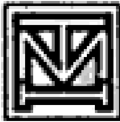
The McGraw·Hill Companies

ARTIFICIAL INTELLIGENCE

Third Edition

Elaine Rich
Kevin Knight
Shivashankar B Nair

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

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 course 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

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, 19801):

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:

Ql: 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

Ql: 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.

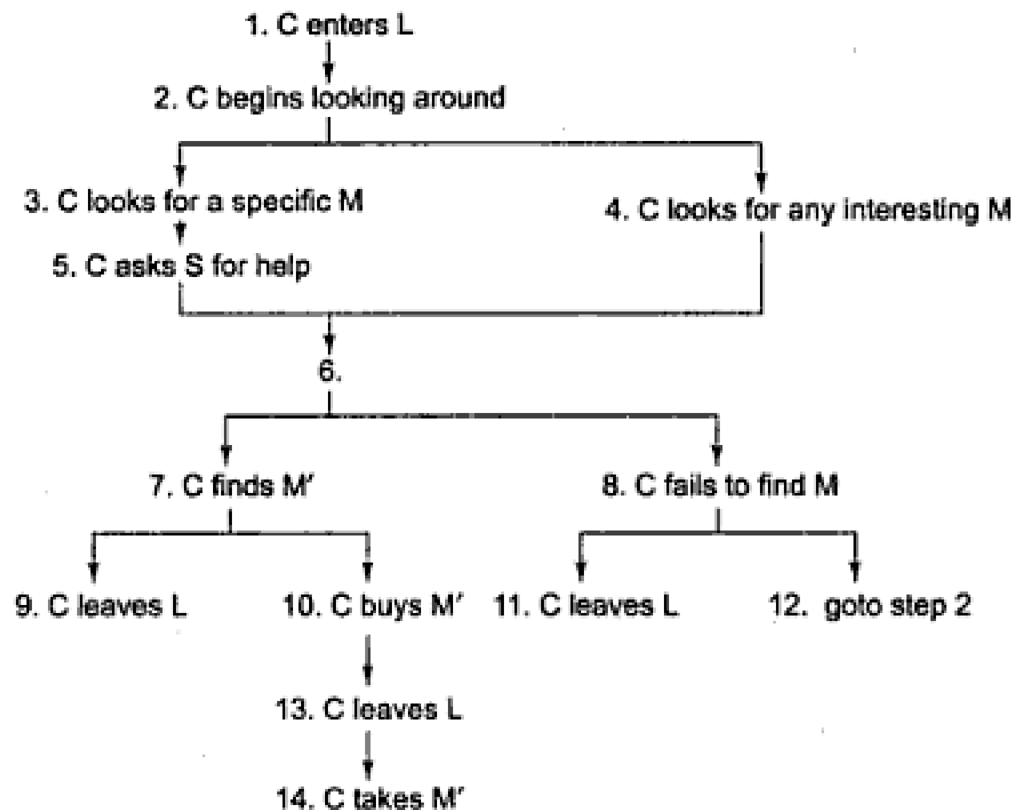


Fig. 1.3 *A Shopping Script*

EnglishKnow — Same as in Program 2.

InputText — The input text in character form.

Copyrighted material

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

For a variety of reasons, LISP has historically been the most commonly used language for AI programming. We say little explicitly about LISP in this book, although we occasionally rely on it as a notation. There used to be several competing dialects of LISP, but Common Lisp is now accepted as a standard. If you are unfamiliar with LISP, consult any of the following sources: *LISP* [Winston and Horn, 1989], *Common Lisp* [Hennessey, 1989], *Common LISPcraft* [Wilensky, 1986], and *Common Lisp: A Gentle Introduction to Symbolic Computation* [Touretzky, 1989a]. For a complete description of Common Lisp, see *Common Lisp: The Reference* [Steele, 1990].

Another language that is often used for AI programming is PROLOG, which is described in Chapter 25 And increasingly, as AI makes its way into the conventional programming world, AI systems are being written in general purpose programming languages such as C. One reason for this is that AI programs are ceasing to be standalone systems; instead, they are becoming components of larger systems, which may include conventional programs and databases of various forms. Real code does not form a big part of this book precisely because it is possible to implement the techniques we discuss in any of several languages and it is important not to confuse the ideas with their specific implementations. But you should keep in mind as you read the rest of this book that both the knowledge structures and the problem-solving strategies we discuss must ultimately be coded and integrated into a working program. This process will definitely throw more light into real world problems faced in the implementation of AI techniques. It is for this reason we have introduced Prolog to ensure that you do not end up just reading and believing.

AI is still a young discipline possibly in the sense that little has been achieved as compared to what was expected. However one must admit a lot more has been learnt about it. We have learnt many things, some of which are presented in this book. But it is still hard to know exactly the perspective from which those things should be viewed. We cannot resist quoting an observation made by Lady Lovelace more than 100 years ago:

In considering any new subject, there is frequently a tendency, first, to *overrate* what we find to be already interesting or remarkable; and, secondly, by a sort of natural reaction, to *undervalue* the true state of the case, when we do discover that our notions have surpassed those that were really tenable. [Lovelace, 1961]

She was talking.about Babbage's Analytical Engine. But she could have been describing artificial intelligence.

While defining AI in terms of symbol processing it would only be right for us to inspect the problem of *Symbol Grounding* [Stevan Harnad, 1990, The Symbol Grounding Problem, Physica, D42, 335-346] and not forget about it while grasping any of the concepts discussed in this book. Harnad defines the symbol grounding problem citing the example of the *Chinese Room* [Searle, 1980]. The basic assumption of symbolic AI is that if a symbol system is able to exhibit behaviors which are indistinguishable from those made by a human being, then it has a mind. Imagine such a system subjected to the Turing test in Chinese. If the system can respond to all Chinese symbol string inputs in just the manner as a native Chinese speaker, then it means (seems) that the system is able to comprehend the meaning of the Chinese symbols just the way we all comprehend our native languages. Searle argues that this cannot be and poses the question — If he (who knows none of Chinese) is given the same strings and does exactly what the computer did (maybe execute the program manually!), would he be understanding Chinese? The rhetoric only leads to one unambiguous inference — *The computer does not understand a thing*. It is thus important to note that the symbols by themselves do not have any intrinsic meaning (like the symbols in a book). They derive their meanings only when we read and the brain comprehends it. It goes to say that if the meaning of the symbols used in a symbol system are extrinsic, unlike the meanings in our heads, then the model itself has no meaning. As the symbols themselves have no meaning and depend on other symbols whose meanings are also extrinsic, we seem to be reasoning around meaningless entities which itself is a meaningless affair! This is the symbol grounding problem.

In the context of the meaninglessness of the use of symbols, Harnad provides a classic example of learning Chinese. Assume you do not know Chinese and had to learn it using a *Chinese to Chinese* dictionary. You

Copyrighted material

would compare character by character of a given word and find the corresponding word in the dictionary only to find many more (meanings) written in the same language alongside, for which you would repeat the same task. The process would put you on an endless merry-go-round. It would be only by translating it to a language that you understand that your brain can finally perceive what it means. The Chinese symbols in the present case are not *grounded* to its meaning. The moral of the example is simple — *You cannot ground the meaning of a symbol with other meaningless symbols.* Harnad also cites that cryptologists are able to comprehend ancient languages and symbols because their efforts are grounded in their real world domain knowledge as also on some previous language that forms its basis.

Robots form the ultimate test-bed for AI. While AI researchers have brought forth a reasonably large repository of techniques and programs that are based on the symbol system, implementing them on robots have posed several problems. Though this may be beyond the scope of this book we must exercise caution while implementing symbolic AI. For instance on board a robot a symbol 'red' has to be actually grounded to some values reported by the camera or a colour sensor.

Finally one should not forget that research in AI is multidisciplinary. People have been using AI techniques to reap benefits in a gamut of applications. There are still a lot more untrodden paths to be discovered. In the quest to find better techniques, the reader is advised to give imagination a free run so that the marginal and the peripheral are accommodated without losing the grounding of each symbol.

EXERCISES

1. Pick a specific topic within the scope of AI and use the sources described in this chapter to do a preliminary literature search to determine what the current state of understanding of that topic is. If you cannot think of a more novel topic, try one of the following: expert systems for some specific domain (e.g., cancer therapy, computer design, financial planning), recognizing motion in images, using natural (i.e., humanlike) methods for proving mathematical theorems, resolving pronominal references in natural language texts, representing sequences of events in time, or designing a memory organization scheme for knowledge in a computer system based on our knowledge of human memory organization.
2. Explore the spectrum from static to AI-based techniques for a problem other than the two discussed in this chapter. Think of your own problem or use one of the following:
   - Translating an English sentence into Japanese
   - Teaching a child to subtract integers
   - Discovering patterns in empirical data taken from scientific experiments, and suggesting further experiments to find more patterns
3. Imagine that you had been to an aquarium and seen a shark and an octopus. Describe these to a child who has never seen one. What resources and mechanisms does the child use to comprehend the nature of these marine animals?

Copyrighted material

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.

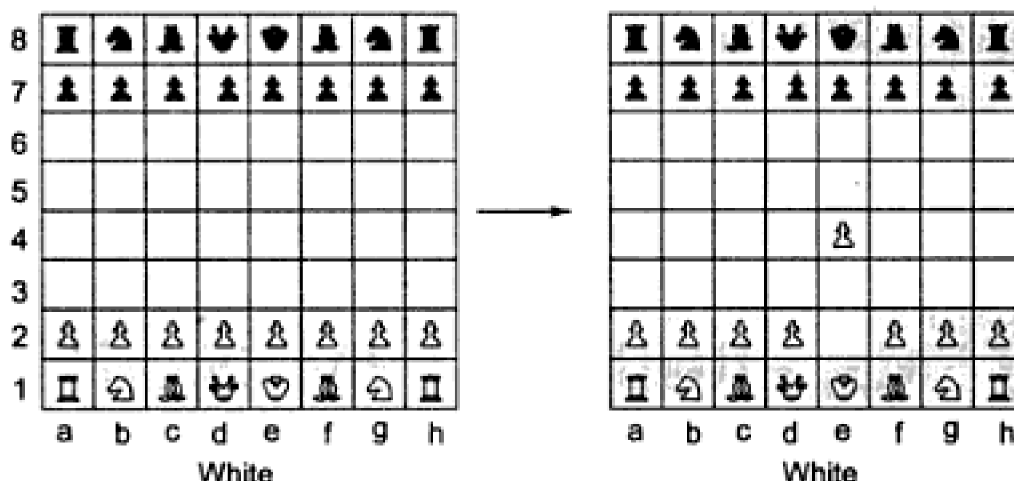


Fig. 2.1 *One Legal Chess Move*

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

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

The extreme of this approach is shown in the first tic-tac-toe program of Chapter 1. Each entry in the move vector corresponds to a rule that describes an operation. The left side of each rule describes a board configuration and is represented implicitly by the index position. The right side of each rule describes the operation to be performed and is represented by a nine-element vector that corresponds to the resulting board configuration. Each of these rules is maximally specific; it applies only to a single board configuration, and, as a result, no search is required when such rules are used. However, the drawback to this extreme approach is that the problem solver can take no action at all in a novel situation. In fact, essentially no problem *solving* ever really occurs. For a tic-tac-toe playing program, this is not a serious problem, since it is possible to enumerate all the situations (i.e., board configurations) that may occur. But for most problems, this is not the case. In order to solve new problems, more general rules must be available.

A second issue is exemplified by rules 3 and 4 in Fig. 2.3. Should they or should they not be included in the list of available operators? Emptying an unmeasured amount of water onto the ground is certainly allowed by the problem statement. But a superficial preliminary analysis of the problem makes it clear that doing so will never get us any closer to a solution. Again, we see the tradeoff between writing a set of rules that describe just the problem itself, as opposed to a set of rules that describe both the problem and some knowledge about its solution.

Rules 11 and 12 illustrate a third issue. To see the problem-solving knowledge that these rules represent, look at the last two steps of the solution shown in Fig. 2.4. Once the state (4, 2) is reached, it is obvious what to do next. The desired 2 gallons have been produced, but they are in the wrong jug. So the thing to do is to move them (rule 11). But before that can be done, the water that is already in the 4-gallon jug must be emptied out (rule 12). The idea behind these special-purpose rules is to capture the special-case knowledge that can be used at this stage in solving the problem. These rules do not actually add power to the system since the operations they describe are already provided by rule 9 (in the case of rule 11) and by rule 5 (in the case of rule 12). In fact, depending on the control strategy that is used for selecting rules to use during problem solving, the use of these rules may degrade performance. But the use of these rules may also improve performance if preference is given to special-case rules (as we discuss in Section 6.4.3).

We have now discussed two quite different problems, chess and the water jug problem. From these discussions, it should be clear that the first step toward the design of a program to solve a problem must be the creation of a formal and manipulable description of the problem itself. Ultimately, we would like to be able to write programs that can themselves produce such formal descriptions from informal ones. This process is called *operationalization*. It is not at all well-understood how to construct such programs, but see Section 17.3 for a description of one program that solves a piece of this problem. Until it becomes possible to automate this process, it must be done by hand, however. For simple problems, such as chess or the water jug, this is not very difficult. The problems are artificial and highly structured. For other problems, particularly naturally-occurring ones, this step is much more difficult. Consider, for example, the task of specifying precisely what it means to understand an English sentence. Although such a specification must somehow be provided before we can design a program to solve the problem, producing such a specification is itself a very hard problem. Although our ultimate goal is to be able to solve difficult, unstructured problems, such as natural language understanding, it is useful to explore simpler problems, such as the water jug problem, in order to gain insight into the details of methods that can form the basis for solutions to the harder problems.

Summarizing what we have just said, in order to provide a formal description of a problem, we must do the following:

1. Define a state space that contains all the possible configurations of the relevant objects (and perhaps some impossible ones). It is, of course, possible to define this space without explicitly enumerating all of the states it contains.

Copyrighted material

2. Specify one or more states within that space that describe possible situations from which the problem-solving process may start. These states are called the *initial states.*
3. Specify one or more states that would be acceptable as solutions to the problem. These states are called *goal states.*
4. Specify a set of rules that describe the actions (operators) available. Doing this will require giving thought to the following issues:
   - What unstated assumptions are present in the informal problem description?
   - How general should the rules be?
   - How much of the work required to solve the problem should be precomputed and represented in the rules?

The problem can then be solved by using the rules, in combination with an appropriate control strategy, to move through the problem space until a path from an initial state to a goal state is found. Thus the process of search is fundamental to the problem-solving process. The fact that search provides the basis for the process of problem-solving does not, however, mean that other, more direct approaches cannot also be exploited. Whenever possible, they can be included as steps in the search by encoding them into the rules. For example, in the water jug problem, we use the standard arithmetic operations as single steps in the rules. We do not use search to find a number with the property that it is equal to $y — (4 — x)$. Of course, for complex problems, more sophisticated computations will be needed. Search is a general mechanism that can be used when no more direct method is known. At the same time, it provides the framework into which more direct methods for solving subparts of a problem can be embedded.

2.2 PRODUCTION SYSTEMS

Since search forms the core of many intelligent processes, it is useful to structure AI programs in a way that facilitates describing and performing the search process. Production systems provide such structures. A definition of a production system is given below. Do not be confused by other uses of the word *production,* such as to describe what is done in factories. A *production system* consists of:

- A set of rules, each consisting of a left side (a pattern) that determines the applicability of the rule and a right side that describes the operation to be performed if the rule is applied.[3]
- One or more knowledge/databases that contain whatever information is appropri- ate for the particular task. Some parts of the database may be permanent, while other parts of it may pertain only to the solution of the current problem. The information in these databases may be structured in any appropriate way.
- A control strategy that specifies the order in which the rules will be compared to the database and a way of resolving the conflicts that arise when several rules match at once.
- A rule applier.

So far, our definition of a production system has been very general. It encompasses a great many systems, including our descriptions of both a chess player and a water jug problem solver. It also encompasses a family of general production system interpreters, including:

- Basic production system languages, such as OPS5 [Brownston *et al.*, 1985] and ACT* [Anderson, 1983].
- More complex, often hybrid systems called *expert system shells,* which provide complete (relatively speaking) environments for the construction of knowledge- based expert systems.
- General problem-solving architectures like SOAR [Laird *et al.*, 1987], a system based on a specific set of cognitively motivated hypotheses about the nature of problem-solving.

[3] This convention for the use of left and right sides is natural for forward rules. As we will see later, many backward rule systems reverse the sides.

Copyrighted material

All of these systems provide the overall architecture of a production system and allow the programmer to write rules that define particular problems to be solved. We discuss production system issues further in Chapter 6.

We have now seen that in order to solve a problem, we must first reduce it to one for which a precise statement can be given. This can be done by defining the problem's state-space (including the start and goal states) and a set of operators for moving in that space. The problem can then be solved by searching for a path through the space from an initial state to a goal state. The process of solving the problem can usefully be modeled as a production system. In the rest of this section, we look at the problem of choosing the appropriate control structure for the production system so that the search can be as efficient as possible.

2.2.1 Control Strategies

So far, we have completely ignored the question of how to decide which rule to apply next during the process of searching for a solution to a problem. This question arises since often more than one rule (and sometimes fewer than one rule) will have its left side match the current state. Even without a great deal of thought, it is clear that how such decisions are made will have a crucial impact on how quickly, and even whether, a problem is finally solved.

- *The first requirement of a good control strategy is that it causes motion.* Consider again the water jug problem of the last section. Suppose we implemented the simple control strategy of starting each time at the top of the list of rules and choosing the first applicable one. If we did that, we would never solve the problem. We would continue indefinitely filling the 4-gallon jug with water Control strategies that do not cause motion will never lead to a solution.
- *The second requirement of a good control strategy is that it be systematic.* Here is another simple control strategy for the water jug problem: On each cycle, choose at random from among the applicable rules. This strategy is better than the first. It causes motion. It will lead to a solution eventually. But we are likely to arrive at the same state several times during the process and to use many more steps than are necessary. Because the control strategy is not systematic, we may explore a particular useless sequence of operators several times before we finally find a solution. The requirement that a control strategy be systematic corresponds to the need for global motion (over the course of several steps) as well as for local motion (over the course of a single step). One systematic control strategy for the water jug problem is the following. Construct a tree with the initial state as its root. Generate all the offspring of the root by applying each of the applicable rules to the initial state. Fig. 2.5 shows how the tree looks at this point. Now for each leaf node, generate all its successors by applying all the rules that are appropriate. The tree at this point is shown in Fig. 2.6.[4] Continue this process until some rule produces a goal state. This process, called *breadth-first search*, can be described precisely as follows.

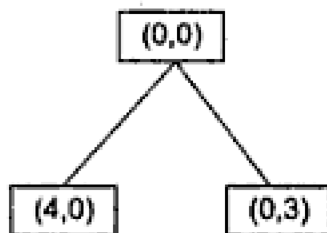


Fig. 2.5 *One Level of a Breadth-First Search Tree*

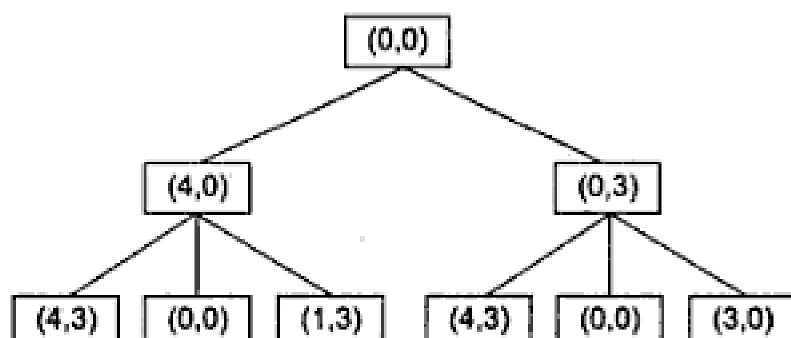


Fig. 2.6 *Two Levels of a Breadth-First Search Tree*

[4] Rule 3, 4, 11, and 12 have been ignored in constructing the search tree.

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

2.3.1 Is the Problem Decomposable?

Suppose we want to solve the problem of computing the expression

$$\int (x^2 + 3x + \sin^{2*} \cdot \cos^2 x)\, dx$$

We can solve this problem by breaking it down into three smaller problems, each of which we can then solve by using a small collection of specific rules. Figure 2.9 shows the problem tree that will be generated by the process of problem decomposition as it can be exploited by a simple recursive integration program that works as follows: At each step, it checks to see whether the problem it is working on is immediately solvable. If so, then the answer is returned directly. If the problem is not easily solvable, the integrator checks to see whether it can decompose the problem into smaller problems. If it can, it creates those problems and calls itself recursively on them. Using this technique of *problem decomposition,* we can often solve very large problems easily.

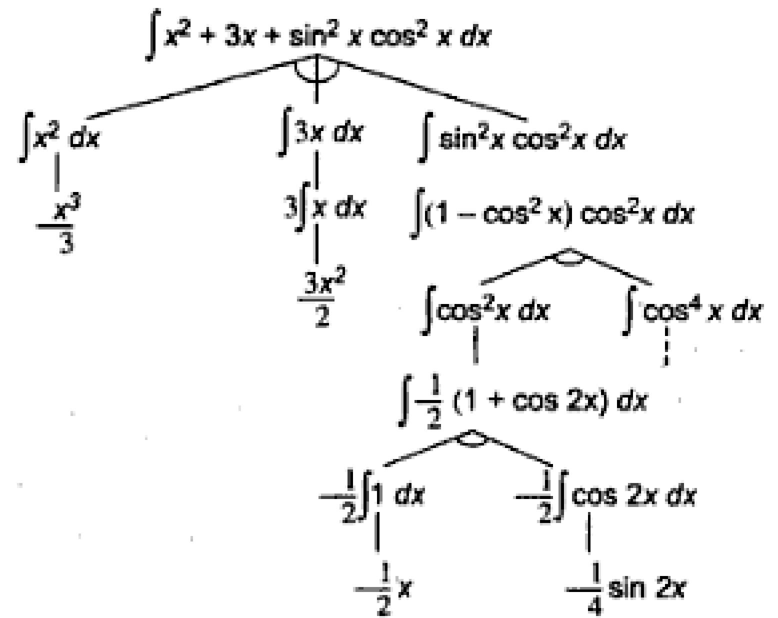


Fig. 2.9 *A Decomposable Problem*

Now consider the problem illustrated in Fig. 2.10. This problem is drawn from the domain often referred to in AI literature as the *blocks world.* Assume that the following operators are available:

1. CLEAR (x) [block x has nothing on it] → ON (x, Table) [pick up x and put it on the table]
2. CLEAR (x) and CLEAR (y) → ON (x, y) [put x on y]

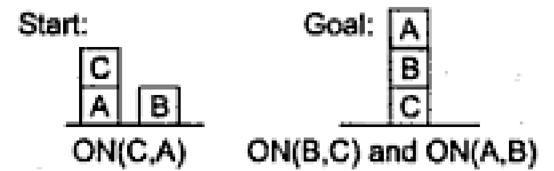


Fig. 2.10 *A Simple Blocks World Problem*

Applying the technique of problem decomposition to this simple blocks world example would lead to a solution tree such as that shown in Fig. 2.11. In the figure, goals are underlined. States that have been achieved are not underlined. The idea of this solution is to reduce the problem of getting B on C and A on B to two separate problems. The first of these new problems, getting B on C, is simple, given the start state. Simply put B on C. The second subgoal is not quite so simple. Since the only operators we have allow us to pick up single blocks at a time, we have to clear off A by removing C before we can pick up A and put it on B. This can easily be done. However, if we now try to combine the two subsolutions into one solution, we will fail. Regardless of which one we do first, we will not be able to do the second as we had planned. In this problem, the two subproblems are not independent. They interact and those interactions must be considered in order to arrive at a solution for the entire problem.

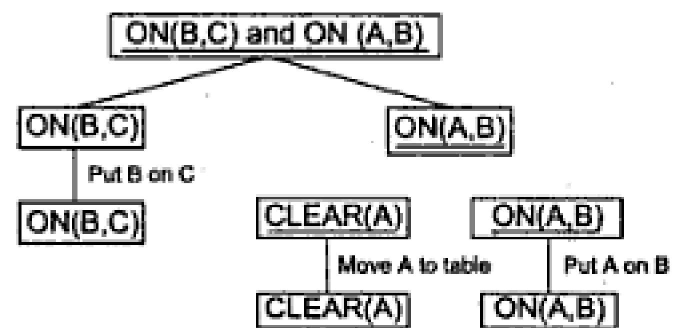


Fig. 2.11 *A Proposed Solution for the Blocks Problem*

These two examples, symbolic integration and the blocks world, illustrate the difference between decomposable and nondecomposable problems. In Chapter 3, we present a specific algorithm for problem decomposition, and in Chapter 13, we look at what happens when decomposition is impossible.

2.3.2 Can Solution Steps Be Ignored or Undone?

Suppose we are trying to prove a mathematical theorem. We proceed by first proving a lemma that we think will be useful. Eventually, we realize that the lemma is no help at all. Are we in trouble?

Copyrighted material

No. Everything we need to know to prove the theorem is still true and in memory, if it ever was. Any rules that could have been applied at the outset can still be applied. We can just proceed as we should have in the first place. All we have lost is the effort that was spent exploring the blind alley.

Now consider a different problem.

> **The 8-Puzzle:** The 8-puzzle is a square tray in which are placed, eight square tiles. The remaining ninth square is uncovered. Each tile has a number on it. A tile that is adjacent to the blank space can be slid into that space. A game consists of a starting position and a specified goal position. The goal is to transform the starting position into the goal position by sliding the tiles around.

A sample game using the 8-puzzle is shown in Fig. 2.12. In attempting to solve the 8-puzzle, we might make a stupid move. For example, in the game shown above, we might start by sliding tile 5 into the empty space. Having done that, we cannot change our mind and immediately slide tile 6 into the empty space since the empty space will essentially have moved. But we can backtrack and undo the first move, sliding tile 5 back to where it was, Then we can move tile 6. Mistakes can still be recovered from but not quite as easily as in the theorem-proving problem. An additional step must be performed to undo each incorrect step, whereas no action was required to "undo" a useless lemma. In addition, the control mechanism for an 8-puzzle solver must keep track of the order in which operations are performed so that the operations can be undone one at a time if necessary. The control structure for a theorem prover does not *need* to record all that information.

Start

2	8	3
1	6	4
7		5

Goal

1	2	3
8		4
7	6	5

Fig. 2.12 *An Example of the 8-Puzzle*

Now consider again the problem of playing chess. Suppose a chess-playing program makes a stupid move and realizes it a couple of moves later. It cannot simply play as though it had never made the stupid move. Nor can it simply back up and start the game over from that point. All it can do is to try to make the best of the current situation and go on from there.

These three problems—theorem proving, the 8-puzzle, and chess—illustrate the differences between three important classes of problems:

- Ignorable (e.g., theorem proving), in which solution steps can be ignored
- Recoverable (e.g., 8-puzzle), in which solution steps can be undone
- Irrecoverable (e.g., chess), in which solution steps cannot be undone

These three definitions make reference to the steps of the solution to a problem and thus may appear to characterize particular production systems for solving a problem rather than the problem itself. Perhaps a different formulation of the same problem would lead to the problem being characterized differently. Strictly speaking, this is true. But for a great many problems, there is only one (or a small number of essentially equivalent) formulations that *naturally* describe the problem. This was true for each of the problems used as examples above. When this is the case, it makes sense to view the recoverability of a problem as equivalent to the recoverability of a natural formulation of it.

The recoverability of a problem plays an important role in determining the complexity of the control structure necessary for the problem's solution. Ignorable problems can be solved using a simple control structure that never backtracks. Such a control structure is easy to implement. Recoverable problems can be solved by a slightly more complicated control strategy that does sometimes make mistakes. Backtracking will be necessary to recover from such mistakes, so the control structure must be implemented using a push-down stack, in which decisions are recorded in case they need to be undone later. Irrecoverable problems, on the other hand, will need to be solved by a system that expends a great deal of effort making each decision since the decision must be final. Some irrecoverable problems can be solved by recoverable style methods used in a *planning* process, in which an entire sequence of steps is analyzed in advance to discover where it will lead before the first step is actually taken. We discuss next the kinds of problems in which this is possible.

Copyrighted material

2.3.3 Is the Universe Predictable?

Again suppose that we are playing with the 8-puzzle. Every time we make a move, we know exactly what will happen. This means that it is possible to plan an entire sequence of moves and be confident that we know what the resulting state will be. We can use planning to avoid having to undo actual moves, although it will still be necessary to backtrack past those moves one at a time during the planning process. Thus a control structure that allows backtracking will be necessary.

However, in games other than the 8-puzzle, this planning process may not be possible. Suppose we want to play bridge. One of the decisions we will have to make is which card to play on the first trick. What we would like to do is to plan the entire hand before making that first play. But now it is not possible to do such planning with certainty since We cannot know exactly where all the cards are or what the other players will do on their turns. The best we can do is to investigate several plans and use probabilities of the various outcomes to choose a plan that has the highest estimated probability of leading to a good score on the hand.

These two games illustrate the difference between certain-outcome (e.g., 8-puzzle) and uncertain-outcome (e.g., bridge) problems. One way of describing planning is that it is problem-solving without feedback from the environment. For solving certain-outcome problems, this open-loop approach will work fine since the result of an action can be predicted perfectly. Thus, planning can be used to generate a sequence of operators that is guaranteed to lead to a solution. For uncertain-outcome problems, however, planning can at best generate a sequence of operators that has a good probability of leading to a solution. To solve such problems, we need to allow for a process of *plan revision* to take place as the plan is carried out and the necessary feedback is provided. In addition to providing no guarantee of an actual solution, planning for uncertain-outcome problems has the drawback that it is often very expensive since the number of solution paths that need to be explored increases exponentially with the; number of points at which the outcome cannot be predicted.

The last two problem characteristics we have discussed, ignorable versus recoverable versus irrecoverable and certain-outcome versus uncertain-outcome, interact in an interesting way. As has already been mentioned, one way to solve irrecoverable problems is to plan an entire solution before embarking on an implementation of the plan. But this planning process can only be done effectively for certain-outcome problems. Thus one of the hardest types of problems to solve is the irrecoverable, uncertain-outcome. A few examples of such problems are:

- Playing bridge. But we can do fairly well since we have available accurate estimates of the probabilities of each of the possible outcomes.
- Controlling a robot arm. The outcome is uncertain for a variety of reasons. Someone might move something into the path of the arm. The gears of the arm might stick. A slight error could cause the arm to knock over a whole stack of things.
- Helping a lawyer decide how to defend his client against a murder charge. Here we probably cannot even list all the possible outcomes, much less assess their probabilities.

2.3.4 Is a Good Solution Absolute or Relative?

Consider the problem of answering questions based on a database of simple facts, such as the following:

1. Marcus was a man.
2. Marcus was a Pompeian.
3. Marcus was born in 40 A.D.
4. All men are mortal.
5. All Pompeians died when the volcano erupted in 79 A.D.
6. No mortal lives longer than 150 years.
7. It is now 1991 A.D.

Copyrighted material

Suppose we ask the question "Is Marcus alive?" By representing each of these facts in a formal language, such as predicate logic, and then using formal inference methods we can fairly easily derive an answer to the question.[7] In fact, either of two reasoning paths will lead to the answer, as shown in Fig. 2.13. Since all we are interested in is the answer to the question, it does not matter which path we follow. If we do follow one path successfully to the answer, there is no reason to go back and see if some other path might also lead to a solution.

		Justification
1.	Marcus was a man.	axiom 1
4.	All men are mortal.	axiom 4
8.	Marcus is mortal.	1, 4
3.	Marcus was born in 40 A.D.	axiom 3
7.	It is now 1991 A.D.	axiom 7
9.	Marcus' age is 1951 years.	3, 7
6.	No mortal lives longer than 150 years.	axiom 6
10.	Marcus is dead.	8, 6, 9
	OR	
7.	It is now 1991 A.D.	axiom 7
5.	All Pompeians died in 79 A.D.	axiom 5
11.	All Pompeians are dead now.	7, 5
2.	Marcus was a Pompeian.	axiom 2
12.	Marcus is dead.	11, 2

Fig. 2.13 *Two Ways of Deciding That Marcus Is Dead*

But now consider again the traveling salesman problem. Our goal is to find the shortest route that visits each city exactly once. Suppose the cities to be visited and the distances between them are as shown in Fig. 2.14.

	Boston	New York	Miami	Dallas	S.F.
Boston		250	1450	1700	3000
New York	250		1200	1500	2900
Miami	1450	1200		1600	3300
Dallas	1700	1500	1600		1700
S.F.	3000	2900	3300	1700	

Fig. 2.14 *An Instance of the Traveling Salesman Problem*

One place the salesman could start is Boston. In that case, one path that might be followed is the one shown in Fig. 2.15, which is 8850 miles long. But is this the solution to the problem? The answer is that we cannot be sure unless we also try all other paths to make sure that none of them is shorter. In this case, as can be seen from Fig. 2.16, the first path is definitely not the solution to the salesman's problem.

These two examples illustrate the difference between any-path problems and best- path problems. Best-path problems are, in general, computationally harder than any-path problems. Any-path problems can often be solved in a reasonable amount of time by using heuristics that suggest good paths to explore. (See the discussion of best-first search in Chapter 3 for one way of doing this.) If the heuristics are not perfect, the search for a solution may not be as direct as possible, but that does not matter. For true best-path problems, however, no heuristic that could possibly miss the best solution can be used. So a much more exhaustive search will be performed.

[7] Of course, representing these statements so that a mechanical procedure could exploit them to answer the question also requires the explicit mention of other facts, such as "dead implies not alive." We do this in Chapter 5.

Copyrighted material

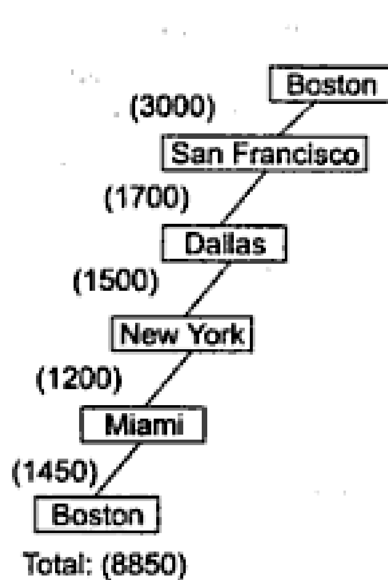


Fig. 2.15 *One Path among the Cities*

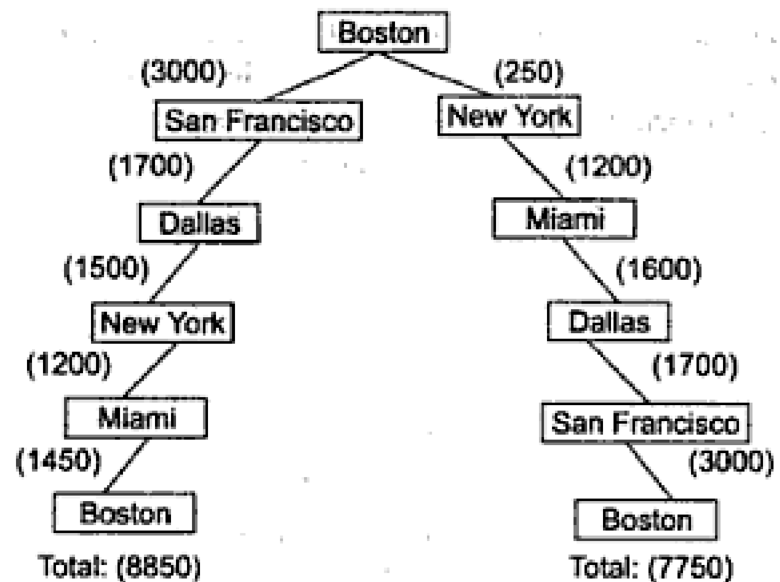


Fig. 2.16 *Two Paths Among the Cities*

2.3.5 Is the Solution a State or a Path?

Consider the problem of finding a consistent interpretation for the sentence

The bank president ate a dish of pasta salad with the fork.

There are several components of this sentence, each of which, in isolation, may have more than one interpretation. But the components must form a coherent whole, and so they constrain each other's interpretations. Some of the sources of ambiguity in this sentence are the following:

The word "bank" may refer either to a financial institution or to a side of a river. But only one of these may have a president.

- The word "dish" is the object of the verb "eat." It is possible that a dish was eaten. But it is more likely that the pasta salad in the dish was eaten.
- Pasta salad is a salad containing pasta. But there are other ways meanings can be formed from pairs of nouns. For example, dog food does not normally contain dogs.
- The phrase "with the fork" could modify several parts of the sentence. In this case, it modifies the verb "eat." But, if the phrase had been "with vegetables," then the modification structure would be different. And if the phrase had been "with her friends," the structure would be different still.

Because of the interaction among the interpretations of the constituents of this sentence, some search may be required to find a complete interpretation for the sentence. But to solve the problem of finding the interpretation we need to produce only the interpretation itself. No record of the processing by which the interpretation was found is necessary.

Contrast this with the water jug problem. Here it is not sufficient to report that we have solved the problem and that the final state is (2, 0). For this kind of problem, what we really must report is not the final state but the path that we found to that state. Thus a statement of a solution to this problem must be a sequence of operations (sometimes called *apian)* that produces the final state.

These two examples, natural language understanding and the water jug problem, illustrate the difference between problems whose solution is a state of the world and problems whose solution is a path to a state. At one level, this difference can be ignored and all problems can be formulated as ones in which only a state is required to be reported. If we do this for problems such as the water jug, then we must redescribe our states so that each state represents a partial path to a solution rather than just a single state of the world. So this question

Copyrighted material

- Solitary, in which the computer is given a problem description and produces an answer with no intermediate communication and with no demand for an explanation of the reasoning process
- Conversational, in which there is intermediate communication between a person and the computer, either to provide additional assistance to the computer or to provide additional information to the user, or both

Of course, this distinction is not a strict one describing particular problem domains. As we just showed, mathematical theorem proving could be regarded as either. But for a particular application, one or the other of these types of systems will usually be desired and that decision will be important in the choice of a problem-solving method.

2.3.8 Problem Classification

When actual problems are examined from the point of view of all of these questions, it becomes apparent that there are several broad classes into which the problems fall. These classes can each be associated with a generic control strategy that is appropriate for solving the problem. For example, consider the generic problem of *classification.* The task here is to examine an input and then decide which of a set of known classes the input is an instance of. Most diagnostic tasks, including medical diagnosis as well as diagnosis of faults in mechanical devices, are examples of classification. Another example of a generic strategy is *propose and refine.* Many design and planning problems can be attacked with this strategy.

Depending on the granularity at which we attempt to classify problems and control strategies, we may come up with different lists of generic tasks and procedures. See Chandrasekaran [1986] and McDermott [1988] for two approaches to constructing such lists. The important thing to remember here, though, since we are about to embark on a discussion of a variety of problem-solving methods, is that there is no one single way of solving all problems. But neither must each new problem be considered totally *ab initio.* Instead, if we analyze our problems carefully and sort our problem-solving methods by the kinds of problems for which they are suitable, we will be able to bring to each new problem much of what we have learned from solving other, similar problems.

2.4 PRODUCTION SYSTEM CHARACTERISTICS

We have just examined a set of characteristics that distinguish various classes of problems. We have also argued that production systems are a good way to describe the operations that can be performed in a search for a solution to a problem. Two questions we might reasonably ask at this point are:

1. Can production systems, like problems, be described by a set of characteristics that shed some light on how they can easily be implemented?
2. If so, what relationships are there between problem types and the types of pro- duction systems best suited to solving the problems?

The answer to the first question is yes. Consider the following definitions of classes of production systems. A *monotonic production system* is a production system in which the application of a rule never prevents the later application of another rule that could also have been applied at the time the first rule was selected. A *nonmonotonic production system* is one in which this is not true. A *partially commutative production system* is a production system with the property that if the application of a particular sequence of rules transforms state x into state y, then any permutation of those rules that is allowable (i.e., each rule's preconditions are satisfied when it is applied) also transforms state x into state y. A *commutative production system* is a production system that is both monotonic and partially commutative.[8]

[8] This corresponds to the definition of a commutative production system given in Nilsson [1980].

Copyrighted material

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.

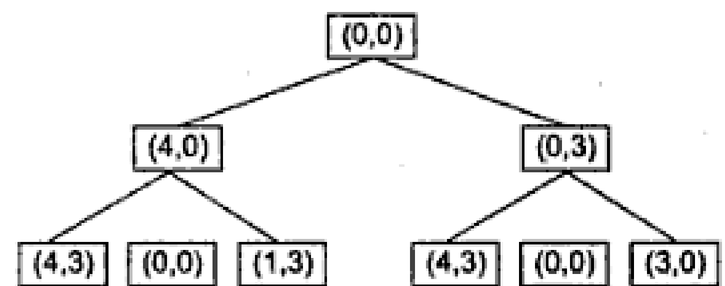


Fig. 2.18 *A Search Tree for the Water Jug Problem*

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.

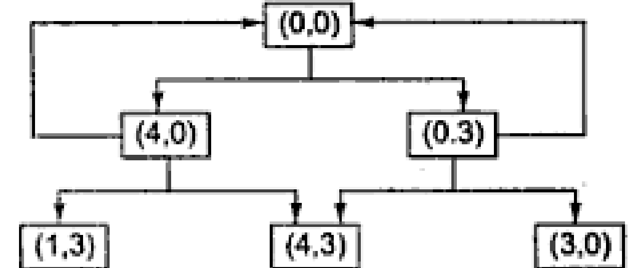


Fig. 2.19 *A Search Graph for the Water Jug Problem*

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.

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:

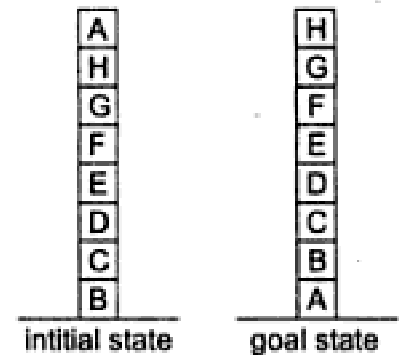


Fig. 3.1 *A Hill-Climbing Problem*

> **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.

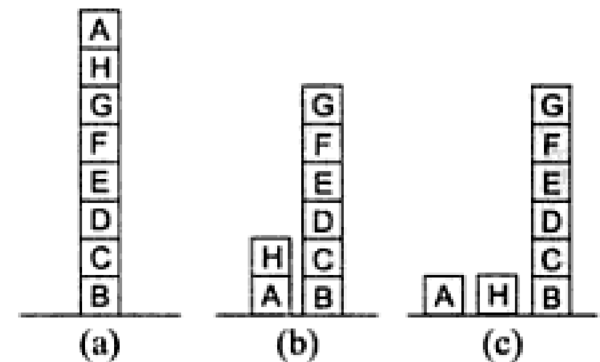


Fig. 3.2 *Three Possible Moves*

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

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

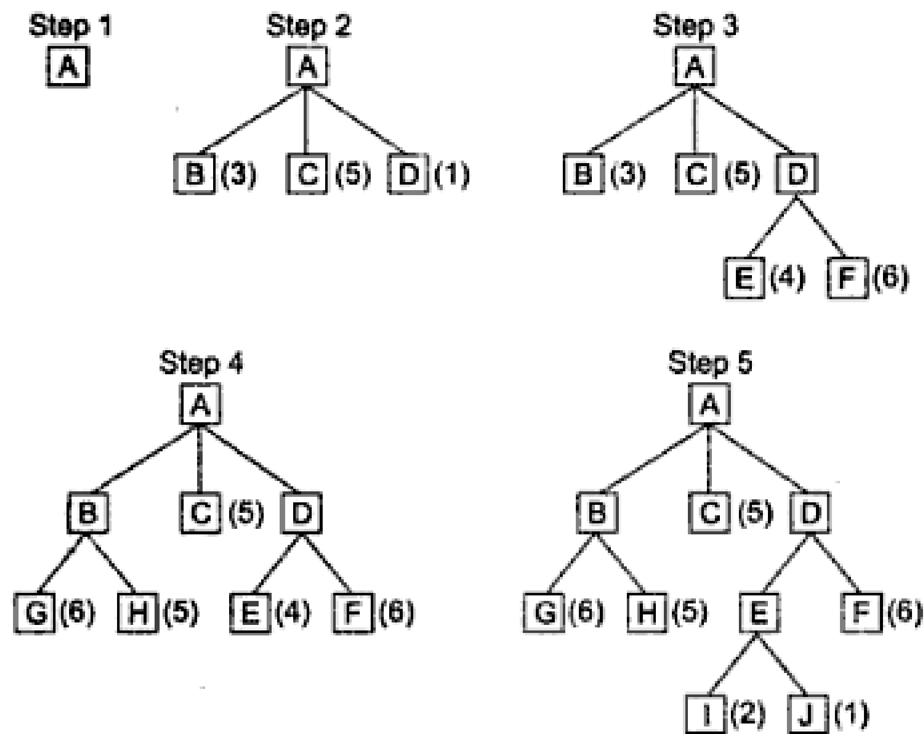


Fig. 3.3 *A Best-First Search*

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 asf'(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.

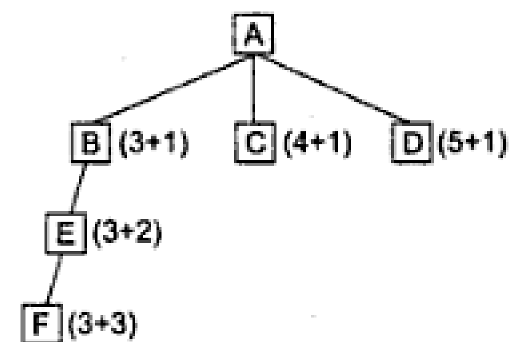


Fig. 3.4 *h′ Underestimates h*

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:

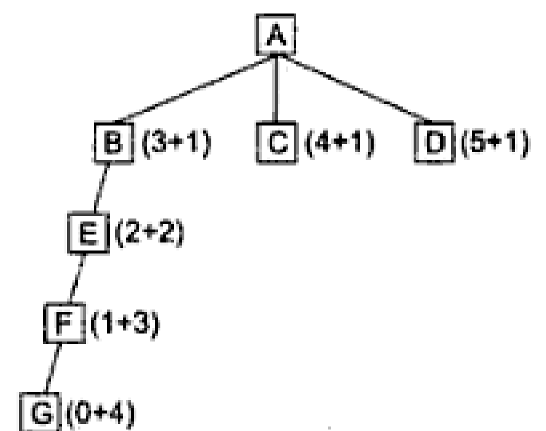


Fig. 3.5 *h′ Overestimates h*

> **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

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.

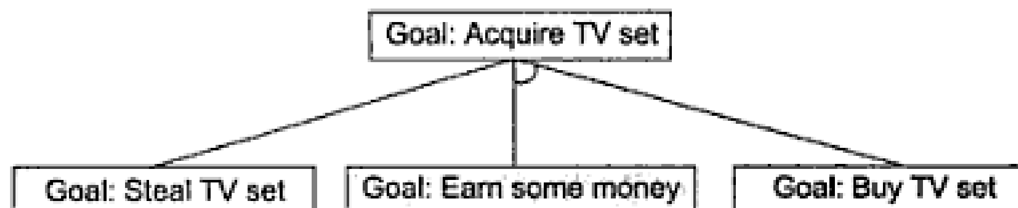


Fig. 3.6 *A Simple AND-OR Graph*

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

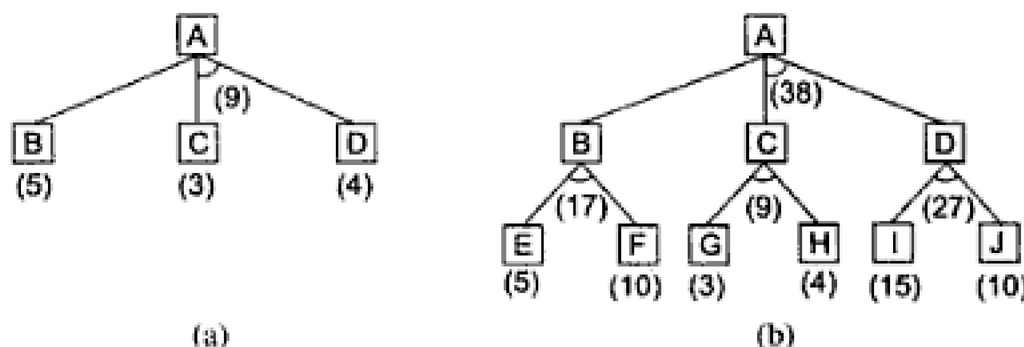


Fig. 3.7 *AND-OR Graphs*

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

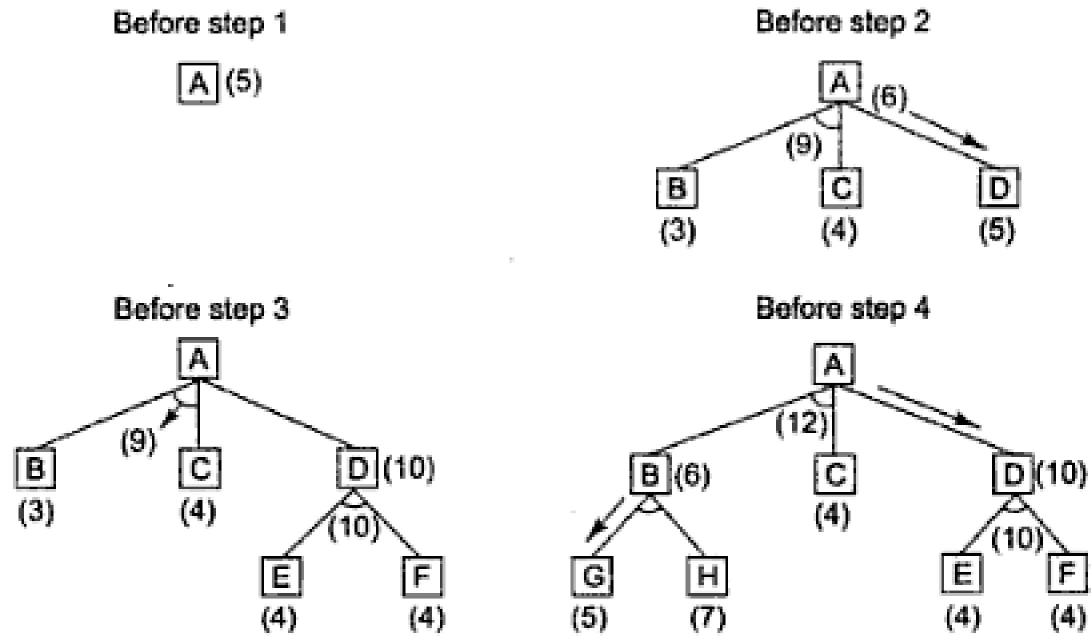


Fig. 3.8 *The Operation of Problem Reduction*

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.

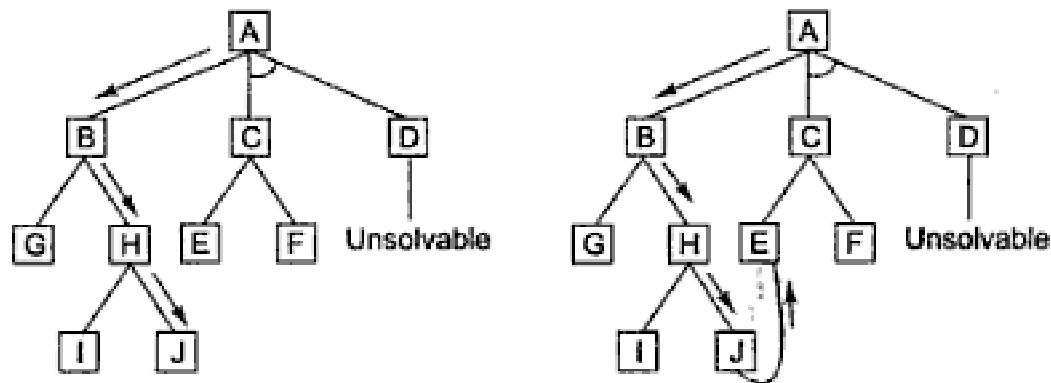


Fig. 3.9 *A Longer Path May Be Better*

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.

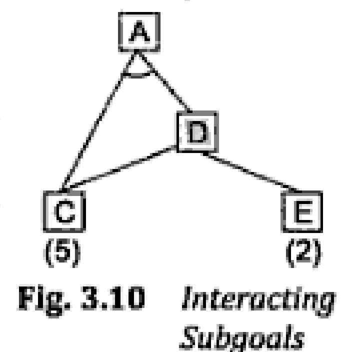


Fig. 3.10 *Interacting Subgoals*

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.

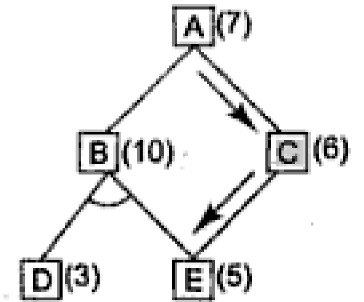
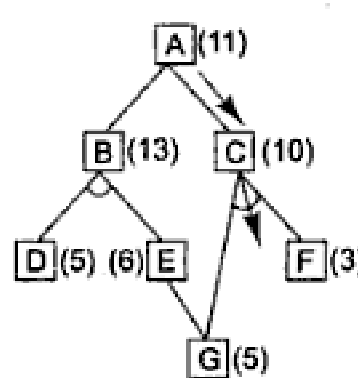
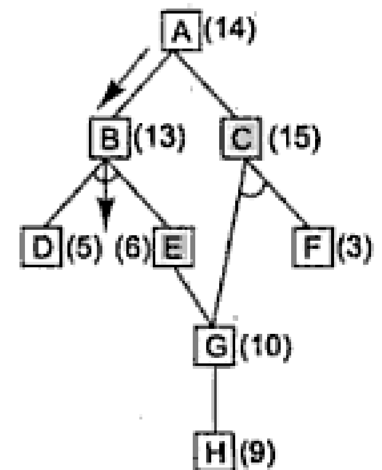


Fig. 3.11 *An Unnecessary Backward Propagation*





(a) (b)

Fig. 3.12 *A Necessary 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.

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(l) + 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

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.

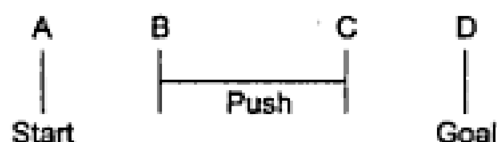


Fig. 3.17 *The Progress of the Means-Ends Analysis Method*

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.

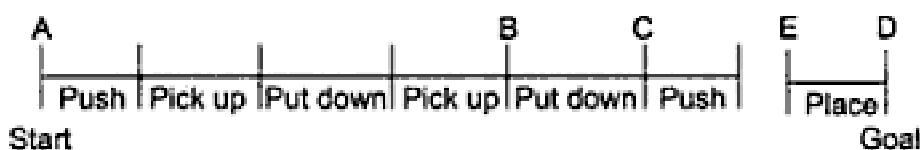


Fig. 3.18 *More Progress of the Means-Ends Method*

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:

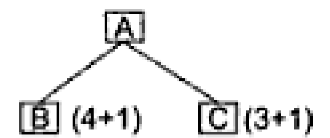
Start

1	2	3
8	5	6
4	7	

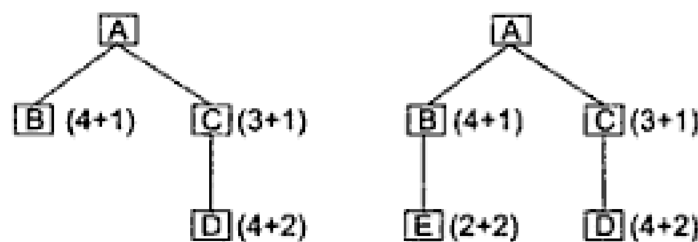
Goal

1	2	3
4	5	6
7	8	

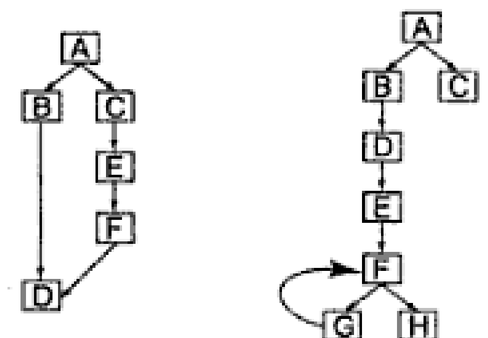
4. Describe the behavior of a revised version of the steepest ascent hill climbing algorithm in which step 2(c) is replaced by "set current state to best successor."
5. Suppose that the first step of the operation of the best-first search algorithm results in the following situation ($a + b$ means that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



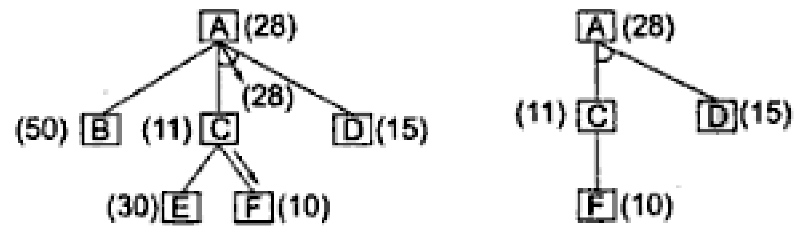
   (a) What node will be expanded at the next step?
   (b) Can we guarantee that the best solution will be found?
6. Why must the A* algorithm work properly on graphs containing cycles? Cycles could be prevented if when a new path is generated to an existing node, that path were simply thrown away if it 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



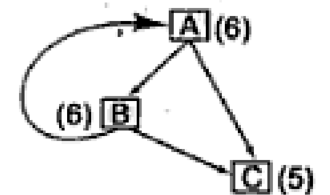
Copyrighted material

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?

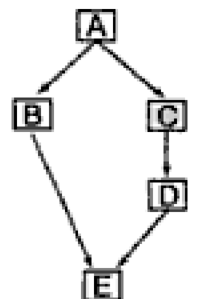
7. Formalize the Graceful Decay of Admissibility Corollary and prove that it is true of the A* algorithm.
8. In step 2(*a*) of the AO* algorithm, a random state at the end of the current best path is chosen for expansion. But there are heuristics that can be used to influence this choice. For example, 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 (includin^a each agent's behaviors and current goals) are described.

Copyrighted material

- The *symbol level,* at which representations of objects at the knowledge level are defined in terms of symbols that can be manipulated by programs.

See Newell [1982] for a detailed exposition of this view in the context of agents and their goals and behaviors. In the rest of our discussion here, we will follow a model more like the one shown in Fig. 4.1. Rather than thinking of one level on top of another, we will focus on facts, on representations, and on the two-way mappings that must exist between them. We will call these links *representation mappings*. The forward representation mapping maps from facts to representations. The backward representation mapping goes the other way, from representations to facts.

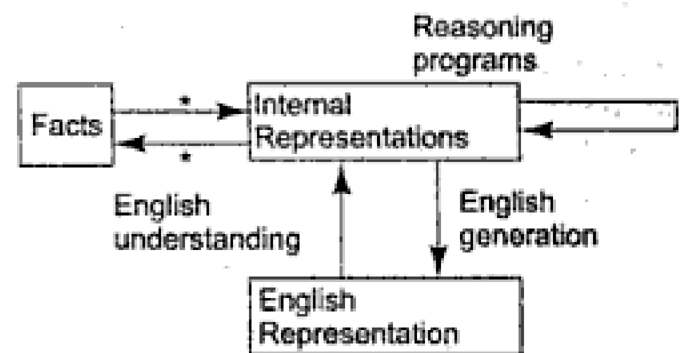


Fig. 4.1 *Mappings between Facts and Representations*

One representation of facts is so common that it deserves special mention: natural language (particularly English) sentences. Regardless of the representation for facts that we use in a program, we may also need to be concerned with an English representation of those facts in order to facilitate getting information into and out of the system. In this case, we must also have mapping functions from English sentences to the representation we are actually going to use and from it back to sentences. Figure 4.1 shows how these three kinds of objects relate to each other.

Let's look at a simple example using mathematical logic as the representational formalism. Consider the English sentence:

Spot is a dog.

The fact represented by that English sentence can also be represented in logic as:

$dog(Spot)$

Suppose that we also have a logical representation of the fact that all dogs have tails:

$\forall x : dog(x) \rightarrow hastail(x)$

Then, using the deductive mechanisms of logic, we may generate the new representation object:

$hastail(Spot)$

Using an appropriate backward mapping function, we could then generate the English sentence:

Spot has a tail.

Or we could make use of this representation of a new fact to cause us to take some appropriate action or to derive representations of additional facts.

It is important to keep in mind that usually the available mapping functions are not one-to-one. In fact, they are often not even functions but rather many-to-many relations. (In other words, each object in the domain may map to several elements in the range, and several elements in the domain may map to the same element of the range.) This is particularly true of the mappings involving English representations of facts. For example, the two sentences "All dogs have tails" and "Every dog has a tail" could both represent the same fact, namely,

Copyrighted material

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.

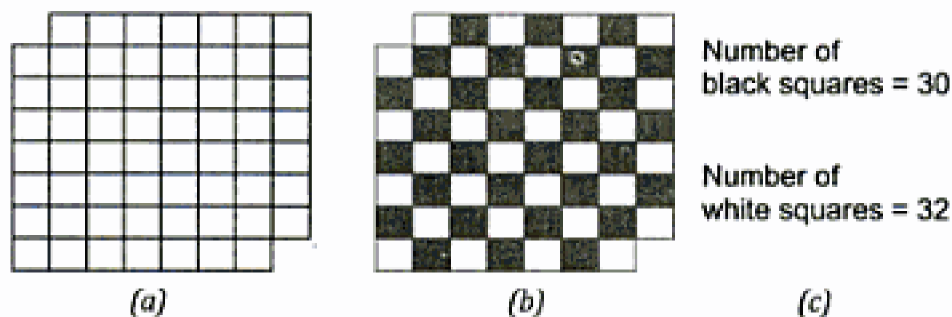


Fig. 4.2 *Three Representations of a Mutilated Checker board*

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.

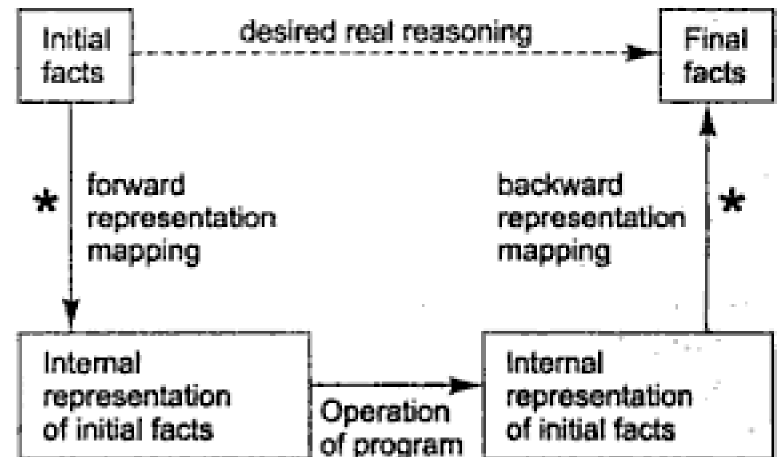


Fig. 4.3 *Representation of Facts*

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.

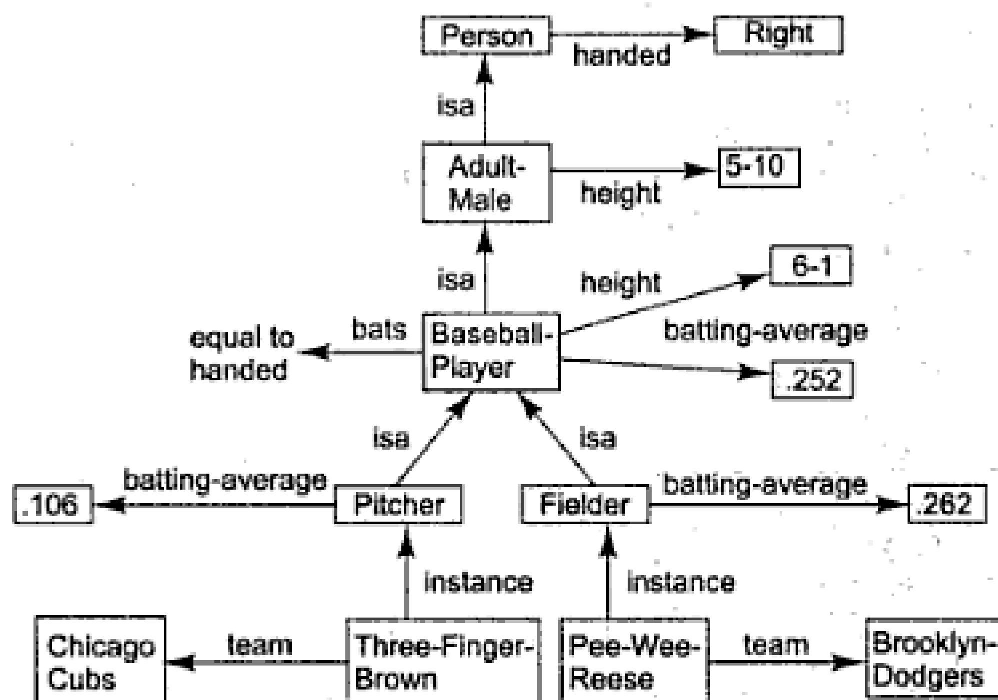


Figure 4.5 *Inheritable Knowledge*

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

"John punched Mary."

(a)

"Mary punched John."

(b)

Fig. 4.10 *Redundant Representations*

There are several arguments against the use of low-level primitives. One is that simple high-level facts may require a lot of storage when broken down into primitives. Much of that storage is really wasted since the low-level rendition of a particular high- level concept will appear many times, once for each time the high-level concept is referenced. For example, suppose that actions are being represented as combinations of a small set of primitive actions. Then the fact that John punched Mary might be represented as shown in Fig. 4.10(a). The representation says that there was physical contact between John's fist and Mary. The contact was caused by John propelling his fist toward Mary, and in order to do that John first went to where Mary was.[2] But suppose we also know that Mary punched John. Then we must also store the structure shown in Fig. 4.10(b). If, however, punching were represented simply as punching, then most of the detail of both structures could be omitted from the structures themselves. It could instead be stored just once in a common representation of the concept of punching.

A second but related problem is that if knowledge is initially presented to the system in a relatively high-level form, such as English, then substantial work must be done to reduce the knowledge into primitive form.

[2] The representation shown in this example is called *conceptual dependency* and is discussed in detail in Section 10.1.

Copyrighted material

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

(We could, of course, have decided instead to substitute x for y, since they are both just dummy variable names. The algorithm will simply pick one of these two substitutions.) But now, if we simply continue and match x and z, we produce tne substitution z/x. But we cannot substitute bothy and z for x, so we have not produced a consistent substitution.

What we need to do after finding the first substitutiony/x is to make that substitution throughout the literals, giving

$P(y, y)$
$P(y, z)$

Now we can attempt to unify arguments y and z, which succeeds with the substitution z/y. The entire unification process has now succeeded with a substitution that is the composition of the two substitutions we found. We write the composition as

$(z/y)(y/x)$

following standard notation for function composition. In general, the substitution $(a_1/a_2, a_3/a_4, ...)(b_1/b_2, b_3/b_4,...)$... means to apply all the substitutions of the right- most list, then take the result and apply all the ones of the next list, and so forth, until all substitutions have been applied.

The object of the unification procedure is to discover at least one substitution that causes two literals to match. Usually, if there is one such substitution there are many. ' For example, the literals

hate(x, y)
hate$(Marcus, z)$

could be unified with any of the following substitutions:

$(Marcus/x, z/y)$
$(Marcus/x, y/z)$
$(Marcus/x, Caesar/y, Caesar/z)$
$(Marcus/x, Polonius/y, Polonius/z)$

The first two of these are equivalent except for lexical variation. But the second two, although they produce a match, also produce a substitution that is more restrictive than absolutely necessary for the match. Because the final substitution produced by the unification process will be used by the resolution procedure, it is useful to generate the most general unifier possible. The algorithm shown below will do that.

Having explained the operation of the unification algorithm, we can now state it concisely. We describe a procedure Unify(Ll, L2), which returns as its value a list representing the composition of the substitutions that were performed during the match. The empty list, NIL, indicates that a match was fpund without any substitutions. The list consisting of the single value FAIL indicates that the unification procedure failed.

Algorithm: Unify(Ll, L2)

1. If $L1$ or $L2$ are both variables or constants, then:
   (a) If $L1$ and $L2$ are identical, then return NIL.
   (b) Else if $L1$ is a variable, then if $L1$ occurs in $L2$ then return {FAIL}, else return $(L2/L1)$.
   (c) Else if $L2$ is a variable then if $L2$ occurs in $L1$ then return {FAIL}, else return $(L1/L2)$.
   (d) Else return {FAIL}.

Copyrighted material

2. If the initial predicate symbols in $L1$ and $L2$ are not identical, then return {FAIL}.
3. If $L1$ and $L2$ have a different number of arguments, then return {FAIL}.
4. Set *SUBST* to NIL. (At the end of this procedure, *SUBST* will contain all the substitutions used to unify $L1$ and $L2$.)
5. For $i \leftarrow 1$ to number of arguments in $L1$:
   (a) Call Unify with the ith argument of $L1$ and the ith argument of $L2$, putting result in S.
   (b) If S contains FAIL then return {FAIL}.
   (c) If S is not equal to NIL then:
      (i) Apply S to the remainder of both $L1$ and $L2$.
      (ii) *SUBST* := APPEND(S, *SUBST*).
6. Return *SUBST*.

The only part of this algorithm that we have not yet discussed is the check in steps 1(*b*) and 1(*c*) to make sure that an expression involving a given variable is not unified with that variable. Suppose we were attempting to unify the expressions

$$f(x,x)$$
$$f(g(x),g(x))$$

If we accepted $g(x)$ as a substitution for x, then we would have to substitute it for x in the remainder of the expressions. But this leads to infinite recursion since it will never be possible to eliminate x.

Unification has deep mathematical roots and is a useful operation in many AI programs, for example, theorem provers and natural language parsers. As a result, efficient data structures and algorithms for unification have been developed. For an introduction to these techniques and applications, see Knight [1989].

5.4.5 Resolution in Predicate Logic

We now have an easy way of determining that two literals are contradictory—they are if one of them can be unified with the negation of the other. So, for example, $man(x)$ and $\neg man(Spot)$ are contradictory, since $man(x)$ and $man(Spot)$ can be unified. This corresponds to the intuition that says that $man(x)$ cannot be true for all x if there is known to be some x, say Spot, for which $man(x)$ is false. Thus in order to use resolution for expressions in the predicate logic, we use the unification algorithm to locate pairs of literals that cancel out.

We also need to use the unifier produced by the unification algorithm to generate the resolvent clause. For example, suppose we want to resolve two clauses:

1. $man(Marcus)$
2. $\neg man(x_1) \vee mortal(x_1)$

The literal $man(Marcus)$ can be unified with the literal $man(x_1)$ with the substitution $Marcus/x_1$, telling us that for $x_1 = Marcus$, $\neg man(Marcus)$ is false. But we cannot simply cancel out the two *man* literals as we did in propositional logic and generate the resolvent $mortal(x_1)$. Clause 2 says that for a given x_1, either $\neg man(x_1)$ or $mortal(x_1)$. So for it to be true, we can now conclude only that $mortal(Marcus)$ must be true. It is not necessary that $mortal(x_1)$ be true for all x_1, since for some values of x_1, $\neg man(x_1)$ might be true, making $mortal(x_1)$ irrelevant to the truth of the complete clause. So the resolvent generated by clauses 1 and 2 must be $mortal(Marcus)$, which we get by applying the result of the unification process to.the resolvent. The resolution process can then proceed from there to discover whether $mortal(Marcus)$ leads to a contradiction with other available clauses.

This example illustrates the importance of standardizing variables apart during the process of converting expressions to clause form. Given that that standardization has bee'n done, it is easy to determine how the

Copyrighted material

unifier must be used to perform substitutions to create the resolvent. If two instances of the same variable occur, then they must be given identical substitutions.

We can now state the resolution algorithm for predicate logic as follows, assuming a set of given statements *F* and a statement to be proved *P*:

Algorithm: Resolution

1. Convert all the statements of *F* to clause form.
2. Negate *P* and convert the result to clause form. Add it to the set of clauses obttfHied in 1.
3. Repeat until either a contradiction is found, no progress can be made, or a prede- termined amount of effort has been expended.
   (a) Select two clauses. Call these the parent clauses.
   (b) Resolve them together. The resolvent will be the disjunction of all the literals of both parent clauses with appropriate substitutions performed and with the following exception: If there is one pair of literals *T1* and ¬*T2* such that one of the parent clauses contains *T2* and the other contains *T1* and if *T1* and *T2* are unifiable, then neither *T1* nor *T2* should appear in the resolvent. We call *T1* and *T2* *Complementary literals.* Use the substitution produced by the unification to create the resolvent. If there is more than one pair of complementary literals, only one pair should be omitted from the resolvent.
   (c) If the resolvent is the empty clause, then a contradiction has been found. If it is not, then add it to the set of clauses available to the procedure.

If the choice of clauses to resolve together at each step is made in certain systematic ways, then the resolution procedure will find a contradiction if one exists. However, it may take a very long time. There exist strategies for making the choice that can speed up the process considerably:

- Only resolve pairs of clauses that contain complementary literals, since only such resolutions produce new clauses that are harder to satisfy than their parents. To facilitate this, index clauses by the predicates they contain, combined with an indication of whether the predicate is negated. Then, given a particular clause, possible resolvents that contain a complementary occurrence of one of its predi- cates can be located directly.
- Eliminate certain clauses as soon as they are generated so that they cannot partic- ipate in later resolutions. Two kinds of clauses should be eliminated: tautologies (which can never be unsatisfied) and clauses that are subsumed by other clauses (i.e., they are easier ;o satisfy. For example, $P \vee Q$ is subsumed by *P*.)
- Whenever possible, resolve either with one of the clauses that is part of the statement we are trying to refute or with a clause generated by a resolution with such a clause. This is called the *set-of-support strategy* and corresponds to the intuition that the contradiction we are looking for must involve the statement we are trying to prove. Any other contradiction would say that the previously believed statements were inconsistent.
- Whenever possible, resolve with clauses that have a single literal. Such resolutions generate new clauses with fewer literals than the larger of their parent clauses and thus are probably closer to the goal of a resolvent with zero terms. This method is called the *unit-preference strategy.*

Let's now return to our discussion of Marcus and show how resolution can be used to prove new things about him. Let's first consider the set of statements introduced in Section 5.1. To use them in resolution proofs, we must convert them to clause form as described in Section 5.4.1. Figure 5.9(*a*) shows the results of that conversion. Figure 5.9(*b*) shows a resolution proof of the statement

hate(*Marcus*,*Caesar*)

Copyrighted material

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

Fig. 6.5 *Another Way to Describe Chess Moves*

All of this does not mean that indexing cannot be helpful even when the preconditions of rules are stated as fairly high-level predicates. In PROLOG and many theorem-proving systems, for example, rules are indexed by the predicates they contain, so all the rules that could be applicable to proving a particular fact can be accessed fairly quickly. In the chess example, rules can be indexed by pieces and their positions. Despite some limitations of this approach, indexing in some form is very important in the efficient operation of rule-based systems.

6.4.2 Matching with Variables

The problem of selecting applicable rules is made more difficult when preconditions are not stated as exact descriptions of particular situations but rather describe properties (of varying complexity) that the situations must have. It often turns out that discovering whether there is a match between a particular situation and the preconditions of a given rule must itself involve a significant search process.

If we want to match a single condition against a single element in a state description, then the unification procedure of Section 5.4.4 will suffice. However, in many rule-based systems, we need to compute the whole set of rules that match the current state description. Backward-chaining systems usually use depth-first backtracking to select individual rules, but forward-chaining systems generally employ sophisticated *conflict resolution strategies* to choose among the applicable rules.[3] While it is possible to apply .unification repeatedly over the cross product of preconditions and state description elements, it is more efficient to consider the *many-many* match problem, in which many rules are matched against many elements in the state description simultaneously.

One efficient many-many match algorithm is RETE, which gains efficiency from three major sources:

- The temporal nature of data. Rules usually do not alter the state description radically. Instead, a rule will typically add one or two elements, or perhaps delete one or two, but most of the state description remains the same. (Recall our discussion of this as part of our treatment of the frame problem in Section 4.4.) If a rule did not match in the previous cycle, it will most likely fail to apply in the current cycle. RETE maintains a network of rule conditions, and it uses changes in the state description to determine which new rules might apply (and which rules might no longer apply). Full matching is only pursued for candidates that could be affected by incoming or outgoing data.
- Structural similarity in rules. Different rules may share a large number of pre-conditions. For example, consider rules for identifying wild animals. One rule concludes *jaguar(x)* if *mammal(x), feline(x), carnivorous(x),* and *has-spots(x).* Another rule concludes *tiger(x)* and is identical to the first rule except that it replaces *has-spots* with *has-stripes.* If we match the two rules independently, we will repeat a lot of work unnecessarily. RETE stores the rules so that they share structures in memory; sets of conditions that appear in several rules are matched (at most) once per cycle.
- Persistence of variable binding consistency. While all the individual preconditions of a rule might be met, there may be variable binding conflicts that prevent the rule from firing. For example, suppose we know the facts *son(Mary, Joe)* and *son(Bill, Bob).* The individual preconditions of the rule

[3] Conflict resolution is discussed in the next section.

Copyrighted material

son(*x*, *y*) $\wedge$ *son*(*y*, *z*) $\rightarrow$ *grandparent*(*x*, *z*)

can be matched, but not in a manner that satisfies the constraint imposed by the variable *y*. Fortunately, it is not necessary to compute binding consistency from scratch every time a new condition is satisfied. RETE remembers its previous calculations and is able to merge new binding information efficiently.

For more details about the RETE match algorithm, see Forgy [1982]. Other matching algorithms (e.g., Miranker [1987] and Oflazer [1987]) take different standson how much time to spend on saving state information between cycles. They can be more or less efficient than RETE, depending on the types of rules written for the domain and on the degree of hardware parallelism available.

6.4.3 Complex and Approximate Matching

A more complex matching process is required when the preconditions of a rule specify required properties that are not stated explicitly in the description of the current state. In this case, a separate set of rules must be used to describe how some properties can be inferred from others.

An even more complex matching process is required if rules should be applied if their preconditions *approximately* match the current situation. This is often the case in situations involving physical descriptions of the world. For example, a speech- understanding program must contain rules that map from a description of a physical waveform to phones (instances of English phonemes, such as *p* or *d*). There is so much variability in the physical signal, as a result of background noise, differences in the way individuals speak, and so forth, that one can hope to find only an approximate match between the rule that describes an ideal sound and the input that describes an unideal world. Approximate matching is particularly difficult to deal with because as we increase the tolerance allowed in the match, we also increase the number of rules that will match, thus increasing the size of the main search process. But approximate matching is nevertheless superior to exact matching in situations such as speech understanding, where exact matching may often result in no rules being matched and the search process coming to a grinding halt. Although symbolic techniques for approximate matching exist, there is another, very different approach that can be used to solve this problem. We discuss it in detail in Chapter 18 where we describe connectionist systems (also called neural nets).

For some problems, almost all the action is in the matching of the rules to the problem state. Once that is done, so few rules apply that the remaining search is trivial. This was the case, for example, in ELIZA [Weizenbaum, 1966], an early AI program that simulated the behavior of a Rogerian therapist. A fragment of a dialogue between ELIZA and a user is shown in Figure 6.6. ELIZA's knowledge about both English and psychology was coded in a set of simple rules. Figure 6.7 shows some ELIZA-like rules.

Person:	Men are all alike.
ELIZA:	In what way?
Person:	They're always bugging us about something or other.
ELIZA:	Can you think of a specific example?
Person:	Well, my boyfriend made me come here.
ELIZA:	Your boyfriend made you come here.
Person:	He says I'm depressed much of the time.
ELIZA:	I am sorry to hear you are depressed.
Person:	It's true. I am unhappy.
ELIZA:	Do you think coming here will help you not to be unhappy?
Person:	I need some help, that much seems certain.
ELIZA:	What would it mean to you if you got some help?
Person:	Perhaps I could learn to get along with my mother.
ELIZA:	Tell me more about your family.

Fig. 6.6 *A Bit of a Dialogue with ELIZA*

Copyrighted material

(X me Y)	→ (X you Y)
(I remember X)	→ (Why do remember X just now?)
(My {family-member} is Y)	→ (Who else in your family is Y?)
(X {family-member} Y)	→ (Tell me more about your family)

Fig. 6.7 *Some ELIZA-like rules*

ELIZA operated by matching the left sides of the rules against the user's last sentence and using the appropriate right side to generate a response. For example, if the user typed "My brother is mean to me," ELIZA might respond, "Who else in your family is mean to you?" or "Tell me more about your family." The rules were indexed by keywords so only a few had actually to be matched against a particular sentence. Some of the rules had no left side, so the rule could apply anywhere. These rules were used if no other rules matched and they generated replies such as "Tell me more about that". Notice that the rules themselves cause a form of approximate matching to occur. The patterns ask about specific words in the user's sentence. They do not need to match entire sentences. Thus a great variety of sentences can be matched by a single rule, and the grammatical complexity of English is pretty much ignored. This accounts both for ELIZA's major strength, its ability to say something fairly reasonable almost all of the time, and its major weakness, the superficiality of its understanding and its ability to be led completely astray. Approximate matching can easily lead to both these results.

As if the matching process were not already complicated enough, recall the frame problem mentioned in Chapter 4. One way of dealing with the frame problem is to avoid storing entire state descriptions at each node but instead to store only the changes from the previous node. If this is done, the matching process will have to be modified to scan backward from a node through its predecessors, looking for the required objects.

6.4.4 Conflict Resolution

The result of the matching process is a list of rules whose antecedents have matched the current state description along with whatever variable bindings were generated by the matching process. It is the job of the search method to decide on the order in which rules will be applied. But sometimes it is useful to incorporate some of that decision making into the matching process. This phase of the matching process is then called *conflict resolution.*

There are three basic approaches to the problem of conflict resolution in a production system:

- Assign a preference based on the rule that matched.
- Assign a preference based on the objects that matched.
- Assign a preference based on the action that the matched rule would perform.

Preferences Based on Rules

There are two common ways of assigning a preference based on the rules themselves. The first, and simplest, is to consider the rules to have been specified in a particular order, such as the physical order in which they are presented to the system. Then priority is given to the rules in the order in which they appear. This is the scheme used in PROLOG.

The other common rule-directed preference scheme is to give priority to special case rules over rules that are more general. We ran across this in Chapter 2, in the case of the water jug problem of Fig. 2.3. Recall that rules 11 and 12 were special cases of rules 9 and 5, respectively. The purpose of such specific rules is to allow for the kind of knowledge that expert problem solvers use when they solve problems directly, without search. If we consider all rules that match, then the addition of such special-purpose rules will increase the size of the search rather than decrease it. In order to prevent that, we build the matcher so that it rejects rules that are more general than other rules that also match. How can the matcher decide that one rule is more general than another? There are a few easy ways:

Copyrighted material

Our beliefs (1) through (4) are inconsistent, so we must choose one for rejection. Which has the weakest evidence? The basis for (1) in the hotel register is good, since it is a fine old hotel. The basis for (2) is weaker, since Babbitt's brother-in-law might be lying. The basis for (3) is perhaps twofold: that there is no sign of burglary and that only Abbott, Babbitt, and Cabot seem to have stood to gain from the murder apart from burglary. This exclusion of burglary seems conclusive, but the other consideration does not; there could be some fourth beneficiary. For (4), finally, the basis is conclusive: the evidence from television. Thus (2) and (3) are the weak points. To resolve the inconsistency of (1) through (4) we should reject (2) or (3), thus either incriminating Babbitt or widening our net for some new suspect.

See also how the revision progresses downward. If we reject (2), we also revise our previous underlying belief, however tentative, that the brother-in-law was telling the truth and Babbitt was in Brooklyn. If instead we reject (3), we also revise our previous underlying belief that none but Abbott, Babbitt, and Cabot stood to gain from the murder apart from burglary.

Finally, a certain arbitrariness should be noted in the organization of this analysis. The inconsistent beliefs (1) through (4) were singled out, and then various further beliefs were accorded a subordinate status as underlying evidence: a belief about a hotel register, a belief about the prestige of the hotel, a belief about the television, a perhaps unwarranted belief about the veracity of the brother-in-law, and so on. We could instead have listed this full dozen of beliefs on an equal footing, appreciated that they were in contradiction, and proceeded to restore consistency by weeding them out in various ways. But the organization lightened our task. It focused our attention on four prominent beliefs among which to drop one, and then it ranged the other beliefs under these four as mere aids to choosing which of the four to drop.

The strategy illustrated would seem in general to be a good one: divide and conquer. When a set of beliefs has accumulated to the point of contradiction, find the smallest selection of them you can that still involves contradiction; for instance, (1) through (4). For we can be sure that we are going to have to drop some of the beliefs in that subset, whatever else we do. In reviewing and comparing the evidence for the beliefs in the subset, then, we will find ourselves led down in a rather systematic way to other beliefs of the set. Eventually we find ourselves dropping some of them too.

In probing the evidence, where do we stop? In probing the evidence for (1) through (4) we dredged up various underlying beliefs, but we could have probed further, seeking evidence in turn for them. In practice, the probing stops when we are satisfied how best to restore consistency: which ones to discard among the beliefs we have canvassed.

This story illustrates some of the problems posed by uncertain, fuzzy, and often changing knowledge. A variety of logical frameworks and computational methods have been proposed for handling such problems. In this chapter and the next, we discuss two approaches:

- Nonmonotonic reasoning, in which the axioms and/or the rules of inference are extended to make it possible to reason with incomplete information. These systems preserve, however, the property that, at any given moment, a statement is either believed to be true, believed to be false, or not believed to be either.
- Statistical reasoning, in which the representation is extended to allow some kind of numeric measure of certainty (rather than simply TRUE or FALSE) to be associated with each statement.

Other approaches to these issues have also been proposed and used in systems. For example, it is sometimes the case that there is not a single knowledge base that captures the beliefs of all the agents involved in solving a problem. This would happen in our murder scenario if we were to attempt to model the reasoning of Abbott, Babbitt, and Cabot, as well as that of the police investigator. To be able to do this reasoning, we would require a technique for maintaining several parallel *belief spaces,* each of which would correspond to the beliefs of one agent. Such techniques are complicated by the fact that the belief spaces of the various agents, although

Copyrighted material

not identical, are sufficiently similar that it is unacceptably inefficient to represent them as completely separate knowledge bases. In Section 15.4.2 we return briefly to this issue. Meanwhile, in the rest of this chapter, we describe techniques for nonmonotonic reasoning.

Conventiotnal reasoning systems, such first-order predicate logic, are designed to work with information that has three important properties:

- It is complete with respect to the domain of interest. In other words, all the facts that are necessary to solve a problem are present in the system or can be derived from those that are by the conventional rules of first-order logic.
- It is consistent.
- The only way it can change is that new facts can be added as they become available. If these new facts are consistent with all the other facts that have already been asserted, then nothing will ever be retracted from the set of facts that are known to be true. This property is called *monotonicity.*

Unfortunately, if any of these properties is not satisfied, conventional logic-based reasoning systems become inadequate. Nonmonotonic reasoning systems, on the other hand, are designed to be able to solve problems in which all of these properties may be missing.

In order to do this, we must address several key issues, including the following:

1. *How can the knowledge base be extended to allow inferences to be made on the basis of lack of knowledge as well as on the presence of it?* For example, we would like to be able to say things like, "If you have no reason to suspect that a particular person committed a crime, then assume he didn't," or "If you have no reason to believe that someone is not getting along with her relatives, then assume that the relatives will try to protect her." Specifically, we need to make clear the distinction between:
   - It is known that $\neg P$.
   - It is not known whether P.

   First-order predicate logic allows reasoning to be based on the first of these. We need an extended system that allows reasoning to be based on the second as well. In our new system, we call any inference that depends on the lack of some piece of knowledge a *nonmonotonic inference.*[1]

   Allowing such reasoning has a significant impact on a knowledge base. Nonmonotonic reasoning systems derive their name from the fact that because of inferences that depend on lack of knowledge, knowledge bases may not grow monotonically as new assertions are made. Adding a new assertion may invalidate an inference that depended on the absence of that assertion. First-order predicate logic systems, on the other hand, are monotonic in this respect. As new axioms are asserted, new wff's may become provable, but no old proofs ever become invalid.

   In other words, if some set of axioms T entails the truth of some statement w, then T combined with another set of axioms N also entails w. Because nonmonotonic reasoning does not share this property, it is also called *defeasible:* a nonmonotonic inference may be defeated (rendered invalid) by the addition of new information that violates assumptions that were made during the original reasoning process. It turns out, as we show below, that making this one change has a dramatic impact on the structure of the logical system itself. In particular, most of our ideas of what it means to find a proof will have to be reevaluated.
2. *How can the knowledge base be updated properly when a new fact is added to the system (or when an old one is removed)?* In particular, in nonmonotonic systems, since the addition of a fact can cause

[1] Recall that in Section 2.4, we also made a monotonic/nonmonotonic distinction. There the issue was classes of production systems. Although we are applying the distinction to different entities here, it is essentially the same distinction in both cases, since it distinguishes between systems that never shrink as a result of an action (monotonic ones) and ones that can (nonmonotonic ones).

Copyrighted material

previously discovered proofs to be become invalid, how can those proofs, and all the conclusions that depend on them be found? The usual solution to this problem is to keep track of proofs, which are often called *justifications.* This makes it possible to find all the justifications that depended on the absence of the new fact, and those proofs can be marked as invalid. Interestingly, such a recording mechanism also makes it possible to support conventional, monotonic reasoning in the case where axioms must occasionally be retracted to reflect changes in the world that is being modeled. For example, it may be the case that Abbott is in town this week and so is available to testify, but if we wait until next week, he may be out of town. As a result, when we discuss techniques for maintaining valid sets of justifications, we talk both about nonmonotonic reasoning and about monotonic reasoning in a changing world.

3. *How can knowledge be used to help resolve conflicts when there are several in consistent nonmonotonic inferences that could be drawn?* It turns out that when inferences can be based on the lack of knowledge as well as on its presence, contradictions are much more likely to occur than they were in conventional logical systems in which the only possible contradictions were those that depended on facts that were explicitly asserted to be true. In particular, in nonmonotonic systems, there are often portions of the knowledge base that are locally consistent but mutually (globally) inconsistent. As we show below, many techniques for reasoning nonmonotonically are able to define the alternatives that could be believed, but most of them provide no way to choose among the options when not all of them can be believed at once.

To do this, we require additional methods for resolving such conflicts in ways that are most appropriate for the particular problem that is being solved. For example, as soon as we conclude that Abbott, Babbitt, and Cabot all claim that they didn't commit a crime, yet we conclude that one of them must have since there's no one else who is believed to have had a motive, we have a contradiction, which we want to resolve in some particular way based on other knowledge that we have. In this case, for example, we choose to resolve the conflict by finding the person with the weakest alibi and believing that he committed the crime (which involves believing other things, such as that the chosen suspect lied).

The rest of this chapter is divided into five parts. In the first, we present several logical formalisms that provide mechanisms for performing nonmonotonic reasoning. In the last four, we discuss approaches to the implementation of such reasoning in problem-solving programs. For more detailed descriptions of many of these systems, see the papers in Ginsberg [1987].

7.2 LOGICS FOR NONMONOTONIC REASONING

Because monotonicity is fundamental to the definition of first-order predicate logic, we are forced to find some alternative to support nonmonotonic reasoning. In this section, we look at several formal approaches to doing this. We examine several because no single formalism with all the desired properties has yet emerged (although there are some attempts, e.g., Shoham [1987] and Konolige [1987], to present a unifying framework for these several theories). In particular, we would like to find a formalism that does all of the following things:

- Defines the set of possible worlds that could exist given the facts that we do have. More precisely, we will define an *interpretation* of a set of wff's to be a domain (a set of objects) D, together with a function that assigns: to each predicate, a relation (of corresponding arity); to each n-ary function, an operator that maps from D^n into D; and to each constant, an element of D. A *model* of a set of wff's is an interpretation that satisfies them. Now we can be more precise about this requirement. We require a mechanism for defining the set of models of any set of wff's we are given.
- Provides a way to say that we prefer to believe in some models rather than others.

Copyrighted material

have a contradiction if we do not have at least one murder suspect. Thus a contradiction might have the justification shown in Fig. 7.10, where the node *Other Suspects* means that there are suspects other than Abbott, Babbitt, and Cabot. This is one way of explicitly representing an instance of the closed world assumption. Later, if we discover a long-lost relative, this will provide a valid justification for *Other Suspects*. But for now, it has none and must be labeled OUT. Fortunately, even though Abbott and Babbitt are not suspects, *Suspect Cabot* is labeled IN, invalidating the justification for the contradiction. While the contradiction is labeled OUT, there is no contradiction to resolve.

Now we learn that Cabot was seen on television attending the ski tournament. Adding this to the dependency network first illustrates the fact that nodes can have more than one justification as shown in Fig. 7.11. Not only does Cabot say he was at the ski slopes, but he was seen there on television, and we have no reason to believe that this was an elaborate forgery. This new valid justification of *Alibi Cabot* causes it to be labeled IN (which also causes *Tells Truth Cabot* to come IN). This change in state propagates to *Suspect Cabot*, which goes OUT. Now we have a problem.

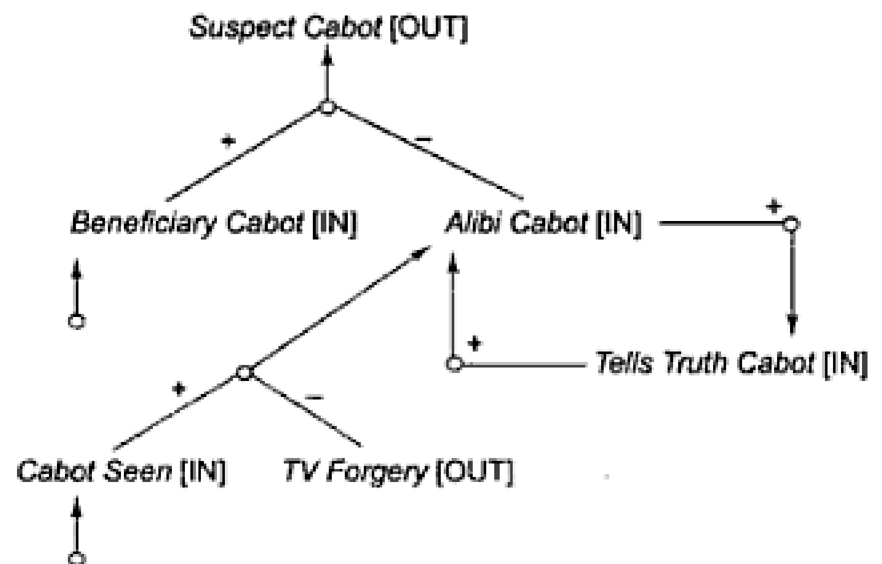


Fig. 7.11 *A Second Justification*

The justification for the contradiction is now valid and the contradiction is IN. The job of the TMS at this point is to determine how the contradiction can be made OUT again. In a TMS network, a node can be made OUT by causing all of its justifications to become invalid. Monotonic justifications cannot be made invalid without retracting explicit assertions that have been made to the network. Nonmonotonic justifications can, however, be invalidated by asserting some fact whose absence is required by the justification. We call assertions with nonmonotonic justifications *assumptions*. An assumption can be retracted by making IN some element of its justification's OUT-list (or recursively in some element of the OUT-list of the justification of some element in its IN-list). Unfortunately, there may be many such assumptions in a large dependency network. Fortunately, the network gives us a way to identify those that are relevant to the contradiction at hand. Dependency-directed backtracking algorithms, of the sort we described in Section 7.5.1, can use the dependency links to determine an AND/OR tree of assumptions that might be retracted and ways to retract them by justifying other beliefs.

In Fig. 7.10, we see that the contradiction itself is an assumption whenever its justification is valid. We might retract it by believing there were other suspects or by finding a way to believe again that either Abbott, Babbitt, or Cabot was a suspect. Each of the last three could be believed if we disbelieved their alibis, which in turn are assumptions. So if we believed that the hotel register was a forgery, that Babbitt's brother-in-law lied, or that the television pictures were faked, we would have a suspect again and the contradiction would go back OUT. So there are four things we might believe to resolve the contradiction. That is as far as DDB will take us. It reports there is an OR tree with four nodes. What should we do?

A TMS has no answer for this question. Early TMSs picked an answer at random. More recent architectures take the more reasonable position that this choice was a problem for the same problem-solving agent that created the dependencies in the first place. But suppose we do pick one. Suppose, in particular, that we choose to believe that Babbitt's brother-in-law lied. What should be the justification for that belief? If we believe it just because not believing it leads to a contradiction, then we should install a justification that should be valid only as long as it needs to be. If later we find another way that the contradiction can be labeled OUT, we will not want to continue in our abductive belief.

Copyrighted material

For instance, suppose that we believe that the brother-in-law lied, but later we discover that a long-lost relative, jilted by the family, was in town the day of the murder. We would no longer have to believe the brother-in-law lied just to avoid a contradiction. A TMS may also have algorithms to create such justifications, which we call abductive since they are created using abductive reasoning. If they have the property that they are not unnecessarily valid, they are said to be *complete.* Figure 7.12 shows a complete abductive justification for the belief that Babbitt's brother-in-law lied. If we come to believe that Abbott or Cabot is a suspect, or we find a long-lost relative, or we somehow come to believe that Babbitt's brother-in-law didn't really say Babbitt was at his house, then this justification for lying will become invalid.

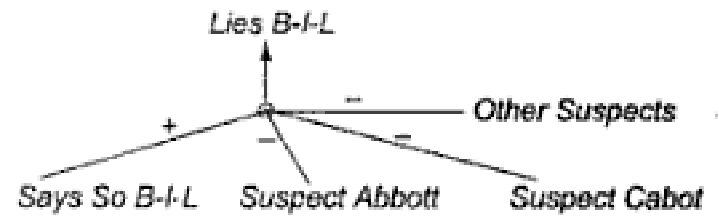


Fig. 7.12 *A Complete Abductive Justification*

At this point, we have described the key reasoning operations that are performed by a JTMS:

- consistent labeling
- contradiction resolution

We have also described a set of important reasoning operations that a JTMS does not perform, including:

- applying rules to derive conclusions
- creating justifications for the results of applying rules (although justifications are created as part of contradiction resolution)
- choosing among alternative ways of resolving a contradiction
- detecting contradictions

All of these operations must be performed by the problem-solving program that is using the JTMS. In the next section, we describe a slightly different kind of TMS, in which, although the first three of these operations must still be performed by the problem-solving system, the last can be performed by the TMS.

7.5.3 Logic-Based Truth Maintenance Systems

A *logic-based truth maintenance system* (LTMS) [McAllester, 1980] is very similar to a JTMS. It differs in one important way. In a JTMS, the nodes in the network are treated as atoms by the TMS, which assumes no relationships among them except the ones that are explicitly stated in the justifications. In particular, a JTMS has no problem simultaneously labeling both P and $\neg P$ IN. For example, we could have represented explicitly both *Lies B-I-L* and *Not Lies B-I-L* and labeled both of them IN. No contradiction will be detected automatically. In an LTMS, on the other hand, a contradiction would be asserted automatically in such a case. If we had constructed the ABC example in an LTMS system, we would not have created an explicit contradiction corresponding to the assertion that there was no suspect. Instead we would (replace the contradiction node by one that asserted something like *No Suspect.* Then we would assert *Suspect.* When *No Suspect* came IN, it would cause a contradiction to be asserted automatically.

7.6 IMPLEMENTATION: BREADTH-FIRST SEARCH

The *assumption-based truth maintenance system* (ATMS) [de Kleer, 1986] is an alternative way of implementing nonmonotonic reasoning. In both JTMS and LTMS systems, a single line of reasoning is pursued at a time, and dependency-directed backtracking occurs whenever it is necessary to change the system's assumptions. In an ATMS, alternative paths are maintained in parallel. Backtracking is avoided at the expense of maintaining multiple contexts, each of which corresponds to a set of consistent as-sumptions. As reasoning proceeds in an ATMS-based system, the universe of consistent contexts is pruned as contradictions are discovered. The remaining consistent contexts are used to label assertions, thus indicating the contexts in which each assertion has a valid justification. Assertions that do not have a valid justification in any consistent context can be

Copyrighted material

pruned from consideration by the problem solver. As the set of consistent contexts gets smaller, so too does the set of assertions that can consistently be believed by the problem solver. Essentially, an ATMS system works breadth-first, considering all possible contexts at once, while both JTMS and LTMS systems operate depth-first.

The ATMS, like the JTMS, is designed to be used in conjunction with a separate problem solver. The problem solver's job is to:

- Create nodes that correspond to assertions (both those that are given as axioms and those that are derived by the problem solver).
- Associate with each such node one or more justifications, each of which describes a reasoning chain that led to the node.
- Inform the ATMS of inconsistent contexts.

Notice that this is identical to the role of the problem solver that uses a JTMS, except that no explicit choices among paths to follow need be made as reasoning proceeds. Some decision may be necessary at the end, though, if more than one possible solution still has a consistent context.

The role of the ATMS system is then to:

- Propagate inconsistencies, thus ruling out contexts that include subcontexts (sets of assertions) that are known to be inconsistent.
- Label each problem solver node with the contexts in which it has a valid justification. This is done by combining contexts that correspond to the components of a justification. In particular, given a justification of the form

$$A1 \wedge A2 \wedge \ldots \wedge An \rightarrow C$$

  assign as a context for the node corresponding to C the intersection of the contexts corresponding to the nodes $A1$ through An.

Contexts get eliminated as a result of the problem-solver asserting inconsistencies and the ATMS propagating them. Nodes get created by the problem-solver to represent possible components of a problem solution. They may then get pruned from consideration if all their context labels get pruned. Thus a choice among possible solution components gradually evolves in a process very much like the constraint satisfaction procedure that we examined in Section 3.5.

One problem with this approach is that given a set of n assumptions, the number of possible contexts that may have to be considered is 2^n. Fortunately, in many problem-solving scenarios, most of them can be pruned without ever looking at them. Further, the ATMS exploits an efficient labeling system that makes it possible to encode a set of contexts as a single context that delimits the set. To see how both of these things work, it is necessary to think of the set of contexts that are defined by a set of assumptions as forming a lattice, as shown for a simple example with four assumptions in Fig. 7.13. Lines going upward indicate a subset relationship.

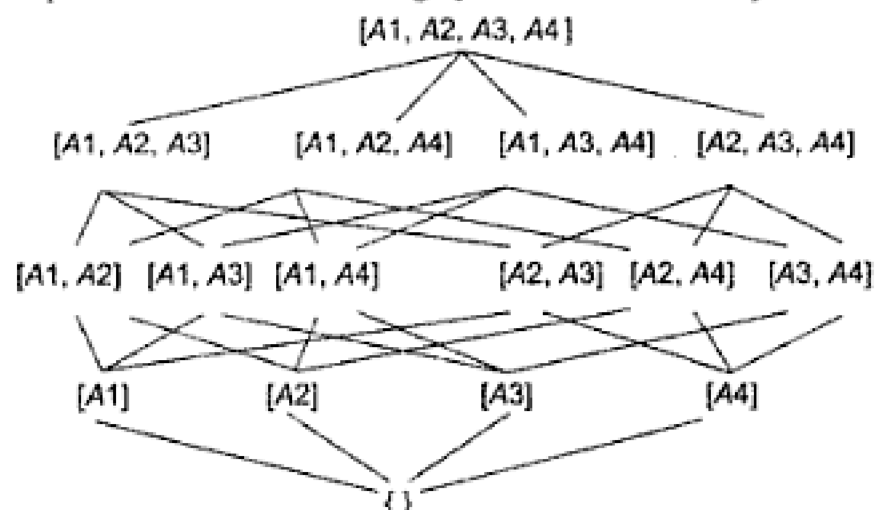


Fig. 7.13 *A Context Lattice*

The first thing this lattice does for us is to illustrate a simple mechanism by which contradictions (inconsistent contexts) can be propagated so that large parts of the space of 2^n contexts can be eliminated. Suppose that the

Copyrighted material

context labeled {$A2$, $A3$} is asserted to be Inconsistent. Then all contexts that include it (i.e., those that are above it) must also be inconsistent.

Now consider how a node can be labeled with all the contexts in which it has a valid justification. Suppose its justification depends on assumption $A1$. Then the context labeled {$A1$} and all the contexts that include it are acceptable. But this can be indicated just by saying {$A1$}. It is not necessary to enumerate its supersets. In general, each node will be labeled with the greatest lower bounds of the contexts in which it should be believed.

Clearly, it is important that this lattice not be built explicitly but only used as an implicit structure as the ATMS proceeds.

As an example of how an ATMS-based problem-solver works, let's return to the ABC Murder story. Again, our goal is to find a primary suspect. We need (at least) the following assumptions:

- $A1$. Hotel register was forged.
- $A2$. Hotel register was not forged.
- $A3$. Babbitt's brother-in-law lied.
- $A4$. Babbitt's brother-in-law did not lie.
- $A5$. Cabot lied.
- $A6$. Cabot did not lie.
- $A7$. Abbott, Babbitt, and Cabot are the only possible suspects.
- $A8$. Abbott, Babbitt, and Cabot are not the only suspects.

The problem-solver could then generate the nodes and associated justifications shown in the first two columns of Fig. 7.14. In the figure, the justification for a node that corresponds to a decision to make assumption N is shown as {N}. Justifications for nodes that correspond to the result of applying reasoning rules are shown as the rule involved. Then the ATMS can assign labels to the nodes as shown in the second two columns. The first shows the label that would be generated for each justification taken by itself. The second shows the label (possibly containing multiple contexts) that is actually assigned to the node given all its current justifications. These columns are identical in simple cases, but they may differ in more complex situations as we see for nodes 12, 13, and 14 of our example.

	Nodes	Justifications		Node Labels
[1]	Register was not forged	{A2}	{A2}	{A2}.
[2]	Abbott at hotel	[1] → [2]	{A2}	{A2}
[3]	B-I-L didn't lie	{4}	{A4}.	{A4}
[4]	Babbitt at B-I-L	[3] → [4]	{A4}	{A4}
[5]	Cabot didn't lie	{6}	{A6}	{A6}
[6]	Cabot at ski show	[5] → [6]	{A6}	{A6}
[7]	A, B, C only suspects	{A7}	{A7}	{A7}
[8]	Prime Suspect Abbott	[7] ∧ [13] ∧ [14] → [8]	{A7, A4, A6}	{A7, A4, A6}
[9]	Prime Suspect Babbitt	[7] ∧ [12] ∧ [14] → [9]	{A7, A2, A6}	{A7, A2, A6}
[10]	Prime Suspect Cabot	[7] ∧ [12] ∧ [13] → [10]	{A7, A2, A4}	{A7, A2, A4}
[11]	A, B,C not only suspects	{A8}	{A8}	{A8}
[12]	Not prime suspect Abbott	[2] → [12]	{A2}	{A2}, {A8}
		[11] → [12]	{A8}	
		[9] → [12]	{A7, A2, A6}	
		[10] → [12]	{A7, A2, A4}	
[13]	Not prime suspect Babbitt	[4] → [13]	{A4}	{A4}, {A8}
		[11] → [13]	{A8}	
		[8] → [13]	{A7, A4, A6}	
		[10] → [13]	{A7, A4, A2}	
[14]	Not prime suspect Cabot	[6] → [14]	{A6}	{A6}, {A8}
		[11] → [14]	{A8}	
		[8] → [14]	{A7, A4, A6}	
		[9] → [14]	{A7, A2, A6}	

Fig. 7.14 *Nodes and Their Justifications and Labels*

Copyrighted material

Unfortunately, in an arbitrarily complex world, the size of the set of joint probabilities that we require in order to compute this function grows as 2^n if there are n different propositions being considered. This makes using Bayes' theorem intractable for several reasons:

- The knowledge acquisition problem is insurmountable; too many probabilities have to be provided. In addition, there is substantial empirical evidence (e.g., Tversky and Kahneman [1974] and Kahneman *et al.* [1982]) that people are very poor probability estimators.
- The space that would be required to store all the probabilities is too large.
- The time required to compute the probabilities is too large.

Despite these problems, though, Bayesian statistics provide an attractive basis for an uncertain reasoning system. As a result, several mechanisms for exploiting its power while at the same time making it tractable have been developed. In the rest of this chapter, we explore three of these:

- Attaching certainty factors to rules
- Bayesian networks
- Dempster-Shafer theory

We also mention one very different numerical approach to uncertainty, fuzzy logic.

There has been an active, strident debate for many years on the question of whether pure Bayesian statistics are adequate as a basis for the development of reasoning programs. (See, for example, Cheeseman [1985] for arguments that it is and Buchanan and Shortliffe [1984] for arguments that it is not.) On the one hand, non-Bayesian approaches have been shown to work well for some kinds of applications (as we see below). On the other hand, there are clear limitations to all known techniques. In essence, the jury is still out. So we sidestep the issue as much as possible and simply describe a set of methods and their characteristics.

8.2 CERTAINTY FACTORS AND RULE-BASED SYSTEMS

In this section we describe one practical way of compromising on a pure Bayesian system. The approach we discuss was pioneered in the MYCIN system [Shortliffe, 1976; Buchanan and Shortliffe, 1984; Shortliffe and Buchanan, 1975], which attempts to recommend appropriate therapies for patients with bacterial infections. It interacts with the physician to acquire the clinical data it needs. MYCIN is an example of an *expert system*, since it performs a task normally done by a human expert. Here we concentrate on the use of probabilistic reasoning; Chapter 20 provides a broader view of expert systems.

MYCIN represents most of its diagnostic knowledge as a set of rules. Each rule has associated with it a *certainty factor*, which is a measure of the extent to which the evidence that is described by the antecedent of the rule supports the conclusion that is given in the rule's consequent. A typical MYCIN rule looks like:

```
If:   (1) the stain of the organism is gram-positive, and
      (2) the morphology of the organism is coccus, and
      (3) the growth conformation of the organism is clumps,
          then there is suggestive evidence (0.7) that
          the identity of the organism is staphylococcus.
```

This is the form in which the rules are stated to the user. They are actually represented internally in an easy-to-manipulate LISP list structure. The rule we just saw would be represented internally as

```
PREMISE:   ($AND    (SAME CNTXT GRAM GRAMPOS)
                    (SAME CNTXT MORPH COCCUS)
                    (SAME CNTXT CONFORM CLUMPS))
ACTION:             (CONCLUDE CNTXT IDENT STAPHYLOCOCCUS TALLY 0.7)
```

Copyrighted material

MYCIN uses these rules to reason backward to the clinical data available from its goal of finding significant disease-causing organisms. Once it finds the identities of such organisms, it then attempts to select a therapy by which the disease (s) may be treated. In order to understand how MYCIN exploits uncertain information, we need answers to two questions: "What do certainty factors mean?" and "How does MYCIN combine the estimates of certainty in each of its rules to produce a final estimate of the certainty of its conclusions?" A further question that we need to answer, given our observations about the intractability of pure Bayesian reasoning, is, "What compromises does the MYCIN technique make and what risks are associated with those compromises?" In the rest of this section we answer all these questions.

Let's start first with a simple answer to the first question (to which we return with a more detailed answer later). A certainty factor ($CF[h, e]$) is defined in terms of two components:

- $MB[h, e]$—a measure (between 0 and 1) of belief in hypothesis h given the evidence e. MB measures the extent to which the evidence supports the hypothesis. It is zero if the evidence fails to support the hypothesis.
- $MD[h,e]$—a measure (between 0 and 1) of disbelief in hypothesis h given the evidence e. MD measures the extent to which the evidence supports the negation of the hypothesis. It is zero if the evidence supports the hypothesis.

From these two measures, we can define the certainty factor as

$$CF[h, e] = MB[h, e] - MD[h, e]$$

Since any particular piece of evidence either supports or denies a hypothesis (but not both), and since each MYCIN rule corresponds to one piece of evidence (although it may be a compound piece of evidence), a single number suffices for each rule to define both the MB and MD and thus the CF.

The CF's of MYCIN's rules are provided by the experts who write the rules. They reflect the experts' assessments of the strength of the evidence in support of the hypothesis. As MYCIN reasons, however, these CF's need to be combined to reflect the operation of multiple pieces of evidence and multiple rules applied to a problem. Figure 8.1 illustrates three combination scenarios that we need to consider. In Fig. 8.1(a), several rules all provide evidence that relates to a single hypothesis. In Fig. 8.1(b), we need to consider our belief in a collection of several propositions taken together. In Fig. 8.1(c), the output of one rule provides the input to another.

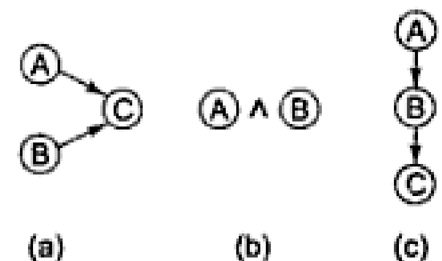


Fig. 8.1 *Combining Uncertain Rules*

What formulas should be used to perform these combinations? Before we answer that question, we need first to describe some properties that we would like the combining functions to satisfy:

- Since the order in which evidence is collected is arbitrary, the combining functions should be commutative and associative.
- Until certainty is reached, additional confirming evidence should increase MB (and similarly for disconfirming evidence and MD).
- If uncertain inferences are chained together, then the result should be less certain than either of the inferences alone.

Having accepted the desirability of these properties, let's first consider the scenario in Fig. 8.1(a), in which several pieces of evidence are combined to determine the CF of one hypothesis. The measures of belief and disbelief of a hypothesis given two observations s_1 and s_2 are computed from:

Copyrighted material

$$MB[h, s_1 \wedge s_2] = \begin{cases} 0 & \text{if } MD[h, s_1 \wedge s_2] = 1 \\ MB[h, s_1] + MB[h, s_2] \cdot (1 - MB[h, s_1]) & \text{otherwise} \end{cases}$$

$$MD[h, s_1 \wedge s_2] = \begin{cases} 0 & \text{if } MB[h, s_1 \wedge s_2] = 1 \\ MD[h, s_1] + MD[h, s_2] \cdot (1 - MD[h, s_1]) & \text{otherwise} \end{cases}$$

One way to state these formulas in English is that the measure of belief in h is 0 if h is disbelieved with certainty. Otherwise, the measure of belief in h given two observations is the measure of belief given only one observation plus some increment for the second observation. This increment is computed by first taking the difference between 1 (certainty) and the belief given only the first observation. This difference is the most that can be added by the second observation. The difference is then scaled by the belief in h given only the second observation. A corresponding explanation can be given, then, for the formula for computing disbelief. From *MB* and *MD*, *CF* can be computed. Notice that if several sources of corroborating evidence are pooled, the absolute value of *CF* will increase. If conflicting evidence is introduced, the absolute value of *CF* will decrease.

A simple example shows how these functions operate. Suppose we make an initial observation that confirms our belief in h with $MB = 0.3$. Then $MD[h,s_1] = 0$ and $CF[h, s_1] = 0.3$. Now we make a second observation, which also confirms h, with $MB[h,s_2] = 0.2$. Now:

$$\begin{aligned} MB[h,s_1 \wedge s_2] &= 0.3 + 0.2 \cdot 0.7 \\ &= 0.44 \\ MD[h,s_1 \wedge s_2] &= 0.0 \\ CF[h,s_1 \wedge s_2] &= 0.44 \end{aligned}$$

You can see from this example how slight confirmatory evidence can accumulate to produce increasingly larger certainty factors.

Next let's consider the scenario of Fig. 8.1(b), in which we need to compute the certainty factor of a combination of hypotheses. In particular, this is necessary when we need to know the certainty factor of a rule antecedent that contains several clauses (as, for example, in the staphylococcus rule given above). The combination certainty factor can be computed from its *MB* and *MD*. The formulas MYCIN uses for the *MB* of the conjunction and the disjunction of two hypotheses are:

$$MB[h_1 \wedge h_2, e] = \min(MB[h_1, e], MB[h_2, e])$$

$$MB[h_1 \wedge h_2, e] = max(MB[h_1,e], MB[h_2,e])$$

MD can be computed analogously.

Finally, we need to consider the scenario in Fig. 8.1(c), in which rules are chained together with the result that the uncertain outcome of one rule must provide the input to another. Our solution to this problem will also handle the case in which we must assign a measure of uncertainty to initial inputs. This could easily happen in situations where the evidence is the outcome of an experiment or a laboratory test whose results are not completely accurate. In such a case, the certainty factor of the hypothesis must take into account both the strength with which the evidence suggests the hypothesis and the level of confidence in the evidence. MYCIN provides a chaining rule that is defined as follows. Let $MB'[h, s]$ be the measure of belief in h given that we are absolutely sure of the validity of s. Let e be the evidence that led us to believe in s (for example, the actual readings of the laboratory instruments or the results of applying other rules). Then:

Copyrighted material

9.1.5 The Evolution into Frames

The idea of a semantic net started out simply as a way to represent labeled connections among entities. But, as we have just seen, as we expand the range of problem-solving tasks that the representation must support, the representation itself necessarily begins to become more complex. In particular, it becomes useful to assign more structure to nodes as well as to links. Although there is no clear distinction between a semantic net and a frame system, the more structure the system has, the more likely it is to be termed a frame system. In the next section we continue our discussion of structured slot-and-filler representations by describing some of the most important capabilities that frame systems offer.

9.2 FRAMES

A frame is a collection of attributes (usually called slots) and associated values (and possibly constraints on values) that describe some entity in the world. Sometimes a frame describes an entity in some absolute sense; sometimes it represents the entity from a particular point of view (as it did in the vision system proposal [Minsky, 1975] in which the term *frame* was first introduced). A single frame taken alone is rarely useful. Instead, we build frame systems out of collections of frames that are connected to each other by virtue of the fact that the value of an attribute of one frame may be another frame. In the rest of this section, we expand on this simple definition and explore ways that frame systems can be used to encode knowledge and support reasoning

9.2.1 Frames as Sets and Instances

The Set theory provides a good basis for understanding frame systems. Although not all frame systems are defined this way, we do so here. In this view, each frame represents either a class (a set) or an instance (an element of a class). To see how this works, consider the frame system shown in Fig. 9.5, which is a slightly modified form of the network we showed in Fig. 9.5. In this example, the frames *Person, Adult-Male, ML-Baseball-Player* (corresponding to major league baseball players), *Pitcher,* and *ML-Baseball-Team* (for major league baseball team) are all classes. The frames *Pee-Wee-Reese* and *Brooklyn-Dodgers* are instances.

The *isa* relation that we have been using without a precise definition is in fact the *subset* relation. The set of adult males is a subset of the set of people. The set of major league baseball players is a subset of the set of adult males, and so forth. Our *instance* relation corresponds to the relation *element-of.* Pee Wee Reese is an element of the set of fielders. Thus he is also an element of all of the supersets of fielders, including major league baseball players and people. The transitivity of *isa* that we have taken for granted in our description of property inheritance follows directly from the transitivity of the subset relation.

Both the *isa* and *instance* relations have inverse attributes, which we call *subclasses* and *all-instances.* We do not bother to write them explicitly in our examples unless we need to refer to them. We assume that the frame system maintains them automatically, either explicitly or by computing them if necessary.

Because a class represents a set, there are two kinds of attributes that can be associated with it. There are attributes about the set itself, and there are attributes that are to be inherited by each element of the set. We indicate the difference between these two by prefixing the latter with an asterisk (*). For example, consider the class *ML-Baseball-Player.* We have shown only two properties of it as a set: It is a subset of the set of adult males. And it has cardinality 624 (i.e., there are 624 major league baseball players). We have listed five properties that all major league baseball players have (*height, bats, batting-average, team,* and *uniform-color*), and we have specified default values for the first three of them. By providing both kinds of slots, we allow a class both to define a set of objects and to describe a prototypical object of the set.

Sometimes, the distinction between a set and an individual instance may not seem clear. For example, the team *Brooklyn-Dodgers,* which we have described as an instance of the class of major league baseball teams,

Copyrighted material

could be thought of as a set of players. In fact, notice that the value of the slot *players* is a set. Suppose, instead, that we want to represent the Dodgers as a class instead of an instance. Then its instances would be the individual players. It cannot stay where it is in the *isa* hierarchy; it cannot be a subclass of *ML-Baseball-Team*, because if it were, then its elements, namely the players, would also, by the transitivity of subclass, be elements of *ML-Baseball-Team*, which is not what we want to say. We have to put it somewhere else in the *isa* hierarchy. For example, we could make it a subclass of major league baseball players. Then its elements, the players, are also elements of *ML-Baseball-Player*, *Adult-Male*, and *Person*. That is acceptable. But if we do that, we lose the ability to inherit properties of the Dodgers from general information about baseball teams. We can still inherit attributes for the elements of the team, but we cannot inherit properties of the team as a whole, i.e., of the set of players. For example, we might like to know what the default size of the team is,

```
Person
    isa :                   Mammal
    cardinality :           6,000,000,000
    * handed :              Right
Adult-Male
    isa :                   Person
    cardinality :           2,000,000,000
    * height :              5-10
ML-Baseball-Player
    isa :                   Adult-Male
    cardinality :           624
    *height :               6-1
    * bats :                equal to handed
    * batting-average :     .252
    * team :
    * uniform-color :
Fielder
    isa :                   ML-Baseball-Player
    cardinality :           376
    *batting-average :      .262
Pee-Wee-Reese
    instance :              Fielder
    height :                5-10
    bats :                  Right
    batting-average :       .309
    team :                  Brooklyn-Dodgers
    uniform-color :         Blue
ML-Baseball-Team
    isa:                    Team
    cardinality :           26
    * team-size :           24
    * manager :
Brooklyn-Dodgers
    instance :              ML-Baseball-Team
    team-size :             24
    manager :               Leo-Durocher
    players :               {Pee-Wee-Reese,...}
```

Fig. 9.5 *A Simplified Frame System*

that it has a manager, and so on. The easiest way to allow for this is to go back to the idea of the Dodgers as an instance of *ML-Baseball-Team*, with the set of players given as a slot value.

But what we have encountered here is an example of a more general problem. A class is a set, and we want to be able to talk about properties that its elements possess. We want to use inheritance to infer those properties

Copyrighted material

from general knowledge about the set. But a class is also an entity in itself. It may possess properties that belong not to the individual instances but rather to the class as a whole. In the case of *Brooklyn-Dodgers,* such properties included team size and the existence of a manager. We may even want to inherit some of these properties from a more general kind of set. For example, the Dodgers can inherit a default team size from the set of all major league baseball teams. To support this, we need to view a class as two things simultaneously: a subset (*isa*) of a larger class that also contains its elements and an instance (*instance*) of a class of sets, from which it inherits its set-level properties.

To make this distinction clear, it is useful to distinguish between regular classes, whose elements are individual entities, and *metaclasses,* which are special classes whose elements are themselves classes. A class is now an element of (*instance*) some class (or classes) as well as a subclass (*isa*) of one or more classes. A class inherits properties from the class of which it is an instance, just as any instance does. In addition, a class passes inheritable properties down from its superclasses to its instances.

Let us consider an example. Figure 9.6 shows how we could represent teams as classes using this distinction. Figure 9.7 shows a graphic view of the same classes. The most basic metaclass is the class *Class.* It represents the set of all classes. All classes are instances of it, either directly or through one of its subclasses. In the example, *Team* is a subclass (subset) of *Class* and *ML-Baseball-Team* is a subclass of *Team.* The class *Class* introduces the attribute *cardinality,* which is to be inherited by all instances of *Class* (including itself). This makes sense since all the instances of *Class* are sets and all sets have a cardinality.

Class
  instance : *Class*
  isa : *Class*
  * *cardinality* :

Team
  instance : *Class*
  isa : *Class*
  cardinality : {the number of teams that exist}
  **team-size* : {each team has a size}

ML-Baseball-Team
  isa : *Mammal*
  instance : *Class*
  isa : *Team*
  cardi nality : 26 {the number of baseball teams that exist}
  * *team-size* : 24 {default 24 players on a team}
  * *manager* :

Brooklyn-Dodgers
  instance : *ML-Baseball-Team*
  isa : *ML-Baseball-Player*
  team-size : 24
  manager : *Leo-Durocher*
  * *uniform-color* : *Blue*

Pee-Wee-Reese
  instance : *Brooklyn-Dodgers*
  instance : *Fielder*
  uniform-color : *Blue*
  batting-average : .309

Fig. 9.6 *Representing the Class of All Teams as a Metaclass*

Copyrighted material

Consider a resolution theorem prover running with assertions in the ABox. A standard operation in resolution is determining when pairs of literals such as $f(x)$ and $\neg f(x)$ are inconsistent. Standard resolution requires that the literals be textually unifiable (except for the negation sign). KRYPTON extends the idea of textual inconsistency to *terminological* inconsistency in order to make the theorem prover more efficient. The TBox can tell that the two assertions *triangle*(x) and *rectangle*(x) are inconsistent and can thus be resolved against each other. The TBox can also determine the inconsistency of *triangle*(x) and $\neg$*polygon*(x); moreover, the two assertions $\neg$*rectangle*(x) and *polygon*(x) can be resolved against each other as long as we add to the resolvent the fact that x must have an angle which is not 90 degrees. If TBox computations are very efficient, then ABox proofs will be generated much faster than they would be in a pure logic framework.

11.3 OTHER REPRESENTATIONAL TECHNIQUES

In the last several chapters, we have described various techniques that can be used to represent knowledge. But our survey is by no means complete. There are other ways of representing knowledge; some of them are quite similar to the ones we have discussed and some are quite different. In this section we briefly discuss three additional methods: constraints, simulation models, and subsymbolic systems. Keep in mind throughout this discussion that it is not always the case that these various representational systems are mutually inconsistent. They often overlap, either in the way they use component representational mechanisms, the reasoning algorithms they support, or the problem-solving tasks for which they are appropriate.

11.3.1 Representing Knowledge as Constraints

Much of what we know about the world can be represented as sets of constraints. We talked in Section 3.5 about a very simple problem, cryptarithmetic, that can be described this way. But constraint-based representations are also useful in more complex problems. For example, we can describe an electronic circuit as a set of constraints that the states of various components of the circuit impose on the states of other components by virtue of being connected together. If the state of one of these components changes, we can propagate the effect of the change throughout the circuit by using the constraints. As a second example, consider the problem of interpreting visual scenes. We can write down a set of constraints that characterize the set of interpretations that can make sense in our physical world. For example, a single edge must be interpreted consistently, at both of its ends, as either a convex or a concave boundary. Finally, as we saw in Section 8.3, there are several kinds of relationships that can be represented as sets of constraints on the likelihoods that we can assign to collections of interdependent events.

In some sense, everything we write in any representational system is a constraint on the world models or problem solutions that we want our program to accept. For example, a wff [e.g., $\forall x : man(x) \rightarrow mortal(x)$] constrains the set of consistent models to those that do not include any man who is not mortal. But there is a very specific sense in which it is useful to talk about a specific class of techniques as constraint-based. Recall that in Section 3.5 we presented an algorithm for constraint satisfaction that was based on the notion of propagating constraints throughout a system until a final state was reached. This algorithm is particularly effective precisely when knowledge is represented in a way that makes it efficient to propagate constraints. This will be true whenever it is easy to locate the objects that a given object influences. This occurs when the objects in the system are represented as a network whose links correspond to constraints among the objects. We considered one example of this when we talked about Bayesian networks in Section 8.3. We consider other examples later in this book. For example, we return to the problem of simulating physical processes, such as electronic circuits, in Section 19.1. We present in Section 14.3 a constraint-propagation solution (known as the Waltz algorithm) to a simple vision problem. And in Section 15.5 we outline a view of natural language understanding as a constraint satisfaction task.

Copyrighted material

11.3.2 Models and Model-Based Reasoning

For many kinds of problem-solving tasks, it is necessary to model the behavior of some object or system. To diagnose faults in physical devices, such as electronic circuits or electric motors, it is necessary to model the behavior of both the correctly functioning device and some number of ill-functioning variants of it. To evaluate potential designs of such devices requires the same capability. Of course, as soon as we begin to think about modeling such complex entities, it becomes clear that the best we will be able to do is create an approximate model. There are various techniques that we can use to do that.

When we think about constructing a model of some entity in the world, the issue of what we mean by a model soon arises. To what extent should the structure of the model mirror the structure of the object being modeled? Some representational techniques tend to support models whose structure is very different from the structure of the objects being modeled. For example, in predicate logic we write wff's such as $\forall x : raven(x) \rightarrow black(x)$. In the real world, though, this single fact has no single realization; it is distributed across all known ravens. At the other extreme are representations, such as causal networks, in which the physical structure of the world is closely modeled in the structure of the representation.

There are arguments in favor of both ends of this spectrum (and many points in the middle). For example, if the knowledge structure closely matches the problem structure, then the frame problem may be easier to solve. Suppose, for example, that we have a robot-planning program and we want to know if we move a table into another room, what other objects also change location. A model that closely matches the structure of the world (as shown in Fig. 11.1 (a)) will make answering this question easy, while alternative representations (such as the one shown in Fig. 11.1 (b)) will not. For more on this issue, see Johnson-Laird [1983]. There are, however, arguments for representations whose structures do not closely model the world. For example, such representations typically do a better job of capturing generalizations and thus of making predictions about some kinds of novel situations.

```
(Livingroom1 :
    contains :
        (Table 1 :
            made-of : Wood
            has-on :      (Vase1 :
                                 made-of : Glass)
                          (Lamp1: ...))
        (Table2:
            has-on: (Vase2: ...)))
```

(a)

```
in(Table1, Livingroom)
made-of (Table1, Wood)
on(Vase1, Table1)
made-of (Vase1, Glass)
on(Vase2, Table2)
on(Lamp1, Table1)
```

(b)

Fig. 11.1 *Capturing Structure in Models*

11.3.3 Subsymbolic Systems

So far, all of the representations that we have discussed are symbolic, in the sense we defined in Section 1.2. There are alternative representations, many of them based on a neural model patterned after the human brain. These systems are often called neural nets or connectionist systems. We discuss such systems in Chapter 18.

Copyrighted material

11.4 SUMMARY OF THE ROLE OF KNOWLEDGE

In the last several chapters we have focused on the kinds of knowledge that may be useful to programs and on ways of representing and using that knowledge within programs. To sum up, for now, our treatment of knowledge within AI programs, let us return to a brief discussion of the two roles that knowledge can play in those programs.

- It may define the search space and the criteria for determining a solution to a problem.. We call this knowledge *essential knowledge.*
- It may improve the efficiency of a reasoning procedure by informing that procedure of the best places to look for a solution. We call that knowledge *heuristic knowledge.*

In formal tasks, such as theorem proving and game playing, there is only a small amount of essential knowledge and the need for a large amount of heuristic knowledge may be challenged by several brute force programs that perform quite successfully (e.g., the chess programs HITECH [Berliner and Ebeling, 1989] and DEEP THOUGHT [Anantharaman *et al.,* 1990]). The real knowledge challenge arises when we tackle naturally occurring problems, such as medical diagnosis, natural language processing, or engineering design. In those domains, substantial bodies of both essential and heuristic knowledge are absolutely necessary.

EXERCISES

1. Artificial intelligence systems employ a variety of formalisms for representing knowledge and reasoning with it. For each of the following sets of sentences, indicate the formalism that best facilitates the representation of the knowledge given in the statements in order to answer the question that is posed. Explain your choice briefly. Show how the statements would be encoded in the formalism you have selected. Then, show how the question could be answered.

    John likes fruit.
    Kumquats are fruit.
    People eat what they like.
    Does John eat kumquats?

    Assume that candy contains sugar unless you know
        specifically that it is dietetic.
    M&M's are candy.
    Diabetics should not eat sugar.
    Bill is a diabetic.
    Should Bill eat M&M's?

    Most people like candy.
    Most people who give parties like to serve food that.
        their guests like.
    Tom is giving a party.
    What might Tom like to serve?

    When you go to a movie theatre, you usually buy a ticket,
        hand the ticket to the ticket taker, and then go and
        find a seat.
    Sometimes you buy popcorn before going to your seat
    When the movie is over, you leave the theatre.
    John went to the movies.
    Did John buy a ticket?

Copyrighted material

depends critically on the correctness of that node's value, then the node is expanded one extra ply. This technique allows the search program to concentrate on tactical, forcing combinations. It employs a purely syntactic criterion, choosing interesting lines of play without recourse to any additional domain knowledge. The DEEP THOUGHT chess computer [Anantharaman *et al.*, 1990] has used singular extensions to great advantage, finding midgame mating combinations as long as thirty-seven moves, an impossible feat for fixed-depth minimax.

12.4.3 Using Book Moves

For complicated games taken as wholes, it is, of course, not feasible to select a move by simply looking up the current game configuration in a catalogue and extracting the correct move. The catalogue would be immense and no one knows how to construct it. But for some segments of some games, this approach is reasonable. In chess, for example, both opening sequences and endgame sequences are highly stylized. In these situations, the performance of a program can often be considerably enhanced if it is provided with a list of moves (called *book moves*) that should be made. The use of book moves in the opening sequences and endgames, combined with the use of the minimax search procedure for the midgame, provides a good example of the way that knowledge and search can be combined in a single program to produce more effective results than could either technique on its own.

12.4.4 Alternatives to Minimax

Even with the refinements above, minimax still has some problematic aspects. For instance, it relies heavily on the assumption that the opponent will always choose the optimal move. This assumption is acceptable in winning situations where a move that is guaranteed to be good for us can be found. But, as suggested in Berliner [1977], in a losing situation it might be better to take the risk that the opponent will make a mistake. Suppose we must choose between two moves, both of which, if the opponent plays perfectly, lead to situations that are very bad for us, but one is slightly less bad than the other. But further suppose that the less promising move could lead to a very good situation for us if the opponent makes a single mistake. Although the minimax procedure would choose the guaranteed bad move, we ought instead to choose the other one, which is probably slightly worse but possibly a lot better. A similar situation arises when one move appears to be only slightly more advantageous than another, assuming that the opponent plays perfectly. It might be better to choose the less advantageous move if it could lead to a significantly superior situation if the opponent makes a mistake. To make these decisions well, we must have access to a model of the individual opponent's playing style so that the likelihood of various mistakes can be estimated. But this is very hard to provide.

As a mechanism for propagating estimates of position strengths up the game tree, minimax stands on shaky theoretical grounds. Nau. [1980] and Pearl [1983] have demonstrated that for certain classes of game trees, e.g., uniform trees with random terminal values, the deeper the search, the *poorer* the result obtained by minimaxing. This "pathological" behavior of amplifying/error-prone heuristic estimates has not been observed in actual game-playing programs, however. It seems that game trees containing won positions and nonrandom distributions of heuristic estimates provide environments that are conducive to minimaxing.

12.5 ITERATIVE DEEPENING

A number of ideas for searching two-player game trees have led to new algorithms for single-agent heuristic search, of the type described in Chapter 3. One such idea is *iterative deepening*, originally used in a program called CHESS 4.5 [Slate and Atkin, 1977]. Rather than searching to a fixed depth in the game tree, CHESS 4.5 first searched only a single ply, applying its static evaluation function to the result of each of its possible moves. It then initiated a new minimax search, this time to a depth of two ply. This was followed by a three-

Copyrighted material

ply search, then a four-ply search, etc. The name "iterative deepening" derives from the fact that on each iteration, the tree is searched one level deeper. Figure 12.10 depicts this process.

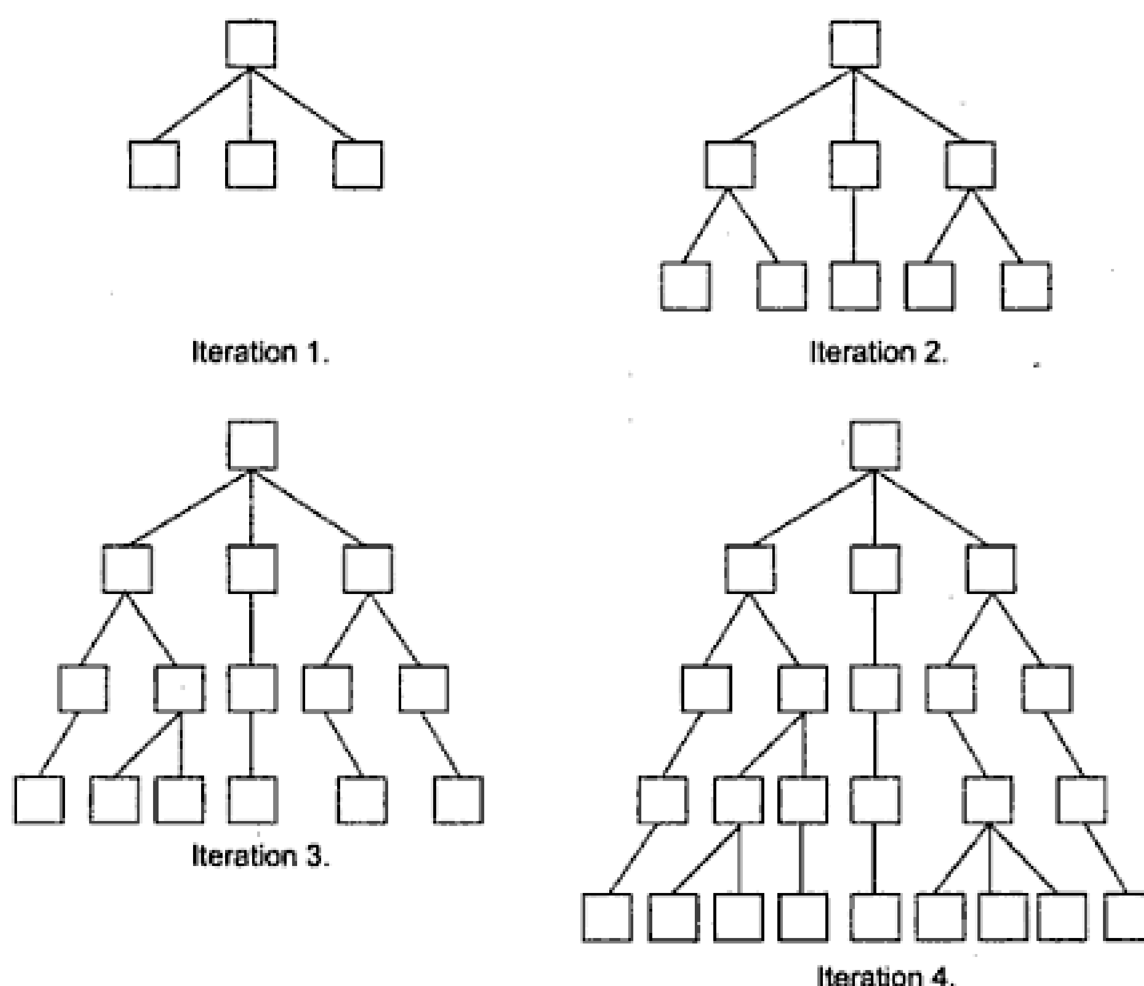


Fig. 12.10 *Iterative Deepening*

On the face of it, this process seems wasteful. Why should we be interested in any iteration except the final one? There are several reasons. First, game-playing programs are subject to time constraints. For example, a chess program may be required to complete all its moves within two hours. Since it is impossible to know in advance how long a fixed-depth tree search will take (because of variations in pruning efficiency and the need for selective search), a program may find itself running out of time. With iterative deepening, the current search can be aborted at any time and the best move found by the previous iteration can be played. Perhaps more importantly, previous iterations can provide invaluable move-ordering constraints. If one move was judged to be superior to its siblings in a previous iteration, it can be searched first in the next iteration. With effective ordering, the alpha-beta procedure can prune many more branches, and total search time can be decreased drastically. This allows more time for deeper iterations.

Years after CHESS 4.5's success with iterative deepening, it was noticed [Korf, 1985a] that the technique could also be applied effectively to single-agent search to solve problems like the 8-puzzle. In Section 2.2.1, we compared two types of uninformed search, depth-first search and breadth-first search. Depth-first search was efficient in terms of space but required some cutoff depth in order to force backtracking when a solution was not found. Breadth-first search was guaranteed to find the shortest solution path but required inordinate amounts of space because all leaf nodes had to be kept in memory. An algorithm called depth-first iterative deepening (DFID) combines the best aspects of depth-first and breadth-first search.

Copyrighted material

Algorithm: Depth-First Iterative Deepening

1. Set SEARCH-DEPTH = 1.
2. Conduct a depth-first search to a depth of SEARCH-DEPTH. If a solution path is found, then return it.
3. Otherwise, increment SEARCH-DEPTH by 1 and go to step 2.

Clearly, DFID will find the shortest solution path to the goal state. Moreover, the maximum amount of memory used by DFID is proportional to the number of nodes in that solution path. The only disturbing fact is that all iterations but the final one are essentially wasted. However, this is not a serious problem. The reason is that most of the activity during any given iteration occurs at the leaf-node level. Assuming a complete tree, we see that there are as many leaf nodes at level n as there are total nodes in levels 1 through n. Thus, the work expended during the nth iteration is roughly equal to the work expended during all previous iterations. This means that DFID is only slower than depth-first search by a constant factor. The problem with depth-first search is that there is no way to know in advance how deep the solution lies in the search space. DFID avoids the problem of choosing cutoffs without sacrificing efficiency, and, in fact, DFID is the optimal algorithm (in terms of space and time) for uninformed search.

But what about informed, heuristic search? Iterative deepening can also be used to improve the performance of the A* search algorithm [Korf, 1985a]. Since the major practical difficulty with A* is the large amount of memory it requires to maintain the search node lists, iterative deepening can be of considerable service.

Algorithm: Iterative-Deepening-A*

1. Set THRESHOLD = the heuristic evaluation of the start state.
2. Conduct a depth-first search, pruning any branch when its total cost function ($g + h'$) exceeds THRESHOLD.[4] If a solution path is found during the search, return it.
3. Otherwise, increment THRESHOLD by the minimum amount it was exceeded during the previous step, and then go to Step 2.

Like A*, Iterative-Deepening-A* (IDA*) is guaranteed to find an optimal solution, provided that h' is an admissible heuristic. Because of its depth-first search technique, IDA* is very efficient with respect to space. IDA* was the first heuristic search algorithm to find optimal solution paths for the 15-puzzle (a 4x4 version of the 8-puzzle) within reasonable time and space constraints.

12.6 REFERENCES ON SPECIFIC GAMES

In this chapter we have discussed search-based techniques for game playing. We discussed the basic minimax algorithm and then introduced a series of refinements to it. But even with these refinements, it is still difficult to build good programs to play difficult games. Every game, like every AI task, requires a careful combination of search and knowledge.

Chess

Research on computer chess actually predates the field we call artificial intelligence. Shannon [1950] was the first to propose a method for automating the game, and two early chess programs were written by Greenblatt *et al.* [1967] and Newell and Simon [1972].

Chess provides a well-defined laboratory for studying the trade-off between knowledge and search. The more knowledge a program has, the less searching it needs to do. On the other hand, the deeper the search, the less knowledge is required. Human chess players use a great deal of knowledge and very little search—they

[4] Recal g stands for the cost so far in reaching the current node, and h' stands for the heuristic estimate of the distance from the node to the goal.

Copyrighted material

lead nowhere. For example, if, in trying to satisfy goal A, the program eventually reduces its problem to the satisfaction of goal A as well as goals B and C, it has made little progress. It has produced a problem even harder than its original one, and the path leading to this problem should be abandoned.

Repairing an Almost Correct Solution

The kinds of techniques we are discussing are often useful in solving *nearly* decomposable problems. One good way of solving such problems is to assume that they are completely decomposable, proceed to solve the subproblems separately, and then check that when the subsolutions are combined, they do in fact yield a solution to the original problem. Of course, if they do, then nothing more need be done. If they do not, however, there are a variety of things that we can do. The simplest is just to throw out the solution, look for another one, and hope that it is better. Although this is simple, it may lead to a great deal of wasted effort.

A slightly better approach is to look at the situation that results when the sequence of operations corresponding to the proposed solution is executed and to compare that situation to the desired goal. In most cases, the difference between the two will be smaller than the difference between the initial state and the goal (assuming that the solution we found did some useful things). Now the problem-solving system can be called again and asked to find a way of eliminating this new difference. The first solution can then be combined with this second one to form a solution to the original problem.

An even better way to patch up an almost correct solution is to appeal to specific knowledge about what went wrong and then to apply a direct patch. For example, suppose that the reason that the proposed solution is inadequate is that one of its operators cannot be applied because at the point it should have been invoked, its preconditions were not satisfied. This might occur if the operator had two preconditions and the sequence of operations that makes the second one true undid the first one. But perhaps, if an attempt were made to satisfy the preconditions in the opposite order, this problem would not arise.

A still better way to patch up incomplete solutions is not really to patch them up at all but rather to leave them incompletely specified until the last possible moment. Then when as much information as possible is available, complete the specification in such a way that no conflicts arise. This approach can be thought of as a *least-commitment* strategy. It can be applied in a variety of ways. One is to defer deciding on the order in which operations will be performed. So, in our previous example, instead of arbitrarily choosing one order in which to satisfy a set of preconditions, we could leave the order unspecified until the very end. Then we would look at the effects of each of the subsolutions to determine the dependencies that exist among them. At that point, an ordering can be chosen.

13.4 GOAL STACK PLANNING

One of the earliest techniques to be developed for solving compound goals that may interact was the use of a goal stack. This was the approach used by STRIPS. In this method, the problem solver makes use of a single stack that contains both goals and operators that have been proposed to satisfy those goals. The problem solver also relies on a database that describes the current situation and a set of operators described as PRECONDITION, ADD, and DELETE lists. To see how this method works, let us carry it through for the simple example shown in Fig. 13.4.

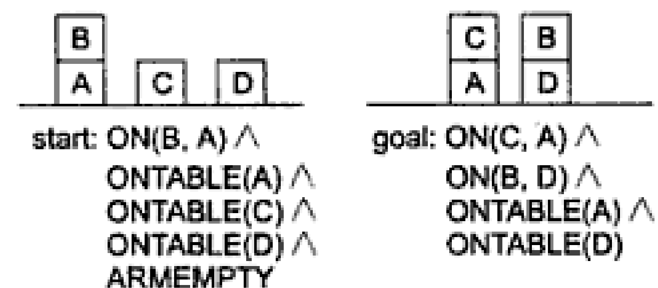


Fig. 13.4 *A Very Simple Blocks World Problem*

When we begin solving this problem, the goal stack is simply

ON(C, A) ∧ ON(B, D) ∧ ONTABLE(A) ∧ ONTABLE(D)

Copyrighted material

But we want to separate this problem into four subproblems, one for each component of the original goal. Two of the subproblems, ONTABLE(A) and ONTABLE(D), are already true in the initial state. So we will work on only the remaining two. Depending on the order in which we want to tackle the subproblems, there are two goal stacks that could be created as our first step, where each line represents one goal on the stack and OTAD is an abbreviation for ONTABLE(A) ∧ ONTABLE(D):

ON(C, A)	ON(B, D)
ON(B, D)	ON(C, A)
ON(C, A) ∧ ON(B, D) ∧ OTAD	ON(C, A) ∧ ON(B,D) ∧ OTAD
[1]	[2]

At each succeeding step of the problem-solving process, the top goal on the stack will be pursued. When a sequence of operators that satisfies it is found, that sequence is applied to the state description, yielding a new description. Next, the goal that is then at the top of the stack is explored and an attempt is made to satisfy it, starting from the situation that was produced as a result of satisfying the first goal. This process continues until the goal stack is empty. Then, as one last check, the original goal is compared to the final state derived from the application of the chosen operators. If any components of the goal are not satisfied in that state (which they might not be if they were achieved at one point and then undone later), then those unsolved parts of the goal are reinserted onto the stack and the process resumed.

To continue with the example we started above, let us assume that we choose first to explore alternative 1. Alternative 2 will also lead to a solution. In fact, it finds one so trivially that it is not very interesting. Exploring alternative 1, we first check to see whether ON(C, A) is true in the current state. Since it is not, we check for operators that could cause it to be true. Of the four operators we are considering, there is only one, STACK, and it would have to be called with C and A. So we place STACK(C, A) on the stack in place of ON(C, A), yielding

STACK(C, A)
ON(B, D)
ON(C, A) ∧ ON(B, D) ∧ OTAD

STACK(C, A) replaced ON(C, A) because after performing the STACK we are guaranteed that ON(C, A) will hold. But in order to apply STACK(C, A), its preconditions must hold, so we must establish them as subgoals. Again we must separate a compound goal

CLEAR(A) ∧ HOLDING(C)

into its components and choose an order in which to work on them. At this point, it is useful to exploit some heuristic knowledge. HOLDING(x) is very easy to achieve. At most, it is necessary to put down something else and then to pick up the desired object. But HOLDING is also very easy to undo. In order to do almost anything else, the robot will need to use the arm. So if we achieve HOLDING first and then try to do something else, we will most likely end up with HOLDING no longer true. So we exploit the heuristic that if HOLDING is one of several goals to be achieved at once, it should be tackled last. This produces the new goal stack

CLEAR(A)
HOLDING(C)
CLEAR(A) ∧ HOLDING(C)
STACK(C, A)
ON(B, D)
ON(C, A) ∧ ON(B, D) ∧ OTAD

Copyrighted material

This kind of heuristic information could be contained in the precondition list itself by stating the predicates in the order in which they should be achieved.

Next we check to see if CLEAR(A) is true. It is not. The only operator that could make it true is UNSTACK (B, A). So we will attempt to apply it. This produces the goal stack

ON(B, A)
CLEAR(B)
ARMEMPTY
ON(B, A) $\wedge$ CLEAR(B) $\wedge$ ARMEMPTY
UNSTACK(B, A)
HOLDING(C)
CLEAR(A) $\wedge$ HOLDING(C)
STACK(C, A)
ON(B, D)
ON(C, A) $\wedge$ ON(B, D) $\wedge$ OTAD

This time, when we compare the top element of the goal stack, ON(B, A), to the world model, we see that it is satisfied. So we pop it off and consider the next goal, CLEAR(B). It, too, is already true in the world model, although it was not stated explicitly as one of the initial predicates. But from the initial predicates and the blocks world axiom that says that any block with no blocks on it is clear, a theorem prover could derive CLEAR(B). So that goal, too, can be popped from the stack. The third precondition for UNSTACK(B, A) remains. It is ARMEMPTY, and it is also true in the current world model, so it can be popped off the stack. The next element on the stack is the combined goal representing all of the preconditions for UNSTACK(B, A). We check to make sure it is satisfied in the world model. It will be unless we undid one of its components in attempting to satisfy another. In this case, there is no problem and the combined goal can be popped from the stack.

Now the top element of the stack is the operator UNSTACK(B, A). We are now guaranteed that its preconditions are satisfied, so it can be applied to produce a new world model from which the rest of the problem-solving process can continue. This is done using the ADD and DELETE lists specified for UNSTACK. Meanwhile we record that UNSTACK(B, A) is the first operator of the proposed solution sequence. At this point, the database corresponding to the world model is

ONTABLE(A) $\wedge$ ONTABLE(C) $\wedge$ ONTABLE(D) $\wedge$
HOLDING(B) $\wedge$ CLEAR(A)

The goal stack now is

HOLDING(C)
CLEAR(A) $\wedge$ HOLDING(C)
STACK(C, A)
ON(B, D)
ON(C, A) $\wedge$ ON(B, D) $\wedge$ OTAD

We now attempt to satisfy the goal HOLDING(C). There are two operators that might make HOLDING(C) true: PICKUP(C) and UNSTACK(C, *x*), where *x* could be any block from which C could be unstacked.

Copyrighted material

Without looking ahead, we cannot tell which of these operators is appropriate, so we create two branches of the search tree, corresponding to the following goal stacks:

ONTABLE(C)	ON(C, x)
CLEAR(C)	CLEAR(C)
ARMEMPTY	ARMEMPTY
ONTABLE(C) ∧ CLEAR(C) ∧ ARMEMPTY	ON(C, x) ∧ CLEAR(C) ∧ ARMEMPTY
PICKUP(C)	**UNSTACK(C, x)**
CLEAR(A) ∧ HOLDING(C)	CLEAR(A) ∧ HOLDING(C)
STACK(C, A)	**STACK(C, A)**
ON(B, D)	ON(B, D)
ON(C, A) ∧ ON(B, D) ∧ OTAD	ON(C, A) ∧ ON(B, D) ∧ OTAD
[1]	[2]

Notice that for alternative 2, the goal stack now contains a variable x, which appears in three places. Although any block could be substituted for x, it is important that the same one be matched to each of the x's. Thus it is important that each time a variable is introduced into the goal stack, it be given a name distinct from any other variables already in the stack. And whenever a candidate object is chosen to match a variable, the binding must be recorded so that other occurrences of the same variable will be bound to the same object.

How should our program choose now between alternative land alternative 2? We can tell that picking up C (alternative 1) is better than unstacking it because it is not currently on anything. So to unstack it, we would first have to stack it. Although this could be done, it would be a waste of effort. But how could a program know that? Suppose we decided to pursue alternative 2 first. To satisfy ON(C, x), we would have to STACK C onto some block x. The goal stack would then be

CLEAR(x)
HOLDING(C)
CLEAR(x) ∧ HOLDING(C)
STACK(C, x)
CLEAR(C)
ARMEMPTY
ON(C, x) ∧ CLEAR(C) ∧ ARMEMPTY
UNSTACK(C, x)
CLEAR(A) ∧ HOLDING(C)
STACK(C, A)
ON(B, D)
ON(C, A) ∧ ON(B, D) ∧ OTAD

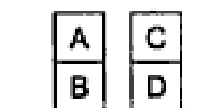
But now notice that one of the preconditions of STACK is HOLDING(C). This is what we were trying to achieve by applying UNSTACK, which required us to apply STACK so that the precondition ON(C, x) would be satisfied. So we are back to our original goal. In fact, we now have additional goals as well, since other predicates have also been added to the stack. At this point, this path can be terminated as unproductive. If, however, block C had been on another block in the current state, ON(C, x) would have been satisfied immediately with no need to do a STACK and this path would have led to a good solution.

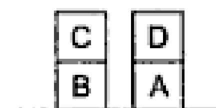
Copyrighted material

- Triangle Tables [Fikes *et al.*, 1972; Nilsson, 1980]—Provide a way of recording the goals that each operator is expected to satisfy as well as the goals that must be true for it to execute correctly. If something unexpected happens during the execution of a plan, the table provides the information required to patch the plan.
- Metaplanning [Stefik, 1981a]—A technique for reasoning not just about the problem being solved but also about the planning process itself.
- Macro-operators [Fikes and Nilsson, 1971]—Allow a planner to build new operators that represent commonly used sequences of operators. See Chapter 17 for more details.
- Case-Based Planning [Hammond, 1986]—Re-uses old plans to make new ones. We return to case-based planning in Chapter 19.

EXERCISES

1. Consider the following blocks world problem:





start: ON(C, B) ∧
ON(D, A) ∧
ONTABLE(B) ∧
ONTABLE(A) ∧
ARMEMPTY

goal: ON(C, B) ∧
ON(D, A) ∧
ONTABLE(B) ∧
ONTABLE(A)

   (a) Show how STRIPS would solve this problem.
   (b) Show how TWEAK would solve this problem.
   (c) Did these processes produce optimal plans? If not, could they be modified to do so?
2. Consider the problem of devising a plan for cleaning the kitchen.
   (a) Write a set of STRIPS-style operators that might be used. When you describe the operators, take into account such considerations as:
      - Cleaning the stove or the refrigerator will get the floor dirty.
      - To clean the oven, it is necessary to apply oven cleaner and then to remove the cleaner.
      - Before the floor can be washed, it must be swept.
      - Before the floor can be swept, the garbage must be taken out.
      - Cleaning the refrigerator generates garbage and messes up the counters.
      - Washing the counters or the floor gets the sink dirty.
   (b) Write a description of a likely initial state of a kitchen in need of cleaning. Also write a description of a desirable (but perhaps rarely obtained) goal state.
   (c) Show how the technique of planning using a goal stack could be used to solve this problem. (Hint—you may want to modify the definition of an ADD condition so that when a condition is added to the database, its negation is automatically deleted if present.)
3. In Section 13.4, we showed an example of a situation in which a search path could be terminated because it led back to one of its earlier goals. Describe a mechanism by which a program could detect this situation.
4. Consider the problem of swapping the contents of two registers, A and B. Suppose that there is available the single operator ASSIGN(x, v, lv, ov), which assigns the value v, which is stored in location lv, to location x, which previously contained the value ov:

Copyrighted material

ASSIGN(x v, lv, ov)
P: CONTAINS(lv, v) $\wedge$ CONTAINS(x, ov)
D: CONTAINS(x, ov)
A: CONTAINS(x, v)

Assume that there is at least one additional register, C, available.

(a) What would STRIPS do with this problem?
(b) What would TWEAK do with this problem?
(c) How might you design a program to solve this problem?

Copyrighted material

CHAPTER 14

UNDERSTANDING

All truths are easy to understand once they are discovered; the point is to discover them.

—Galileo Galilei
(1564-1642), Italian physicist, mathematician, astronomer and philosopher

14.1 WHAT IS UNDERSTANDING?

To understand something is to transform it from one representation into another, where this second representation has been chosen to correspond to a set of available actions that could be performed and where the mapping has been designed so that for each event, an *appropriate* action will be performed. There is very little absolute in the notion of understanding. If you say to an airline database system "I need to go to New York as soon as possible," the system will have "understood" if it finds the first available plane to New York. If you say the same thing to your best friend, who knows that your family lives in New York, she will have "understood" if she realizes that there may be a problem in your family and you may need some emotional support. As we talk about understanding, it is important to keep in mind that the success or failure of an "understanding" program can rarely be measured in an absolute sense but must instead be measured with respect to a particular task to be performed. This is true both of language-understanding programs and also of understanders in other domains, such as vision.

For people, understanding applies to inputs from all the senses. Computer understanding has so far been applied primarily to images, speech, and typed language. In this chapter we discuss issues that cut across all of these modalities. In Chapter 15, we explore the problem of typed natural language in more detail, and in Chapter 21, we look at speech and vision problems. Although we have defined understanding above as the process of mapping into appropriate *actions,* we are not precluding a view of understanding in which inputs are simply interpreted and stored for later. In such a system, the appropriate action is to store the proper representation. This view of understanding describes what occurs in most image understanding programs and some language understanding programs. Taking direct action describes what happens in systems in which language, either typed or spoken, is used in the interface between user and computer.

Copyrighted material

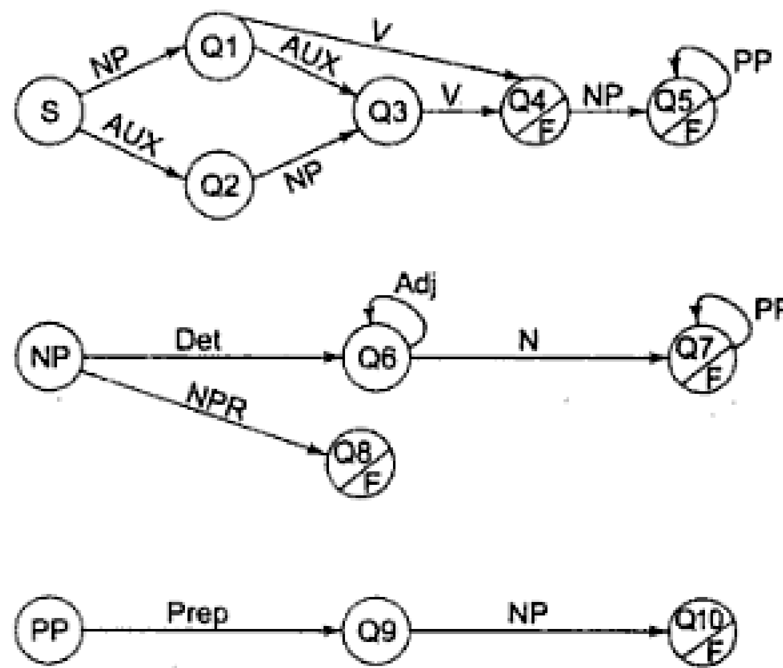


Fig. 15.8 *An ATN Network for a Fragment of English*

```
(S/     (PUSH NP/T
            (SETR SUBJ *)
            (SETR TYPE (QUOTE DCL))
            (TO Q1))
        (CAT AUX T
            (SETR AUX *)
            (SETR TYPE (QUOTE Q))
            (TO Q2)))
(Q1     (CAT V T
            (SETR AUX NIL)
            (SETR V *)
            (TO Q4))
        (CAT AUX T
            (SETR AUX *)
            (TO Q3)))
(Q2     (PUSH NP/ T
            (SETR SUBJ *)
            (TO Q3)))
(Q3     (CAT V T
            (SETR V *)
            (TO Q4)))
(Q4     (POP (BUILDQ (S + + + (VP +))
                TYPE SUBJ AUX V) T)
        (PUSH NP/ T
            (SETR VP (BUILDQ (VP (V +) *) V))
            (TO Q5)))
(Q5     (POP (BUILDQ (S + + + +)
                TYPE SUBJ AUX VP) T)
        (PUSH PP/ T
            (SETR VP (APPEND (GETR VP) (LIST *)))
            (TO Q5)))
```

Fig. 15.9 *An ATN Grammar in List Form*

Copyrighted material

6. This test succeeds, so append "long" to the list contained in the ADJS register. (This list was previously empty.) Stay in state Q6.
7. Do a category test to see if "file" is an adjective. This test fails.
8. Do a category test to see if "file" is a noun. This test succeeds, so set the NOUN register to "file" and go to state Q7.
9. Push to PP.
10. Do a category test to see if "has" is a preposition. This test fails, so pop and signal failure.
11. There is nothing else that can be done from state Q7, so pop and return the structure
    (NP (FILE (LONG) DEFINITE))
    The return causes the machine to be in state Ql, with the SUBJ register set to the structure just returned and the TYPE register set to DCL.
12. Do a category test to see if "has" is a verb. This test succeeds, so set the AUX register to NIL and set the V register to "has." Go to state Q4.
13. Push to state NP. Since the next word, "printed," is not a determiner or a proper noun, NP will pop and return failure.
14. The only other thing to do in state Q4 is to halt. But more input remains, so a complete parse has not been found. Backtracking is now required.
15. The last choice point was at state Ql, so return there. The registers AUX and V must be unset.
16. Do a category test to see if "has" is an auxiliary. This test succeeds, so set the AUX register to "has" and go to state Q3.
17. Do a category test to see if "printed" is a verb. This test succeeds, so set the V register to "printed." Go to state Q4.
18. Now, since the input is exhausted, Q4 is an acceptable final state. Pop and return the structure
    (S DCL (NP (FILE (LONG) DEFINITE)) HAS (VP PRINTED))
    This structure is the output of the parse.

This example grammar illustrates several interesting points about the use of ATNs. A single subnetwork need only occur once even though it is used in more than one place. A network can be called recursively. Any number of internal registers may be used to contain the result of the parse. The result of a network can be built, using the function BUILDQ, out of values contained in the various system registers. A single state may be both a final state, in which a complete sentence has been found, and an intermediate state, in which only a part of a sentence has been recognized. And, finally, the contents of a register can be modified at any time.

In addition, there are a variety of ways in which ATNs can be used which are not shown in this example:

- The contents of registers can be swapped. For example, if the network were expanded to recognize passive sentences, then at the point that the passive was detected, the current contents of the SUBJ register would be transferred to an OBJ register and the object of the preposition "by" would be placed in the SUBJ register. Thus the final interpretation of the following two sentences would be the same.
  - Bill printed the file.
  - The file was printed by Bill.
- Arbitrary tests can be placed on the arcs. In each of the arcs in this example, the test is specified simply as T (always true). But this need not be the case. Suppose that when the first NP is found, its number is determined and recorded in a register called NUMBER. Then the arcs labeled V could have an additional test placed on them that checked that the number of the particular verb that was found is equal to the value stored in NUMBER. More sophisticated tests, involving semantic markers or other semantic features, can also be performed.

Copyrighted material

15.2.3 Unification Grammars

ATN grammars have substantial procedural components. The grammar describes the order in which constituents must be built. Variables are explicitly given values, and they must already have been assigned a value before they can be referenced. This procedurality limits the effectiveness of ATN grammars in some cases, for example: in speech processing where some later parts of the sentence may have been recognized clearly while earlier parts are still unknown (for example, suppose we had heard, "The long * * * file printed."), or in systems that want to use the same grammar to support both understanding and generation (e.g., Appelt [1987], Shieber [1988], and Barnett *et al.* [1990]). Although there is no clear distinction between declarative and procedural representations (as we saw in Section 6.1), there is a spectrum and it often turns out that more declarative representations are more flexible than more procedural ones are. So in this section we describe a declarative approach to representing grammars.

When a parser applies grammar rules to a sentence, it performs two major kinds of operations:

- Matching (of sentence constituents to grammar rules)
- Building structure (corresponding to the result of combining constituents)

Now think back to the unification operation that we described in Section 5.4.4 as part of our theorem-proving discussion. Matching and structure building are operations that unification performs naturally. So an obvious candidate for representing grammars is some structure on which we can define a unification operator. Directed acyclic graphs (DAGs) can do exactly that.

Each DAG represents a set of attribute-value pairs. For example, the graphs corresponding to the words "the" and "file" are:

```
[CAT: DET          [CAT: N
 LEX: the]          LEX: file
                    NUMBER: SING]
```

Both words have a lexical category (CAT) and a lexical entry. In addition, the word "file" has a value (SING) for the NUMBER attribute. The result of combining these two words to form a simple NP can also be described as a graph:

```
[NP: [DET: the
      HEAD: file
      NUMBER: SING]]
```

The rule that forms this new constituent can also be represented as a graph, but to do so we need to introduce a new notation. Until now, all our graphs have actually been trees. To describe graphs that are not trees, we need a way to label a piece of a graph and then point to that piece elsewhere in the graph. So let $\{n\}$ for any value of n be a label, which is to be interpreted as a label for the next constituent following it in the graph. Sometimes, the constituent is empty (i.e., there is not yet any structure that is known to fill that piece of the graph). In that case, the label functions very much like a variable and will be treated like one by the unification operation. It is this degenerate kind of a label that we need in order to describe the NP rule:

NP → DET N

We can write this rule as the following graph:

Copyrighted material

```
[CONSTITUENT1: [CAT: DET
                LEX: {1}]
CONSTITUENT2: [CAT: N
                LEX: {2}
                NUMBER: {3}]
BUILD: [NP:[DET: {1}
           HEAD: {2}
           NUMBER: {3}]]]
```

This rule should be read as follows: Two constituents, described in the subgraphs labeled CONSTITUENT1 and CONSTITUENT2, are to be combined. The first must be of CAT DET. We do not care what its lexical entry is, but whatever it is will be bound to the label {1}. The second constituent must be of CAT N. Its lexical entry will be bound to the label {2}, and its number will be bound to the label {3}. The result of combining these two constituents is described in the subgraph labeled BUILD. This result will be a graph corresponding to an NP with three attributes: DET, HEAD, and NUMBER. The values for all these attributes are to be taken from the appropriate pieces of the graphs that are being combined by the rule.

Now we need to define a unification operator that can be applied to the graphs we have just described. It will be very similar to logical unification. Two graphs unify if, recursively, all their subgraphs unify. The result of a successful unification is a graph that is composed of the union of the subgraphs of the two inputs, with all bindings made as indicated. This process bottoms out when a subgraph is not an attribute-value pair but is just a value for an attribute. At that point, we must define what it means for two values to unify. Identical values unify. Anything unifies with a variable (a label with no attached structure) and produces a binding for the label. The simplest thing to do is then to say that any other situation results in failure. But it may be useful to be more flexible. So some systems allow a value to match with a more general one (e.g., PROPER-NOUN matches NOUN). Others allow values that are disjunctions [e.g., (MASCULINE $\vee$ FEMININE)], in which case unification succeeds whenever the intersection of the two values is not empty.

There is one other important difference between logical unification and graph unification. The inputs to logical unification are treated as logical formulas. Order matters, since, for example, $f(g(a), h(b))$ is a different formula than $f(h(b), g(a))$. The inputs to graph unification, on the other hand, must be treated as sets, since the order in which attribute-value pairs are stated does not matter. For example, if a rule describes a constituent as

```
[CAT: DET
LEX: {1}]
```

we want to be able to match a constituent such as

```
[LEX: the
CAT: DET]
```

Algorithm: Graph-Unify

1. If either G_1 or G_2 is an attribute that is not itself an attribute-value pair then:
   (a) If the attributes conflict (as defined above), then fail.
   (b) If either is a variable, then bind it to the value of the other and return that value.
   (c) Otherwise, return the most general value that is consistent with both the original values. Specifically, if disjunction is allowed, then return the inter section of the values.

Copyrighted material

2. Otherwise, do:
   (a) Set variable *NEW* to empty.
   (b) For each attribute *A* that is present (at the top level) in either *G*1 or *G*2 do
      (i) If *A* is not present at the top level in the other input, then add *A* and its value to *NEW*.
      (ii) If it is, then call Graph-Unify with the two values for *A*. If that fails, then fail. Otherwise, take the new value of *A* to be the result of that unification and add *A* with its value to *NEW*.
   (c) If there are any labels attached to *G*1 or *G*2, then bind them to *NEW* and return *NEW*.

A simple parser can use this algorithm to apply a grammar rule by unifying CONSTITUENT 1 with a proposed first constituent. If that succeeds, then CONSTITUENT2 is unified with a proposed second constituent. If that also succeeds, then a new constituent corresponding to the value of BUILD is produced. If there are variables in the value of BUILD that were bound during the matching of the constituents, then those bindings will be used to build the new constituent.

There are many possible variations on the notation we have described here. There are also a variety of ways of using it to represent dictionary entries and grammar rules. See Shieber [1986] and Knight [1989] for discussions of some of them.

Although we have presented unification here as a technique for doing syntactic analysis, it has also been used as a basis for semantic interpretation. In fact, there are arguments for using it as a uniform representation for all phases of natural language understanding. There are also arguments against doing so, primarily involving system modularity, the noncompositionality of language in some respects (see Section 15.3.4), and the need to invoke substantial domain reasoning. We will not say any more about this here, but to see how this idea could work, see Allen [1989].

15.3 SEMANTIC ANALYSIS

Producing a syntactic parse of a sentence is only the first step toward understanding it. We must still produce a representation of the *meaning* of the sentence. Because understanding is a mapping process, we must first define the language into which we are trying to map. There is no single, definitive language in which all sentence meanings can be described. All of the knowledge representation systems that were described in Part II are candidates, and having selected one or more of them, we still need to define the vocabulary (i.e., the predicates, frames, or whatever) that will be used on top of the structure. In the rest of this chapter, we call the final meaning representation language, including both the representational framework and the specific meaning vocabulary, the *target language*. The choice of a target language for any particular natural language understanding program must depend on what is to be done with the meanings once they are constructed. There are two broad families of target languages that are used in NL systems, depending on the role that the natural language system is playing in a larger system (if any).

When natural language is being considered as a phenomenon on its own, as, for example, when one builds a program whose goal is to read text and then answer questions about it, a target language can be designed specifically to support language processing. In this case, one typically looks for primitives that correspond to distinctions that are usually made in language. Of course, selecting the right set of primitives is not easy. We discussed this issue briefly in Section 4.3.3, and in Chapter 10 we looked at two proposals for a set of primitives, conceptual dependency and CYC.

When natural language is being used as an interface language to another program (such as a database query system or an expert system), then the target language must be a legal input to that other program. Thus the design of the target language is driven by the backend program. This was the case in the simple example we discussed in Section 15.1.1. But even in this case, it is useful, as we showed in that example, to use an intermediate knowledge-based representation to guide the overall process. So, in the rest of this section, we assume that the target language we are building is a knowledge-based one.

Copyrighted material

Although the main purpose of semantic processing is the creation of a target language representation of a sentence's meaning, there is another important role that it plays. It imposes constraints on the representations that can be constructed, and, because of the structural connections that must exist between the syntactic structure and the semantic one, it also provides a way of selecting among competing syntactic analyses. Semantic processing can impose constraints because it has access to knowledge about what makes sense in the world. We already mentioned one example of this, the sentence, "Is the glass jar peanut butter?" There are other examples in the rest of this section.

Lexical Processing

The first step in any semantic processing system is to look up the individual words in a dictionary (or *lexicon)* and extract their meanings. Unfortunately, many words have several meanings, and it may not be possible to choose the correct one just by looking at the word itself. For example, the word "diamond" might have the following set of meanings:

- A geometrical shape with four equal sides
- A baseball field
- An extremely hard and valuable gemstone

To select the correct meaning for the word "diamond" in the sentence,

> Joan saw Susan's diamond shimmering from across the room.

it is necessary to know that neither geometrical shapes nor baseball fields shimmer, whereas gemstones do.

Unfortunately, if we view English understanding as mapping from English words into objects in a specific knowledge base, lexical ambiguity is often greater than it seems in everyday English. For, example, consider the word "mean." This word is ambiguous in at least three ways: it can be a verb meaning "to signify"; it can be an adjective meaning "unpleasant" or "cheap"; and it can be a noun meaning "statistical average." But now imagine that we have a knowledge base that describes a statistics program and its operation. There might be at least two distinct objects in that knowledge base, both of which correspond to the "statistical average" meaning of "mean." One object is the statistical concept of a mean; the other is the particular function that computes the mean in this program. To understand the word "mean" we need to map it into some concept in our knowledge base. But to do that, we must decide which of these concepts is meant. Because of cases like this, lexical ambiguity is a serious problem, even when the domain of discourse is severely constrained.

The process of determining the correct meaning of an individual word is called *word sense disambiguation* or *lexical disambiguation.* It is done by associating, with each word in the lexicon, information about the contexts in which each of the word's senses may appear. Each of the words in a sentence can serve as part of the context in which the meanings of the other words must be determined.

Sometimes only very straightforward information about each word sense is necessary. For example, the baseball field interpretation of "diamond" could be marked as a LOCATION. Then the correct meaning of "diamond" in the sentence "I'll meet you at the diamond" could easily be determined if the fact that *at* requires a TIME or a LOCATION as its object were recorded as part of the lexical entry for *at.* Such simple properties of word senses are called *semantic markers.* Other useful semantic markers are

- PHYSICAL-OBJECT
- ANIMATE-OBJECT
- ABSTRACT-OBJECT

Using these markers, the correct meaning of "diamond" in the sentence "I dropped my diamond" can be computed. As part of its lexical entry, the verb "drop" will specify that its object must be a PHYSICAL-

Copyrighted material

17.5.3 Decision Trees

A third approach to concept learning is the induction of *decision trees*, as exemplified by the ID3 program of Quinlan [1986]. ID3 uses a tree representation for concepts, such as the one shown in Fig. 17.13. To classify a particular input, we start at the top of the tree and answer questions until we reach a leaf, where the classification is stored. Fig. 17.13 represents the familiar concept "Japanese economy car." ID3 is a program that builds decision trees automatically, given positive and negative instances of a concept.[4]

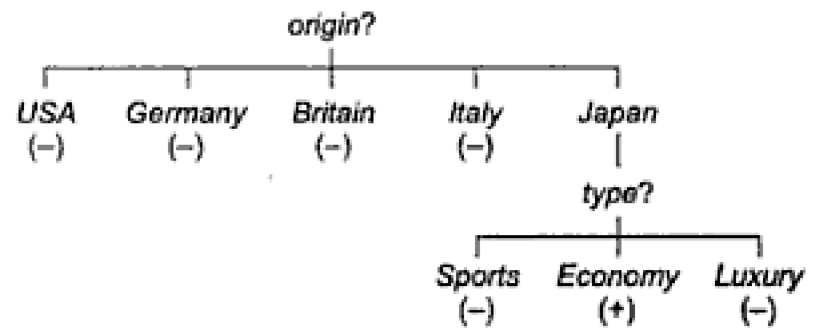


Fig. 17.13 *A Decision Tree*

ID3 uses an iterative method to build up decision trees, preferring simple trees over complex ones, on the theory that simple trees are more accurate classifiers of future inputs. It begins by choosing a random subset of the training examples. This subset is called the *window*. The algorithm builds a decision tree that correctly classifies all examples in the window. The tree is then tested on the training examples outside the window. If all the examples are classified correctly, the algorithm halts. Otherwise, it adds a number of training examples to the window and the process repeats. Empirical evidence indicates that the iterative strategy is more efficient than considering the whole training set at once.

So how does ID3 actually construct decision trees? Building a node means choosing some attribute to test. At a given point in the tree, some attributes will yield more information than others. For example, testing the attribute *color* is useless if the color of a car does not help us to classify it correctly. Ideally, an attribute will separate training instances into subsets whose members share a common label (e.g., positive or negative). In that case, branching is terminated, and the leaf nodes are labeled.

There are many variations on this basic algorithm. For example, when we add a test that has more than two branches, it is possible that one branch has no corresponding training instances. In that case, we can either leave the node unlabeled, or we can attempt to guess a label based on statistical properties of the set of instances being tested at that point in the tree. Noisy input is another issue. One way of handling noisy input is to avoid building new branches if the information gained is very slight. In other words, we do not want to overcomplicate the tree to account for isolated noisy instances. Another source of uncertainty is that attribute values may be unknown. For example a patient's medical record may be incomplete. One solution is to guess the correct branch to take; another solution is to build special "unknown" branches at each node during learning.

When the concept space is very large, decision tree learning algorithms run more quickly than ther version space cousins. Also, disjunction is more straightforward. For example, we can easily modify Fig. 17.13 to represent the disjunctive concept "American car or Japanese economy car," simply by changing one of the negative (—) leaf labels to positive (+). One drawback to the ID3 approach is that large, complex decision trees can be difficult for humans to understand, and so a decision tree system may have a hard time explaining the reasons for its classifications.

17.6 EXPLANATION-BASED LEARNING

The previous section illustrated how we can induce concept descriptions from positive and negative examples. Learning complex concepts using these procedures typically requires a substantial number of training instances.

[4] Actually, the decision tree representation is more general: Leaves can denote any of a number of classes, not just positive and negative.

Copyrighted material

The McGraw·Hill Companies

ARTIFICIAL INTELLIGENCE

Third Edition

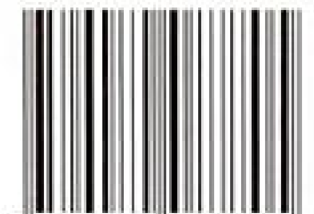
This hallmark text presents both theoretical foundations of Artificial Intelligence and ways in which current techniques can be used in application programs. The new edition has been enriched with specific chapters describing upcoming areas that have found variety of uses under the domain of Artificial 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



Tata McGraw-Hill

Copyrighted material