 430, 433
Hart 423
Haussler 373
Hayes 224, 415, 417
Hayes-Roth 22, 145, 342
Heckerman 177
Helder Knidel 331
Hendrix 191, 302
Hendrix and Lewis 302
Hennessey 23
Hewett 342
Hindin 430
Hintikka 317
Hinton 377, 398, 401, 404
Hoare 130
Hobbs 414
Holland 374

Copyrighted material

Hon 437, 438
Hood 432
Hopfield 377, 398
Horn 23, 436
Hsu 336
Huhns 346

I

Ishiguro 489

J

Jagannathan 344
Johnson-Laird 226
Jordan 399, 400
Joshi 330

K

Kaczmarek 205
Kaebling 269
Kahneman 174
Kanal 186
Kandel and Schwartz, 1985 417
Kasif 336
Kautz 413, 414
Kearns 373
King 330, 430
Kirkpatrick 55
Knight 116, 300
Knuth 240
Kolodner 419, 420
Konolige 150
Korf 243, 244, 340, 353, 354, 433
Kornfeld and Hewitt 346
Krogh 439
Kulikowski 430
Kumar 336

L

Laffey 430
Laird 30, 143, 334, 443
Langley 370, 432
Lansky 443
Lauritzen 180
LeCun 389
Lee 98, 112, 245, 437, 438
Lehnert 212
Lehrberger and Bourbeau 331
Lemmer 186
Lenat 34, 205, 216, 218, 368, 369, 408
Lesser 342, 437
Lesser and Corkill 340
Levesque 97, 223
Levy 246
LFG Bresnan 292
Lifschitz 157
Lindsay 51, 91
Littlestone 373
Lovelace 6, 347
Lowerre 437
Lozano 441
Lozano-Perez 439
Lytinen 419

M

Mahajan 245
Malone 346
Marcus 294, 426, 428
Marr 436
Martelli 62, 67
Mason 440
Mason and Johnson 346
Mayer 433
McCarthy 157
McClelland 390
McCulloch 19, 376
McDermott 21, 43, 151, 170, 426, 428
McDonald and Bole 331
McKeown and Swartout 331
Mellish 131
Minsky 19, 94, 193, 233, 346, 376, 384
Minton 144, 267, 354, 355
Miranker 140
Misra 130
Mitchell 205, 269, 358, 363, 365, 429, 442, 443
Montanari 67
Moore 126, 152, 240
MYCIN 422

N

N.K. Jerne 485
Nagao 331
Nau. 242
Newell 6, 48, 72, 80, 142, 244, 263
Niblack 434
Nilsson 21, 22, 67, 151, 186, 224, 252, 262, 268, 270, 352

Copyrighted material

Nirenburg 331
Norman 20
Nyberg 205

O

Oehrle 292
Oflazer 140
Owens 212

P

Pamela McCordiick 22
Papert 19, 384
Parker 389
Paulos 7
Pawley 398
Pearl 179, 180, 181, 186, 240, 242
Pereira and Warren 295
Perez 441
Pitts 19, 376
Pohl 136
Polya 35, 136, 375
Pomerleau 398
Pople 179
Prerau 430
Putnam 109

Q

Quillian 189, 417
Quine 92, 147

R

Rauch 430
Reiter 151, 153, 155, 170
Rich 153
Ritchie 369
Roberts 205
Robinson 113
Rosenberg 396
Rosenblatt 376, 379, 383
Rosenbloom 245, 355
Rosenschein and Breese 340
Rumelhart 389, 390
Russell 98

S

Sabot 335
Sacerdoti 262, 268
Samuel 231, 245, 348, 351
Santos 331
Schank 22, 89, 207, 209, 210, 212, 213, 418, 420
Schmolze 205
Searle 23, 319
Sejnowski 245, 396, 398
Selfridge 376
Shafer 181, 186, 439, 442
Shannon 244
Shapiro 21, 335
Shieber 300
Shoham 150
Shortliffe 136, 174, 177, 424
Shrobe 22
Shultz 179
Sidner 319
Siklossy 22
Simmons 442
Simon 6, 22, 34, 49, 76, 244, 248, 263
Sint 136
Slate 242
Slocum 331
Smolensky 7
Sowa 205
Spiegelhalter 180
Springer 331
Stallman 160
Steele 23, 170, 423
Stefik 262, 270
Stickel 113
Sussman 160, 260, 262, 264
Susumu Tonegawa 483

T

Tadepalli 367
Tambe 355
Tanimoto 21
Tank 398
Tate 262
Tesauro 245, 398
The Society of Mind 346
Thompson 310, 336
Thorpe 439
Toshinori Munakata 22
Touretzky 23, 153, 396, 404
Turing 20
Tversky 174

U

Ullian 147

Copyrighted material

V

Valiant 372, 373
van Melle 179
Vitter and Simons 336
Vladimir Levenshtein 327

W

Wagner 327
Waibel 437
Waltz algorithm 278, 282
Waterman 22, 145, 430
Webber 22
Weiss 179, 430
Weizenbaum 140
Weld 411
Werbos 389
Whitehead 98
Whittaker 439, 442
Wilcox 245
Wilensky 23, 319
Wilks 89, 212, 302
Williams 170
Wilson 398
Winograd 97, 205, 330
Winograd and Flores 332
Winston 23, 356
Woods 92, 97, 295
Wos 98

Z

Zadeh 185

Copyrighted material

SUBJECT INDEX

A

a cut threshold 454
A Pick-and-Place Task 440
A T-Cell Receptor 484
A Water Jug Problem 27
A* algorithm 433
Abduction 153
abstraction spaces 268
Abstrips 268
actions 272, 438, 491
activation record 342
acute angles 280
ad hoc 294
Adaptive Immune System 481
add one smoothing 325
affected-entity 310
Affinity 482, 483, 485, 487, 491, 492, 495
  Modelling 486
Agendas 62
agent 220, 312, 517
agent_close 523
agent_data 522
Agent
  Definition 517
  Characteristics 517
  Chimera 517, 518, 526
  Handler 518
  Creation 518
Link 518
agent_dict 520
AI Skeptics 529
Algorithm: Real-Time-A* 433
All Paths 293
Alleles 457
alpha-beta pruning 336
alpha cut threshold
Alternatives to Minimax 242
AM: Theory-Driven Discovery 368
ambiguous 275
An Envisionment 411
An Isa Hierarchy of Attributes 88
Analogy 371
anaphora 314
anaphoric references 314
AND-OR Graphs 64
AND-parallelism 335
annealing 55
annealing schedule 55
ant 520
Ant Algorithms 476
Ant Colony Optimization (ACO) algorithm 476
Antibodies 481, 482, 484, 486, 491, 492, 496
antibody molecule 483
anticipatory 427
Antigen Presenting Cells (APCs) 481
Antigen
Antigens 480
antonym 520
AO* Algorith 67
apian 41
apoptosis 482, 485, 494
appropriate 272
Appropriateness Conditions 320

Copyrighted material

Arch 357
Arithmetic Operators 500
Arity 499
ARROW 280
Artificial intelligence 3
ARTIS (ARTificial Immune System) 479, 486, 487
assert 513-515, 521
assumption-based truth maintenance system (ATMS) 166
ATN Grammar 296
ATN Network 296
Atoms 499
Attributes 87
Augmented transition networks 295
Autoepistemic Logic 152
Autoimmune 494
autonomous reinforcement 485
autonomous reinforcement learning 485

B

B-cell Receptor (BCR) 483
B-cells 482
  generation 482
  receptor 483
Backgammon 245
Backpropagation Networks 385
Backtracking 503, 505-507
Backward Propagation 68
backward reasoning 45
Backward-Chaining Rule Systems 137
BACON: Data-Driven Discovery 370
Bayes' Theorem 172
Bayesian Networks 179
belief spaces 148
beliefs 220
Best Path with Backtracking 294
Best Path with Patchup 294
Best-First Search 57
bias 359
bidirectional search 136
bigram 323
Bigram counts 324
billion-component 377
BKG 245
Blackboard systems 341
Blocks World 250
body 321
Boltzmann Machines 390
book and books (singular and plural) 322
Book Moves 242
BookkeepingSlots 219
Bottom-Up Parsing 293
bottom-up parsing with top-down filtering 293
Bounded rationality 339
branch- and-bound 33
breadth-first search 31
Brittleness 430
BUILD 299
built-in predicates (BIPS) 515

C

c-space 439
candidate elimination algorithm 360
case analysis 350
Case Grammars 305
Case-Based Reasoning 271, 419
case-based reasoning (CBR) systems 419
CAT DET 299
CAT N 299
causal chain 213
CBR 419, 420, 421
CD Structure 308
Certainty Factors 174
chain smoker, a hard nut, extremely beautiful 321
Character 499
Checkers 245
CHEF 420
Chemotaxis 494
chiasma 460
Chimera 517
Chimera Agent 518
Chimera Agent Engine 516
Chinese Room 23
Christmas gifts 321
Chromosome 457, 458
Chromosome representation 464
chronological backtracking 160
Chunking 354
Cilia 494
circumscription 156, 157
Clause Form 109
Clauses 499
Clonal Selection 484, 485, 487
Concentration
  dynamic 492
  decay in 492
closed world assumption 134, 155
Clustering 370
CODGER 442
cognitive science 20, 22

Copyrighted material

Collection 219
collocations 321, 322
combinatorial explosion 33
Combinatorial Problems 398
Combining Forward and Backward Reasoning 137
Combining Mapping Knowledge 311
Common Sense 408
Common sense Ontologies 411
commonsense reasoning 4
commutative production system 43
Complement 447
completeness 125
compliant motion 441
Components of an AI Program 529
Composite 219
compositional semantics 310
Compound goals 502, 503
Computable Functions 105
Concave Edge 278
concept learning 356
concept space 359
Conceptual Dependency 207
conceptual dependency model 273
Conceptual parsing 307
Conceptual Tenses 211
Conclusion 529
concordance 321
Concurrent PROLOG 335
conditional logics 317
configuration space 439
Conflict Resolution 141
connectionist 19
Connectionist AI 403
Connectionist Models 376
Connectionist Speech 396
Connectionist Vision 397
Constants 499
Constituent1 299
Constituent2 299
constrains 428
constraint posting 262
Constraint Satisfaction 68, 278, 336
consult 516
Content-Addressable Memory 378
Contract Nets 339
contributes-to 428
contributes-to, constrains 428
Control and Meta-Knowledge 218
Control Knowledge 142
Control Strategies 31
Control Structures 500
conversational postulates 319, 320
Convex Edge 278
Corpora 321
corpus 321
  Counting elements of 322
  Parallel 322
covering 427
credit assignment problem 233, 351
Crisp Sets 445
Criteria for Success 20
Crossover 458, 460, 461, 466, 468-471, 473-475
Crossover is 460
Crowding 471
Crowding Factor 471
Cryptarithmetic 48
Cryptarithmetic Problem 71
Cut 505
Cuts 505
CYC 216
CYCL 217
CYRUS 419
cytokines 481, 494

D

Darwinian Evolution 458
Databases
  dynamic 512-514
data retrieval system 274
Dead Ends 254
decision phase 334
Decision Trees 364
declarative representation 129
Declobbering 265
decomposable 248
Decomposable Problem 37
Default Logic 153
Default Reasoning 151
defeasible 149
defined goals 517
DefiningSlots 219
Defuzzification 450, 451, 452
Defuzzifier 450
Deletion 326
demon 342
Dempster-Shafer Theory 181
DENDRAL 21, 51, 429
Dendritic cells 494
Dependency-Directed Backtracking 160
depth-first iterative deepening (DFID) 243
depth-first search 32
Derivational Analogy 372

Copyrighted material

DESIGN ADVISOR 423, 424
DET, HEAD 299
Dialog Editor option 524
Dialogue
  Creation 524
  Editor 524
Dialogues 524
difference table 72
Directed acyclic graphs (DAGs) 179, 298
Discourse Integration 287, 290
Discovery 367
Distributed, Asynchronous Control 378
distributed planning 339
Distributed Problems 337
Distributed Reasoning Algorithms 345
distributed reasoning systems 336
Distributed Representations 378, 400
distributed systems 333
Domain Knowledge 422
Domain Theory 365
Down-counting 507
DRAGON 437
Dynamic Databases 512

E

Ease-of-Adaptation Preference 420
Efficiency 337
elaboration phase 334
element-of 316
ELIZA 13, 140
EMYCIN 424
Encyclopedia of Artificial Intelligence 21
EnglishKnow 14
envisionment 411
EPAM 19
epistemological level 218
Epithelium 495
Epitope 484-486, 495
epitopes 483
Equations 410
essential knowledge 529
EURISKO 369
Evening Star 92
Event 312
Event and Process 219
expectation-driven 307
expert system 429
expert system shells 30, 424
Expert Systems 5, 422, 430
Expert Tasks 5
Explanation 425
explanation-based learning 354, 364
Expressions 410
Extrinsic 219

F

faces 415
Factorial of a number 507
facts 497, 499
fail 504
Fast Computer Architectures 337
Fault Tolerance 378
feedforward networks 385
Fitness criteria 465
fitness function 458, 459, 462, 465, 472, 473, 475
FORK 280, 282
Formal Learning Theory 372
Formal Tasks 5
Forward versus Backward Reasoning 134
forward-backward algorithm 437
Forward-Chaining Rule Systems 137
Four Trihedral Vertex 280
Frame Languages 205
frame of discernment 181
Frame Problem 47, 96, 248
frame system 84
frames 83, 193
Frequence Preference 420
fully connected 385
functionally acpurate, cooperative (FA/C) 340
FUTILITY 65
futility cutoff 240
Fuzzification 450
Fuzzy Hedges 454
Fuzzy Inference
  Mamdani 448
  Sugeno 453
Fuzzy Logic 184
Fuzzy Logic Control 447
Fuzzy Logic Systems 445
Fuzzy regions 446, 450, 453, 456
Fuzzy Room Cooler 448
Fuzzy rules 449, 453-455
Fuzzy Set 446
  Support of 447
  Operations 447
Fuzzy Set Operations 447
Fuzzy Sets 446, 447

Copyrighted material

G

Game Playing 231
garden path sentence 294
Gelernter 4
Gene 457
General Problem Solver (GPS) 4, 72
Generalization 349, 389
Generate-and-Test 50
Genetic Algorithms (GAs) 374, 457, 458
  Theoretical Grounding 474
Genetic Learning 373
Genetic Operators 470
Genome 495
Genotypes 457, 458
get/1 515
Global 54
Global Ontology 219
Go 245
Goal 497
Goal Concept 365
Goals
  compound
Goal Stack Planning 255
Goal-Directed preference 420
Goals 497, 499
grab/1 506
Graceful Decay of Admissibility 61
gradient descent 383
gradient search 53
grammar 291
grammatical sentences 292
grams 324
granularity 89, 414
Granulocytes 495
Graph-Unify 299

H

HACKER 262
HARPY 437
HEARSAY 437
HEARSAY-II Blackboard 342, 343
Hamming Distance 487, 488
Herbrand's theorem 112
Heterogeneous Reasoning 337
heuristic 34, 35
heuristic function 35, 55
heuristic knowledge 227, 529
heuristic level 218
Heuristic Search 33
Heuristic Search Techniques 50
hidden Markov modeling 437
Hierarchical Planning 268
Hill Climbing 52
history 411
HMM 437
Hopfield Networks 377
horizon effect 241
Horn clause 131
human intelligence 334
hypothesis 341

I

idiotope 485, 491, 492
Idiotypic theory 485
IEEE Expert 22
IEEE Transactions on Systems 22
image 278
immune 479
Immune memory 481, 483, 493
Immune Network Theory 485, 486
immunis 479
Immunity 480
Implementation 166
Implementation Issues 157
Indexing 138
indirect speech acts 319
IndividualObject 219
Induction 355
Inferential Knowledge 85
Inheritable Knowledge 83
Inheritance 154, 202
initial states 30
Initial Structure 94
Innate Immune System 480
Input/Output and Streams 515
Insertion 326
instance 312
Intangible 219
intelligence 530
Intelligent Agent 516
intelligent being 530
intense 322
Interacting Subgoals 66
interference effect 403
International Conference on Artificial Immune
  Syst 493
International Joint Conference on Artificial Intel 22
Intersection 447
Intersection Search 189

Copyrighted material

Intrinsic 219
Inverses 87
Isa Relationships 103
Isolated-Word Error Correction 327
Iterative Deepening 242

J

Job-Shop Scheduling 458, 459, 475
Jordan Network 399
justification-based truth maintenance system
(JTMS 345
Justification-Based Truth Maintenance Systems 162
justifications 62, 150

K

Knowledge 42
knowledge acquisition 347, 427, 430
knowledge level 79
knowledge representation 14, 79, 222
knowledge representation problem 47
Kohonen
Neural Network 394
Layer 394
Kohonen Neural Network Model 394

L

labeling 163
Lack of Meta-Knowledge 430
layered 385
Learning 347, 485
learning from examples 348
Learning: Generalization of an Input-Output table 462
Leucocytes 495
Levenshtein 327
lexical disambiguation 301
Lexical Processing 301
LISP 232
list 509
appending an element to a 509
appending an list to another 510
lenght in a 511
member of a 511
N^{th} member of a 511
Lists 508
Local 54
Local Ambiguity 277
local maximum 53
local minima 389
Logic and Slot-and-Filler Structures 224
Logic Programming 131
Logic Programming Associates (LPA) PROLOG 516
Logic Theorist 4
Logic-Based Truth Maintenance Systems 166
long-term memory (LTM) 417
lower-level routines 278
LUNAR system 295
lymphokines 481, 484, 495
Lysis 495

M

MACE 344
Machine intelligence quotient 445
Machines Who Think 22
Macro-operators 271, 252
macrophages 480, 495
Major Histocompatibility Complex (MHC) 481
Manipulation 439
manipulator 440
many-to-many mapping 275
Markov assumption 323
matching 47, 138
Matching in PROLOG 502
Matching with Variables 139
Materials 415
Mathematics and Humor 7
mating pool 460
meaning 300
Means-Ends Analysis 72
Membership function 447, 454
Memory Organization 417
memory organization packets (MOPs) 418
message-passing distributed system 344
Message-passing systems 341
META-DENDRAL 429
Metaplanning 271
million-component 377
minimal model 155
Minimalist Reasoning 155
Minimum Edit Distance 327
Minimum Edit Distance Technique 327
modal logic 317
modal truth criterion 266
Model-Based Reasoning 226
Modeling Individual Beliefs 316
Modeling Shared Beliefs 316
Modelling Affinities 486
Models 226
MOLE 427, 428

Copyrighted material

MOLE-p 428
MOLGEN 262
Monkey and Bananas Problem 47
monotonic production system 43
monotonicity 149
Montague Semantics 309
MOP 418, 419
MOPs 418, 419
MOPTRANS 419
Morphological Analysis 287, 288
Motivations 216
multi-agent planning 338
Multi-Agent System (MAS) 520
Multilisp 335
Multiple Contiguous Bit Pattern 488
Multiple Perspectives 337
Mundane Tasks 5
Mutation 458, 461, 467-471, 473, 474, 476, 478
Mutilated Checker board Problem 81
MYCIN 136, 422, 423, 424, 425

N

Natural Deduction 124, 125
Natural Killer (NK) Cells 481
natural language parsing 336
natural language processing 286
natural language understanding 4
Natural Screening Process 458
Navigation 438
NAVLAB 442
nearest neighbor heuristic 34
nearly 248
necessarily true 317
negation as failure 133
Negative selection 482, 483, 493
NETL 400
Network Security 487
Neural Net Learning 373
neural networks 334, 472
Neuro Fuzzy Systems 455
NEUROGAMMON 245
neutrophils 480, 495
NEW 300
Newell 4
N-GRAMS 322
niching 471
Niching and Speciation 471
NOAH 262
Noise 277
Non-Word Error Detection 327
Nonbinary Predicates 189
NONLIN 262
Nonlinear Planning 262
nonmonotonic inference 149
Nonmonotonic Logic 151
nonmonotonic production system 43
nonself 483, 488, 496
nonserializable subgoals 353
nonterminal symbols 292
Nontrihedral 279
noun phrase (NP) 295
NUMBER 297, 299

O

object 312
objective function 55
Obscuring Edge 278
obtuse angles 280
Ontology 219
operationalization 29
operationalized 348
Operationally Criterion 365
Optimization with constraints 464
OR Graphs 57
OR-parallelism 335
Organized Storage of Information 349
originate 347
Orthographic errors 326
Othello 245

P

parallel computation 333
parallel corpora 321
Parallel implementations 335
Parallel LISP models 335
Parallel PROLOG 335
parallel relaxation 383
Paratope 485, 486, 490-493, 495
paratopes 485
PARRY 19
parse tree 292
parser 291
PARSIFAL 294
parsing 291
partial match 350
partially commutative production system 43
Partitioned Semantic Nets 191

Copyrighted material

Pathogens 495
path planning 439
pathogens 480
Pattern-Directed Inference 145
Payoff Matrix 340
Penalty 491, 492
Peptides 495
Perception 434
Perception and Action 431
perceptron convergence theorem 383
Perceptrons 19, 379
Peristalsis 495
perplexity 438

Phagocytes 495
Phenotype 457, 458
Pheromone 476
  Trail 477
  update 476
Phonetic errors 326
physical symbol system hypothesis 6
physical-part-of 316
pick-and-place 440
Pimpernel 323
pixel 434
plan modification operations 266
plan revision 39
plan-generate-test 51
planning 38, 247, 248
Planning System 250
plateau 53
plausible-move generator 232
POLITICS 11
portal 415
Positive selection 482
potential fields 439
Pragmatic Analysis 287, 290
preconditions 72
Predicate Logic 98, 116
Predicates 105, 499
preference heuristics 420
preference semantics 302
prepositional phrase (PP) 295
Principia Mathematica 4, 98
printed-thing 310
Probability 172
Problem 36
Problem Classification 43
problem decomposition 37
Problem Reduction 64
Problem Spaces 25
Problems 25
Procedural Knowledge 85
procedural representation 130
PRODIGY 143, 354
production system 30
Production System Characteristics 43
PROLOG 23, 131, 295, 496, 512, 528
Prolog 496
  facts 496
  goals 497
  predicates 496, 499
  clauses 499
  atoms 499, 503, 523
  character 499
  strings 499
  arity 499
  variables 499, 500
  constants 499
  control structures 500
  arithmetic operators 500, 501
  matching in 502
  lists in 508
Prolog Terminology 499
promotion 264
property inheritance 83
Propositional Logic 113
PROSPECTOR 423
PRS 443
Puralation Model 335

Q

QLISP 335
qualitative 409
Qualitative Physics 409
qualitative simulation 410
Quantity Spaces 410
Question Answering 11, 121
Quiescence 240

R

r-Contiguous Bit Pattern 488
Rates of Change 410
rational 339
reactive systems 249, 269
read/1 515
Real-Time Search 433
Real-Time-A* (RTA*) 433
Reasonableness Conditions 320

Copyrighted material

Reasoning with Qualitative Information 410
Recency Preference 420
receptive field 401
Receptors 483
reconsult 516
Recurrent Networks 399
Recursion 506-508, 512, 514
Redundant Representations 90
reference markers 288
refutation 109
Reinforcement Learning 392
relevant properties 414
Reliability 337
representation mappings 80
Representing Knowledge as Constraints 225
Representing Qualitative Information 410
Reproduction 458, 460, 468, 470, 471, 474, 475
resolution 85, 108, 112
resolution procedure 42
RETE 335
retract 512-514
Reward 491, 492
ribbon 414
ridge 53
R1 423
RLL (Representation Language Language) 218
RM1 289, 311
Road following 439
Robo-SOAR 443
Robot Architectures 441
Robot Control and Navigation 489
robotics 443
rorn 329
rote learning 348
Roulette Wheel 460
RTA* 433
Rule-Based Systems 174
rules 497, 499

S

Salient-Feature Preference 420
SALT 426, 427, 428
satisficers 34
Scarlet 323
scheduler 342
Schema theorem 476
Schemata 474
  Defining length 474, 475
  Order 474, 476
Scripts 212
Search 25
Secondary Search 241
see 515
see/1 515
seen 515
seen/0 515
Self 488
  cells 488
Semantic Analysis 287, 289
semantic grammar 302
Semantic Nets 188
semantic network 83, 84
semantic systems 222
sensor fusion 435
sentence 520
Sentence-Level Processing 302
separation 266
severe smoker 322
Shape Space 486
  Integer 487
  Symbolic 487
  Hamming 487
  Real valued 487
Shaw 4
shells 179, 424
short-term memory (STM) 417
Similarity Net 96
Simon 4
simple establishment 265
Simple Hill Climbing 52
Simulated annealing 55, 389
Sincerity Conditions 320
Single-Valued Attributes 88
singular extensions 241
Skill based Employee Allocation 464
skill refinement 347
slate space 26
slot-and-filler structure 83
Slot-Values 202
Slots 197, 219
Smoothing 325
  Add one 325
SOAR 30, 143, 334
somatic hypermutation 495
Soundex Algorithm 328
Soundex code 328
Space 413
Speciation 471
Species 471
Specificity Preference 420
speech 277

Copyrighted material

speech acts 319
Speech Recognition 436
Spelling 325
Spelling Errors 326
SPHINX 437, 438
squaring iff 34
States 410
Static Control 525
static evaluation function 232
Statistical Natural Language Processing 321
Statistical Reasoning 172
Steepest-Ascent Hill Climbing 53
Stevan Harnad 23
Storing Backed-Up Values 349
story-understanding program SAM 316
Streams
  Input/Output 515
Strings 499
Strips 262, 352, 438
Strong Slot-and-filler Structures 207
Sublanguage Corpora 321
Substance 219
Substitution 326
Subsymbolic Systems 226
Sugano fuzzy inference processing model 453
suggests-revision-of 428
suicide iff 34
Sussman Anomaly 260
Symbol
  Grounding 23
symbol level 80
Symbolic AI 403
Symbolic Reasoning 147
synonym 520
Syntactic Analysis 287, 288
Syntactic Processing 291
syntactic systems 222
System Modularity 337

T

T-Cell Receptor (TCR) 484
T-Cells 481
  generation 482
  receptor 484
Takagi-Sugeno-Kang 453
Tangible 219
tangled hierarchies 202
target language 300
TCA 442
Techniques for Reasoning about Values 88
TEIRESIAS 425, 426
TEIRESIAS-MYCIN 425, 426
tell 515, 516
tell/1 515
temporal logics 317
terminal symbols 292
terminological information 224
The 8-Puzzle 38
The A* Algorithm 59
The Dialog Editor 524
The Level of the Model 18
The Minimax Search Procedure 233
The Missionaries and Cannibals Problem 47
The Scarlet Pimpernel 322
The Traveling Salesman Problem 33
THEO-Agent 442
Thing 219
Three-Agent Scenario 520
Tic-Tac-Toe 8, 29
Time 219, 411
told 515, 516
told/0 515
Tools 220
Top-Down Parsing 293
Tower of Hanoi 47
Training Example 365
Transformational Analogy 371
Traveling Salesman Problem 40
Treebank 321
tremendously beautiful 322
Triangle Tables 270
triggers 342
trigram grammar 438
trihedral 279
Turing Test 20
TWEAK 262, 266
two-word 323
Types and Tokens 322
Typographic errors 326

U

Understanding 272
Unification Algorithm 114
Unification Grammars 298
unify 503
Union 447
Universe of Discourse (U) 447, 454
Unsupervised Learning 392

Copyrighted material

User068 290
User073 290
Utility Problem 354

V

Validation 430
Variables 410, 499
Verb-ACT Dictionary 307
Version Spaces 358
vertex 280
visibility graph 439
Vision 434

W

Wait and See 294
Waltz algorithm 225
Water Jug Problem 45
Weak Slot-And-Filler Structures 188
well-formed formulas 99
well-foundedness 164
White Blood Corpuscles 480
WIN-PROLOG 524, 528
Windows sockets (Winsock) 518
Winston, 1984 21
Winston's Learning Program 356
Witten-Bell Discounting 325
word sense disambiguation 301
word-pair grammar 438
WorldModel 16

Copyrighted material

The McGraw-Hill Companies

ARTIFICIAL INTELLIGENCE

Third Edition

This hallmark text presents both theoretical foundations of Artificial Intelligence and ways in which current techniques can be used in application programs. The new edition has been enriched with specific chapters describing upcoming areas that have found variety of uses under the domain of Artificial Intelligence.

Salient features

- Four new chapters on Fuzzy Logic Systems, Genetic Algorithms, Artificial Immune Systems, and PROLOG
- Important Heuristic Techniques, including Hill Climbing, BFS, and Generate and Test covered explicitly
- Cases on Network Security, Robot Control, and Navigation
- Excellent pedagogy includes
  - 161 Review questions
  - 279 Illustrations

http://www.mhhe.com/rich/ai3

www.tatamcgrawhill.com

ISBN-13: 978-0-07-008770-5

ISBN-10: 0-07-008770-9

9780070087705

Tata McGraw-Hill

Copyrighted material