The Addison-Wesley Series in
Artificial Intelligence

Buchanan and Shortliffe (eds.): *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. (In preparation.)

Clancey and Shortliffe (eds.): *Readings in Medical Artificial Intelligence: The First Decade*. (In preparation.)

Pearl: *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. (1984)

Sager: *Natural Language Information Processing: A Computer Grammar of English and Its Applications*. (1981)

Wilensky: *Planning and Understanding: A Computational Approach to Human Reasoning*. (1983)

Winograd: *Language as a Cognitive Process Vol. I: Syntax*. (1983)

Winston: *Artificial Intelligence*, Second Edition. (1984)

Winston and Horn: *LISP*. (1981)

HEURISTICS

Intelligent Search Strategies for Computer Problem Solving

Judea Pearl

Department of Computer Science
University of California
Los Angeles, California

ADDISON-WESLEY PUBLISHING COMPANY
Reading, Massachusetts • Menlo Park, California
London • Amsterdam • Don Mills, Ontario • Sydney

To
Ruth, Tammy, Danny, and Michelle

This book is in the Addison-Wesley Series in Artificial Intelligence.

Library of Congress Cataloging in Publication Data

Pearl, Judea.
Heuristics : intelligent search strategies for computer problem solving.

(Artificial intelligence series)
Bibliography: p.
Includes index.
1. Artificial intelligence. 2. Operations research. 3. Problem-solving—Data processing. 4. Heuristic programming. I. Title. II. Series: Artificial intelligence series (Addison-Wesley Publishing Company)
Q335.P38 1984 001.53'5 83-12217
ISBN 0-201-05594-5

Copyright © 1984 by Addison-Wesley Publishing Company, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Published simultaneously in Canada.

ABCDEFGHIJ-MA-8987654

Preface

This book is about *heuristics*, popularly known as rules of thumb, educated guesses, intuitive judgments or simply *common sense*. In more precise terms, heuristics stand for strategies using readily accessible though loosely applicable information to control problem-solving processes in human beings and machine. This book presents an analysis of the nature and the power of typical heuristic methods, primarily those used in artificial intelligence (AI) and operations research (OR) to solve problems of search, reasoning, planning and optimization on digital machines.

The discussions in this book follow a three-phase pattern: Presentation, characterization, and evaluation. We first present a set of general-purpose problem-solving strategies guided by heuristic information (Chapters 1 and 2), then highlight the general principles and properties that characterize this set (Chapters 3 and 8) and, finally, we present mathematical analyses of the performances of these strategies in several well-structured domains (Chapters 5, 6, 7, 9, and 10). Some psychological aspects of how people discover and use heuristics are discussed briefly in Chapters 1 and 4.

The original intention in writing this book was to provide a cohesive and hospitable package for the theoretical results obtained at the UCLA Cognitive Systems Laboratory in the past three years. Some of these results were scattered in various reports, proceedings, and archival journals, and others buried in notebooks awaiting a sufficient incentive for refinement and publication. The compilation of this body of research into a single volume under a consistent notation and a unified logical thread now offers readers easier access to the main results and a better opportunity to assess their range of applicability. These original works are covered in Chapters 5, 6, 7, 9 and 10 and the last sections of Chapters 3 and 8.

Under the persuasive influence of colleagues and students, I later broadened the scope of the book to include some introductory material: an overview of heuristics in typical problem-solving situations (Chapter 1), a description and taxonomy of the basic heuristic search strategies used in AI (Chapters 2 and 8), a formal exposition of their main properties (Chapter 3), and a general discussion on the nature of heuristics (Chapter 4).

As it now stands, the book can fulfill three roles—it can serve as a monograph, as a reference, or as a textbook. The researcher or practitioner familiar

with the standard literature on heuristic search can skip the introductory sections and turn directly to the theoretical topics, which are relatively self-contained. Casual readers, curious enough to gain an understanding of the type of problems and techniques emerging from the AI brewery, are advised to follow the introductory material at their own pace and select the advanced reading which best matches their tastes, backgrounds, and ultimate objectives. Special care was taken to adhere to traditional mathematical notation (avoiding programming-based jargons), thereby ensuring that readers from engineering, operations research, mathematics, or computer science will find the presentation familiar and comfortable. Finally, as a textbook, the book can be used as a reference for an introductory course in AI, and as a full text in a more advanced, graduate-level class on AI control strategies or the analysis of algorithms. At UCLA, for example, we have used Chapters 1, 2, 3 (Section 3.1), and 8 to cover roughly half of the first graduate course in artificial intelligence. The other chapters are used as a text in a graduate course on heuristic algorithms that serves the curriculum of two major fields: machine intelligence and computer science theory. These chapters should also be ideal for an operations research class that focuses on the taxonomy and analysis of combinatorial search techniques.

A large part of the material is presented in the language of mathematics, mainly that of elementary probability theory. It is assumed, therefore, that the reader is familiar with the basic concepts of events, random variables, distribution functions, expectations, and generating functions, which correspond to a first course in probability theory. Chapters 5 and 6 make use of some results from the theory of branching processes. These are described and summarized in Appendix 5-A, which readers are advised to read before proceeding with the text of Sections 5.4 and 6.3. Familiarity with graph theory can be helpful but is not essential because the concepts needed for our purposes are summarized in the introductory sections to Chapters 2 and 3.

The mathematical nature of our analyses should not by any means prevent the less mathematically trained reader from following the basic concepts and techniques, or from appreciating the impact of the main results. A conscious effort was made to introduce each topic in an intuitive—even metaphorical—style so that the presentation as a whole would be both technically correct and easy to follow. The "theorem-proof" structure of some chapters should not automatically suggest dryness. Most theorems either summarize the discussions that precede them, or facilitate valuable shortcuts in the discussions that follow. In all cases, however, I kept in mind that a theorem's primary mission is to resolve some curious dilemma, and it is in the description of these dilemmas that my main effort was invested.

The choice of algorithm description language deserves an explanatory remark. The commitment to make this book accessible to readers from all areas of the applied sciences has resulted in the following guideline: whenever possible algorithms should be as readable by nonprogrammers as the rest of the text. For that reason I have avoided using any of the existing formal

languages and, moreover, have purposely attempted to give algorithms an *informal*, proselike flavor. Along this line I have avoided an excessive use of recursions, have made repetitive references to the underlying data structures, have emphasized the purpose of steps and the meaning of the variables, and have consistently used *redundant* connecting phrases such as: "else continue," "as soon as," and others. I hope readers will agree that the improved legibility of these descriptions more than makes up for their "impure" appearance; besides, in the not very distant future computers too will be able to accept these natural descriptions and convert them into executable programs.

I take great pleasure in closing this preface by acknowledging those who assisted me with this book. First, I would like to thank the members of the Cognitive Systems Laboratory at UCLA upon whose works and ideas many of the sections are based; Rina Dechter, Nam Huyn, Jin Kim, Scott Kurman, Gerard Michon, Igor Roizen, Michael Tarsi, Chuck Siska, and Rick Verstraete. Acknowledgment is due the National Science Foundation for sponsoring the research that led to many of these results. My academic and professional colleagues have all been very supportive of this endeavor: Nils Nilsson has provided continuous encouragement and Richard Karp is responsible for everything I know about branching processes, especially the ideas of Section 5.4. The comments of John McCarthy, Donald Michie, Allen Newell, and Herbert Simon helped improve the presentation of many sections, especially those relating the historical developments of the key ideas. The largest portion of this book was typed and corrected on a UNIX system by Lorna Freeman, assisted by Tovah Hollander, Terry Peters and Anna Gibbons. Last, I owe a great debt of thanks to my family for their support and understanding: to Danny for insightful suggestions on the first draft, to Michelle for her keen proofreading, and especially to my wife Ruth for tolerating, supporting, and encouraging my longest project since our wedding day.

Tel Aviv, Israel
January 1984

J.P.

Prologue

The study of heuristics draws its inspiration from the ever-amazing observation of how much people can accomplish with that simplistic, unreliable information source known as *intuition*. We drive our cars with hardly any thought of how they function and only a vague mental picture of the road conditions ahead. We write complex computer programs while attending to only a fraction of the possibilities and interactions that may take place in the actual execution of these programs. Even more surprisingly, we maneuver our way successfully in intricate social situations having only a guesswork expectation of the behavior of other persons around and even less certainty of their expectations of us. Yet, when these expectations fail we are able to master the great power of humor and recover gracefully. No computer program has yet been designed that exhibits such capabilities. Evidently, the little information we possess is so effectively organized that its poor state of reliability hardly hinders our normal, everyday activities.

Early attempts to equip machines with humanlike intelligence have quickly revealed that machines, too, cannot possibly be expected to operate with only precise and detailed knowledge of their task environment. Rather, this knowledge must, in some way, be summarized or abstracted so as to provide, like human intuition, a steady stream of tentative, unpolished, yet swift and informative *advice* for managing the primitive computational steps that make up a problem-solving process. The information content of this advice came to be known as *heuristic knowledge*. The science of heuristics, then, comprises both empirical and theoretical studies aimed toward understanding the workings of heuristic knowledge; how it is acquired, stored, and used by people, how it can be represented and utilized by machines, and what makes one heuristic succeed where others fail. This book focuses on the latter two topics.

Contents

PART I
Problem-Solving Strategies and the Nature of Heuristic Information

1. Heuristics and Problem Representations **3**

1.1 Typical Uses of Heuristics in Problem Solving 3

1.1.1 The 8-Queens Problem 4 / 1.1.2 The 8-Puzzle 6 / 1.1.3 The Road Map Problem 9 / 1.1.4 The Traveling Salesman Problem (TSP) 10 / 1.1.5 The Counterfeit Coin Problem 12

1.2 Search Spaces and Problem Representations 14

1.2.1 Optimizing, Satisficing, and Semi-Optimizing Tasks 14 / 1.2.2 Systematic Search and the Split-and-Prune Paradigm 15 / 1.2.3 State-Space Representation 20 / 1.2.4 Problem-Reduction Representations and AND/OR Graphs 21 / 1.2.5 Selecting a Representation 26

1.3 Bibliographical and Historical Remarks 31

Exercises 32

2. Basic Heuristic-Search Procedures **33**

2.1 Hill-Climbing: An Irrevocable Strategy 35

2.2 Uninformed Systematic Search: Tentative Control Strategies 36

2.2.1 Depth-First and Backtracking: LIFO Search Strategies 36 / 2.2.2 Breadth-First: A FIFO Search Strategy 42 / 2.2.3 Uninformed Search of AND/OR Graphs 44

2.3 Informed, Best-First Search: A Way of Using Heuristic Information 46

2.3.1 A Basic Best-First (*BF*) Strategy for State-Space Search 48 / 2.3.2 A General Best-First Strategy for AND/OR Graphs (*GBF*) 49

2.4 Specialized Best-First Algorithms: Z*, A*, AO, and AO* 56

2.4.1 Why Restrict the Evaluation Functions? 56 / 2.4.2 Recursive Weight Functions 57 / 2.4.3 Identifying G_0, The Most Promising Solution-Base Graph 59 / 2.4.4 Specialized Best-First Strategies 61

2.5 Hybrid Strategies 65
2.5.1 *BF-BT* Combinations 66 / 2.5.2 Introducing Irrevocable Decisions 68
2.6 Bibliographical and Historical Remarks 69
Exercises 71

3. Formal Properties of Heuristic Methods 73
3.1 A^*—Optimal Search for an Optimal Solution 75
3.1.1 Properties of f^* 75 / 3.1.2 Termination and Completeness 76 / 3.1.3 Admissibility—A Guarantee for an Optimal Solution 77 / 3.1.4 Comparing the Pruning Power of Several Heuristics 79 / 3.1.5 Monotone (Consistent) Heuristics 82
3.2 Relaxing the Optimality Requirement 86
3.2.1 Adjusting the Weights of g and h 86 / 3.2.2 Two ϵ-Admissible Speedup Versions of A^* 88 / 3.2.3 R_δ^*—A Limited Risk Algorithm Using Information about the Uncertainty of h 90 / 3.2.4 $R_{\delta,\epsilon}^*$—A Speedup Version of R_δ^* 97
3.3 Some Extensions to Nonadditive Evaluation Functions (*BF** and *GBF**) 99
3.3.1 Notation and Preliminaries 100 / 3.3.2 Algorithmic Properties of Best-First Search *BF** 103
3.4 Bibliographical and Historical Remarks 110
Exercises 112

4. Heuristics Viewed as Information Provided by Simplified Models 113
4.1 The Use of Relaxed Models 113
4.1.1 Where Do These Heuristics Come From? 113 / 4.1.2 Consistency of Relaxation-Based Heuristics 115 / 4.1.3 Overconstrained, Analogical, and Other Types of Auxiliary Models 116
4.2 Mechanical Generation of Admissible Heuristics 118
4.2.1 Systematic Relaxation 118 / 4.2.2 Can a Program Tell an Easy Problem When It Sees One? 121 / 4.2.3 Summary 123
4.3 Probability-Based Heuristics 124
4.3.1 Heuristics Based on the Most Likely Outcome 125 / 4.3.2 Heuristics Based on Sampling 126 / 4.3.3 Probability-Based Heuristics in the Service of Semi-Optimization Problems 128
4.4 Bibliographical and Historical Remarks 131
Exercises 133

PART II
Performance Analysis of Heuristic Methods

5. Abstract Models for Quantitative Performance Analysis 137
5.1 Mathematical Performance Analysis, or Test Tubes versus Fruit Flies in the Design of Gothic Cathedrals 137
5.2 Example 1: Finding a Shortest Path in a Regular Lattice with Air-Distance Heuristics 140
5.3 Example 2: Finding a Shortest Path in a Road-Map with Randomly Distributed Cities 146
5.4 Example 3: Searching for an Optimal Path in a Tree with Random Costs 150
5.4.1 Notation and Preliminaries 150 / 5.4.2 Summary of Results 153 / 5.4.3 Branching Processes and the Proofs of Theorems 1-6 154 / 5.4.4 Conclusions 162
5.5 Bibliographical and Historical Remarks 163
Exercises 164
Appendix 5-A: Basic Properties of Branching Processes 165
Appendix 5-B: The Expected Size of an Extinct Family 166
Appendix 5-C: Proof of Theorem 2 167

6. Complexity versus Precision of Admissible Heuristics 169
6.1 Heuristics Viewed as Noisy Information Sources 169
6.1.1 Simplified Models as Sources of Noisy Signals 169 / 6.1.2 A Probabilistic Model for Performance Analysis 171 / 6.1.3 A Formula for the Mean Complexity of A^* 173
6.2 Stochastic Dominance for Random Admissible Heuristics 176
6.3 The Mean Complexity of A^* under Distance-Dependent Errors 179
6.3.1 The Average Complexity under Proportional Errors 179 / 6.3.2 The Average Complexity under General Distance-Dependent Errors 183
6.4 Comparison to Backtracking and the Effect of Multiple Goals 189
6.4.1 The Mean Complexity of Informed Backtracking 190 / 6.4.2 The Effect of Multiple Goals 191
Exercises 193

7. Searching with Nonadmissible Heuristics 194
7.1 Conditions for Node Expansion 194
7.2 When Is One Heuristic Better Than Another If Overestimations Are Possible? 199
7.3 How to Improve a Given Heuristic 202
7.3.1 The Effect of Weighting g and h 202 / 7.3.2 How to Combine Information from Several Heuristic Sources 209 / 7.3.3 When Is It Safe to Use $f = h$ or, Who's Afraid of $w = 1$? 210
Exercises 213
Appendix 7-A: Proof of Lemma 2 214
Appendix 7-B: Proof of Theorem 1 (The Pessimistic Substitution Principles) 216

PART III
Game-Playing Programs

8. Strategies and Models for Game-Playing Programs 221

8.1 Solving and Evaluating Games 222
8.1.1 Game Trees and Game-Playing Strategies 222 / 8.1.2 Bounded Look-Ahead and the Use of Evaluation Functions 226 / 8.1.3 MIN-MAX versus NEG-MAX Notations 228

8.2 Basic Game-Searching Strategies 229
8.2.1 Exhaustive Minimaxing and the Potential for Pruning 229 / 8.2.2 The α-β Pruning Procedure: A Backtracking Strategy 231 / 8.2.3 SSS* – A Best-First Search for an Optimal Playing Strategy 240 / 8.2.4 SCOUT – A Cautious Test-Before-Evaluate Strategy 246

8.3 A Standard Probabilistic Model for Studying the Performance of Game-Searching Strategies 251
8.3.1 The Probability of Winning a Standard Game with Random Win Positions 251 / 8.3.2 Game Trees with an Arbitrary Distribution of Terminal Values 254 / 8.3.3 The Mean Complexity of Solving a Standard (d,b,P_0)-game 259 / 8.3.4 The Mean Complexity of Testing and Evaluating Multivalued Game Trees 268

8.4 Recreational Diversions 270
8.4.1 The Board-Splitting Game – A Physical Embodiment of the Standard Game Tree 270 / 8.4.2 Other Applications of the Minimax Convergence Theorem 273 / 8.4.3 Games as Mazes with Hidden Paths: A Useful Metaphor 277

8.5 Bibliographical and Historical Remarks 285
Exercises 287

9. Performance Analysis for Game-Searching Strategies 288

9.1 The Expected Performance of SCOUT 289
9.1.1 Games with Continuous Terminal Values 289 / 9.1.2 Games with Discrete Terminal Values 292

9.2 The Expected Performance of α-β 293
9.2.1 Historical Background 293 / 9.2.2 An Integral Formula for $I_{\alpha\text{-}\beta}(d,b)$ 295 / 9.2.3 The Branching Factor of α-β and Its Optimality 296 / 9.2.4 How Powerful Is the α-β Pruning? 298

9.3 The Expected Performance of SSS* 300
9.3.1 A Necessary and Sufficient Condition for Node Examination 300 / 9.3.2 The Probability of Examining a Terminal Node 301 / 9.3.3 The Expected Number of Terminal Nodes Examined by SSS* 303 / 9.3.4 The Branching Factor of SSS* 304 / 9.3.5 Numerical Comparison of the Performances of α-β, SSS*, and SCOUT 000

9.4 The Benefit of Successor Ordering 310

9.5 Games with Random Number of Moves 318
9.5.1 The Distribution of the Value of the Game 318 / 9.5.2 Performance Analysis 322 / 9.5.3 Ordering Successors by Branching Degrees 324

9.6 Bibliographical and Historical Remarks 325
Exercises 326
Appendix 9-A: Proof of Theorem 1 327
Appendix 9-B: Proof of Theorem 6 329

10. Decision Quality in Game Searching 332

10.1 Error Propagation through Minimaxing 333
10.1.1 Error-Propagation for Bi-Valued Estimates and Binary Trees 333 / 10.1.2 Extensions to Multivalued Estimates and b-ary Trees 337 / 10.1.3 Limit-Points of (α_k,β_k) 340 / 10.1.4 The Effect of Searching Deeper 346

10.2 When Is Look-Ahead Beneficial? 347
10.2.1 Improved Visibility 347 / 10.2.2 The Effect of Dependencies 348 / 10.2.3 The Avoidance of Traps 350 / 10.2.4 Playing to Win versus Playing Correctly 357
Exercises 361

Bibliography 363

Glossary of Notation 371

Author Index 375

Subject Index 377

PART I

Problem-Solving Strategies and the Nature of Heuristic Information

Heuristics! Patient rules of thumb,
So often scorned: Sloppy! Dumb!
Yet, slowly, common sense become.

(ODE TO AI)

By the pricking of my thumbs,
Something wicked this way comes.

(MACBETH)

Chapter 1

Heuristics and Problem Representations

1.1 TYPICAL USES OF HEURISTICS IN PROBLEM SOLVING

Heuristics are criteria, methods, or principles for deciding which among several alternative courses of action promises to be the most effective in order to achieve some goal. They represent compromises between two requirements: the need to make such criteria simple and, at the same time, the desire to see them discriminate correctly between good and bad choices.

A heuristic may be a **rule of thumb** that is used to guide one's actions. For example, a popular method for choosing ripe cantaloupe involves pressing the spot on the candidate cantaloupe where it was attached to the plant, and then smelling the spot. If the spot smells like the inside of a cantaloupe, it is most probably ripe. This rule of thumb does not guarantee choosing only ripe cantaloupe, nor does it guarantee recognizing each ripe cantaloupe judged, but it is effective most of the time.

As an example of a less clear-cut use of heuristics, consider a chess grand master who is faced with a choice of several possible moves. The grand master may decide that a particular move is most effective because that move results in a board position that "appears" stronger than the positions resulting from the other moves. The criterion of "appearing" stronger is much simpler for the grand master to apply than, say, rigorously determining which move or moves forces a checkmate. The fact that grand masters do not always win indicates that their heuristics do not guarantee selecting the most effective move. Finally, when asked to describe their heuristics, grand masters can only give partial and rudimentary descriptions of what they themselves seem to apply so effortlessly.

Even the decision to begin reading this book reflects a tacit use of heuristics—heuristics that have led the reader to expect greater benefit from engaging in this activity rather than doing something else at this point in time.

It is the nature of good heuristics both that they provide a simple means of indicating which among several courses of action is to be preferred, and that they are not necessarily guaranteed to identify the most effective course of action, but do so sufficiently often.

Most complex problems require the evaluation of an immense number of possibilities to determine an exact solution. The time required to find an exact solution is often more than a lifetime. Heuristics play an effective role in such problems by indicating a way to reduce the number of evaluations and to obtain solutions within reasonable time constraints.

To illustrate the role of heuristics in problem solving, we make use of five puzzlelike problems which have served as expository tools in both psychology and artificial intelligence (AI). The expository power of puzzles and games stems from their combined *richness* and *simplicity*. If we were to use examples taken from complex real-life problems, it would take us more than a few pages just to lay the background and specify the situation in sufficient detail so that the reader could appreciate the solution method discussed. Moreover, even after making such an effort we cannot be sure that different readers, having accumulated different experiences, would not obtain diversely different perceptions of the problem situation described. Games and puzzles, on the other hand, can be stated with sufficient precision so as to lay a common ground for discussion using but a few sentences and, yet, their behavior is sufficiently rich and unpredictable to simulate the complexities encountered in real-life situations. This, in fact, is what makes games and puzzles so addictive.

1.1.1 The 8-Queens Problem

To solve this problem, one must place eight queens on a chess board such that no queen can attack another. This is equivalent to placing the queens so that no row, column, or diagonal contains more than one queen.

We can attempt to solve this problem in many ways, from consulting a mystic to looking up the solution in a puzzle book. However, a more constructive method of attack would be to forgo hopes of obtaining the final solution in one step and, instead, to attempt reaching a solution in an **incremental,** step-by-step manner, in much the same way that a treasure hunter "closes in" on a target following a series of local decisions, each based on new information gathered along the way. We can start, for example, with an arbitrary arrangement of eight queens on the board and transform the initial arrangement iteratively, going from one board configuration to another, until the eight queens are adequately dispersed. As with the treasure-hunt, however, we must also make sure that the sequence of transformations is not random but **systematic** so that we do not generate the same configuration over and over and so that we do not miss the opportunity of generating the desired configuration. The former is a requirement of efficiency, whereas the latter is one of utility.

One way to systematize the search is to attempt to **construct** (rather than transform) board configurations in a step-by-step manner. Starting with the empty board we may attempt to place the queens on the board one at a time, ruling out violations of the problem constraints, until a satisfactory configuration with eight queens is finally achieved. Moreover, because we can easily deduce from the problem specification that there must be exactly one queen in every row, we can assign the first queen to the first row, the second queen to the second row, and so on, thus reducing the number of alternative positions considered for each new queen to less than 8. The feature that allows us to systematize the search in this manner is the fact that we cannot recover from constraint violations by future operations. Once a partial configuration violates any of the problem constraints (i.e., it contains two or more attacking queens), it cannot be rectified by adding more queens to the board, and so many useless configurations can be eliminated at an early stage of the search process.

Assume now that at some stage of the search, the first three queens were positioned in the cells marked by Q's in Figure 1.1 and that we must decide where to position the fourth queen, in cell A, B, or C. The role of heuristics in this context would be to provide rule-of-thumb criteria for deciding, at least tentatively, which of the three positions appears to have the highest chance of leading to a satisfactory solution.

In devising such heuristics, we may reason that to be able to place all the eight queens we need to leave as many options as possible for future additions. That means we need to pay attention to the number of cells in the unfilled rows that remain unattacked by the previously placed queens. A candidate placement is to be preferred if it leaves the highest **number of unattacked cells** in the

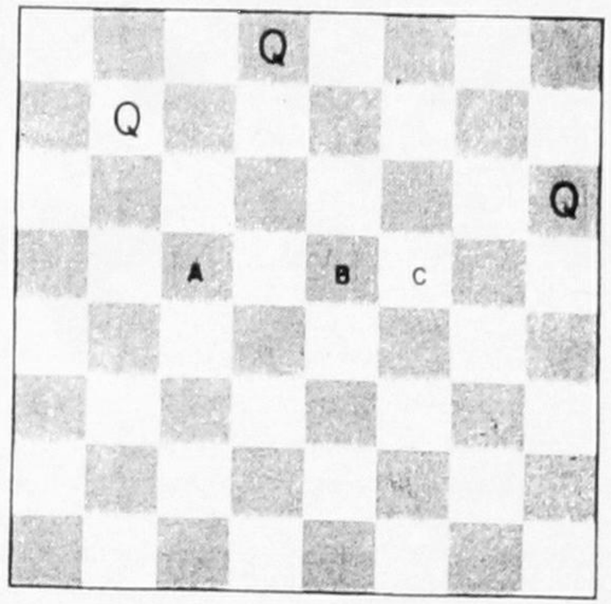

Figure 1.1
A typical decision step in constructing a solution to the 8-Queens problem.

remaining portion of the board. Consequently the number of unattacked cells left by any given alternative constitutes a measure, $f(\cdot)$, of no merit.

If we compute the function $f(\cdot)$ for the alternatives A, B, and C in the preceding example, we obtain

$$f(A) = 8$$

$$f(B) = 9$$

$$f(C) = 10$$

So alternative C should be our preferred choice. Adding credence to the effectiveness of our heuristic is the fact that alternatives B and C both lead to actual eight-queens solutions as shown in Figure 1.2, whereas alternative A leads to no such solution.

A more sophisticated heuristic can be devised by the following consideration. The unfilled rows do not have equal status since those with only few unattacked cells are likely to be "blocked" quicker than rows with many unattacked cells. Consequently, if we wish to minimize the chances of a future blockage, we should focus our attention on the row with the **least number of unattacked cells** and use this number, f', as a figure of merit of each option considered. Performing this computation for the options presented in Figure 1.1, we obtain

$$f'(A) = 1$$

$$f'(B) = 1$$

$$f'(C) = 2$$

which, again, identifies C as the preferred alternative.

Note that if some candidate option makes either f or f' evaluate to 0, there is no sense in pursuing that option any futher; eventually we will come to a **dead end** where no more queens can be added. The function f', however, offers some advantage over f in that it detects all the dead ends predicted by f and many more.

Obviously, for incremental search techniques like the preceding one, there may be cases in which such simplistic heuristic functions would assign the highest value to a futile search avenue. Quite often, however, they would guide the search in the right direction. Moreover, considering the fact that we can recover from bad advice by backtracking to previous decision junctions, the net effect of using such heuristics would be beneficial; they speed up the search without compromising the chances of finding a solution.

The main objective of the studies in this book is to understand and quantify the features of heuristics that make them effective in a given class of problems.

1.1.2 The 8-Puzzle

The objective of the 8-Puzzle is to rearrange a given initial configuration of eight numbered tiles arranged on a 3×3 board into a given final configuration called the **goal state**. The rearrangement is allowed to proceed only by sliding

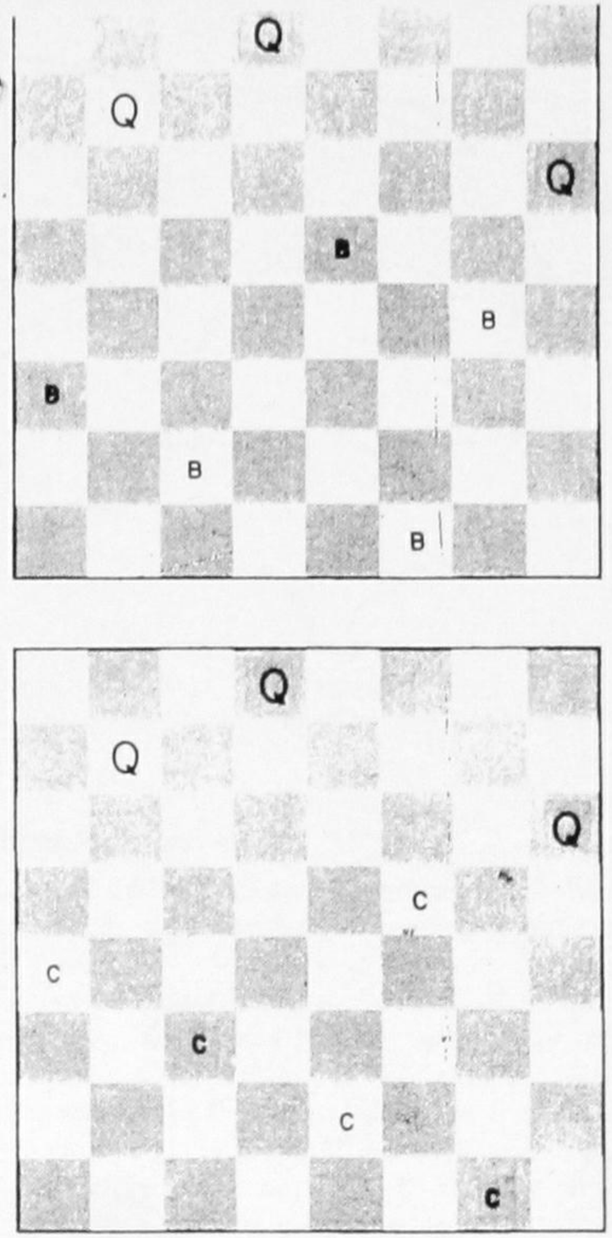

Figure 1.2
Two solutions to the 8-Queens problem extending the options shown in Figure 1.1.

one of the tiles onto the empty square from an orthogonally adjacent position, as shown in Figure 1.3

Which of the three alternatives A, B, or C appears most promising? Of course, the answer can be obtained by exhaustively searching subsequent moves in the puzzle to find out which of the three states leads to the shortest path to the goal. **Exhaustive search**, however, is utterly impractical when the number of states that must be examined is immense. This "combinatorial ex-

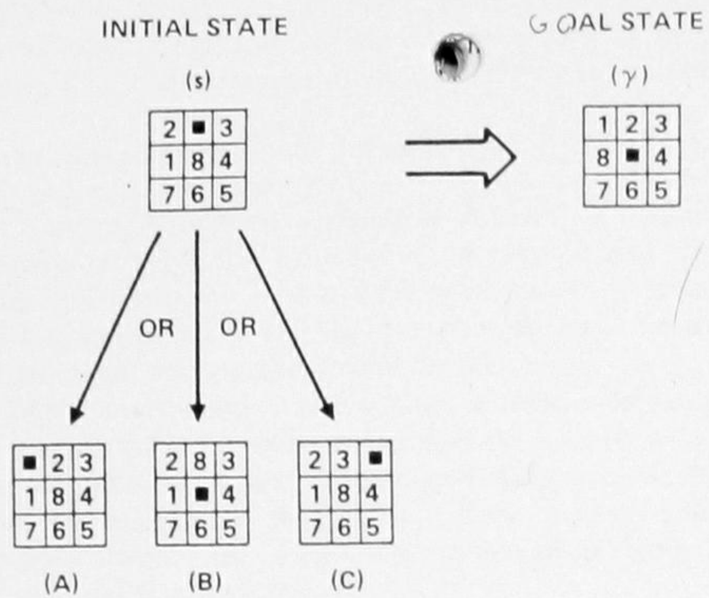

Figure 1.3
A graph description of the first move options in the 8-Puzzle.

plosion" occurs in such puzzles when the path length from the initial state to the goal state is large or when larger boards (e.g., 4×4) are involved. The decision to select the most promising alternative, therefore, cannot rely on exhaustive search, but rather it requires the use of **presearch judgment.**

One of the judgments that we might make in solving a path-finding problem is **estimating how close** a state, or configuration, is to the goal. In the 8-Puzzle there are two very common heuristics, or rules of thumb, that are used for estimating the proximity of one state to another. The first is the **number of mismatched tiles,** those by which the two states differ, which we will call heuristic function h_1. The second is the **sum of the** (horizontal and vertical) **distances** of the mismatched tiles between the two states, which we will call heuristic function h_2. This second heuristic function is also known as the **Manhattan** or **city-block distance.** To see how these heuristic functions work, below are the estimates for the proximities of the states A, B, and C from the goal state in Figure 1.3:

$$h_1(A) = 2 \quad h_1(B) = 3 \quad h_1(C) = 4$$

$$h_2(A) = 2 \quad h_2(B) = 4 \quad h_2(C) = 4$$

Evidently, both heuristics proclaim that state A is the one closest to the goal and that it should, therefore, be explored before B or C.

These heuristic functions are intuitively appealing and readily computable, and may be used to prune the space of possibilities in such a way that only configurations lying close to the solution path will actually be explored. Search strategies that exhibit such pruning through the use of heuristic functions will be discussed in Chapter 2.

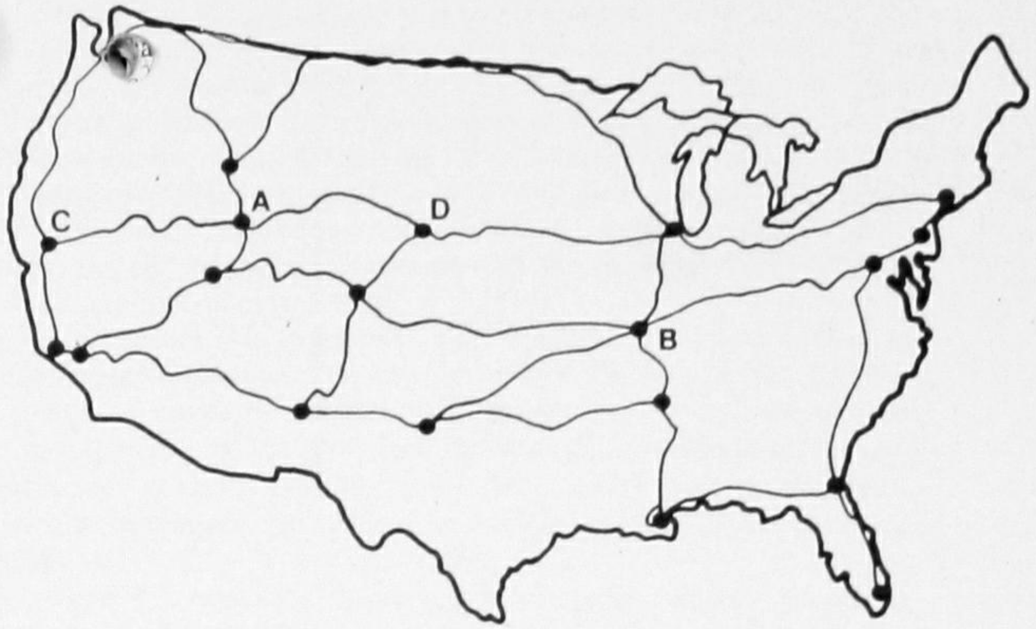

Figure 1.4
A road map, illustrating the use of heuristics in seeking the shortest path from A to B.

1.1.3 The Road Map Problem

Given a road map such as the one shown in Figure 1.4, we want to find the shortest path between city A and city B assuming that the length of each road on the map is specified by a label next to it.

If we are given an actual road map, then in looking for possible paths we would immediately rule out as unpromising those roads that lead away from the general direction of the destination. For example, the fact that city C, unlike city D, lies way out of the natural direction from A to B appears obvious to anyone who glances at the map in Figure 1.4, and so paths going through D will be explored before those containing C. On the other hand, if instead of the map, we are only given a numerical table of the distances $d(i, j)$ between the connected cities, the preference of D over C is not obvious at all; all cities situated at the same distance from A would appear equally meritorious. What extra information does the map provide that is not made explicit in the distance table?

One possible answer is that the human observer exploits vision machinery to estimate the **Euclidean distances** in the map and, since the air distance from D to B is shorter than that between C and B, city D appears as a more promising candidate from which to launch the search. In the absence of a map, we could attempt to simulate these visual computations by supplementing the distance table with extra data of equivalent informational value. For example, because we can easily compute air distances between cities from their coordinates, we can supplement the road-distance table with a heuristic function $h(i)$, which is

the air distance from city i to the goal city B. We can use this extra information to help estimate the merit of pursuing any candidate path from A to some city i, by simply adding $h(i)$ to the distance traversed so far by the candidate path.

For example, starting from city A in Figure 1.4 we find that we have to make a choice between pursuing the search from city C or city D. Since our aim is to minimize the overall path length, our choice should depend on the magnitude of the distance estimate $d(A, C) + h(C)$ relative to the estimate $d(A, D) + h(D)$. Here the respective paths involved are fairly trivial. In general, of course, we might wish to keep track of the shortest distances from the initial city to all the various cities that we have explored in the search.

In contrast to the 8-Queens problem, there is a strong similarity between the Road Map problem and the 8-Puzzle problem. In both the latter two we are asked to find a path from the initial state to some goal state, whereas in the 8-Queens problem we had to construct the goal state itself. However, the 8-Queens problem too was converted into a path-seeking task by our own decision to solve it by incremental search techniques.

1.1.4 The Traveling Salesman Problem (TSP)

Here we must find the cheapest tour, that is, the cheapest path that visits every node once and only once, and returns to the initial node, in a graph of N nodes with each edge assigned a nonnegative cost. The problem is usually stated for a complete graph, that is, one in which every node is linked to each of the other nodes.

The TSP is known to belong to a class of problems called **NP-hard** (Garey and Johnson, 1979) for which all known algorithms require an exponential time in the worst case. However, the use of good **bounding functions** often enables us to find the optimal tour in much less time. What is a bounding function?

Consider the graph in Figure 1.5 where the two paths marked ABC and AED represent two partial tours currently being considered by the search procedure. Which of the two, if properly completed to form a tour, is more likely

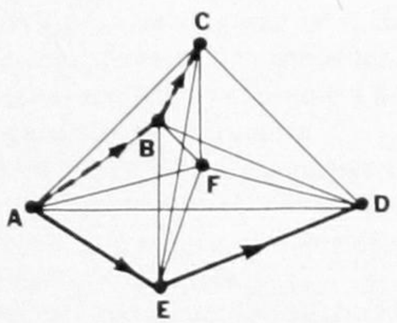

Figure 1.5

Two partial paths as candidates for solving the Traveling Salesman problem.

to be a part of the optimal solution? Clearly the overall solution cost is given by the cost of the initial subtour added to the cost of completing the tour, and so the answer lies in how cheaply we can complete the tour through the remaining nodes. However, since the problem of finding the optimal completion cost is almost as hard as finding the entire optimal tour, we must settle for a *heuristic estimate* of the completion cost. Given such estimates, the decision of which partial tour to extend first should depend on which one of them, after we combine the cost of the explored part with the estimate of its completion, offers a lower *overall* cost estimate.

It can be shown that, if at every stage we select for extension that partial tour with the lowest estimated complete tour cost and if the heuristic function is optimistic (that is, consistently underestimates the actual cost to complete a tour), then the **first** partial tour that is selected for extension and found to already be a complete tour is also a **cheapest** tour. What easily computable function would yield an optimistic, yet not too unrealistic, estimate of the partial tour completion cost?

People, when first asked to "invent" such a function, usually provide very easily computable, but much too simplistic answers. For example, as an estimate of the cost to complete a partial tour, take the cost of the edge from the current end of the partial tour to the initial node. As a marginally more complex example, take the cheapest two-edged path from the current end back to the initial node. These functions, although optimistic, grossly underestimate the completion cost.

Upon deeper thought, more realistic estimates are formed, and the two that have received the greatest attention in the literature are:

1. The cheapest **second degree graph** going through the remaining nodes;
2. The **minimum spanning tree** (MST) through all remaining nodes (Held and Karp, 1971)

The first is obtained by solving the so-called **optimal assignment problem** using on the order of N^3 computational steps, whereas the second requires on the order of N^2 steps.

That these two functions provide optimistic estimates of the completion cost is apparent when we consider that completing the tour requires the choice of a path going through all unvisited cities. A path is both a special case of a graph with degree 2 and also a special case of a spanning tree. Hence, the set of all completion paths is included in the sets of objects over which the optimization takes place, and so the solution found must have a lower cost than that obtained by optimizing over the set of paths only. Figure 1.6(a) shows the shape of a graph that may be found by solving the assignment problem instead of completing subtour AED. The completion part is not a single path but a collection of loops. Figure 1.6(b) shows a minimum-spanning-tree completion of subtour AED. In this case the object found is a tree containing a node with a degree higher than 2.

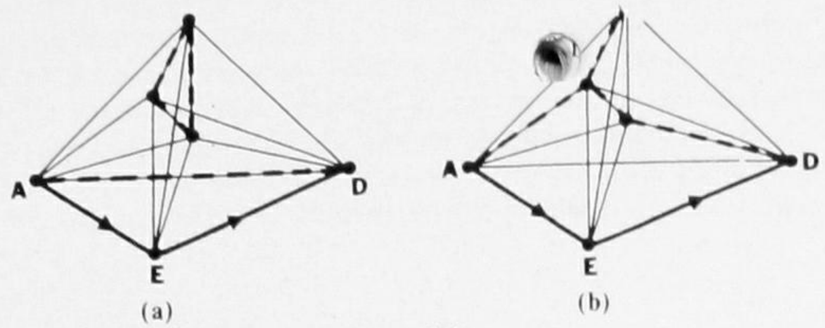

Figure 1.6

Two methods of optimistically estimating the completion cost of candidate $A \rightarrow E \rightarrow D$ in the Traveling Salesman problem of Figure 1.5. Part (*a*) The shortest degree-2 graph containing $A \rightarrow E \rightarrow D$. Part (*b*) The minimum spanning tree containing *A*, *D*, and all remaining (unlabeled) cities.

The shapes of the solutions found by these approximations may not resemble the desired tour, but the costs of these solutions may constitute good optimistic estimates of the optimal tour. Therefore they can be used as heuristics for assessing the relative merit of candidate subtours, thus improving the search efficiency. One of the questions addressed later in Chapter 6 is how the accuracy of such estimates affects the complexity of the search.

1.1.5 The Counterfeit Coin Problem

We are given 12 coins, one of which is known to be heavier or lighter than the rest. Using a two-pan scale, we must find the counterfeit coin and determine whether it is light or heavy in no more than three tests.

To solve this problem we must find a weighing strategy that guarantees that, regardless of the tests' outcomes, the counterfeit coin can be identified and characterized in at most three weighings. By **strategy** is meant a prescription of what to weigh first, what to weigh second for each possible outcome of the first weighing, what to weigh third for each possible outcome of each second weighing, and finally an indication of which coin is counterfeit and whether it is heavy or light for each possible outcome of each of the third weighings. In composing such a strategy we can try to construct it a piece at a time, much as we tackled the 8-Queens problem, by choosing a first weighing, then choosing a second weighing for each outcome, et cetera.

This problem can be solved by an exhaustive enumeration of all decisions and all possible test outcomes. The role of heuristics, however, is to focus attention on the most promising weighing strategies and to attempt to find a solution *without* exploring all possible strategies.

Consider the choice between the following two alternatives:

1. Start by weighing a pair of coins (one in each pan).
2. Start by weighing eight coins (four in each pan).

The second alternative appears superior even to a novice problem solver. The reason is that the first alternative has a high probability of ending up with a balanced scale, leaving us with the hard problem of finding a counterfeit coin among ten suspects using only two weighings. The second alternative, on the other hand, leads to more manageable problems: four suspects in case the scale balances and eight suspects each known to be "not-heavy" or "not-light" in case the scale tips.

As in the 8-Puzzle, the decision to defer the exploration of the first alternative in favor of the second can be obtained from a heuristic assessment of the effort required to complete the strategy from the first alternative relative to that of completing it from the second. For example, the base-3 logarithm of the number of remaining possibilities may provide an optimistic estimate for this effort, since every test may result in one of three possible outcomes: scale balances, scale tips to the right, and scale tips to the left.

Note, however, that every action in this problem requires an assessment of not one but three situations, one for each outcome of the balance. The merit of any action, therefore, should be computed by some combination of the merits assigned to the three situations created by that action. The **combination rule** depends on the problem statement. In our example, since the task is to be accomplished in *not more* than three weighings, the appropriate rule would be to assess any action by the effort required to solve the *hardest* of the three problem situations that may result from that action. On the other hand, if we were given an upper limit on the *average* number of weighings, a more appropriate rule would be to take the *expected value* of the effort estimates associated with those three problem situations.

Another difference between the 8-Puzzle and the Counterfeit Coin problem is that in the former we were only concerned with the issue of which action to explore next. In the Counterfeit Coin problem, we also have to address the following question: given that we decide to explore alternative 2 before alternative 1, which **subproblem** deserves our attention next; the easy one, containing four suspected coins, or the hard one, with the eight coins? Moreover, once we shed additional light on the chosen subproblem, should we then attend to its sibling or reevaluate the parent action to test whether it is still more promising than action 1 which we left unexplored? Deciding control issues of this kind is another role heuristics play in computer-based problem solving.

SUMMARY

Each of the five preceding example problems presents us with repetitive choices among a multitude of exploration paths that might be taken to reach a solution. In each case, we have exhibited a heuristic function that serves to indicate which among all those paths seem the most likely to lead to a solution. The heuristics are simple to calculate relative to the complexity of finding a solution and, although they do not necessarily always guide the search in the

correct direction, they quite often do. Moreover, whenever they do not, we can resort to some recovery scheme to resume the search from a different point. The key is that these heuristics save time by avoiding *many* futile explorations.

But how do you find a good heuristic, and when you have found one, how should you use it most effectively and how would you evaluate its merit?

The main objective of this book is to try to answer such questions and to quantify the characteristics of heuristics that produce the most beneficial results.

1.2 SEARCH SPACES AND PROBLEM REPRESENTATIONS

Each of the five examples discussed in the preceding section represents a class of problems with unique characteristics that may require a different representation or a different problem-solving strategy. Some of these characteristics are classified and discussed in the following sections.

1.2.1 Optimizing, Satisficing, and Semi-Optimizing Tasks

The Road Map and the Traveling Salesman problems differ from the other three examples in Section 1.1 in that the former require **optimization;** the path or tour sought had to be at least as cheap as any other feasible path or tour. In all five cases the problem statement delineated a space of candidate objects in which a solution is to be found. The type of objects sought varied from problem to problem; for the 8-Queens problem we were seeking a board configuration with certain properties, for the 8-Puzzle a sequence of moves, a path for the Road Map problem, a tour for the TSP, and a strategy of actions for the Counterfeit Coin problem. In the two optimization problems, however, the objective has been not merely to exhibit a formal object satisfying an established set of criteria but also to ascertain that it possesses qualities unmatched by any other object in the candidate space. By contrast, what was desired in the other three problems was to discover any qualified object with as little search effort as possible, a task we call **satisficing** (originally coined by H. Simon, 1956.)

Most problems can be posed both as constraint-satisfaction and as optimization tasks. For example, the 8-Queens problem can be stated as follows: find the cheapest placement of eight queens on a chess board where each capturable queen introduces a cost of one unit. Posed in this manner, the problem generally becomes harder since we do not know *a priori* whether a zero-cost solution exists. Moreover, even if we prove that such a solution does not exist, the work is not over yet; we still have to search higher cost configurations for the optimal solution. However, heuristic information pertaining to the *quality* of the solution may also be helpful in reducing the effort of finding one, even in cases in which the quality of the solution is of minor importance. In theorem proving, for example, we would normally be satisfied with the first available proof, however inelegant. Nevertheless, by focusing the search effort toward finding the *shortest* proof, we also install safeguards against aimless explorations in a large space of possibilities, and this normally leads to a smaller search effort in finding a proof. This beneficial principle, which we call **small-is-quick**, plays a major role in heuristic methods for satisficing search. In a later chapter we also demonstrate that algorithms driven by an auxiliary optimization objective (smallness) may exhibit the most efficient (quickest) search even when we know in advance that only one solution exists, or that just any solution will do.

Often the difference in complexity between an optimization task and its satisficing counterpart is very substantial. The TSP is an example of such a case; finding some tour through a set of cities is trivial, but finding an optimal tour is NP-hard. In such cases, one is often forced to relax the optimality requirement and settle for finding a "good" solution using only a reasonable amount of search effort. Whenever the acceptance criterion tolerates a neighborhood about the optimal solution, a **semi-optimization** problem ensues.

Semi-optimization problems fall into one of two categories. If the boundaries of the acceptance neighborhood are sharply defined (e.g., requiring that the cost of the solution found be within a specified factor of the optimal cost), the task is called **near-optimization.** If we further relax the acceptance criterion and insist only that the solution be near optimal with sufficiently high probability, we have an **approximate-optimization** task on our hands.

Most practical problems are of a semi-optimization type, requiring some reasonable balance between the quality of the solution found and the cost of searching for such a solution. Moreover, because the search effort required for many combinatorial optimization problems can easily reach astronomical figures, relaxing optimality is an economic necessity. The basic problem in handling a semi-optimization task is to devise algorithms that guarantee bounds on both the search effort and the extent to which the optimization objective is compromised. A more ambitious task would be to equip such an algorithm with a set of adjustable parameters so that the user can meet changes in emphasis between cost and performance by controlling the tradeoff between the quality of the solution and the amount of search effort.

1.2.2 Systematic Search and the Split-and-Prune Paradigm

Since every problem-solving activity can be regarded as the task of finding or constructing an object with given characteristics, the most rudimentary requirements for computer-based problem solving are:

1. A **symbol structure** or **code** which can represent each candidate object in the space

2. Computational **tools** that are capable of **transforming** the encoding of one object to that of another in order to scan the space of candidate objects systematically
3. An effective method of **scheduling** these transformations so as to produce the desired object as quickly as possible

In the nomenclature common to the artificial intelligence literature, these three components of problem-solving systems are known as

1. The **database**
2. The **operators** or **production rules**
3. The **control strategy**

A control strategy is said to be **systematic** if it complies with the following colloquially stated directives:

1. Do not leave any stone unturned.
2. Do not turn any stone more than once.

The first requirement, called **completeness,** guarantees that we do not miss the opportunity to generate an encoding that represents the desired solution if one exists. The second protects us from the inefficiency of repetitious computations.

Let us first examine what kind of objects the code must be capable of **representing.** Obviously we should have the capabilities of representing those objects that we suspect may themselves constitute the desired solutions to the problem at hand, such as board configurations, paths and strategies. But a code limited to expressing only the final objects will severely restrict the flexibility of search strategies. Although we would still be able to comply with the two basic directives of systemization, such an encoding will prevent us from taking advantage of some useful structural properties of the problem domain.

To illustrate this point, imagine the task of searching for the street address of a telephone number in a typical telephone directory. If we are given only the number, we have no alternative but to scan the book exhaustively. By contrast, if the owner's name is also given, the address can be located in just a few operations. In both cases, the book contains a code for every candidate object, but in the latter case we also use **codes for large subsets of objects.** If we open the directory at a page past the name we seek, we immediately recognize that all the names in the following pages could not qualify for a solution and this entire subset of objects is removed from future considerations by turning the pages back or by placing a book marker in the page just opened. The ability to eliminate or prune large chunks of the candidate space does not exist when we are given just the phone number. True, we can still encode large subsets of objects which we may wish to eliminate, e.g., the set of all phone numbers greater than the one sought, but there is no way to guarantee that future transformations (i.e., page turning operations) will not regenerate elements from the subset we wished eliminated.

This simple example illustrates the need to equip our encoding scheme with facilities to express and manipulate not merely individual solution objects but also **subsets of potential solutions.** Moreover, if we are permitted to apply code-transformations that take us from one subset to another, we should also apply the two requirements of systemization to subsets of potential solutions. The first requirement simply states that all individuals must be included in the collection of subsets expressible in our code, and that each individual be reachable by some operations on the subsets to which it belongs. The second requirement, on the other hand, insists that if subset S_1 has been eliminated from consideration, subsequent operations on other subsets will not generate any member of S_1. Putting these two requirements together implies that the only transformation on subsets we should permit is that of **splitting**. Indeed, by subjecting a given subset to repeated splitting operations, we may eventually reach any individual member of the subset and, at the same time, we cannot generate a new individual not originally included in the parent subset.

Before elaborating on additional features of this **split-and-prune** method, it is instructive to illustrate its basic building blocks in the context of the problems presented in Section 1.1.

In the **8-Queens** problem the individual objects capable of qualifying as solutions can be taken to include all 8×8 board configurations containing eight queens. Upon further reflection, we may narrow down the candidate space by considering only those configurations containing one queen in every row. A partially filled board, like that shown in Figure 1.1, constitutes a code for a *subset of potential solutions* in that it stands for all board configurations whose top three rows coincide with those filled in the partially specified arrangement. Adding a queen to the fourth row further *refines* the parent subset as it constrains the elements represented by the resulting subset to comply with a board of four instead of three specified rows. The three candidate search paths (represented by the letters A, B, and C in Figure 1.1) represent *splitting* of the parent set into a group of three mutually exclusive subsets, each suspected of containing a solution. Note that the subsets corresponding to the remaining five cells on the fourth row need not be considered since they could not possibly contain a legitimate solution.

Our ability to detect and eliminate or **prune** such large subsets of objects at such an early stage of the search is what makes the 8-Queens problem easy to solve. This ability was facilitated partly through the splitting method and partly by the fortunate choice of code. The splitting method guarantees that subsequent operations (i.e., adding new queens to the board) will not regenerate any member of subsets rejected earlier and that no potential solution is overlooked in this process. To appreciate the power of the split-and-prune method, the reader may do well to examine an alternative representation for the 8-Queens problem, one that does not allow splitting. Imagine, for example, that someone decides to solve this puzzle by placing eight queens in some random initial po-

sition and then perturb the original position iteratively by moving one queen at a time. In this representation we do have a code for individual objects but not a code for representing subsets of objects. Consequently, as in the case of trying to find an address in a telephone directory without a name, we can reject individual objects but not large chunks of objects.

The availability of an effective code for subset pruning was also significant in facilitating an easy solution to the 8-Queens problem. Choosing a partially filled board as a code for subsets of potential solutions was fortunate because it provided an early detection of some subsets containing no solution (e.g., if two queens in the partially filled board attack each other). The subset code itself, in this case, already possesses qualities of the final solution upon which the final acceptance tests can be performed, and hence early detections of some dead-end subsets are obtained quickly. To appreciate the importance of code-selection, let us now consider an example in which subset splitting is possible, but the code available for representing subsets of potential solutions does not facilitate an early detection of dead ends.

In the **8-Puzzle**, the individual search objects are sequences of legal transformations starting with the initial configuration s. The solution objects are those sequences that lead from s to the goal configuration. A given chain of moves encodes the subset of potential solutions, which includes all those sequences beginning with the chosen chain. Each time we add a new move to a given chain, we narrow the subset of solutions represented by the chain, and since the subsets represented by all legal moves applicable to a given position are mutually exclusive, we are justified in calling the act of listing the consequences of these moves "splitting."

Again, the splitting method theoretically permits us to prune any subset that is proved hopeless, but in the 8-Puzzle no subset is truly hopeless. Subsequent operations can always undo previous operations and place us back on a path to the goal if such a path exists. By contrast, if we change the original problem specifications and require that the goal state be achieved in *no less than K moves,* some subsets of potential solutions do become hopeless. In fact, every sequence of K moves that does not lead to the goal position can simply be discarded because it cannot be turned into a legitimate solution by adding more moves to it. But can we predict this dead-endness early enough simply by looking at an initial segment of such a sequence? Unfortunately we know of no code that would emit early warning signals so vividly as those obtained in the 8-Queens problem. To produce such early warnings, we must, instead, resort to heuristic information that is not explicitly mentioned in the initial problem statement. For example, using heuristic functions such as h_1 and h_2, we can prune away sequences of length shorter than K. Every sequence of $K - h_1(n)$ (or $K - h_2(n)$) moves not leading to the goal can now be discarded, where n is the last position resulting from the sequence.

The use of heuristic functions imposes additional requirements on the **format** of our chosen **code.** We now must make sure that the code we choose for representing candidate subsets not only identifies these subsets unambiguously, but also facilitates the easy computation of such heuristics. For example, if we choose to codify subsets only by the sequence of splits (or moves) that produced them, such as "up, left, right, down, . . . ," then the computation of h_1 will be very cumbersome. If, on the other hand, we also maintain a (redundant) description of the **state** prevailing after such a sequence, the computation becomes easy. We simply count the number of tiles by which the resulting configuration differs from the goal configuration. And if we also maintain a list of the differing tiles, the computation becomes easier still.

The **Counterfeit Coin** problem introduces a new element. The problem statement requests an identification of a weighing **strategy** that meets certain conditions. A strategy is not simply a sequence of actions but a prescription for choosing an action in response to any possible external event such as an outcome of a test, an action of an adversary, or the output of a complicated computational procedure. Thus, whereas the search objects in the first four examples of Section 1.1 were represented as **paths,** those in the Counterfeit Coin problems are **trees.** A solution to the problem is a tree with depth of at most three which includes all possible outcomes to any of the chosen tests. In addition, each path in the tree must represent a sequence of tests and outcomes that identifies the counterfeit coin unambiguously. To handle such problems, we need, of course, a code for representing subsets of potential solution strategies. A convenient code for such problems is provided by AND/OR graphs, as we will see in Section 1.2.4.

The idea of viewing problem solving as a process of repetitive splitting (also called **branching** or **refinement**) of subsets of potential solutions became popular in operations research and is the basic metaphor behind the celebrated branch-and-bound method (Lawler and Wood, 1966). Artificial intelligence literature rarely mentions this metaphor, preferring to describe problem solving as a "**generate-and-test**" process (Newell and Simon, 1972), **creating** new objects rather than **eliminating** objects from preexisting sets. An artificial intelligence researcher would view a partially developed path, for example, as a building block in the construction of a more elaborate complete solution that will *contain* that path. An operations researcher, on the other hand, views the partial path as (a code for) a huge subset of potential solutions that should eventually be trimmed down to a single element. Note that the use of the word "contain" is reversed in the latter paradigm; the partial path is viewed as an elaborate subset that in itself *contains* complete solutions as structureless points.

The difference between these two approaches is not accidental but reveals a sharp difference of emphasis between the two communities. The split-and-prune paradigm emphasizes the commonality of problem-solving methods over many problems, whereas the object creation paradigm focuses on the unique structure featured by a specific symbolic representation of a given problem. The operations research approach is particularly effective in establishing theoretical guarantees such as completeness and optimality. The artificial intelligence community, on the other hand, has always emphasized the heuristic and implementational aspects of problem solving and its parity with human

style over its mathematical foundations. Indeed, in human terms, the acts of adding a new queen to the board (in the 8-Queens problem), extending a tour to pass through one more city (in the Traveling Salesman problem), or deriving a new lemma in theorem proving are definitely perceived more as acts of *creation* and *construction* than acts of *pruning* or *deletion*.

We will use both paradigms interchangeably. The generate-and-test metaphor will be used to *describe* the various steps used in search procedures, whereas the split-and-prune metaphor will mainly be invoked to *justify* them.

1.2.3 State-Space Representation

Returning to the method of subset splitting, the three basic components needed for the method to work are:

1. A symbol structure called a **code** or a **database** that represents subsets of potential solutions (henceforth called **candidate subsets**)
2. A set of operations or **production rules** that modify the symbols in the database, so as to produce more refined sets of potential solutions
3. A search procedure or **control strategy** that decides at any given time which operation is to be applied to the database

If storage space permits, the database should include codes for several candidate subsets simultaneously. This allows the control strategy to decide which subset would be the most promising to split. When memory space is scarce, the database at any given time may contain a code of only one subset with provisions for backtracking and generating alternate refinements of the parent subset in case a solution cannot be found by a given sequence of refinements.

We may now impose **additional requirements on the format** of the code assigned to a given candidate subset. In principle, since a given sequence of refinement operators corresponds to one and only one candidate subset, that sequence in itself could constitute a sufficient parsimonious code for the resultant subset. We have seen in the 8-Puzzle example, however, that such a code would be very awkward. Although this code is adequate for identifying the candidate subset and for protecting against violations of the two rules of systematic search, it tells us nothing about the nature of the problem which remains to be solved (i.e., finding a solution among the members of the encoded subset). The sequence *s*-up-left-up-right, for example, uniquely specifies one tile-configuration in the 8-Puzzle but contains no explicit information regarding the resultant configuration, its proximity to the goal, whether it has been encountered before in some other candidate subsets, or even which operators are next applicable.

For these reasons we find it convenient to include in the code assigned to each candidate subset additional information that **explicitly defines the remaining subproblem** presented by the subset. The code specifying this additional information is called **a state**. The set of all subproblems obtainable by executing some sequence of refinement operators from a given position is called a **state-space**. If we connect the elements of this space by arcs labeled by the appropriate refinement operators, we obtain a **state-space graph**.

To illustrate this, a typical candidate subset code for the Traveling Salesman problem might look as follows:

$$\underbrace{A \rightarrow B \rightarrow C \rightarrow D}_{\text{candidate subset}} \underbrace{\rightarrow \{E, F\} \rightarrow A}_{\text{state}}$$

The first sequence, $A \rightarrow B \rightarrow C \rightarrow D$, identifies the subset of all tours that begin at city A and then proceed through cities B, C, and D. The part marked "state" specifies the problem of completing the tour: find a path beginning with D, going once through each city in the set $\{E, F\}$ and finally terminating at A. Note that the former is both necessary and sufficient, whereas the latter part is **redundant** and **incomplete**; that is, a solution of the subproblem represented by the state will not identify the entire tour. A subtour description, on the other hand, completely specifies the remaining problem.

The advantages of keeping the state code explicit are apparent. It is this portion of the code that expedites the computation of the heuristic estimates discussed in Section 1.1.4. Moreover, suppose at a certain state of the search, the database also contains the code:

$$\underbrace{A \rightarrow C \rightarrow B \rightarrow D}_{\text{candidate subset}} \underbrace{\rightarrow \{E, F\} \rightarrow A}_{\text{state}}$$

Although the candidate subset represented by this code is different from the preceding one, their states are identical, so we can determine that one subset of candidates is superior to the other without solving the state subproblem. Simply by comparing the costs of the two paths from A to D, we can prune the inferior subset from future consideration. This type of pruning is called **pruning by dominance** (Ibaraki, 1977) and can be shown to result in substantial savings of search effort.

1.2.4 Problem-Reduction Representations and AND/OR Graphs

We have seen that both constraint-satisfaction problems and sequence-seeking problems can be represented as tasks of finding paths in a state-space graph. Other problems, like the strategy-seeking task in the Counterfeit Coin problem, may be more conveniently represented by a technique called **problem-reduction** representation. The unique feature of this class of problems is that each residual problem posed by some subset of candidates can be viewed as a **conjunction of several subproblems** that may be solved independently of each

other, may require specialized treatments, and should, therefore, be represented as separate entities.

To illustrate this point let us return to the Counterfeit Coin problem of Section 1.1.4 and assume that we have chosen the first alternative, that is, start by weighing a single pair. If we are lucky and the balance tips one way or another, we are left with the easy problem of finding which of the two suspects is the counterfeit coin, knowing that the remaining ten coins are honest. This simple problem can, of course, be solved in a single step, weighing one of the suspects against one of the honest coins. The thing to notice, though, is that this two-suspect problem can be **solved in isolation**, independently of the problems that may ensue in case the balance remains neutral. Moreover, the same simple problem will appear frequently at the end of many elaborate strategies, and so, if we recognize it as an individual entity deserving an individual code, the solution to this subproblem can be found once, remembered, and shared among all strategies that lead to it.

This suggests that we treat **subproblems as individual nodes** in some graph, even though none of these subproblems alone can constitute a complete solution to our original problem. In other words, since every action in our example introduces a triplet of subproblems, all of which must be solved, we prefer to represent the situation resulting from an action not as a single node containing a list of subproblems but rather as a triplet of individual nodes, each residing atop its own specialized set of candidate solution strategies. Figure 1.7 illustrates this problem-reduction representation, contrasting it with the state-space representation of Figure 1.8.

Note that although the individual subproblems are assigned separate nodes, they are still bound by the requirement that *all* must be solved before the parent problem is considered solved. The purpose of the curved arcs in Figure 1.7 is to remind us of this fact. Thus the search-space graph in this representation consists of two types of links:

1. Links that represent alternative approaches in handling the problem node from which they emanate.
2. Links that connect a parent problem node to the individual subproblems of which it is composed.

The former are identical to the links in state-space graphs and are called **OR links**. The latter are unique to problem-reduction and are called **AND links**.

All nodes in an AND/OR graph, like those in a state-space graph, represent subproblems to be solved or subgoals to be achieved, and the start node represents the specification of the original problem. Likewise, each path emanating from the start node identifies a unique candidate subset containing those strategies that comply with the sequence of event-response pairs specified by the path. However, unlike paths in state-space, a solution to the problem represented by the tip node of such a path is no longer sufficient for synthesizing a solution to the original problem. The complete solution is represented by an AND/OR subgraph called a **solution graph**, which has the following properties:

1. It contains the start node.
2. All its terminal nodes (nodes with no successors) are primitive problems, known to be solved.
3. If it contains an AND link L, it must also contain the entire group of AND links that are siblings of L.

An AND/OR graph and two of its solution graphs are illustrated in Figure 1.9.

It is sometimes convenient to regard AND/OR graphs as generalizations of ordinary graphs called **hypergraphs**. Instead of links that connect pairs of

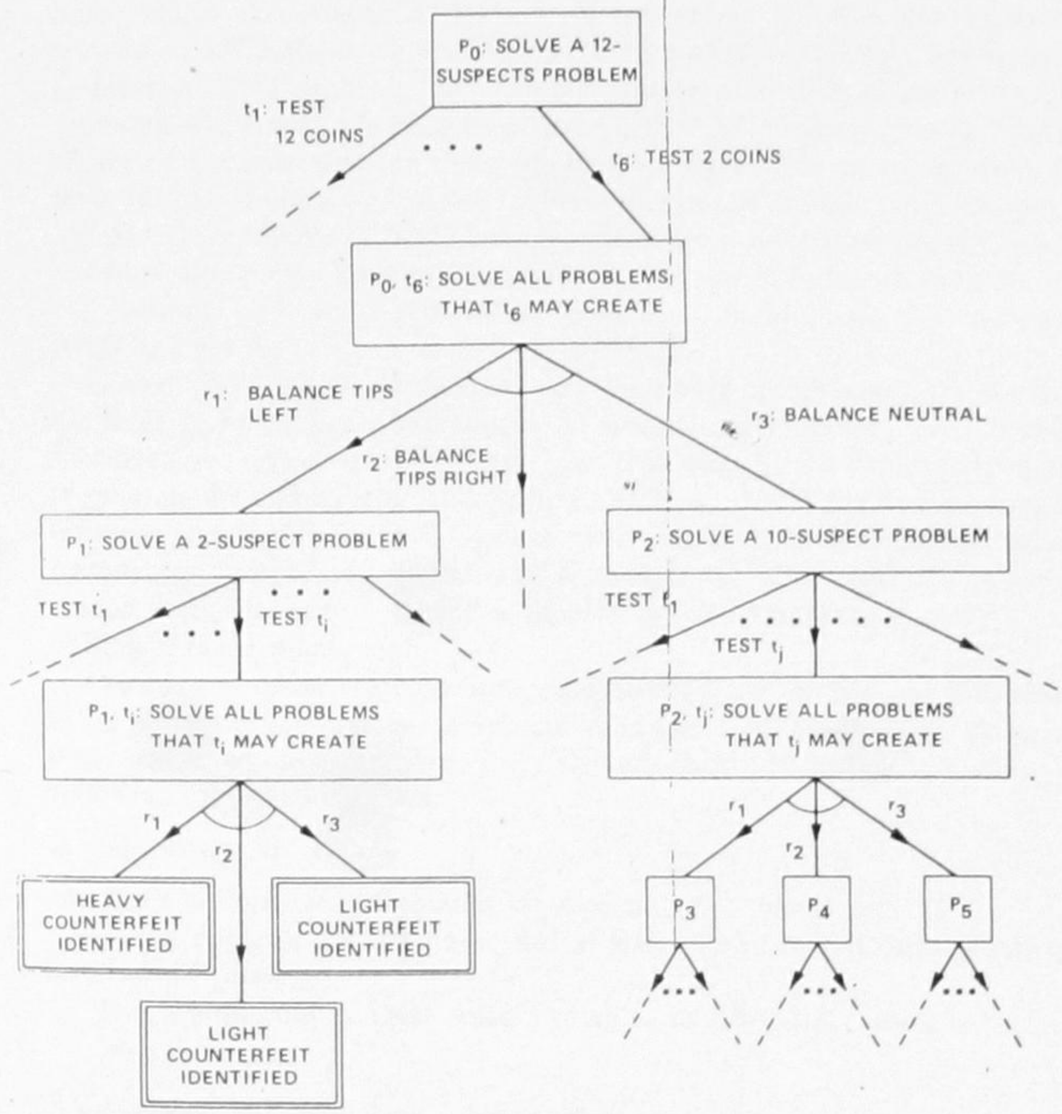

Figure 1.7
Problem-reduction representation for the Counterfeit Coin problem. Each node stands for one subproblem, and the curved arcs indicate which sets of subproblems must be solved conjunctively to make up a complete solution.

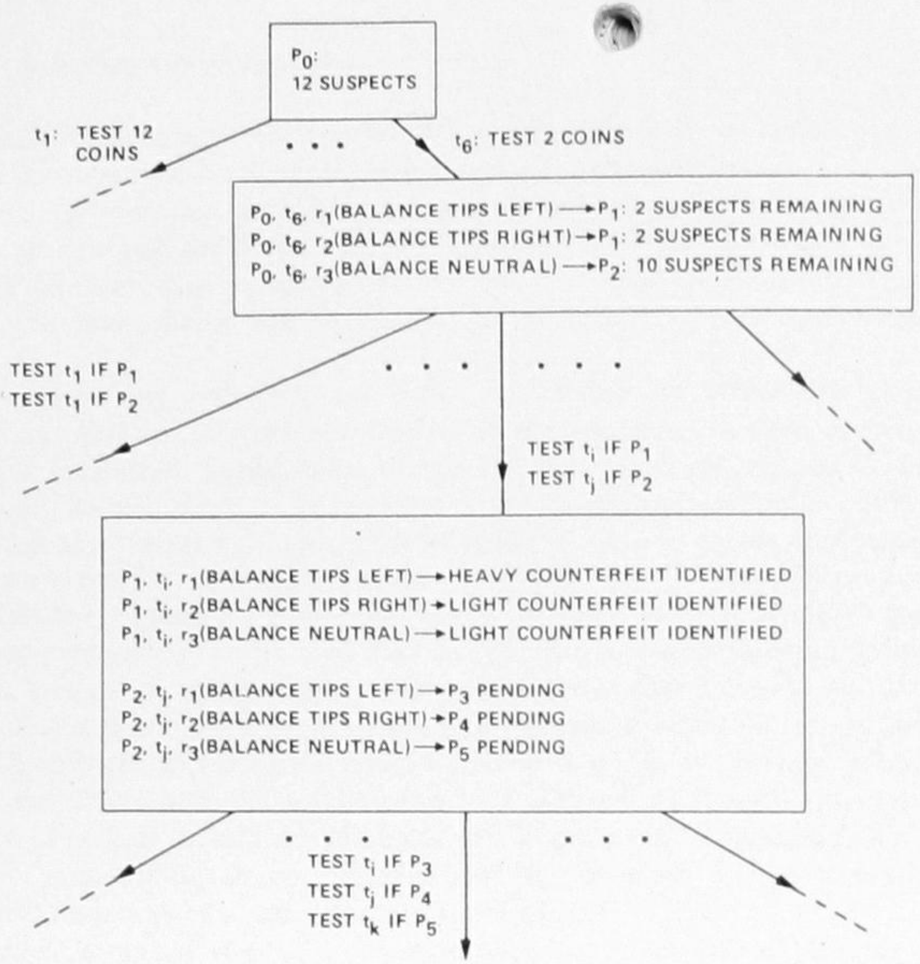

Figure 1.8

State-space representation for the Counterfeit Coin problem. Each node represents the set of all subproblems that may result from the policy (path) leading to that node.

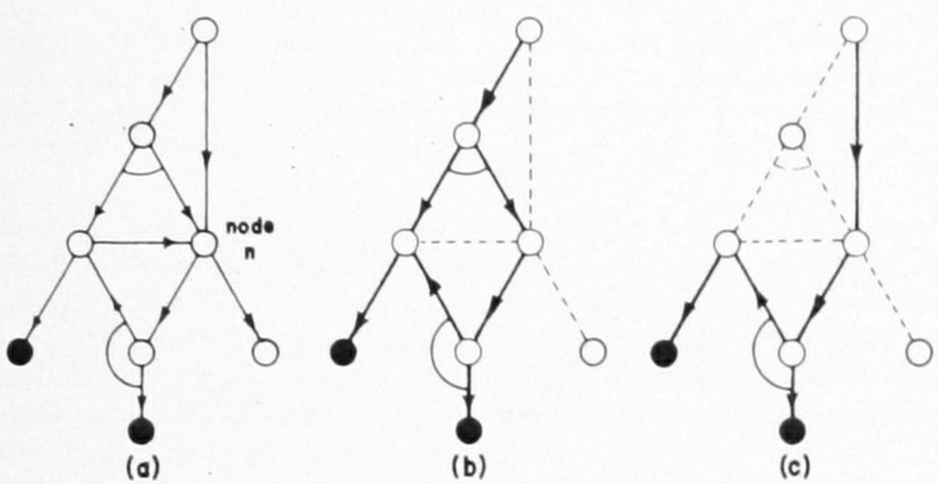

Figure 1.9

An AND/OR graph (*a*) and two of its solution graphs (*b*) and (*c*). Terminal nodes are marked as black dots.

nodes, hypergraphs may contain hyperlinks, called **connectors**, connecting a parent node with a *set* of successor nodes. If we regard each group of AND links as a single connector or hyperlink, the task of finding a solution graph amounts to that of finding a **hyperpath** between the start node and the set of primitive terminal nodes. This analogy is helpful in making concepts and techniques developed for path-finding problems applicable to problem-reduction representations.

In general, an AND link and an OR link may simultaneously point to the same node in an AND/OR graph, such as the node marked (n) in Figure 1.9. In this figure, however, each node issues either OR links or AND links, but not both. When this condition holds, it is convenient to label the nodes **OR nodes** and **AND nodes**, depending on the type of links that they issue. In the Counterfeit Coin problem, for example, this separation is natural; OR links represent the strategist's choices of tests, and AND links represent Nature's choice of test outcomes. In cases where a node gives rise to a mixture of OR links and AND links, we can still maintain the purity of node types by imagining that every bundle of AND links emanates from a separate (dummy) AND node and that these dummy AND nodes are OR linked to the parent node. We will assume this node separation scheme throughout our discussion of AND/OR graphs.

In most problems of practical interest, the entire search-space graph will be too large to be explicitly represented in a computer's memory. The problem specifications, however, will contain rules that will allow a computer program to generate the graph incrementally from the start node in any desired direction thus constructing an explicit representation of a portion of the underlying search-space graph, keeping most of it unexplored. The objective of a well-guided search strategy is to find a solution graph by explicating only a small portion of the implicit graph.

At any stage of the search, a test could be conducted to find out if the explicated portion of the searched graph contains a solution graph. That test is best described as an attempt to *label* the start node "solved" through the use of the following **solve-labeling rules:**

A node is "solved" if:

1. It is a terminal node (representing a primitive problem); or
2. It is a nonterminal OR node, and at least one of its OR links points to a "solved" node, or
3. It is a nonterminal AND node, and all of its AND links point to "solved" nodes.

Equivalently, the explicated portion of the search-graph may contain nodes recognizable as "unsolvable," in which case it is worth testing whether the start node still stands a chance of getting solved by attempting to label it "unsolvable." The labeling rules for "unsolvable" status are similar to those of "solve" except that the roles of AND and OR links are interchanged.

1.2.5 Selecting a Representation

The selection of a representation to fit a given problem is sometimes dictated by the problem specifications and sometimes is a matter of choice.

Generally speaking, problem-reduction representations are more suited to problems in which the final solution is conveniently represented as a tree (or graph) structure. State-space representations are more suited to problems in which the final solution can be specified in the form of a path or as a single node.

Strategy-seeking problems are better represented by AND/OR graphs, where the AND links represent changes in the problem situation caused by external, **uncontrolled conditions** and the OR links represent alternative ways of reacting to such changes. In our Counterfeit Coin problem, the uncontrolled condition was Nature's choice of the weighing results. In games, those conditions are created by the legal moves available to the adversary. In program-synthesis, they consist of the set of outcomes that may result from applying certain computations to unspecified data. In general, a strategy-seeking problem can be recognized by the **format required for presenting the solution.** If the solution is a prescription for responding to observations, we have a strategy-seeking problem on our hands, and an AND/OR graph will be most suitable. If, on the other hand, the solution can be expressed as an unconditional sequence of actions or as a single object with a given set of characteristics, we have a path-finding or constraint-satisfaction problem and a state-space representation may be equally or more appropriate.

Another important class of problems suitable for AND/OR graph representations includes cases in which the solution required is a **partially ordered sequence** of actions. This occurs when each available operator replaces a single component of the database by a set of new components. Take, for example, the following **set-splitting** problem:

> Given a set of numbers, find the cheapest way to separate this set into its individual elements using a sequence of partitioning operators. Each operator partitions one subset into two parts, and the cost of each partition is equal to the sum of the elements in the partitioned subset.

It is not hard to show that the solution to this problem is given by the celebrated Huffman-Code algorithm (Horowitz and Sahni, 1978). However, the feature that makes this example worth mentioning here is that the two subsets created by each partitioning operator can later be handled *in any desired order,* without affecting the cost of the solution. Hence, it would be more natural to represent each candidate solution as an **action-tree**, rather than as an action sequence. Thus, although the solution sought is not a strategy in the strict sense of prescribing a policy of reacting to uncontrolled conditions, it is still in tree form, and, therefore, the problem is well suited for AND/OR graph representation.

A similar structure is possessed by **symbolic-integration** problems (Moses, 1971). Several legal transformations are available in this domain (e.g., integration by parts, long division, etc.) which will split the integrand into a sum of expressions to be integrated separately *in any order.* The set of applicable transformations will be represented as OR links emanating from the node representing the integrand, and the AND links represent individual summands within the integrand, all of which must eventually be integrated.

The tasks of **logical reasoning** and **theorem proving** also give rise to AND/OR structures. We begin with a set of axioms and a set of inference rules that allow us, in each step, to deduce a new statement from a subset of axioms and previously deduced statements. The new statement is added to the database, and the process continues until the desired conclusion (e.g., the theorem) is derived. The solution object pursued by the search is a plan specifying in each step which of the inference rules is to be applied to which subset of statements in the database and what the deduced statement is. This plan, again, is best structured as an **unordered tree** because, when a certain conclusion is derived from a given subset of statements, the internal order in which these statements were themselves derived is of no consequence as long as they reside in the database at the appropriate time. Thus, the solution structure is a tree, and hence, the appropriate search-space would be an AND/OR graph.

In certain cases both state-space and problem-reduction formulations appear equally appropriate and the choice of representation requires additional insight. Take, for instance, the 8-Puzzle (Section 1.1.2) and assume that we wish to formulate it in a problem-reduction representation. Whereas in the state-space approach each tile configuration stood for the problem of transforming the configuration into the goal state, we may attempt now to decompose this problem into a set of more elementary problems. We may regard each condition required to meet the goal criterion as a subproblem on its own right (e.g., tile x_1 on cell c_1, tile x_2 on cell c_2, etc.). However, the highly interdependent nature of these subgoals severely weakens the advantages of the problem-reduction approach. As soon as we accomplish some of these subgoals, we find that they must be violated (at least temporarily) in order to achieve the remaining subgoals. Moreover, the solution found for achieving a given subgoal is **context dependent** and can no longer be saved for future use as we have done in the Counterfeit Coin problem. We conclude, therefore, that a problem-reduction formulation is ineffective for the 8-Puzzle and that state-space is its most natural representation.

This conclusion, however, could not have been drawn by a superficial examination of the problem statement without a detailed analysis of the interactions among the goals. In some sequence-seeking problems, like the 8-Puzzle, the goals are interacting with each other, and yet a problem-reduction representation may substantially reduce the search for a solution. A classical example in this category is the **Tower of Hanoi**:

> There are n disks $D_1, D_2, \ldots, D_n$, of graduated sizes and three pegs 1, 2, and 3. Initially all the disks are stacked on peg 1, with D_1, the smallest, on top and D_n, the

largest, at the bottom as in Figure 1.10. The problem is to transfer the stack to peg 3 given that only one disk can be moved at a time and that no disk may be placed on top of a smaller one.

The problem can easily be represented in state-space by specifying the operators that are permissible from any given disk configuration. Alternatively, the problem may be represented as a list of goals to be achieved: disk D_1 on peg 3, disk D_2 on peg 3, . . . , disk D_n on peg 3. These goals are clearly not independent; placing D_1 initially on peg 3 hinders the transfer of D_2 there. However, this set of goals possesses a unique **linear hierarchy**: once the subset of goals concerning disks $D_k, D_{k+1}, \ldots, D_n$ are completed, the goals of relocating disks $D_1, D_2, \ldots, D_{k-1}$ can be accomplished without undoing the former. Whenever a goal set possesses such a linear structure, the key idea is to begin working on the most critical goal (i.e., setting disk D_n on peg 3).

Having decided to begin with this goal, we must first check whether this can be accomplished by solving a primitive problem. In looking up the list of primitive operators available in our problem and their applicability constraints, we should discover that setting disk D_n on peg 3 is indeed a primitive problem if only two conditions are met:

1. Disk D_n must be clear
2. Peg 3 must be empty.

These two conditions are not satisfied in the initial state and, therefore, they are

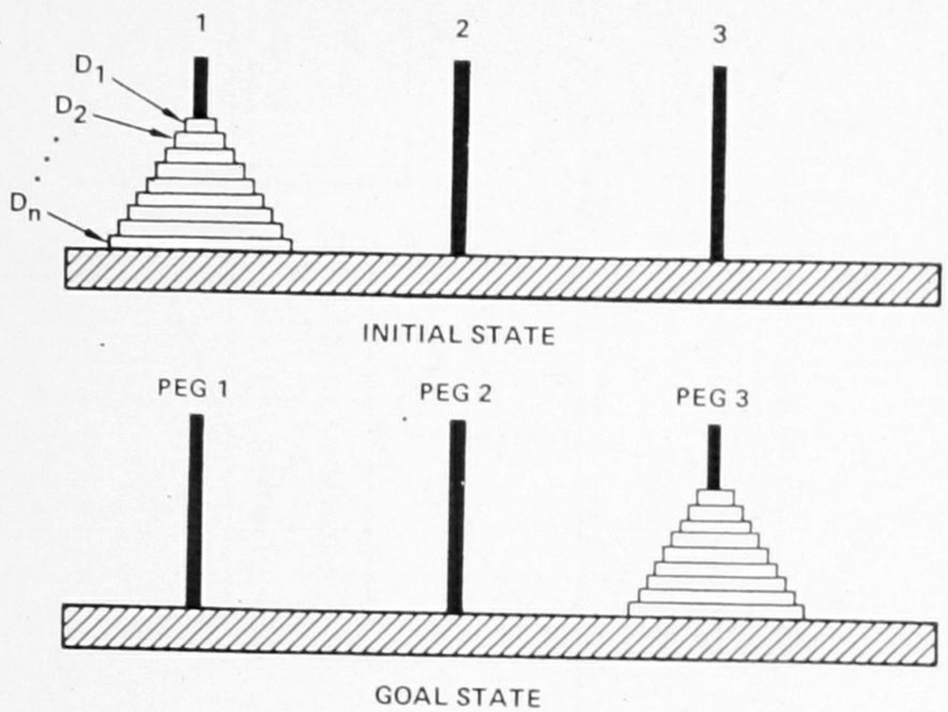

Figure 1.10
The Tower of Hanoi problem.

posited as **auxiliary subgoals**, which can also be formulated as the set of conditions: disk D_1 on peg 2, disk D_2 on peg 2, . . . , disk D_{n-1} on peg 2.

This auxiliary set of subgoals, together with the primitive problem "move D_n onto peg 3" and the set of unaccomplished goals remaining after the latter, now form a conjunction of three subproblems, which are represented by the first level of the tree in Figure 1.11. At this point the power of the problem-reduction representation can be brought into play; although the conjunction of the three subproblems implies an **ordered execution** of their solutions (from left to right), the **search** for these solutions may be conducted **in any order**. That is, we may begin searching for a solution of the rightmost subproblem prior to knowing the solution to the ones on its left; the latter provides no information that is necessary for the former. This property was not present in the 8-Puzzle and for that reason we could not benefit from the problem-reduction representation. In the Tower of Hanoi puzzle, on the other hand, we were able to reduce the original problem involving n disks to three subproblems, one primitive and two involving $n-1$ disks.

This reduction process can be applied to any nonprimitive subproblem until all the nodes in the tree become primitive and then, reading the leaf nodes left to right, we have a complete detailed sequence of elementary moves that constitutes a solution to the initial problem. The tree will undoubtedly be large (exponential in n) simply because the solution requires a large number of elementary moves. However, if we only wish to communicate a parsimonious description of the solution pattern without worrying about its details, we may take the first level of the tree as an **abstract plan** or a blueprint from which the detailed plan can be generated recursively if desired:

To transfer $S = \{D_1, \ldots, D_n\}$ from peg i to peg j, do the following:

1. If S contains a single disk, move that disk from i to j; otherwise
2. Transfer $\{D_1, \ldots, D_{n-1}\}$ from i to k.
3. Move D_n from i to j.
4. Transfer $\{D_1, \ldots, D_{n-1}\}$ from k to j.

The reasoning behind this problem-reduction method is called **means-end analysis**. It is the basis of a program called GPS, the General Problem Solver (Newell, Shaw, and Simon, 1960), subsequent automatic planning programs such as STRIPS (Fikes and Nilsson, 1971) and ABSTRIPS (Sacerdoti, 1974) and a conversational planning system named GODDESS (Pearl, Leal, and Saleh, 1982). The basic difference between this method and the state-space approach is its purposeful behavior: operators are invoked by virtue of their potential in fulfilling a desirable subgoal, and new subgoals are created in order to enable the activation of a desirable operator.

The success of this method depends, of course, on the availability of an ordered set of subgoals $G_1, G_2, \ldots, G_n$, such that the problem is solved when all subgoals are achieved and such that the set $G_1, \ldots, G_k$ can be accomplished

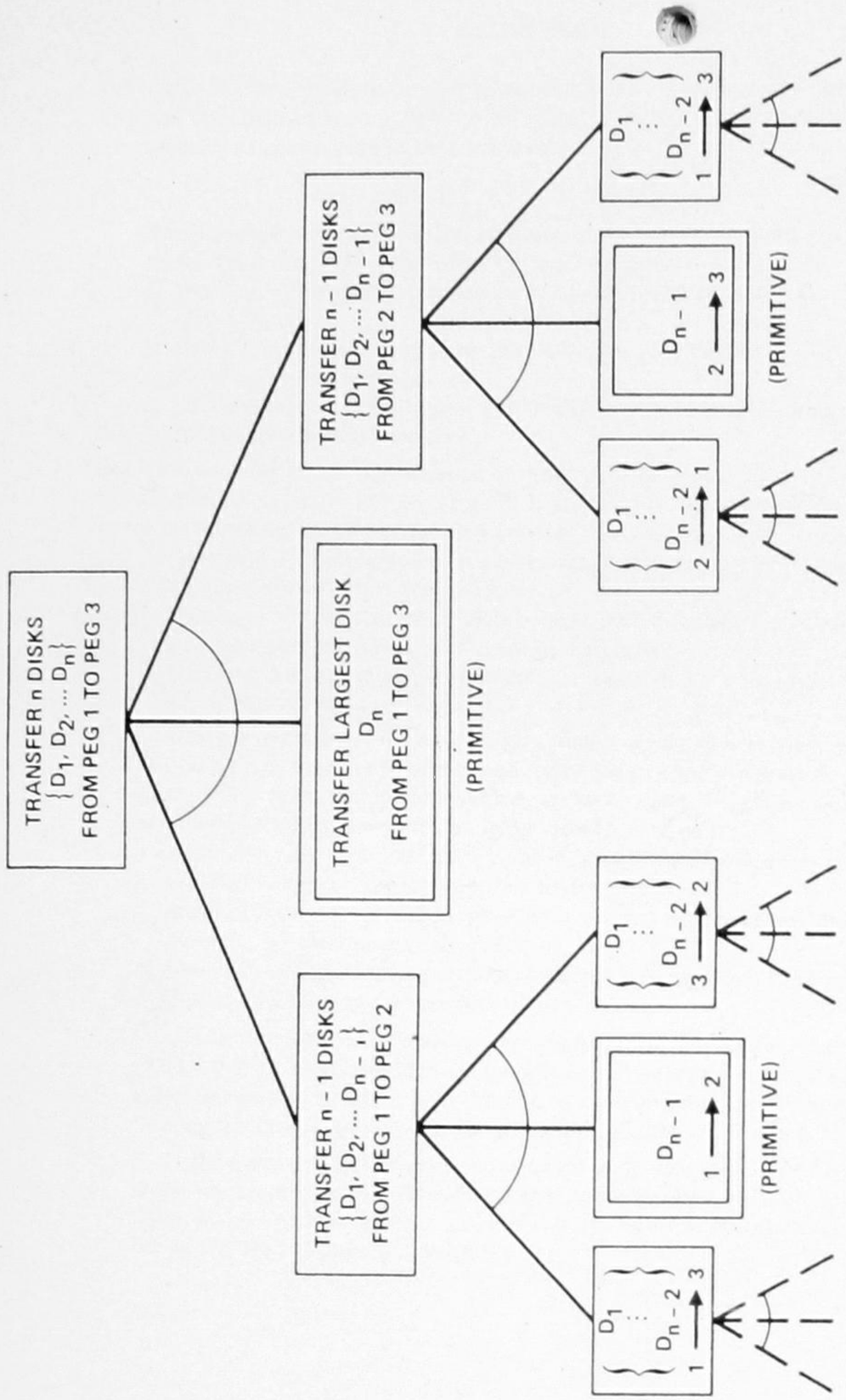

Figure 1.11
Problem-reduction representation of the Tower of Hanoi problem. The problem of transferring n disks is reduced to two subproblems of transferring $n-1$ disks, and one elementary move.

after $G_{k+1}, \ldots, G_n$ without undoing the latter. The problem of finding such an ordered set mechanically for any given problem is extremely hard but seems to be central to the mechanical discovery of heuristics (see Chapter 4).

1.3 BIBLIOGRAPHICAL AND HISTORICAL REMARKS

Some good general books on puzzles, their structure and their solution methods, are those by Gardner (1959, 1961) and by Averbach and Chein (1980). The books by Conway (1976) and Berlekamp, Conway, and Guy (1982) present formal mathematical treatments of the subject. Computer oriented discussions, including descriptions of programs which solve puzzles and games, can by found in Slagle (1971) and Banerji (1980).

The 8-Queens problem or its general n-Queen version, is typical of a class known as constraint-satisfaction or consistent-labeling problems and is often used to demonstrate ideas connected with backtracking search (Section 2.2.1). For example, see Floyd (1967), Fikes (1970), Mackworth (1977), and Gaschnig (1979a). The method of detecting dead ends such as those predicted by the condition $f' = 0$ is known as "forward checking" (Haralick and Elliot, 1980).

The 8-Puzzle is a small version of the 15-Puzzle (using a 4×4 board) first introduced in the 1870s by the American Problematist Sam Loyd (Loyd, 1959). It has served as an arena for testing heuristic methods by many AI researchers, most notably Doran and Michie (1966) and Gaschnig (1979a).

The Traveling Salesman problem (TSP) is typical of scheduling problems arising in operations research (see Wagner, 1975, and Hillier and Lieberman, 1974) and is surveyed in Bellmore and Nemhauser (1968). The use of optimistic cost estimates for finding optimal solutions is the basis of the branch-and-bound method (Lawler and Wood, 1966). The use of the optimal assignment problem to produce lower bounds for the TSP traces back to Eastman (1958), and was shown to be computable in $O(N^3)$ steps by Edmonds and Karp (1972). The use of the MST as a lower bound for the TSP is due to Held and Karp (1971). Additional bounding functions are discussed by Pohl (1973).

The Counterfeit Coin problem is typically used to demonstrate the applications of coding theory (Massey, 1976). A complete solution for the general N-coin problem has been obtained by Linial and Tarsi (1981).

The relations between the split-and-prune paradigm of operations research and the generate-and-test methods of AI are further analyzed by Ibaraki (1978a), Nau, Kumar and Kanal (1982), and Kumar and Kanal (1983). Simon (1983) compares the search paradigm, where the operations are viewed as transformations on data, with the reasoning paradigm of logic, where the elementary operations are viewed as information gathering steps (e.g., accumulating lemmas in proof procedures).

Our treatment of the state-space and problem-reduction representations follows that of Nilsson (1971). The use of AND/OR graphs for representing problems traces back to Amarel (1967) and Slagle (1963). Kowalski (1972) and vanderBrug and Minker (1975) discuss the relation between AND/OR graphs and theorem proving. Amarel (1968, 1982) gives more general expositions of the problem of finding good representations for problems.

EXERCISES

1.1 (*After D. Nau*) Consider a card game in which two players take turns removing one card from either the right or the left end of a row of cards. Initially the row contains five cards each labeled 1 or 0 as shown:

[1] [0] [1] [0] [1]

The player who removes the last card wins if that card is marked 1. Otherwise he loses. The problem is to show that the second player wins this game.

a. Draw an AND/OR graph representing this problem, and identify a solution subgraph.

b. Draw a section of a state-space graph for this problem and indicate the meaning of each node and each arc.

c. Can you find a simple criterion for identifying the winner if initially there are n cards with arbitrary labels?

1.2 The following puzzle is a variation of the card game in exercise 1.1. The cards are labeled with positive integers (e.g., 7, 5, 1, 2, and 8), and a single player removes a card from either end of the row until all cards are removed. His final score is the sum of the labels removed in odd moves of the game. The problem is to find a strategy that maximizes the final score.

a. Show the state-space graph of this problem. Interpret the candidate subset and the state code for a typical node in the graph.

b. Can you think of an optimistic heuristic function that could be of help in the search for an optimal solution?

1.3 (*2-Way Merge Pattern*) Two sorted files containing n and m records, respectively, can be merged together to obtain one sorted file in approximately $m + n$ operations. When more than two sorted files are to be merged together, the merge can be accomplished by repeatedly merging sorted files in pairs.

Given five files with sizes 20, 30, 10, 5, and 30, find a sequence of pairwise merges that minimizes the total number of operations.

a. Illustrate, in the form of a graph section, the state-space representation of this problem; interpret nodes and rules.

b. Illustrate, as in part (a), a problem-reduction formulation for this problem.

c. Discuss which representation is more convenient.

1.4 (*Symbolic Integration*) Consider the problem of integrating the expression $x^4/(1-x^2)^{5/2}$. Select problem reduction operators (e.g., substitutions of variables, integration by parts, long division, completing the square) helpful in performing this task, and represent the problem as an AND/OR graph.

1.5 Solve the Counterfeit Coin problem as stated in the text (Section 1.1.5) and find a graph representation of your solution that could be followed by a layman coin-tester. (*Note*: A clear representation of this solution will be helpful in solving related exercises in Chapters 2 and 4.)

Chapter 2

Basic Heuristic-Search Procedures

In this chapter we describe the basic search strategies used in problem solving with special attention to the way in which they incorporate heuristic information. In Chapter 1, we saw that many types of problems are conveniently described as tasks of finding some properties of graphs. Our current paradigm therefore is to find efficient methods of unraveling some properties of graphs using information from sources outside the graph.

Our discussion will be made easier by compiling some basic nomenclature regarding graphs and graph searching, although some of the concepts were already used in Chapter 1.

Basic Graph-Searching Notation

A graph consists of a set of **nodes** (or **vertices**), which in our context represent encodings of subproblems. Every graph we will consider will have a unique node s called the **start node**, representing the initial problem at hand. Certain pairs of nodes are connected by directed **arcs** (or **links**), which represent operators available to the problem solver. If an arc is directed from node n to node n', node n' is said to be a **successor** (or a **child** or an **offspring**) of n and node n is said to be a **parent** (or a **father**) of n'. The number of successors emanating from a given node is called the **branching degree** (or **degree**) of that node and normally is denoted as b. We assume that the branching degrees of all nodes in our graphs are finite, defining a class known as **locally finite graphs**. A pair of nodes may be successors of each other as in the 8-Puzzle, in which case the two arcs can be replaced by an undirected **edge**.

A **tree** is a graph in which each node (except one **root** node) has only one parent. A node in a tree that has no successors is called a **leaf**, a **tip** node, or a terminal node. A **uniform tree** of depth N is a tree in which every node at depth smaller than N has the same branching degree whereas all nodes at depth N are leaves. Often the arcs are assigned **weights** representing either

costs or **rewards** associated with their inclusion in the final solution. We use the notation $c(n, n')$ to denote the cost of the arc from n to n'.

A sequence of nodes $n_1, n_2, \ldots, n_k$, where each n_i is a successor of n_{i-1}, is called a **path** of length k from node n_1 to node n_k. If a path exists from n_1 to n_k, node n_k is said to be a **descendant** of (or **accessible** from) n_1, and node n_1 is called an **ancestor** of n_k. The cost of a path is normally understood to be the sum of the costs of all the arcs along the path (called the **sum cost**). In special cases other cost measures can be defined, such as the maximum of all the arc costs along the path, or the average cost.

The most elementary step of graph searching that we consider is **node generation**, that is, computing the representation code of a node from that of its parent. The new successor is then said to be **generated** and its parent is said to be **explored**. A coarser computational step of great importance is **node expansion**, which consists of generating *all* successors of a given parent node. The parent is then said to be *expanded*. **Pointers** are normally set up from each generated successor back to its parent node. These pointers form a dynamic network of paths (usually a tree) which facilitates tracing a path from some given node n back to the start node or to some other desired ancestor of n. To distinguish between a pointer-traced path and other paths that may go through a given node, we adopt the following notation: a node n is said to be *along* a **pointer-path** PP at a given time if at that time at least one pointer assigned to n by the search procedure is directed along PP. A node n is said to lie *on* a path P, if the implicit problem graph contains a path P going through n.

A **search procedure,** a **policy,** or a **strategy** is a prescription for determining the order in which nodes are to be generated. We distinguish between a **blind,** or **uninformed,** search and an **informed, guided,** or **directed** search. In the former, the order in which nodes are expanded depends only on information gathered by the search but is unaffected by the character of the unexplored portion of the graph, not even by the goal criterion. The latter uses partial information about the problem domain and about the nature of the goal to help guide the search toward the more promising directions.

Naturally the set of nodes in the graph being searched can at any given time be divided into four disjoint subsets:

1. Nodes that have been expanded,
2. Nodes that have been explored but not yet expanded,
3. Nodes that have been generated but not yet explored,
4. Nodes that are still not generated

Several of the search procedures discussed require a distinction between nodes of the first and third group. Nodes that were expanded (i.e., their successors are available to the search procedure) are called **closed,** whereas nodes that were generated and are awaiting expansion are called **open.** Two separate lists called CLOSED and OPEN are used to keep track of these two sets of nodes.

2.1 HILL-CLIMBING: AN IRREVOCABLE STRATEGY

Hill-climbing, a strategy based on local optimizations, is the simplest and most popular search strategy among human problem solvers. It is called hill-climbing because, like a climber who wishes to reach the mountain peak quickly, it chooses the direction of steepest ascent from its current position. Imagine the task of adjusting the knobs of a television set to obtain a picture of acceptable quality. This task is normally performed by adjusting the knobs one at a time, setting each knob in a position that seems to accomplish the greatest progress toward the desired quality before manipulating the next knob. The procedure is based on continuously monitoring the picture quality against a desired standard and making incremental steps of improvement in a direction that, on the basis of the local information available, seems most promising. The reason this strategy is so popular among human operators is that it requires almost no computational effort; at each setting of the knobs we can begin the adjustment afresh, requiring no memory of past attempts nor of the path that has led us to the current setting.

In terms of our graph-search model, the hill-climbing strategy amounts to repeatedly expanding a node, inspecting its newly generated successors, and choosing and expanding the best among these successors, while retaining no further reference to the father or siblings. Obviously the computational simplicity of this strategy is not without shortcomings. Unless the evaluation function used is extremely informative, we stand a very high chance of violating the first tenet of systematic search, that is, do not leave any stone unturned. Deceptively improving evaluation functions may lure the search into deep, or even infinite, exploration paths that in fact contain no solution. Moreover, as soon as we come to a local maximum (a node more valuable than any of its successors), no further improvement is possible by local perturbations and the process must terminate without reaching a solution. The only way to free ourselves from such a deadlock is to start afresh from what we hope is a completely new node, thus risking violations of the second tenet of systematic search: do not turn any stone more than once. This strategy is called **irrevocable**, because the process does not permit us to shift attention back to previously suspended alternatives, even though they may have offered a greater promise than the alternatives at hand.

Hill-climbing is a useful strategy when we possess a highly informative guiding function to keep us away from local maxima, ridges, and plateaus and to lead us quickly toward the global peak. Hill-climbing is also useful in problems where the state transition operators possess a certain type of independence called **commutativity** (Nilsson, 1980), a condition in which the application of no operator can hinder the future applicability of other operators or alter the state resulting from a sequence of other applicable operators. For example, the operations of expanding nodes in a graph (i.e., generating their successors) are

commutative; the expansion of any one node does not hinder the expansion of other nodes nor alter their descendants. When commutativity holds, irrevocable strategies can be applied without the risk of missing the solution. Applying an inappropriate operator may delay, but never prevent, the eventual discovery of the solution. For that reason, problems that are manageable by a set of commutative operators (e.g., the minimum spanning tree problem) are also blessed with simple optimization algorithms called "greedy." However, the task of fitting such a set of operators to a given problem may be difficult, as is discussed in Chapter 4.

2.2 UNINFORMED SYSTEMATIC SEARCH: TENTATIVE CONTROL STRATEGIES

In contrast to the irrevocable control mechanism employed by hill-climbing, we now discuss a **tentative** scheme to bring our search strategy in line with the requirements of systematization. If for some reason a given search avenue is chosen for exploration, the other candidate alternatives are not discarded but are kept in reserve in case the avenue chosen fails to produce an adequate solution. Naturally this scheme requires more memory space, but it can save time because the memory stores partially developed alternatives which, if discarded, may need to be developed afresh. Many search strategies use this tentative scheme of decision.

The class of strategies described in this section are **uninformed** or "**blind**" in the sense that the order in which the search progresses does not depend on the nature of the solution we seek. In terms of our graph model we usually say that a search strategy is uninformed when the location of the goal node (or nodes) does not alter the order of node expansion, except of course for the termination conditions. Being uninformed, these strategies are rather inefficient and usually are impractical in large problems. They are worth describing, though, as possible standards against which the performances of various informed, heuristically guided, strategies can be compared.

2.2.1 Depth-First and Backtracking: LIFO Search Strategies

In **depth-first** search, as well as in the popular variation called **backtracking**, priority is given to nodes at deeper levels of the search graph. The finest computational step in depth-first search is node expansion, that is, each node chosen for exploration gets all its successors generated before another node is explored.

After each node expansion, one of the newly generated children is again selected for expansion and this forward exploration is pursued until, for some reason, progress is blocked. If blocking occurs, the process resumes from the

deepest of all nodes left behind, namely, from the nearest decision point with unexplored alternatives. This policy works well when solutions are plentiful and equally desirable, or when we have reliable early warning signals to indicate an incorrect candidate direction.

In searching trees the concept of depth is well defined, and a depth-first algorithm should have no difficulty deciding which node in OPEN is the deepest: the deepest node is the one most recently generated. Therefore, if OPEN is structured as a **stack**, a depth-first strategy should put new successors on top of OPEN and select for expansion the topmost node. This last-in-first-out policy guarantees that no node at depth d will be expanded as long as nodes of depths greater than d still reside on OPEN.

It is easy to see that such a policy, if run unchecked, might be dangerous when implemented on large graphs, especially those of infinite depth. It could continue to probe deeper and deeper along some fruitless path and, not having a mechanism to recover from disappointments, would be unable to backup and try a fresh search avenue. For that reason, depth-first algorithms are usually equipped with a **depth-bound**—a stopping rule that, when triggered, returns the algorithm's attention to the deepest alternative not exceeding this bound. Thus the program backtracks under one of two conditions:

1. The depth-bound is exceeded.
2. A node is recognized as a dead end.

The latter event occurs when a node fails to pass a test for some property that must hold true for any node on a path to a solution. For example, requiring that any node explored in the 8-Queens problem should leave at least one unattacked cell in every unfilled row is a legitimate dead-end test.

To summarize, a depth-first policy employs the following sequence of steps:

Depth-First Search

1. Put the start node on OPEN.
2. If OPEN is empty, exit with failure; otherwise continue.
3. Remove the topmost node from OPEN and put it on CLOSED. Call this node n.
4. If the depth of n is equal to the depth bound, clean up CLOSED and go to step 2; otherwise continue.
5. Expand n, generating all of its successors. Put these successors (in no particular order) on top of OPEN and provide for each a pointer back to n.
6. If any of these successors is a goal node, exit with the solution obtained by tracing back through its pointers; otherwise continue.
7. If any of these successors is a dead end, remove it from OPEN and clean up CLOSED.
8. Go to step 2.

The operation "clean up CLOSED" in steps 4 and 7 is performed by purging from CLOSED all those ancestors of the nodes passing the tests in steps 4 and 7

that do not have sons in OPEN. This operation is optional and is designed only to save memory space.

Note that in step 5 placing the successors in an arbitrary order on top of OPEN renders the algorithm totally *uninformed*. An alternative method would be to use heuristics to order these successors, as was discussed in Section 1.1.1. This **partially informed** version of depth-first search will be analyzed in Chapter 6 (Section 6.4).

Figure 2.1 illustrates the sequence of steps taken by a depth-first search of the 4-Queens problem (a 4×4 board version of the 8-Queens problem). Each step is represented by the node being expanded (marked $a, b, \ldots, j, k$) and the status of the explicit portion of the search graph after each expansion. The order of nodes on OPEN can be seen by traversing the leaf nodes from left to right, skipping the dotted lines which represent portions of the graph deleted from memory. The order of node expansion is further illustrated in Figure 2.2.

Note that at any given time the CLOSED list forms a *single* path from the start node to the currently expanded node. This feature reflects the storage economy of depth-first policies; the maximum storage required cannot exceed the product of the depth-bound and the branching degree. The path of CLOSED nodes maintained by the program appears to be traversing the tree,

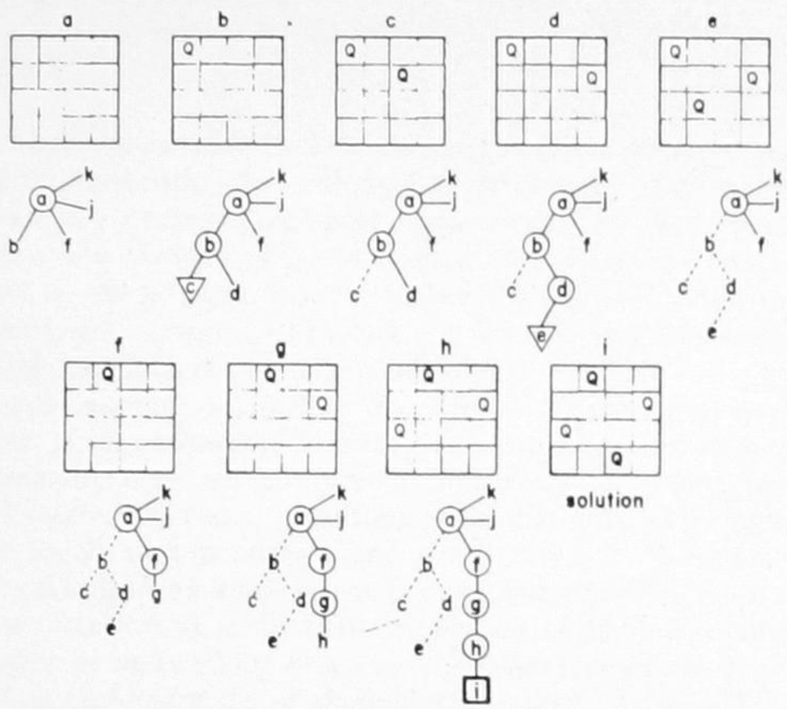

Figure 2.1
Successive steps in a depth-first search of the 4-Queens problem. Circled symbols represent CLOSED nodes, uncircled symbols represent nodes in OPEN, triangles stand for dead ends and boxed nodes are those that satisfy the goal conditions.

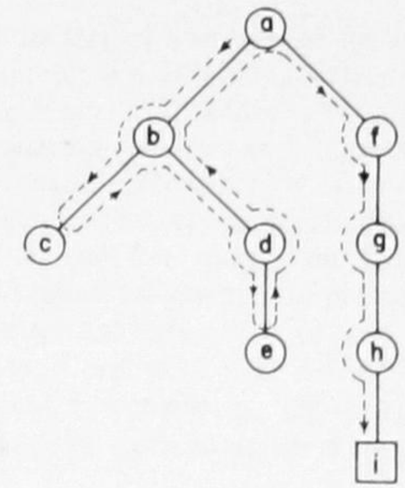

Figure 2.2
Order of node expansion by a depth-first search of the 4-Queens problem.

sweeping across it, from left to right (see Figure 2.2) and is called the **traversal path**.

When implementing depth-first policies on graphs that are not trees, we encounter certain complications. Strictly speaking, the concept of **depth** for a node in a graph is defined recursively as *one plus the depth of its shallowest parent*, with the start node having zero depth. Thus to adhere to a strict depth-first rule the program must first inspect all parents of newly generated nodes, including those parents that were recognized as dead ends and deleted from memory. Such inspection would introduce enormous complications and significantly increase storage requirements. Therefore, in searching graphs we normally compromise the strict depth-first rule (i.e., that no node will be expanded while deeper nodes reside on OPEN) in favor of the last-in-first-out principle, which is much easier to maintain. Thus we still follow the eight-step procedure just outlined, especially step 3, regardless of whether the topmost node is also the deepest node in the strict sense of the word. In step 4, however, where comparison to the depth-bound is required, the depth of a given node is taken as *one plus the depth of the parent designated by its current pointer*. This definition prevents the traversal path from exceeding the depth-bound.

There is still need, however, to check for **node duplications** in order to avoid unnecessary expansions. For example, in the 8-Puzzle, which is an undirected graph, it is important to prevent the search from exploring a repetitive back-and-forth movement of any given tile. This can be done by checking whether any of the newly generated successors already resides on the traversal path and, if a match is found, deleting the matched successor from OPEN. Again, for the sake of maintaining both simplicity and small storage, depth-first strategies normally check for a match only among those nodes residing on the traversal path, allowing duplication of nodes that were generated in the past but since deleted from memory.

Backtracking. Backtracking is a version of depth-first search that applies the last-in-first-out policy to node *generation* instead of node *expansion*. When a node is first selected for exploration, only one of its successors is generated and this newly generated node, unless it is found to be terminal or dead end, is again submitted for exploration. If, however, the generated node meets some stopping criterion, the program backtracks to the closest unexpanded ancestor, that is, an ancestor still having ungenerated successors. This policy can be implemented by modifying the depth-first search procedure as follows:

The Backtracking Procedure

1. Put the start node on OPEN.
2. If OPEN is empty, exit with failure, otherwise continue.
3. Examine the topmost node from OPEN and call it n.
4. If the depth of n is equal to the depth-bound or if all the branches emanating from n have already been traversed, remove n from OPEN and go to step 2; otherwise continue.
5. Generate a new successor of n (along a branch not previously traversed), call it n'. Put n' on top of OPEN and provide a pointer back to n.
6. Mark n to indicate that the branch (n, n') has been traversed.
7. If n' is a goal node, exit with the solution obtained by tracing back through its pointers; otherwise continue.
8. If n' is a dead end, remove it from OPEN.
9. Go to step 2.

OPEN is the only list used in backtracking and this time it serves to store the nodes on the traversal path. The marking of nodes in step 6 is usually done by appending to n an encoding of the transformation rules that have been applied to n in the past. For example, in the 4-Queens puzzle of Figure 2.1, when node c is generated, we should mark node b to indicate that the branch (b, c) has already been traversed. Later on, when node c is found to be a dead end, we will return to b and use this mark to avoid regenerating c.

The main advantage of backtracking over depth-first search lies in achieving an even greater storage economy. Instead of retaining in storage all the successors of an expanded node, we only store a single successor at any given time. In addition, backtracking avoids generating nodes that stand to the right of the solution path found, such as nodes k and j in Figure 2.1. These advantages are marred by the algorithm's inability to use heuristic information for evaluating candidate successors as in the informed version of depth-first search. An informed version of backtracking would have to either temporarily generate all successors of an explored node, or alternatively, to score the available operators on the basis of their own features, without explicating their resultant states.

More sophisticated backtracking strategies also use a technique called **backmarking** with which, after meeting a dead-end condition, they may back up *several levels at once* (Gaschnig, 1979a). This is done by submitting the dead-end condition to a critical analysis to see if the "blame" for that condition can

be placed on one of the earlier ancestors along the traversal path. For example, if a given node at depth 4 of the 8-Queens problem leaves no free cell on the seventh row (see Figure 2.3), and this fact is only discovered at depth 7 (i.e., when trying to place a queen on row 7), backtracking to level 6 or even 5 will not remedy the situation. However, by marking each cell in row 7 with the age of the oldest queen that attacks it, we can detect that the blame really lies with the position of the first four queens and, hence, that we must backtrack directly to level 4 and attempt to change the position of the queen on the fourth row. This situation would not have arisen, of course, were we to insist that each new queen leave at least one unattacked cell in every unfilled row. Meeting this condition, however, would require that we spend extra work on forward testing, whereas backmarking is accomplished with no extra tests.

Backtracking strategies are also useful in **optimization** and semi-optimization problems. If the graph contains many solution paths and our task is to find one with the lowest cost, we can maintain a record of the cheapest solution encountered so far and continue the search until it becomes certain that no cheaper solution may lie ahead of us. For example, if we know that path costs cannot decrease through extension of the path, then finding that the cost of an early portion of a candidate path already exceeds the record we maintain allows us to *prune* that path prior to completion. This pruning is illustrated in Figure 2.4, which describes a backtracking strategy for finding the minimum-sum column in a matrix with nonnegative entries. The search tree, in this case, resembles a curtain where every nonterminal node except the start node is of degree 1. Po-

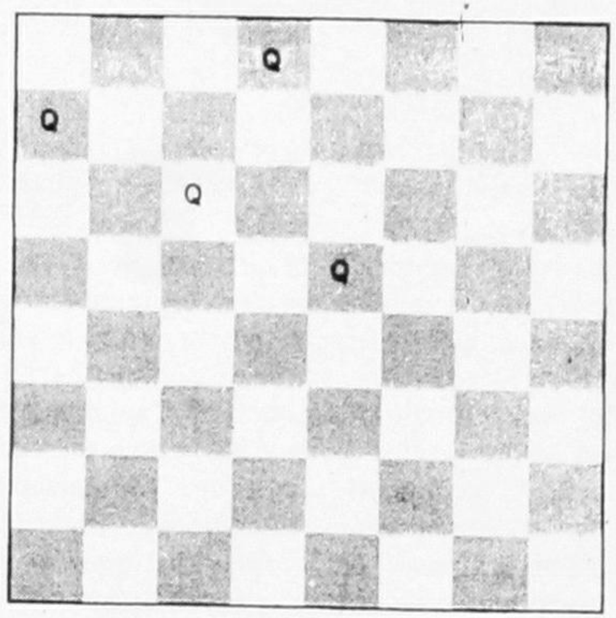

Figure 2.3
A dead-end position in the 8-Queens problem which may not be detected until we try to place a queen on the 7th row.

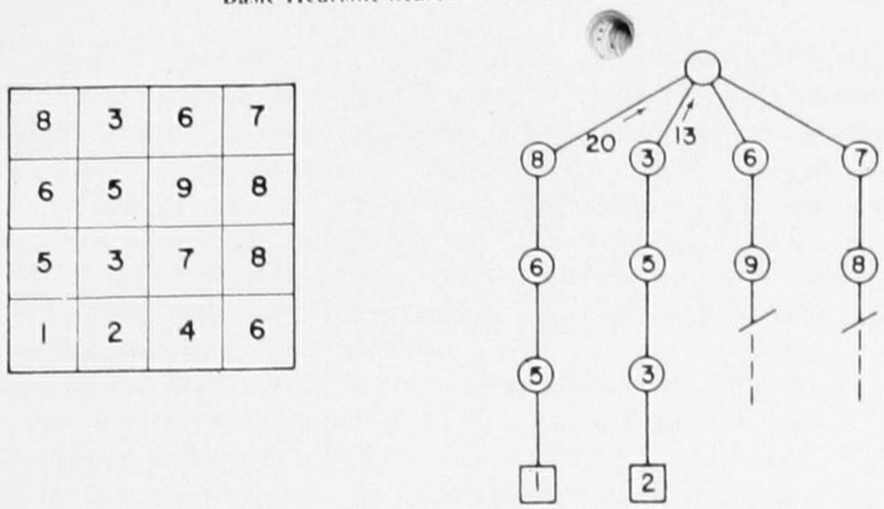

Figure 2.4
Pruning in a backtracking search for the minimum-sum column in a matrix with nonnegative entries.

tential solution paths are pruned as soon as their partial sums exceed the lowest sum encountered on the left. The importance of heuristics for ordering the first group of successors is clearly shown in this example; the amount of pruning increases when we succeed in finding cheap solutions earlier in the search.

2.2.2 Breadth-First: A FIFO Search Strategy

Breadth-first strategies, as opposed to depth-first, assign a higher priority to nodes at the shallower levels of the searched graph, progressively exploring sections of that graph in layers of equal depth. Thus instead of a LIFO policy, breadth-first search is implemented by a first-in-first-out (FIFO) policy, giving first priority to the nodes residing on OPEN for the longest time. The eight-step procedure of the last section can still be followed if in step 5 we place the new successors at the *bottom* of OPEN instead of at the top, thus using OPEN as a queue rather than as a stack.

Figure 2.5 demonstrates the order of node expansions by a breadth-first search on two trees: (*a*) represents the 4-Queens problem including the symmetric half unexplored by backtracking, and (*b*) is a hypothetical tree with a solution node at depth 2. The advantages and weaknesses of breadth-first search can be seen by comparing Figures 2.5(*a*) and 2.1. In the 4-Queens problem, breadth-first is seen to explore almost twice as many nodes as depth-first. This results from having all the solutions situated at depth 5, and no node at depth larger than 5. This property frees depth-first from the danger of following hopeless long paths and, at the same time, commits breadth-first to explore every node of depth smaller than 5. Figure 2.5(*b*), by contrast, represents a situation advantageous to breadth-first search. A solution exists at depth 2 surrounded by paths that may go to a much greater depth. Breadth-first is, again,

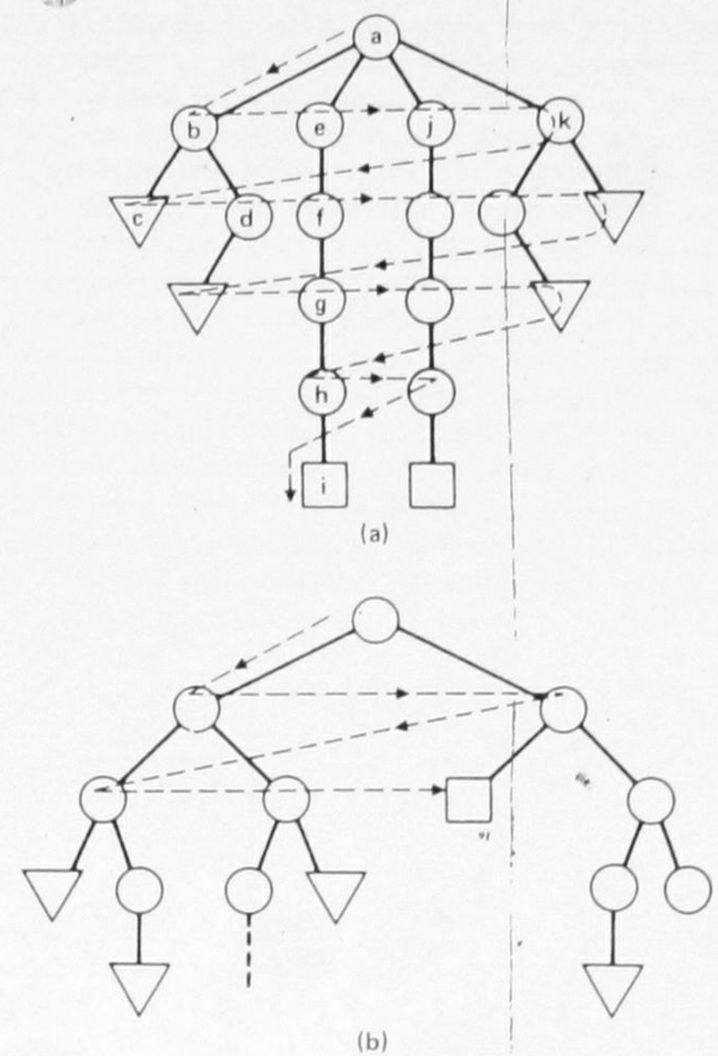

Figure 2.5
Order of node expansion by breadth-first search.

committed to explore every node up to the depth of the shallowest goal, but in this case this commitment is a small price to pay for the safeguard against getting trapped in one of the lengthy paths surrounding the goal.

Unlike depth-first strategies, breadth-first search of locally finite graphs is guaranteed to terminate with a solution if a solution exists. Moreover, it is also guaranteed to find the shallowest possible solution. The price paid for these guarantees is depicted in Figure 2.5. Instead of a single traversal path, breadth-first must retain in storage the entire portion of the graph that it explores. Only by retaining a full copy of the search graph can it shift attention away from newly generated nodes and come back to expand nodes suspended many steps earlier. The need for this abrupt shift of attention, combined with the large storage requirements, is the main reason that breadth-first strategies are rarely adopted by human problem solvers.

The Uniform-Cost Procedure. To exploit the advantages of breadth-first search in optimization problems, an important variation, called the **uniform-cost** or **cheapest-first strategy,** is often employed. Here, instead of progressing in layers of equal depth, we unravel the search space in layers of equal cost. If our task is to find the cheapest path from the start node to a goal and if path costs are nondecreasing with length, we can make sure that no node of cost greater than C is ever expanded while other candidates, promising costs lower than C, are waiting their turn. This guarantees that the first goal node chosen for expansion is also the cheapest solution.

The uniform-cost search procedure is accomplished by selecting for expansion the node in OPEN that has the lowest cost, where the cost of each node is defined as the cost of the path leading to that node from the start node. Figure 2.6 shows the graph explored by the uniform-cost algorithm in searching the minimum-sum-column problem. Note the improved pruning compared to the depth-first search in Figure 2.4.

So far our description has been limited to searches conducted on trees. The modifications required to make these two algorithms suitable for handling graphs will be discussed in Section 2.4.4. as part of the description of the heuristic graph-searching algorithm called A^*. We will see that both breadth-first and uniform-cost are special cases of A^* with its heuristic information suppressed.

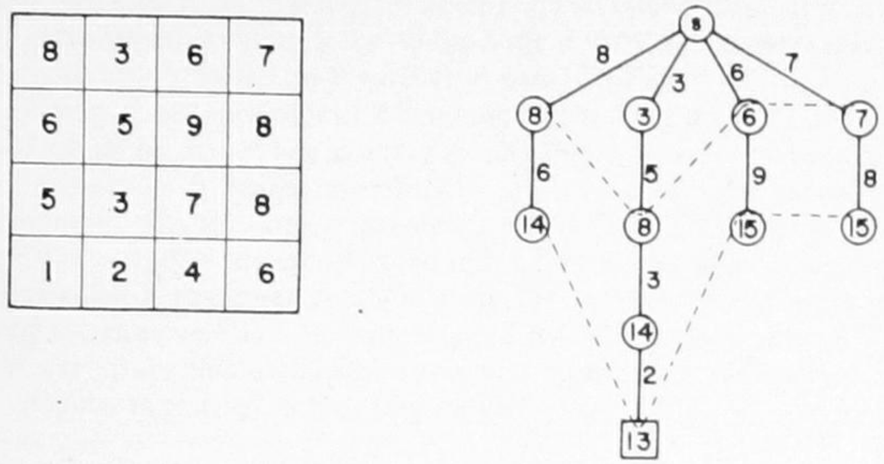

Figure 2.6
Uniform-cost search for the minimum-sum column in the matrix on the left.

2.2.3 Uninformed Search of AND/OR Graphs

Breadth-first and depth-first strategies are easily adaptable to searching AND/OR graphs. The difference lies mainly in the procedures employed for determining the termination conditions. Whereas in state-space searches the

termination test involves the property of a single node, in a problem-reduction formulation we need to ascertain whether a given *set* of solved nodes collectively supports a solution tree. Consequently, the impact of each newly generated group of nodes must be propagated to determine, using the "solved" and "unsolvable" labeling procedures (Section 1.2.3), whether the start node can be labeled. In this propagation process some intermediate nodes are labeled "solved" or "unsolvable," and these can be permanently removed from storage after their impact is transmitted to their parents. Breadth-first transmits the impact of newly generated nodes via the entire AND/OR tree, or graph, that it maintains. Backtracking strategies, on the other hand, transmit that impact through the traversal path that sweeps the graph in a prescribed order.

Both breadth-first and uniform-cost strategies for AND/OR graphs are special cases of a search algorithm called AO^* which will be discussed in Section 2.4.4. We therefore focus our current discussion on backtracking strategies.

Figure 2.7 illustrates the steps taken by backtracking in a left-to-right traversal of a binary AND/OR tree of depth 3. The symbols S and U stand for the

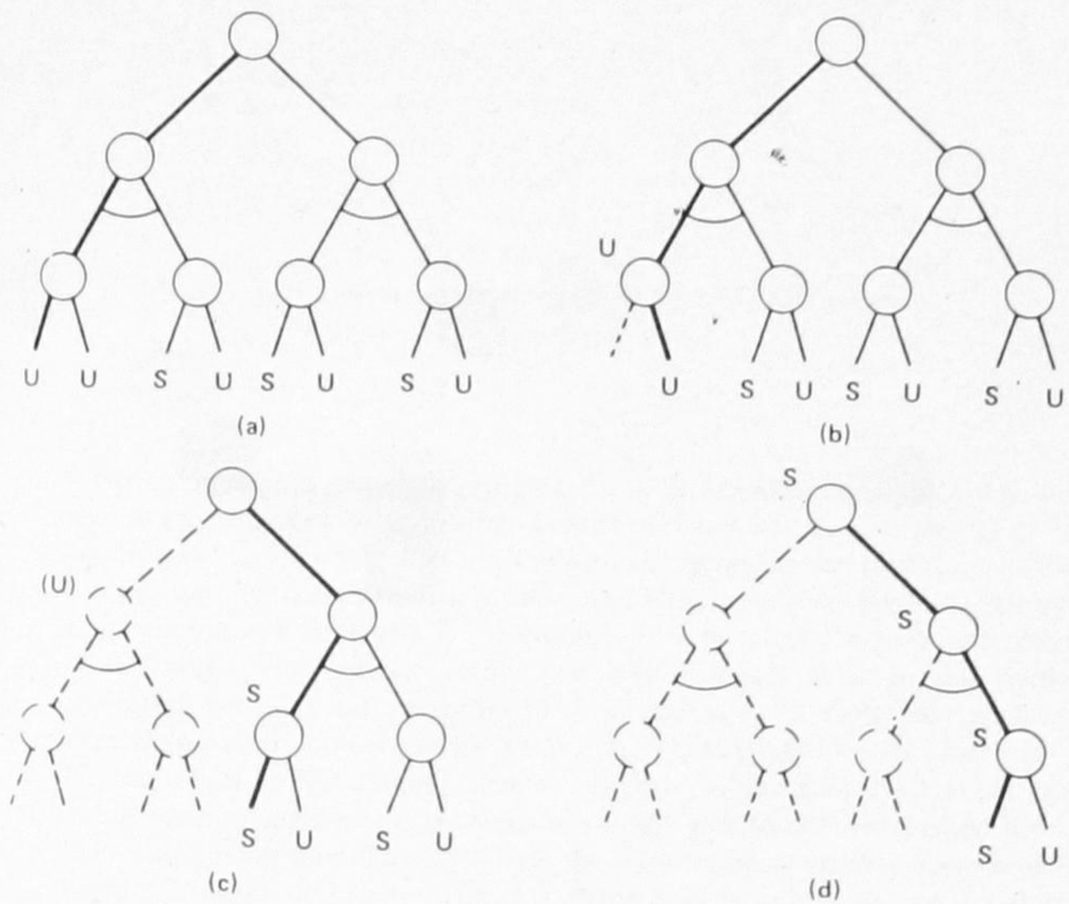

Figure 2.7
Typical steps in the execution of backtracking search of an AND/OR tree. The heavy line represents the traversal path, whereas the broken lines represent portions of the tree that can be purged from memory.

"solve" and "unsolvable" labels that propagate up the traversal path. Note that labeling any intermediate node permits us to remove that node from storage and shift the traversal path one step to the right. This removal, however, although not affecting the correct labeling of the start node, prevents us from reconstructing the actual solution graph once s is labeled as solved. Therefore, if in addition to testing whether a problem can be solved we also wish to exhibit the final solution tree, additional bookkeeping is needed. Prior to its removal, each "solved" node should transmit to its father a code for the solution graph residing below it. These codes are then combined by the various AND nodes and passed upward until the start node, if labeled "solved," receives a code for the entire solution graph beneath it. For example, in Figure 2.7(c) the intermediate node labeled S will transmit to its father the code (1, 1), indicating that a solution can be obtained by going down two levels following the leftmost successors. At stage (d) the AND node assembles the tree code (1, 1) (2, 1), and finally the start node receives the solution tree 2, ((1, 1) (2, 1)).

Since backtracking strategies are committed to maintaining in storage a single traversal path, they cannot take full advantage of the parsimony offered by AND/OR graphs. Identical subproblems encountered along different paths of the graph may need to be solved anew. In Figure 2.8(a), for example, the node labeled A will need to be solved twice, as shown in (b). Cycles can be broken up by discarding any newly generated node that matches those residing in the current traversal path. In so doing we never risk precluding valid, acyclic solutions because the information lost with the node discarded is recoverable from the matching node retained on the traversal path. In Figure 2.8(c), for example, discarding A as a successor of B amounts to breaking up the A-B-A cycle which could not be part of any solution graph. Figure 2.8(d) illustrates how a cycle across siblings is broken after unfolding by a left-to-right backtracking. The legitimate solution tree represented in Figure 2.8(e) is eventually discovered, albeit after some duplicated effort. A breadth-first strategy would have discovered a shallower solution using the terminal nodes at depth 2. However, the extremely low storage required by backtracking strategies normally makes up for this duplicated effort and is the main reason that make them so popular in game-searching (see Part III).

2.3 INFORMED, BEST-FIRST SEARCH: A WAY OF USING HEURISTIC INFORMATION

In Chapter 1 (Section 1.1) we saw examples in which familiarity with the problem domain enabled us to judge certain directions of search more promising than others, thus using knowledge beyond that which is built into the state and operator definitions. In this section, we see how this heuristic knowledge can be used in the formalism of systematic search.

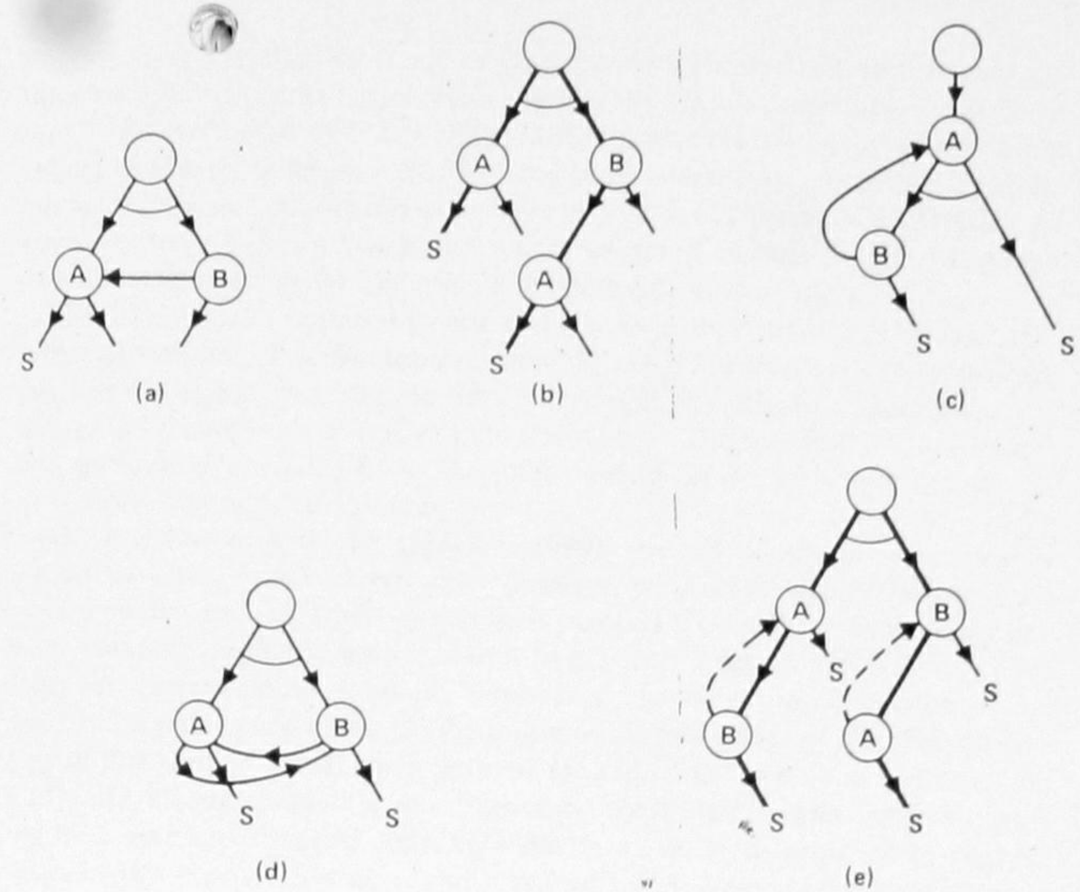

Figure 2.8
Left-to-right backtracking search of AND/OR graphs.

The most natural stage for using heuristic information is in deciding which node to expand next. A somewhat similar decision was also employed by hill-climbing strategies where forward motion was always taken from the last decision through the most promising successor. However, what sets best-first apart from other search strategies is the commitment to select the best from among *all* the nodes encountered so far, no matter where it is in the partially developed tree. To use Winston's (1977) metaphor, "it works like a team of cooperating mountaineers seeking out the highest point in a mountain range: they maintain radio contact, move the highest subteam forward at all times, and divide subteams into sub-subteams at path junctions." To further improve this metaphor we should also add that the mountaineers employ various instruments to estimate the current altitude of the various subteams and the altitude of the paths ahead, and that the readings on these instruments are occasionally corrupted by noise. The noise represents the fact that heuristic information, by its very nature, may occasionally be misleading.

2.3.1 A Basic Best-First (*BF*) Strategy for State-Space Search

The promise of a node can be estimated in various ways. One way is to assess the difficulty of solving the subproblem represented by the node. Another way is to estimate the quality of the set of candidate solutions encoded by the node, that is, those containing the path found leading to that node. A third alternative is to consider the amount of information that we anticipate gaining by expanding a given node and the importance of this information in guiding the overall search strategy. In all these cases, the promise of a node is estimated numerically by a **heuristic evaluation function** $f(n)$ which, in general, may depend on the description of n, the description of the goal, the information gathered by the search up to that point, and most important, on any extra knowledge about the problem domain.

Several best-first strategies differ in the type of evaluation function they employ. The algorithm we describe next is common to all such strategies because no restrictions are placed on the nature of $f(\cdot)$. We assume only that the search space is a general state-space graph, that the node selected for expansion is the one having the *lowest* $f(\cdot)$, and that when two paths lead to the same node, the one with the higher $f(\cdot)$ is discarded. This algorithm is called **best-first** (*BF*) and works as follows:

The Best-First Algorithm BF

1. Put the start node s on a list called OPEN of unexpanded nodes.
2. If OPEN is empty, exit with failure; no solution exists.
3. Remove from OPEN a node n at which f is minimum (break ties arbitrarily), and place it on a list called CLOSED to be used for expanded nodes.
4. Expand node n, generating all its successors with pointers back to n.
5. If any of n's successors is a goal node, exit successfully with the solution obtained by tracing the path along the pointers from the goal back to s.
6. For every successor n' of n:
   a. Calculate $f(n')$.
   b. If n' was neither on OPEN nor on CLOSED, add it to OPEN. Attach a pointer from n' back to n. Assign the newly computed $f(n')$ to node n'.
   c. If n' already resided on OPEN or CLOSED, compare the newly computed $f(n')$ with the value previously assigned to n'. If the old value is lower, discard the newly generated node. If the new value is lower, substitute it for the old (n' now points back to n instead of to its previous predecessor). If the matching node n' resided on CLOSED, move it back to OPEN.
7. Go to step 2.

In step 6c, the decision to avoid node repetitions amounts to maintaining an explicit tree T, called the **search tree** or **traversal tree,** which is a subtree of the

implicit state-space graph G underlying the problem and whose branches are directed opposite to the pointers assigned by the search procedure at any given time. The leaf nodes of T are either OPEN nodes that are generated and not yet expanded, or nodes on CLOSED that were selected for expansion, generating no successors. All the interior nodes of T are, of course, on CLOSED. At any given time, T designates to every node explored a single distinguished path from the start node defined by tracing the pointers head-to-tail.

The computation required by the test for node repetitions in step 6 can be rather substantial. For this reason, some *BF* search procedures avoid making this test by eliminating step 6c altogether, with the result of maintaining in storage duplicate descriptions of identical nodes. The tradeoffs between these two methods of handling node identities depend on the particular features of each individual problem domain.

Another variation of *BF* (Nilsson, 1980), instead of *reopening* previously CLOSED nodes only *redirects* their pointers, propagates new values to their generated descendants and redirects their pointers if necessary. In so doing one must maintain explicitly both a search tree T and a search graph G'. T contains the current pointers system, whereas G' is the explored portion of the underlying state-space graph G (the union of all past traces of T). The advantage of this method is that when pointers are redirected to a node n residing on CLOSED, all successors of n in G' can also adjust their pointers, without waiting for n to be reexpanded. Putting n back on OPEN, as required by step 6c, often invites reexpansion of n and many of its previously generated descendants. Nilsson's variation saves the reexpansion effort at the expense of value propagation and pointer-redirecting effort which may occasionally be superfluous, since some descendants of n in G' may not require regeneration. Most of our discussions in later sections will remain valid for both variations of the *BF* procedure, since the motion of T through G remains the same in both cases.

2.3.2 A General Best-First Strategy for AND/OR Graphs (*GBF*)

The basic best-first principle, namely the commitment to explore the best among *all* available candidates, can also be applied in searching AND/OR graphs. Here, however, we need to define more carefully what we mean by "best" and what we mean by a "candidate." Like in the state-space formulation we consider candidates to be subsets of potential solutions and, since in problem-reduction formulation the objects of pursuit are solution graphs, we ought to treat *subsets of solution graphs* as candidates awaiting exploration. The AND/OR graph formalism encodes each such subset by a special data structure called a **solution base**.

Assume that at a certain phase of the search the explicated portion of the underlying AND/OR graph G is represented by the subgraph G' of G. Since we can only explore G by repeated node expansions starting from s, G' must

be connected, must contain s, and will have a frontier containing all those nodes generated and not yet expanded. Every subgraph of G' that has the potential of being extended into a solution graph of G represents a subset of potential solutions and ought to be regarded as a candidate for exploration. We can enumerate this set of candidate subgraphs by simply assuming that all the frontier nodes of G' that are not already labeled "unsolvable" are in fact "solved," and then find the set of solution graphs that these frontier nodes can support. Thus a **solution base** is any subgraph G'' of G' for which the following conditions hold:

1. G'' contains the start node s.
2. If an expanded AND node n is in G'', then all n's successors are also in G''.
3. If an expanded OR node n is in G'', then exactly one successor of n is also in G''.
4. None of the nodes in G'' is currently labeled "unsolvable."

We now return to consider which among the frontier nodes of G' should be rated most deserving of the next expansion. While executing *BF* on OR graphs we maintained a one-to-one correspondence between the candidates for expansion and the candidate solution bases. At any given time the set of solution bases considered by *BF* were all the (root-to-frontier) paths contained in the traversal tree T, and each one of these paths was represented by a unique node on OPEN. This one-to-one correspondence no longer holds in searching AND/OR graphs; each solution base may contain many nodes that are candidates for expansion, and a given node may participate in several candidate solution bases. Hence the interpretation of what is meant by "best" and the translation of the best-first principle to a working procedure are more complex.

The procedure described in this section applies the best-first principle in two steps. First, it identifies the most promising solution-base graph using a **graph evaluation function** f_1. Then, it expands nodes within that graph using a **node evaluation function** f_2. These two functions, serving in two different roles, provide two different types of estimates: f_1 estimates some properties of the set of solution graphs that may emanate from a given candidate base, whereas f_2 estimates the amount of information that a given node expansion may provide regarding the alleged superiority of its hosting graph. Most works in search theory focus on the computation of f_1, whereas f_2 is usually chosen in an ad hoc manner. The principles underlying the nature of f_1 are discussed later in this section.

Once we select the two guiding functions f_1 and f_2, a best-first search algorithm *GBF* for an AND/OR graph can be stated as follows:

The GBF Algorithm

1. Put the start node s on OPEN.
2. From the explicit search graph G' constructed so far, (initially, just s), compute the most promising solution-base graph G_0, using f_1 and the heuristics h provided in step 4.
3. Using f_2, select a node n that is both on OPEN and in G_0; remove n from OPEN and place it on CLOSED.
4. Expand node n, generating all its immediate successors, and add them to OPEN and to the search graph G' with pointers back to n. For each successor n', provide heuristic information h (e.g., a list of parameters) that characterizes the set of solution graphs rooted at n' (see the text that follows).
5. If any successor n' is a terminal node, then
   a. Label node n' "solved" if a goal or "unsolvable" if not.
   b. If the label of n' induces a label on any of its ancestors, label these ancestors "solved" or "unsolvable" using the labeling procedure that follows.
   c. If the start node is solved, exit with G_0 as a solution graph.
   d. If the start node is labeled "unsolvable," exit with failure.
   e. Remove from G' any nodes whose label can no longer influence the label of s.
6. Go to step 2.

The **labeling procedure** mentioned in step 5 is defined recursively and can be implemented using either depth-first (see Figure 2.7) or breadth-first policies.

The "Solve" Labeling Procedure

1. A terminal node is labeled "solved" if it is a goal node (representing a primitive subproblem); otherwise it is labeled "unsolvable" (representing a subproblem that cannot be reduced any further).
2. A nonterminal AND node is labeled "unsolvable" as soon as one of its successors is labeled "unsolvable"; it is labeled "solved" if all its successors are "solved."
3. A nonterminal OR node is labeled "solved" as soon as one of its successors is labeled "solved"; it is labeled "unsolvable" if all its successors are "unsolvable."

In step 4 of *GBF* each node n' on the frontier of G' is assumed to be characterized by information regarding the set of *solution graphs rooted at* n'. This set (possibly empty) consists of subgraphs $g(n')$ of G where each $g(n')$ satisfies:

1. n' is the root node of $g(n')$.
2. n' can be labeled "solved" by applying the labeling procedure to $g(n')$.

Of course, since n' is not yet expanded, the subgraphs $g(n')$ are not available explicitly and can only be assessed using heuristic information from outside of G. The role of this information is to assess both the difficulty of finding a solution for the subproblem represented by n' and the quality of the solution when found.

Note that the preceding procedure leaves several computations unspecified. Aside from the graph and node selection functions f_1 and f_2, we also left unspecified the heuristic information provided at step 4, without which f_1 and f_2 cannot be computed. In the next section we will see that restrictions placed on these computations define a natural taxonomy of best-first algorithms. But first it may be instructive to demonstrate the major steps of the *GBF* algorithm by illustrating the sequence of solution bases selected in a simple problem.

An Example of Applying GBF. Let us assume that we wish to find a solution graph in the (implicit) AND/OR graph G of Figure 2.9(a). Expanding s in Figure 2.9(b), we unravel three solution bases (i.e., three disjoint subsets of potential solution graphs represented by the three arcs emanating from s). To apply the best-first principle we need to use some criterion (f_1) for assessing the promise offered by these solution bases. Let us assume that we decide to pursue our search within whichever subset appears to contain the solution graph with the *smallest number of edges*. Although the problem itself is not posed as an optimization task (any solution graph will do), we can reasonably argue that the pursuit of the smallest graph automatically guides us toward an early termination.

Now, to rank our three candidates by this criterion, we need to obtain some information regarding the class of solution graphs stemming from each of the three given arcs. If the entire G graph were accessible, we could have ranked our candidates perfectly; all we would need to know is the size of the smallest

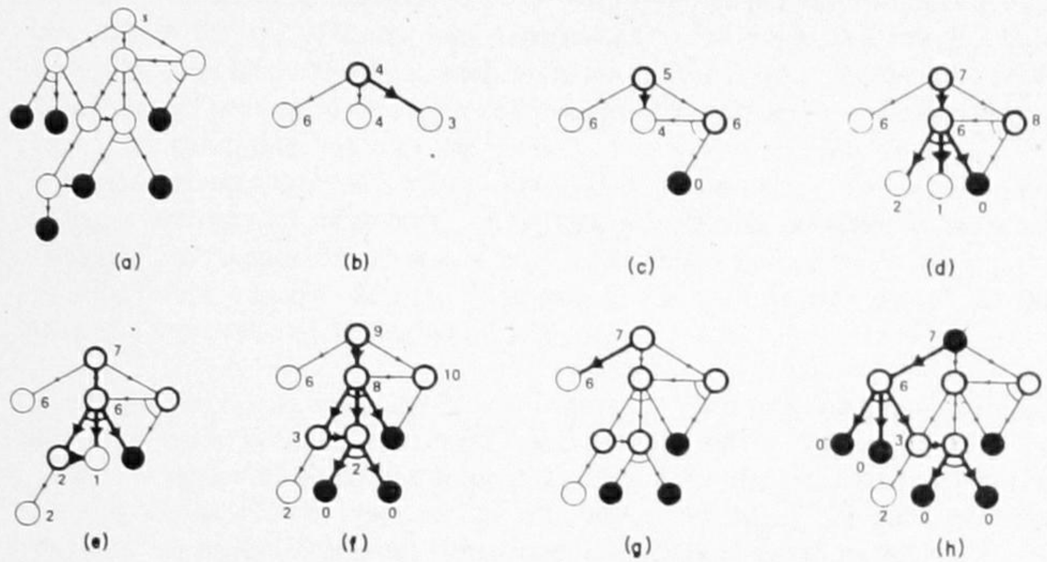

Figure 2.9
Successive steps in the execution of general-best-first (GBF) search on the implicit AND/OR graph of part (a). Solid circles represent solved nodes, heavy hollow circles nodes in CLOSED, and thin circles nodes in OPEN. The heavy lines stand, at each stage, for the current most promising solution base.

solution graph beneath each one of the three nodes currently on OPEN. Not having access to G itself, we must try to estimate the needed information from the descriptions of the nodes themselves, and from our general knowledge of the problem domain.

Assuming that such estimates were computed and gave the values 6, 4, and 3 to the three nodes on OPEN, we now make an additional assumption that these estimates be taken *at face value* for the purpose of computing the merit of each candidate solution base. Recalling that perfect estimates, if available, should be increased by one to give the required size assessment of our candidates, we apply the identical procedure to the imperfect estimates and obtain the values 7, 5, and 4. The best candidate appears to be the rightmost, with an overall estimate of 4, and this estimate can now be stored at s for later use.

This one simple step applied to the three nodes in Figure 2.9 involves two fundamental principles in heuristic programming. The choice of an auxiliary minimization criterion as a means to hasten the search invokes what we call the **small-is-quick** principle. That choice determines the kind of heuristic estimates h that should be provided in step 6 of *GBF*; *h should contain estimates of a set of parameters sufficient for the computation of the minimization criterion chosen.* Substituting these imperfect estimates for the actual values of the parameters constitutes the second principle and would be referred to as the **face-value** principle. Together these two principles determine the graph selection function f_1 in step 2 of *GBF*.

Following step 2, the rightmost candidate solution base is declared most promising, and its (only) OPEN node is expanded (step 3). This allows reevaluation of the candidate using the estimates assigned to its newly generated nodes. In our example (Figure 12.9(c)), the rightmost candidate was reevaluated to 7, rendering it inferior to the middle candidate, valued at 5. Therefore that latter candidate is declared most promising and draws future attention.

The graphs in Figure 2.9 illustrate successive developments of the search graph G' (in thin lines) and the shifting of its most promising solution base G_0 (in heavy lines). Figure 2.9(d) illustrates a situation where the node selection function f_2 is invoked for the first time. The reason for choosing the leftmost tip node of G_0 for expansion stems from the following argument: If the current G_0 is in fact a base of a legitimate solution, all its tip nodes must eventually be expanded, and the order of expansion is immaterial. If it is not, however, the most sensible node to expand would be the one that reveals the error the earliest.

To identify the node most likely to refute G_0's superiority, we need some information regarding the accuracy of each of the estimates. In the absence of such information all we can do is attempt to infer these accuracies from the magnitude of the estimates themselves, assuming, for example, that the errors are proportional to the magnitude of the estimates. In our example we may accordingly argue that the node most likely to prove G_0 inferior is the one with the largest size estimate, since a small percentage error in this estimate may

cause large fluctuations in the value of G_0. More elaborate selection criteria can be applied if additional information is available regarding the candidate tip nodes. For example, it would be appropriate to combine both the likelihood that a node turns unsolvable with the estimate of the effort required to demonstrate it.

Figure 2.9(*e*) raises another dilemma. One successor (marked 1) of the node expanded at this stage is already on OPEN, with size estimate 1. Should this estimate be also attributed to its second father, or ignored because it has already been counted by its first father? The first alternative compromises our decision to seek the solution graph with the smallest number of edges; we may find ourselves chasing graphs with a different measure of size where each edge is counted several times depending on the number of parents it has. The second alternative, counting each edge only once, severely complicates the computation of f_1, making it impossible to calculate the merit of candidate solution bases from the estimates assigned to their frontier nodes even if these estimates were exact.

Take, for instance, the nodes marked 1 and 2 in Figure 2.9(*d*) and assume that these estimates are exact. The value of the parent node would be 6 if the optimal solution graphs beneath these two nodes were disjoint. On the other hand, the value of the parent is only 5 if the two solution graphs share an edge as in Figure 2.9(*e*). This complication implies that the contribution of a given frontier node to the size of a solution base is no longer a constant but may vary depending on the nature of the solution base for which this node is being considered. For example, the node marked 2 in Figure 2.9(*e*) may contribute only one edge to the size of G_0 but may contribute two full edges to a solution base emerging from its other parent, such as that in Figure 2.9(*h*). Thus each candidate solution base must be recalculated separately with each node expansion, even if the candidates share a large portion of the graph in common.

In optimization problems we have no choice but to pursue the second alternative, counting each edge once. In satisficing problems, however, because the choice of optimization pursuit was only an auxiliary device for achieving the solution more quickly, we may compromise the optimization purity for the sake of a more efficient computation of f_1.

***Provisions for Optimality*: GBF*.** In Figure 2.9(*f*) a solution graph is found, the start node is labeled "solved," and the algorithm terminates with the current structure of G_0 whose value is 9. This is not the smallest size solution though; Figure 12.9(*h*) contains one with only seven edges. To guarantee that the *GBF* algorithm terminates with an optimal solution graph as defined by the function f_1, all cost estimates must be optimistic (i.e., underestimating costs and overestimating merit) and the termination condition must be modified. Instead of accepting the first solution that induces a "solved" label on the start node (step 5c), we should check for termination only after G_0 is selected (step 2) to see if G_0, in isolation, constitutes a solution graph. Thus, the modified algorithm, which we call *GBF**, would simply discard step 5c, and modify step 2 to read:

2′. Compute the most promising solution-base graph G_0; if all leaf nodes of G_0 are "solved," exit with G_0 as a solution graph.

Applying *GBF** to the situation in Figure 12.9(*f*) we see that the last expansion has increased the size of G_0 to 9, thus rendering it no longer most promising. Instead, the leftmost branch of Figure 12.9(*g*) promises a size of 7 and now assumes the G_0 status. In the next step, Figure 2.9(*h*), a new G_0 is found that retains its most-promising status and simultaneously proves to be a legitimate solution graph; at this point the algorithm terminates.

The fact that **GBF* always finds an optimal solution graph** (if the f_1 estimates are optimistic) can be seen by viewing the solutions as points in some space, the solution bases as subsets of points in that space and the operation of node expansion as the act of splitting those subsets. Since the estimates assigned to the subsets are optimistic (i.e., they exaggerate the merit of the best point in each subset) and the algorithm terminates only when it attempts to split the best subset and finds that it is a point (where the estimate is accurate), the point found could not be inferior to some other point. If it were, the subset containing that other point would have been chosen for splitting on account of its overly optimistic estimate.

This argument is the basis for the branch-and-bound method in operations research and several results in heuristic search theory. It will be proven formally in Chapter 3 in the context of path-seeking problems. The formal proof, however, essentially mirrors the steps of the set-splitting argument but lacks the latter's generality and appeal.

Relations between GBF and BF. If we let *GBF* search a state-space graph G (where all nodes are OR nodes), its operation would resemble that of *BF*. The two differ, however, in one essential feature: whereas *GBF* traverses the implicit graph G using an explicit *subgraph* G', *BF* traverses G using a *subtree* T. This difference results from step 2 of *BF* where, if a node is found to have two parents, an *irrevocable decision* is made to *discard* the one with the higher f value. That leaves each tip node in T to represent one unique solution base (the path leading to that node) and permits us to characterize the merit of each node with one parameter. Such irrevocable decisions are often justified in state-space search where the relative merits of the two contending solution bases usually will not be altered by new information regarding their common extension and where, more significantly, if one base is solvable, so is the other. These decisions are rarely justified in searching AND/OR graphs though, because one parent can be part of a solution graph, whereas the other parent may not be in one. So, by discarding one parent, we may be removing an edge from the only solution graph possible.

Note, however, that under certain circumstances parent-discarding may not be allowable even in path-seeking problems. For example, if we must find a path containing two edges with equal cost, we run the risk that the discarded subpath will turn out to contain the cost-matching edge needed for a solution. For that reason, *BF* can be guaranteed not to miss a solution only when the criterion qualifying a path as a solution invokes the properties of the *last* node on the path, but not when it depends on global properties of the path as a whole. This condition is assumed throughout our discussions, thus ruling out problems such as: "find the shortest path containing two equal edges."

In the preceding section we saw that if an optimal solution is required, *GBF* should be modified by retarding its termination test, thus leading to algorithm *GBF**. A similar modification can be performed on *BF* if we only delay its termination until a goal node is actually selected for expansion. The resulting procedure would naturally be called *BF**, but its optimality guarantees are more involved than those of *GBF** because of the irrevocable parent-selection step of *BF*. For example, if we seek a path to a goal that has the smallest difference between the most costly and the least costly edges, we cannot discard one parent from future consideration simply because it currently represents a path with a higher difference than a competitor parent. The preference can easily be reversed through extension of the subpath common to the two parents (see exercise 2.4). In the next section and in Chapter 3 we establish conditions that guarantee that *BF** terminates with an optimal solution.

2.4 SPECIALIZED BEST-FIRST ALGORITHMS: Z*, A*, AO, AND AO*

2.4.1 Why Restrict the Evaluation Functions?

In their current general form, the algorithms describing *BF* and *GBF* are merely skeletons of strategies and are far from exhibiting the details necessary for implementation. Since the preference functions, f for *BF* and both f_1 and f_2 for *GBF*, remain arbitrary, the algorithms do not specify how these functions are computed, where the information needed for deciding the "best" choice comes from, or how it propagates through the graph. These issues constitute a major component of the search effort and will, therefore, play an important role in the taxonomy of best-first algorithms.

In large graphs, the computation of f_1 would be a hopeless task if each candidate subgraph had to be evaluated separately or if the entire set of candidates had to be reevaluated afresh with each node expansion. In the example of Figure 2.9, however, the computation of f_1 was facilitated by two main features:

1. *Shared computations* —intermediate computations in the evaluation of some solution candidates could be saved and reused in the evaluation of other candidates.

2. *Selective updating* —only ancestors of newly expanded nodes required updating, whereas all other nodes retained their values unaltered.

These two useful features are results of the **recursive** nature of the cost function chosen as a target of pursuit, that is, because the cost of solving node n could be determined from the costs of solving its successors. Indeed, we saw in Figure 2.9(*e*) how attempting to minimize the number of graph edges (a nonrecursive cost measure) led to computational difficulties, and how these difficulties disappeared as soon as we settled for minimizing a compromised measure, the number of edges in the tree-unfolding of the graph, which can be computed recursively. Much greater complications would have arisen had we chosen a more elaborate minimization measure, such as the median weight of the edges in the graph. (The median of a set of n numbers is the $(n/2)$th smallest number in the set.) This measure is notoriously difficult because it defies recursive formulation; the median of a set cannot be computed from the medians of its subsets but requires the values of every individual element in the set. (!)

Fortunately common cost or merit criteria do not possess these difficulties. They exhibit a certain regularity in the way they combine, which greatly simplifies the implementation of best-first strategies. This regularity is formalized in the next section using the domain of AND/OR graphs.

2.4.2 Recursive Weight Functions

For a given solution graph G, we designate its weight by W_G, where W_G is that property of G chosen as an optimization measure, representing either merit (Q) or cost (C). If we remove from G all nodes but those that are descendants of some given node n, the remaining portion of the graph is a solution graph for n, and its weight is denoted by $W_G(n)$. In general, the weight of any solution graph may be a complex function of all the quantities in the graph: node weights (e.g., representing processing delays), arc weights (e.g., representing transition costs), and terminal node weights (e.g., representing final payoffs).

DEFINITION: *A weight function $W_G(n)$ is* **recursive** *if for every node n in the graph*

$$W_G(n) = F[\mathbf{E}(n);\ W_G(n_1), W_G(n_2), \ldots, W_G(n_b)] \qquad (2.1)$$

where: $n_1, n_2, \ldots, n_b$ *are the immediate successors of n.*
$\mathbf{E}(n)$ *stands for a set of local properties characterizing the node n.*
F is an arbitrary combination function.

If such a combining function F exists, it is possible to evaluate the merit of any given solution graph from the bottom up, starting at the payoffs associated with the terminal nodes and working upward until the merit of the entire solu-

tion graph is computed at the root node. The process is similar to the "solve" labeling procedure described in Section 1.2.3 and will henceforth be called the **cost labeling, merit labeling,** or **weight labeling** procedure, as the case requires. F is sometimes called the **rollback function**.

Let us demonstrate the weight-labeling procedure on a few typical examples:

1. **The Counterfeit Coin problem with the maximum number of tests as a weight measure**. The solution objects are test-specification graphs where the terminal nodes represent an identified coin and the nonterminal nodes are of two types: *action nodes* (OR nodes with a single successor specifying the type of test to be conducted) and *outcome nodes* (AND nodes whose AND links point at the three possible outcomes of each test). The overall cost of a given solution graph G is taken to be the number of tests along the longest path in the graph. Thus the cost combining function can be written:

$$C_G(n) = \begin{cases} 0 & \text{if } n \text{ is terminal} \\ 1 + C_G(n_1) & \text{if } n \text{ is an OR node} \\ \max\limits_{i=1,2,3} [C_G(n_i)] & \text{if } n \text{ is an AND node} \end{cases} \tag{2.2}$$

2. **The Counterfeit Coin problem with the expected testing cost as a weight measure.** Let $c(n, n')$ be the cost of the test represented by the arc from n to n', and let p_1, p_2, and p_3 be the probabilities associated with the three outcomes of any given test. The expected cost of the strategy represented by a solution graph G rooted at n is given by:

$$C_G(n) = \begin{cases} 0 & \text{if } n \text{ is terminal} \\ c(n, n_1) + C_G(n_1) & \text{if } n \text{ is an OR node} \\ \sum\limits_{i=1,2,3} p_i C_G(n_i) & \text{if } n \text{ is an AND node} \end{cases} \tag{2.3}$$

3. **Game-playing strategies**. The quality Q of a given strategy S is determined by the payoff $v(n)$ given to player 1 from the terminal node n which is eventually reached. Assuming, however, that player 2 acts in such a way as to minimize this payoff, we have:

$$Q_S(n) = \begin{cases} v(n) & \text{if } n \text{ is terminal} \\ Q_S(n_1) & \text{if } n \text{ admits moves by player 1} \\ \min\limits_i Q_S(n_i) & \text{if } n \text{ admits moves by player 2} \end{cases} \tag{2.4}$$

Thus the quality of any given strategy is given by the lowest payoff over all the terminal nodes in the graph representing this strategy.

These three examples represent the use of the two most common cost-combination rules: maximum-cost and (weighted) sum-cost. In these examples the local properties $\mathbf{E}(n)$ that influence the combination rule are the costs and probabilities associated with the arcs and some characteristics of the node itself, for example, whether it is a terminal node, an OR node, or an AND node.

2.4.3 Identifying G_0, The Most Promising Solution-Base Graph

We will now demonstrate how the recursive nature of the weight measure W simplies the task of rating the candidate solution bases. We assume that the weight signifies the quality or merit of a given solution with the understanding that *cost measures are simply the negative of quality*.

If the entire search space had been explored, an *optimal* solution graph could be constructed and its quality $Q^*(s)$ could be computed by taking the maximum of $Q_G(s)$ over all solution graphs rooted at s. This maximization can be performed recursively by the following **merit-labeling procedure**:

1. If n is a terminal node, then $Q^*(n) = v(n)$, where $v(n)$ is the terminal payoff associated with n.
2. If n is a nongoal node without successors thus representing an unsolvable subproblem, then $Q^*(n)$ is $-\infty$.
3. If n is an AND node with successors $n_1, n_2, \ldots, n_b$, then $Q^*(n) = F[\mathbf{E}(n); Q^*(n_1), Q^*(n_2), \ldots, Q^*(n_b)]$.
4. If n is an OR node with successors $n_1, n_2, \ldots, n_b$, then $Q^*(n) = \max\limits_i F[\mathbf{E}(n), Q^*(n_i)]$.

According to this definition, $Q^*(n)$ is finite if and only if the problem represented by node n is solvable. For each solvable n, $Q^*(n)$ gives the quality of an optimal solution graph rooted at n. If s is the start node, then $Q^*(s)$ is the quality of an optimal solution to the initial problem.

For example, in example 2 of the Counterfeit Coin problem in Section 2.4.2, the optimal cost $C^*(n)$ associated with subproblem n is computed by the cost-labeling procedure:

$$C^*(n) = \begin{cases} 0 & \text{if } n \text{ is terminal} \\ \min\limits_i [c(n, n_i) + C^*(n_i)] & \text{if } n \text{ is an OR node} \\ \sum\limits_i p_i C^*(n_i) & \text{if } n \text{ is an AND node} \end{cases} \tag{2.5}$$

In game-playing (example 3 in Section 2.4.2), the merit-labeling procedure yields the celebrated minimax rule:

$$Q^*(n) = \begin{cases} v(n) & \text{if } n \text{ is terminal} \\ \max_i [Q^*(n_i)] & \text{if } n \text{ admits moves by player 1} \\ \min_i [Q^*(n_i)] & \text{if } n \text{ admits moves by player 2} \end{cases} \tag{2.6}$$

The **optimal solution graph** G^* is likewise defined in terms of Q^*.

DEFINITION OF G^*:

1. *The start node s is in G^*.*
2. *If an AND node is in G^*, all its (AND) successors are in G^*.*
3. *If an OR node n is in G^*, one successor n' of n is in G^* such that* $F[\mathbf{E}(n); Q^*(n')] = \max_i F[\mathbf{E}(n); Q^*(n_i)]$.

If the search space is only partially explored, there is no way, of course, to compute the optimal solution or even to identify which among the solution bases unraveled leads to an optimal solution. However, if each node n at the search frontier is assigned an *estimate* $h(n)$ of $Q^*(n)$, we may invoke the face-value principle and compute what appears to be the **most promising potential solution**. This solution is computed in two steps:

1. Assign to each tip node n of a solution base an evaluation function $h(n)$ which estimates the quality of the best solution rooted at n.
2. Use the combining function F and the merit-labeling procedure *as if* the estimates $h(n)$ were true terminal payoffs (the face-value principle).

These two steps offer constructive definitions of the function f_1 and the most promising solution base G_0, both of which are used in step 2 of *GBF*. First, we define a **backed-up evaluation function** $e(n)$ which gives an estimate of $Q^*(n)$ for all nodes of the explored graph:

$$e(n) = \begin{cases} h(n) & \text{if } n \text{ is in OPEN} \\ F[\mathbf{E}(n); e(n_1) \cdots e(n_b)] & \text{if } n \text{ is a CLOSED AND node} \\ \max_i F[\mathbf{E}(n); e(n_i)] & \text{if } n \text{ is a CLOSED OR node} \end{cases} \tag{2.7}$$

This evaluation function gives rise to the following top-down definition of G_0.

DEFINITION OF G_0:

1. *The start node s is in G_0.*
2. *If a CLOSED AND node is in G_0, all its (AND) successors are in G_0.*
3. *If a CLOSED OR node n is in G_0, one (OR) successor n' of n is in G_0, such that* $e(n) = F[\mathbf{E}(n); e(n')]$.

In practice, the identification of G_0 can be done bottom-up, together with the computation of $e(n)$, by simply marking the OR successor that has the highest value of e among its siblings.

In this procedure, G_0 is identified without making explicit use of the graph selection function f_1, however, if one wishes to assess the promise $f_1(G')$ of any candidate solution-base graph G', all that is required is to compute e bottom-up along the arcs in G' (each OR node will have only one successor since G' is a solution base). The value finally assigned to the start node $e_{G'}(s)$, is equal to the required promise estimate $f_1(G')$:

$$f_1(G') = e_{G'}(s) \tag{2.8}$$

It is interesting to see how *BF* can use this procedure to compute the node selection function $f(n)$ while traversing an OR graph. Since *BF* explores the graph using a traversal tree T, each node n on OPEN takes part in only one solution base, given by the path $P(n)$ from s to n, currently in T. To decide (in step 3 of *BF*) which node warrants expansion, the evaluation function $e_{P(n)}(s)$ must be assigned to each node on OPEN, where $e_{P(n)}(s)$ is computed bottom-up (using F as in Eq. 2.7) along the unique path connecting s and n. Thus the function $f(n)$, computed in step 6 of *BF*, is merely the negative of $e_{P(n)}(s)$:

$$f(n) = -e_{P(n)}(s) \tag{2.9}$$

(Tradition dictates that f should represent costs and therefore be minimized, whereas e represents merits and is to be maximized.)

Under normal circumstances (e.g., when the solution weight is determined by the sum cost of its edges), we need not traverse the entire path from n to s to compute $e_{P(n)}(s)$. A few auxiliary parameters stored at the parent of n would permit $e_{P(n)}(s)$ to be calculated locally and be transmitted, from father to son, with each node expansion. These parameters correspond to the notion of state description in dynamic programming (Dreyfus and Law, 1977). For example, in the case of the additive cost measure, we need to transmit only one parameter, $g(n)$, which is the overall cost along the path $P(n)$, and this gives rise to the celebrated A^* algorithm (Nilsson, 1971) with heuristic function:

$$f(n) = -e_{P(n)}(s) = g(n) + h(n) \tag{2.10}$$

for each node on OPEN.

2.4.4 Specialized Best-First Strategies

If the selection functions f_1 (for *GBF*) and f (for *BF*) are computed recursively by the rollback procedure of Eqs. (2.7) and (2.8), these algorithms assume special names; *BF* becomes what we will call algorithm **Z** and *GBF* becomes

AO. If we further modify these algorithms by delaying the termination test as was done in GBF^*, Z becomes $\mathbf{Z}^*$ and AO becomes $\mathbf{AO}^*$.

The motivation for delaying the termination test until G_0 is selected lies, of course, in the hope of returning an optimal solution when one is required. This hope will be realized for AO^* if the heuristic estimates h of the tip nodes are optimistic (i.e., overestimating merits and underestimating costs), because under such conditions the estimates $f_1(G')$ are also optimistic for all solution bases, and, invoking the same set-splitting argument used for GBF^*, AO^* cannot return a suboptimal solution.

The optimality condition for Z^* is slightly more involved due to the parent-selection step of BF. In addition to requiring that the estimates $h(n)$ be optimistic, we still need to ascertain that by discarding the parent with the inferior f value we do not in fact throw away the optimal solution path. This would be guaranteed when the rank order of two contending parents remains independent of the path rooted at their common child. Therefore, we need also to require that F, the rollback function that defines the cost of any given path, satisfies:

$$F(\mathbf{E}_1, h) \geqslant F(\mathbf{E}_2, h) \Longrightarrow F(\mathbf{E}_1, h') \geqslant F(\mathbf{E}_2, h') \qquad (2.11)$$

for all $\mathbf{E}_1$, $\mathbf{E}_2$, h and h'. If the path-cost definition satisfies the **order-preserving** property of Eq. (2.11) and the estimates $h(n)$ are optimistic, Z^* terminates with an optimal (lowest cost) path. Fortunately the two most commonly used weight measures—the additive cost (where $F = c(n, n') + h(n')$) and the max cost (where $F = \max[c(n, n'), h(n')]$)—do satisfy this property. A more general discussion of the order-preserving restriction is given in Section 3.3, pertaining to the optimality of BF^*.

So far we have encountered eight variations of best-first strategies, four suitable for AND/OR graphs (GBF, GBF^*, AO, and AO^*) and four limited to searching OR graphs (BF, BF^*, Z, and Z^*). Before introducing our last strategy, the celebrated specialization of Z^* called A^*, it is instructive to reiterate the relationships between the algorithms defined so far. They are illustrated in the hierarchical diagram of Figure 2.10, where each arrow specifies the restriction placed on the parent algorithm in order to produce the more specialized successor.

Figure 2.10 also shows the position of A^* in this hierarchy; it is a specialization of Z^* where the target of pursuit is the path of minimum sum-cost. A^* may be used both for optimization problems and satisficing problems because the shortest path is the most natural choice for the small-is-quick principle. The reader may wonder why Figure 2.10 does not contain an additive-cost successor of AO^*, which, in fact, was described in the example of Figure 2.9. The reason is simply that the additive-cost condition does not introduce a significant enough change in the algorithm to warrant a new name. A^*, too, is only slightly different from Z^*; however, in order to keep our notation consistent with the

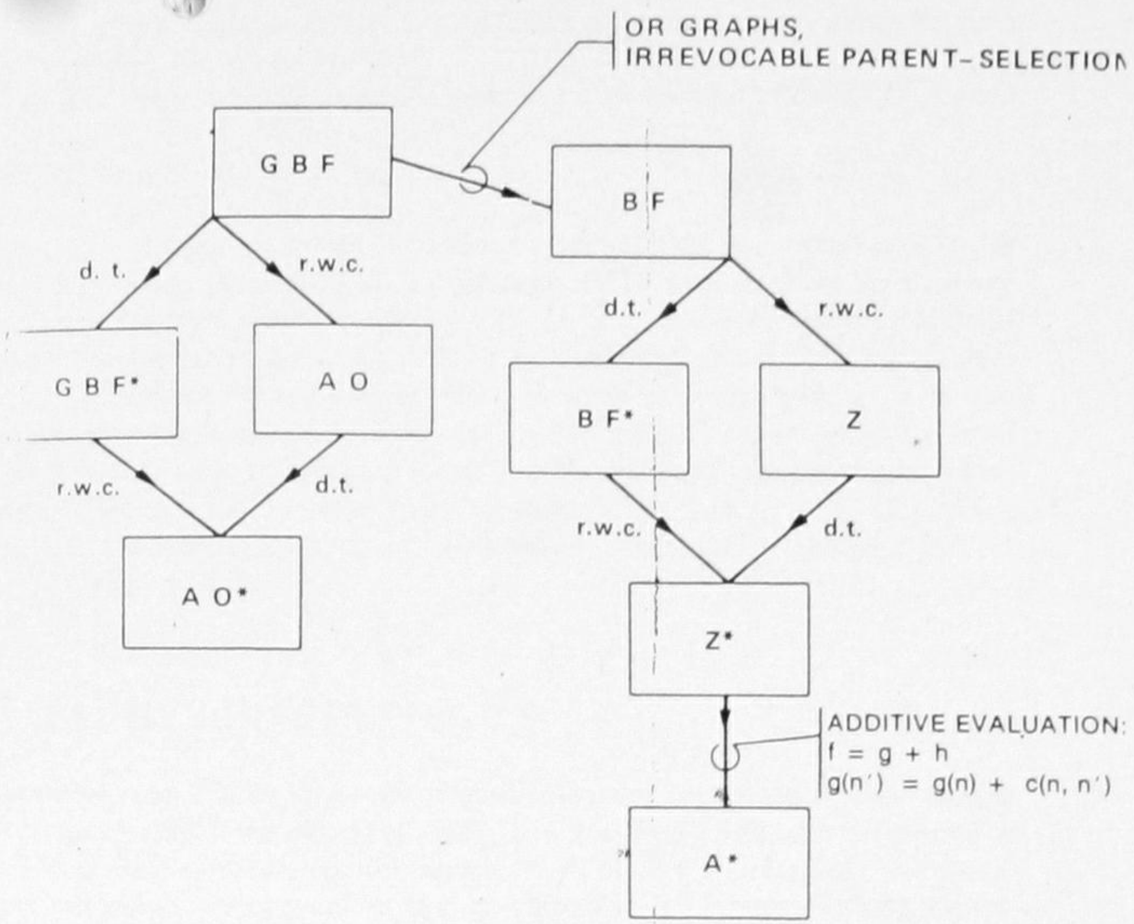

Figure 2.10
Hierarchical diagram showing the relationships between the nine best-first algorithms discussed in Section 2.4 (d.t. stands for delayed-termination; r.w.c. stands for recursive-weight-computation).

large body of literature on A^* and out of respect for its inventors (Hart, Nilsson, and Raphael, 1968) we prefer to keep its traditional name intact.

Since AO^* and A^* are the most popular strategies in use, we now give an explicit description of these two algorithms, followed by a few words of explanation. A formal treatment of the properties of A^* and BF^* is to be found in Chapter 3.

The AO^* Algorithm

1. Create a search graph G' initially consisting of the start node s. Put s on OPEN. Let G_0 initially include just s.
2. Trace down the marked connectors of a subgraph G_0 (G_0 stands for the most promising solution-base graph, and its connectors are marked in step 7) and inspect its tip nodes.

3. If OPEN $\cap$ G_0 is empty, exit with G_0 as a solution graph; otherwise
4. Select an OPEN node n in G_0 using a selection function f_2. Remove n from OPEN and place it on CLOSED.
5. Expand n, generating all its successors; put them on OPEN and add them to the search graph G' with pointers back to n.
6. Calculate for each nonterminal successor, n', a heuristic merit estimate $h(n')$. If n' is terminal with payoff $v(n')$, set $h(n') = v(n')$ and move n' from OPEN to CLOSE; if n' is unsolvable, set $h(n') = -\infty$. If n' is already in G', set $h(n') = e(n')$.
7. Revise the evaluation $e(\cdot)$ of n and all its ancestors, using the rollback function F. Mark the best arc from every updated OR node to identify G_0.
8. If $e(s)$ is updated to $-\infty$, exit with failure. Otherwise remove from G' all nodes that can no longer influence the value of s.
9. Go to step 2.

AO^* terminates when the current G_0 has no more OPEN tip nodes, that is, when all its tip nodes are solved terminals (if any of its terminals were unsolvable, s would be marked $-\infty$). If the weight of a solution graph G is given recursively by Eq. (2.1) and the estimates $h(n)$ are optimistic assessments of the weight of the optimal solution to n, then AO^* will exit with the optimal-weight solution.

In step 8 the process of pruning away nodes that cannot provide useful information remains unspecified. Indeed, with a general AND/OR graph and an arbitrary F, not many nodes would meet this pruning criterion, since subsequent expansions may significantly alter the values of all nodes on OPEN and their ancestors. Only unsolvable nodes are truly safe from future alterations and can be pruned away. If, however, F is known to comply with additional restrictions (e.g., that $e(n)$ cannot increase by expansion) or if the graph searched is a tree, this pruning can result in substantial savings. We therefore postpone description of the pruning process until we have discussed some specific types of F functions.

The A Algorithm*

1. Put the start node s on OPEN.
2. If OPEN is empty, exit with failure.
3. Remove from OPEN and place on CLOSED a node n for which f is minimum.
4. If n is a goal node, exit successfully with the solution obtained by tracing back the pointers from n to s.
5. Otherwise expand n, generating all its successors, and attach to them pointers back to n. For every successor n' of n:
   a. If n' is not already on OPEN or CLOSED, estimate $h(n')$ (an estimate of the cost of the best path from n' to some goal node), and calculate $f(n') = g(n') + h(n')$ where $g(n') = g(n) + c(n, n')$ and $g(s) = 0$.

   b. If n' is already on OPEN or CLOSED, direct its pointers along the path yielding the lowest $g(n')$.
   c. If n' required pointer adjustment and was found on CLOSED, reopen it.
6. Go to step 2.

The more general algorithm Z^* follows essentially the same steps as A^* with one modification. In step 5a the calculation of $f(n')$ may invoke an arbitrary function of the form $f(n') = F[\mathbf{E}(n), f(n), h(n')]$, where $\mathbf{E}(n)$ is a set of local parameters characterizing n (e.g., $c(n, n')$). Similarly, in step 5b Z^* will direct pointers along the path of lowest $f(n')$ (instead of lowest $g(n')$). It is also worth noting that the breadth-first strategy (Section 2.2.2) is a special case of A^* with $h = 0$ and $c(n, n') = 1$ for all successors. Similarly, the uniform-cost strategy (Section 2.2.2) is also a special case of A^* with $h = 0$. Depth-first strategies, on the other hand, can be obtained from Z^* by setting $f(n') = f(n) - 1$, $f(s) = 0$.

2.5 HYBRID STRATEGIES

The three main search strategies described in the preceding section, that is, **hill-climbing** (HC), **backtracking** (BT), **and best-first** (BF), can be viewed as three extreme points of a continuous spectrum of search strategies. To demonstrate this point, it is convenient to characterize search strategies along the following two dimensions:

1. **Recovery of pursuit** (R): The degree to which a search strategy allows recovery from disappointing search avenues to reaccess previously suspended alternatives.
2. **Scope of evaluation** (S): The number of alternatives considered in each decision.

Along the R dimension we find HC at one extreme, permitting no recovery whatsoever of suspended alternatives, and BT and BF at the other extreme, where all decisions are tentative and all suspensions are revocable. Along the S dimension we find HC and BT focusing narrowly on the set of newly available alternatives, whereas BF examines before each decision the entire set of available alternatives, those newly generated as well as all those suspended in the past.

These extreme cases are illustrated schematically in Figure 2.11. The shaded area represents a continuous spectrum of search strategies that combine some characteristics from each of the three prototypes in order to achieve a better mix of their computational features. To recapitulate, recall that HC spends the minimal amount of computation at the risk of missing the solution. BF

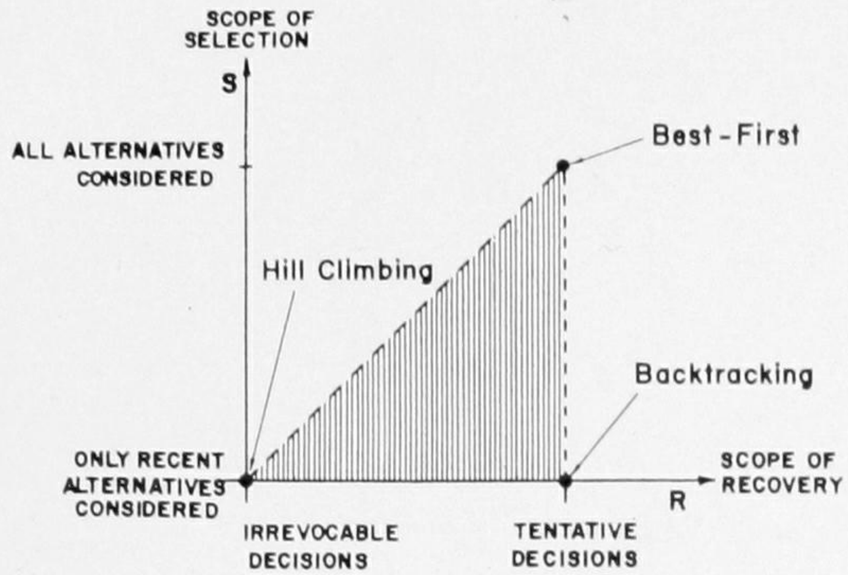

Figure 2.11
Hill-climbing (HC), backtracking (BT), and best-first (BF) strategies as three extreme points in a 2-dimensional space of hybrid strategies.

proceeds wisely and cautiously; it will find the solution (if one exists) after the fewest possible decisions but must pay dearly in node storage for these judicious decisions. *BT* is committed to maintaining in storage only a single path containing the set of alternatives leading to the current decision point. It proceeds forward heedlessly by considering only a narrow scope of young alternatives, and so it usually pays for its memory economy in increased runtime.

2.5.1 *BF-BT* Combinations

If one cannot afford the memory space required by a pure *BF* strategy, various *BF-BT* combinations can be implemented that cut down the storage requirement at the expense of narrowing the evaluation scope. One such combination is depicted in Figure 2.12(*a*), where the *BF* strategy is applied at the top of the search graph (represented by the shaded area with the irregular frontier) and a *BT* strategy at the bottom (represented by the left-to-right arrow). As soon as the memory space allotted to the *BF* strategy is exhausted, the *BT* search takes over from the best node on OPEN until the entire graph beneath that node is traversed. If *BT* fails to find a solution, this node is declared unsolvable and the second best node on OPEN is taken as a new root for a *BT* search, and so on.

An opposite approach is shown in Figure 2.12(*b*). Here *BT* is employed at the top of the graph, whereas *BF* is used at the bottom. We begin *BT* until a depth-bound d_0 is reached. At this point, instead of backing up, we employ a

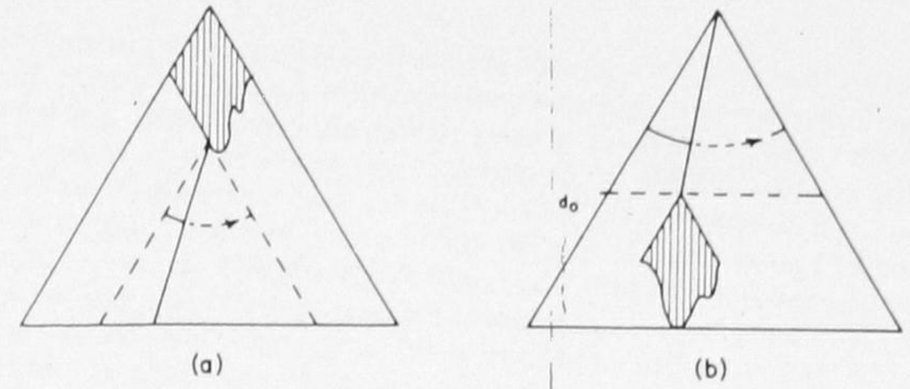

Figure 2.12
Schematic representations of two BF-BT hybrid strategies. Part (*a*) represents a BF search on top (shaded area) followed by BT ending. Part (*b*) represents a BT start (left-to-right arrow) followed by BF ending.

BF search from the node at d_0 until it returns a solution or exits with failure. If it fails to find a solution, we return to backtracking and again use *BF* upon reaching the depth-bound d_0. The parameter d_0 is chosen in such a way that the memory consumed by the *BF* strategy will not exceed the available allotment. It is harder to control than the strategy in Figure 2.12(*a*), where memory utilization itself was the mechanism that triggered the transition from *BF* to *BT*, but employing *BF* at the bottom of the graph has an advantage. *BF* performance is at its best when its guiding heuristic is more informed, and this usually happens at the bottom of the search graph, where the differences between promising nodes and dead-end nodes are more apparent.

A more elaborate scheme is shown in Figure 2.13, where *BF* is performed locally and *BT* globally. We begin searching in a best-first manner until a memory allotment M_0 is exhausted. At this point we regard all the nodes on OPEN as direct successors of the root node and submit them to a *BT* search. *BT* selects the best among these successors and "expands" it using *BF* search, that is, it submits the chosen node to a local *BF* search until a new memory allotment M_0 is consumed and treats the new nodes on OPEN as direct successors of the node "expanded." In summary, this strategy amounts to running an informed depth-first search where each node expansion is accomplished by a memory-limited *BF* search and the nodes on OPEN are defined as children. The overall memory required for this strategy is equal to M_0 times the maximum length of the global traversal path. It is linear with the depth of the search graph, in contrast to the exponential memory growth required by purely *BF* strategies.

Ibaraki (1978b) has proposed another scheme of mixing *BF* and *BT* strategies. Whereas informed depth-first seach considers for expansion only the newly generated nodes, Ibaraki's scheme expands the scope of selection to include, in addition to the set of new successors, the k best alternatives from the set considered at the previous decision. Thus, at the ith decision, the node ex-

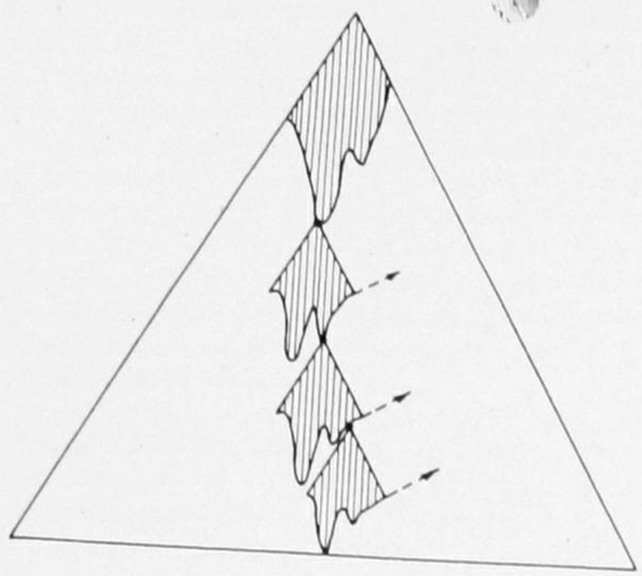

Figure 2.13
Schematic representation of a hybrid BF-BT strategy. Local BF searches (shaded areas) are nested in a global BT traversal.

panded is the best within a subset of OPEN, which we will call FOCAL(t), containing the set of newly generated nodes together with the k best nodes of FOCAL($t-1$). Clearly, the case $k = 0$ corresponds to pure depth-first while $k = \infty$ amounts to pure *BF*. Although this scheme does not possess an elegant geometrical representation such as those shown in Figures 2.12 and 2.13, Ibaraki was able to show that the storage consumed by this strategy is equal to bN^{k+1} where N is the depth of the search tree and b is its branching degree. One can, therefore, adjust k to meet practical storage limitations.

2.5.2 Introducing Irrevocable Decisions

The need to combine *BF* and *BT* strategies was a result of computational considerations aimed at finding a good storage-time mix when storage space is limited. These combinations retained the purely tentative nature of all decisions. As soon as we introduce irrevocable decisions of the hill-climbing type, permanently discarding alternatives before they are proven fruitless, we also introduce a risk of missing a solution and exiting with failure, or returning a bad solution when an optimal one is required. Occasions may arise in which even the storage required for local best-first search is too large and purely depth-first strategies are too time consuming. In such cases, it may be desirable to take a risk and prune the tree so far generated by the search in order to free needed storage space and allow the search to continue deeper.

A pruning technique that has gained some popularity is called **staged search** (Nilsson, 1971) and proceeds as follows. Instead of maintaining in storage the

entire search tree generated by a *BF* search, only the most promising subtree is retained. At the end of each storage reclamation stage, some subset of nodes on OPEN (e.g., those having the lowest values of f) are marked for retention. The best paths of these nodes are remembered, and the rest of the tree is thrown away. The search then continues in a best-first manner until the storage allotment is again exhausted, thus marking the end of a new storage reclamation stage, and so on. This strategy can be regarded as a hybrid between best-first and hill-climbing; whereas pure hill-climbing discards all but a single best candidate, staged search discards all but a *set* of best candidates. Still, following the best-first philosophy, the scope of selection now includes all candidates available at the decision time.

An important variant of staged search also introduces an element of *BT* search into the *HC-BF* mixture. It is tailored after the hybrid strategy of Figure 2.13, but instead of submitting the entire frontier of the local *BF* search to a global *BT*, only the best subset of these frontier nodes are retained whereas the rest are discarded permanently. Of course, even if A^* is used in each stage and the whole process does terminate in a solution path, there is now no guarantee that it is an optimal path. In Chapter 5 we analyze a version of this search strategy and show that under certain conditions one can choose the search parameters (e.g., the size of the local search tree and the threshold of pruning) in such a way that the search will run in linear average time and the solution found will almost always fall in an arbitrarily close neighborhood of the optimal solution.

2.6 BIBLIOGRAPHICAL AND HISTORICAL REMARKS

The book by Horowitz and Sahni (1978) contains a thorough discussion of backtracking and other search methods including a more detailed specification of the computational steps involved. Improvements on the basic backtracking algorithm are reported and analyzed by Gaschnig (1979a), Haralick and Elliot (1980), and Nudel (1983). Look-ahead methods of solving satisficing problems, sometimes called "relaxation," "constraint propagation," or "consistency checking," were introduced by Rosenfeld, Hummel, and Zucker (1976), Waltz (1975), Montanari (1974), and Mackworth (1977). These are especially effective in object-recognition tasks where constraints have a local character (e.g., only few lines meet in each vertex of a polyhedran object).

Systematic, uninformed graph-searching procedures are normally discussed under the topic "shortest path problems." Moore's (1959) maze searching algorithm is essentially a breadth-first strategy, while Dijkstra's (1959) two-point shortest path algorithm is what we call uniform-cost (or A^* with $h = 0$). An up-to-date survey of shortest path methods is given in Deo and Pang (1982). Dynamic programming (Bellman and Dreyfus, 1962) is a recursive formulation of breadth-first search which, for search spaces with regular structures, may yield analytical expressions for the optimal cost or the optimal policy.

The use of heuristic information to increase search efficiency has been studied both in artificial intelligence (*AI*) and in operations research. In operations research the most common representation for this information has been the bounding functions which govern the branch-and-bound search (Lawler and Wood, 1966). In AI, general heuristic search methods applicable to a variety of problem domains were developed by Newell, Shaw, and Simon (1957, 1960). The use of evaluation functions to direct search in graphs was proposed by Lin (1965) who applied it to the Traveling Salesman problem. Doran and Michie (1966) formulated and experimented with a *BF* search algorithm (the Graph Traverser) guided by a distance-to-goal estimate. Hart, Nilsson, and Raphael (1968) recognized the applicability of the algorithm to optimization problems when the estimates are optimistic and have established the basic properties of the A^* algorithm (to be discussed in Section 3.1).

A^* is one of several versions of heuristic shortest-path algorithms introduced by several workers (Michie and Ross, 1970; Slagle and Bursky, 1968; Pohl, 1970; and Nilsson, 1980). Our description of A^* follows that of Nilsson (1971). It differs from that of Nilsson (1980) where the condition $h \leqslant h^*$ is imposed (otherwise the algorithm is called A) and where a search graph (in addition to a search tree) is maintained (see discussion in Section 2.3.1). Bidirectional versions of A^*, which simultaneously explicate the graph from both the start node and from the set of goal nodes, were introduced by Pohl (1971), and further investigated by de Champeaux and Sint (1977).

Techniques for searching AND/OR graphs were first introduced by Amarel (1967), Nilsson (1969), and Chang and Slagle (1971). Our description of AO^*, and to a great extent also *GBF* and GBF^*, is based on that of Nilsson (1971). More recent versions of AO^*, making use of the monotonic properties of the cost functions, have been introduced by Martelli and Montanari (1973, 1978). Of the nine best-first algorithms mentioned in Section 2.4 (see Figure 2.10), only A^* and AO^* can be found in the general literature under identical acronyms. The names given to the other seven algorithms were contrived so as to best convey their similarities and differences.

EXERCISES

2.1 Consider the 8-Puzzle with start node $s = \begin{bmatrix} 2 & 8 & 3 \\ 1 & 6 & 4 \\ 7 & & 5 \end{bmatrix}$ and goal node $\gamma = \begin{bmatrix} 1 & 2 & 3 \\ 8 & & 4 \\ 7 & 6 & 5 \end{bmatrix}$

Execute the A^* algorithm using the following heuristics:

a. $h = 0$

b. h_1 = number of misplaced tiles

c. h_2 = sum of Manhattan distances

d. h_3 = 3 times seq(n) where seq(n) counts 1 for a central tile and counts 2 for each tile on the board perimeter that is not followed (in clockwise order) by its proper successor.

In each of the preceding cases count the number of nodes expanded and display the explicit search tree at termination.

e. Evaluate the performance of hill-climbing strategies using the heuristics in (a), (b), and (c).

2.2 A language L has a four-character vocabulary $V = \{\epsilon, A, B, C\}$ where ϵ is the empty symbol. The probability that character v_i will be followed by v_j is given by the following matrix:

$P(v_j \mid v_i) =$

v_i \ v_j	ϵ	A	B	C
ϵ	1/4	1/4	1/4	1/4
A	1/2	0	1/4	1/4
B	1/8	1/2	1/8	1/4
C	1/4	1/8	1/2	1/8

Do parts (a), (b), and (c) using the uniform-cost algorithm.

a. Find the most likely five-symbol string that starts with ϵ.

b. Find the most likely five-symbol string that starts and ends with ϵ.

In transmitting messages from L, some characters may be corrupted by noise and be confused with others. The probability that a transmitted character v_j will be interpreted as v_k is given by the confusion matrix:

$P_c(v_k \mid v_j) =$

v_j \ v_k	ϵ	A	B	C
ϵ	.9	.1	0	0
A	.1	.8	.1	0
B	0	.1	.8	.1
C	0	.1	.1	.8

c. Find the message most probably to have been transmitted given that the string $\epsilon\ \epsilon\ B\ C\ A\ A\ \epsilon\ \epsilon$ is received, and knowing that all messages must begin and end with ϵ.

d. Repeat parts (a), (b), and (c) using backtracking. Compare the number of nodes generated and the storage required by the two algorithms.

e. Can you think of a reasonable heuristic function h to be used in A^* on part (c)?

(Remark: You may wish to consult the literature on Viterbi's algorithm, e.g., Forney, 1973; or Bahl, Jelinek, and Mercer, 1983.)

2.3 a. Find a reasonable evaluation function e and execute AO^* to find the solution to the card game of exercise 1.1.

b. Consider a modified version of the game where the cards are labeled 7, 5, 8, 2, and 1, and the first player wins as many points as the number labeled on the last remaining card. Find an optimistic evaluation function and execute AO^* to find the best first move. Specify a pruning criterion for step 8 of AO^*.

c. Repeat part (b) for another modification of the game: each player receives a score equal to the sum of the card values that he removed minus those removed by the opponent.

d. Repeat part (c) assuming that the second player, in his turn, flips a coin to decide his move.

e. Repeat part (b) assuming that the second player always removes the leftmost card, that the game begins with the set 4, 3, 2, 5, 7, 1, 8, 6, and that the first player receives a score equal to the sum of the card values that he removes.

f. Solve part (e) using A^*.

2.4 This problem requires the execution of GBF^* on an OR graph. Consider the following variation of the card game, played by a single player. In each move the player removes one element from either end of a given list of numbers. His cost is equal to the difference between the highest and the lowest number removed at odd moves.

a. Write a detailed description of a GBF^* algorithm that can solve instances of this problem. Determine the set of heuristic parameters $\boldsymbol{h}$ necessary to guarantee optimality, specify the selection functions f_1 and f_2, and devise a convenient code for storing solution bases.

b. Under what conditions can candidate solution bases be purged from future considerations?

c. Find the optimal sequence of moves by executing the algorithm in part (a) on the list 8, 3, 9, 7, 2, 6, 1.

2.5 a. Find the expected number of tests required by the solution to the Counterfeit Coin problem (exercise 1.6).

b. Prove that the value found in part (a) is optimal, i.e., the strategy that minimizes the worst-case number of tests also minimizes its average case value.

c. Find the expected cost of the solution to exercise 1.6, assuming that the cost of each test is equal to the number of coins weighed in that test.

d. Assume that the original Counterfeit Coin problem is submitted to an informed depth-first strategy using the evaluation function $f(n) = \log_3 S_n$ where S_n stands for the total number of possible outcomes to subproblem n. Which test will be the first to be explored by this strategy? How many first-level tests will eventually be explored by this strategy?

e. Repeat part (c) if the objective is to find a strategy with the least expected number of tests.

2.6 Reformulate AO^* to be executed on OR graphs with the objective of finding the lowest (sum) cost path from s to a set Γ of goal nodes, given that h is optimistic. How will it differ from A^*? Discuss the advantages of the way each algorithm handles nodes which are generated and found already on CLOSED.

Chapter 9

Formal Properties of Heuristic Methods

It is often said that heuristic methods are unpredictable; they work wonders *most* of the time, but may fail miserably *some* of the time. Indeed, some heuristics greatly reduce search effort but occasionally fail to find an optimal or even a near-optimal solution. Others work efficiently on most problems in a given domain yet a rare combination of conditions and data may cause the search to continue forever.

In Parts II and III of this book we will use probabilistic analyses to assess how sure we can be that a given heuristic method will perform in accordance with expectations. By contrast, this chapter discusses those properties of heuristic methods that can be guaranteed to hold *all the time*, that is, over all problems in a given class. We will see that, contrary to a common misconception, some desirable properties of heuristics can be *guaranteed in advance*. For instance, a simple test on the type of information used by A^* could guarantee that an optimal path will eventually be found. A second simple test guarantees that A^* will retain all its virtues even without appraising duplicate nodes on CLOSED. Still another test may reveal that one heuristic function is consistently better than another in that it always causes A^* to expand a smaller number of nodes.

Although many of the formal properties discussed in this chapter are shared by several best-first strategies, we will focus our attention on A^* and only mention generalizations when the occasion arises. Thus, unless noted otherwise, we will assume that our search space is an OR graph, that our task is to find a path from s to some goal node, and that a heuristic estimate $h(n)$ is computable for every node in the graph.

More Notations on Graphs. We will be dealing exclusively with graphs in which every node has only a finite number of successors, namely, with **locally finite graphs**. This restriction is natural in problems in which only a finite number of decisions are to be made in every stage of the search process. It

excludes the class of problems in which continuous parameters must be chosen, such as in many resource allocation problems. Practically speaking, however, even continuous quantities have finite practical bounds and can be controlled only by a finite degree of precision. Therefore they can be quantized into a finite number of compartments and fit into the locally finite graph formalism.

We will find it useful to condense the verbal phrase "node n_j is a successor of the node n_i" by writing $n_j \in \mathrm{SCS}(n_i)$, stating that n_j is a member of the finite set SCS (n_i) of successors to n_i. SCS is also called the **successor operator** because, when applied to n_i, it generates all its successors.

Our graphs will be characterized by having a single start node s. In cases where a set $\{s_i\}$ of multiple starting nodes exists, we can always regard them as successors of a dummy node s with zero-cost links (see Figure 2.4). This dummy node device may not work as well in the case of multiple goal nodes because the addition of a dummy goal node may change the whole character of the search space (e.g., from a tree to a graph). For that reason, we will assume explicitly that the object of the search is to find a cheapest path from s to any **goal node** γ in the **set** Γ **of goal nodes.** Every arc will be quantified with a positive cost label $c(n_i, n_j) \geqslant \delta > 0$, and the cost of a path will be understood to be the sum-cost of all its arcs.

If node n_j is accessible from node n_i, we will let $\mathbf{P}_{n_i - n_j}$ stand for **the set of all paths going from** n_i **to** n_j and let $\mathbf{P}_{n - \Gamma}$ denote the set of paths going from n to the set Γ. The symbol $P_{n - \gamma}$ stands for *any path* from n to γ, i.e., $P_{n - \gamma} \in \mathbf{P}_{n - \Gamma}$. The **set of cheapest paths** from n_i to n_j will be denoted by $\mathbf{P}^*_{n_i - n_j}$ and the *cost* of a cheapest path by $k(n_i, n_j)$, (as in Nilsson, 1980).

The costs of paths containing s or an element of Γ will be given special names:

- $g^*(n)$—The cheapest cost of paths going from s to some node n; thus, $g^*(n) = k(s, n)$
- $h^*(n)$—The cheapest cost of paths going from n to Γ, thus, $h^*(n) = \min_{\gamma \in \Gamma} k(n, \gamma)$
- C^*—The cheapest cost of paths going from s to Γ, i.e., $C^* = h^*(s)$
- Γ^*—The set of *optimal* goals, i.e., the subset of goal nodes accessible from s by a path or paths that cost C^*

A^* explores (as does every *BF* strategy) the state-space graph G using a traversal tree T. At any stage of the search, T is defined by the pointers that A^* assigns to the nodes generated, with the branches in T directed opposite to the pointers. Whenever it is desirable to stress the fact that a given path is part of T (at some phase of the search) we will denote this pointer-path by PP, e.g., $PP_{n_1 - n_2} \in T$. Similarly, we will distinguish between a node lying *along* a path P (if P is in T) and a node lying *on* P (when P may not be in T). The union of all the branches in the traversal trees constructed during the search makes up

the **explicated** subgraph of G and will be denoted by G_e. Note that a path P can be in G_e and not in any of the traversal trees that make up G_e.

In the following sections we will examine some basic algorithmic properties of A^*. These are defined as follows.

- **Completeness**—An algorithm is said to be **complete** if it terminates with a solution when one exists.
- **Admissibility**—An algorithm is **admissible** if it is guaranteed to return an **optimal** solution whenever a solution exists.
- **Dominance**—An algorithm A_1 is said to **dominate** A_2 if every node expanded by A_1 is also expanded by A_2. Similarly, A_1 **strictly dominates** A_2 if A_1 dominates A_2 and A_2 does not dominate A_1. We will also use the phrase **"more efficient than"** interchangeably with **dominates.**
- **Optimality**—An algorithm is said to be **optimal** over a class of algorithms if it dominates all members of that class.

3.1 A^*—OPTIMAL SEARCH FOR AN OPTIMAL SOLUTION

3.1.1 Properties of f^*

The node-selection function f used by A^* consists of adding two parts, $g(n)$ and $h(n)$, where

$g(n)$ = the cost of $PP_{s - n}$, the *current* path from s to n, with $g(s) = 0$

$h(n)$ = an estimate of $h^*(n)$, such that $h(\gamma) = 0$

When we wish to specify the *actual* costs of $P_{s - n}$ and $P_{n - \gamma}$ along a particular path P, we shall write $g_P(n)$ and $h_P(n)$, respectively. Clearly, for every path P we have

(no cost sh)

$$g_P(n) \geqslant g^*(n) \qquad \text{and} \qquad h_P(n) \geqslant h^*(n) \tag{3.1}$$

When g and h coincide with their optimal values, the resulting f assumes a special meaning. Calling

$$f^*(n) = g^*(n) + h^*(n) \tag{3.2}$$

we see that $f^*(n)$ measures the **optimal cost over all solution paths constrained to go through** n. In particular, for s and γ we have

$$f^*(s) = h^*(s) = g^*(\gamma) = f^*(\gamma) = C^* \quad \text{for all } \gamma \in \Gamma^* \tag{3.3}$$

and, moreover, if n^* is any node on an optimal path from s to γ, it must satisfy

$$f^*(n^*) = C^* \qquad n^* \in P^*_{s - \Gamma} \tag{3.4}$$

To show the validity of Eq. (3.4) we reason as follows: $n' \in P^*_{s-\Gamma}$ implies that there is a solution path P through n costing C^*, that is, along this path $g_P(n)+h_P(n) = C^*$. Using the optimality of g^* and h^* (Eq. 3.1) we have $g^*(n)+h^*(n) \leqslant C^*$, but this inequality cannot be strict or else there would be a path P' along which $g_{P'}(n)+h_{P'}(n) < C^*$, thus contradicting the optimality of C^*.

The importance of $f^*(n)$ stems from its behavior on **off-track** nodes, that is, nodes not lying on any optimal solution path. Let n be any such node, it must satisfy

$$f^*(n) > C^* \qquad n \notin P^*_{s-\Gamma} \tag{3.5}$$

The reason behind Eq. (3.5) is similar to that for Eq. (3.4); suppose $g^*(n)+h^*(n) \leqslant C^*$, then there should exist a solution path P' through n such that $g_{P'}(n)+h_{P'}(n) \leqslant C^*$, qualifying P' as an optimal solution path and contradicting our assumption that n is an off-track node.

The properties in 3.4 and 3.5 amount to the **principle of optimality** of dynamic programming, stating that a path is optimal if and only if every segment of it is optimal, and is a unique feature of the recursive and order-preserving properties of the additive cost measure. (The reader who wishes to witness a counter-example may examine the problem of finding a minimal median path, mentioned in Section 2.4.1.) These properties single out f^* as a **perfect discriminator**; it could provide perfect early-warning signals $f^*(n) > C^*$ for all off-track nodes encountered along the search and, since we always have some nodes around for which $f^*(n) = C^*$ (e.g., $f^*(s) = C^*$), following the minimum f^* would constitute a perfect search strategy leading straight toward an optimal solution without ever getting sidetracked.

The trouble, of course, is that we normally do not possess the extra information or insight required to compute f^*, especially the h^* part. If we try to compute it by searching the graph itself (as is actually done in dynamic programming), we defeat the main purpose, which is to avoid excessive search, because many off-track nodes need to be explored in the computation of h^*. There still remains the possibility of *approximating* f^* by an easily computable function f, and this is exactly what A^* attempts to achieve via the combination $f = g+h$.

3.1.2 Termination and Completeness

A^* always terminates on finite graphs. The reason is that the number of acyclic paths in each such graph is finite and with every node expansion A^* *adds* new links to its traversal tree. Each newly added link represents a new acyclic path and so the reservoir of paths must eventually be exhausted. Note that even reopened nodes represent new paths because A^* only reopens a node on CLOSED when it finds a *strictly* cheaper path to it. This argument is valid for any best-first strategy that prunes cyclic paths; that is, if the cost of the path without the cycle is not higher than that of the path with the cycle.

A^* is also **complete** on finite graphs. A failure is returned only when OPEN is found empty and, if a solution path $P_{s-\gamma}$ exists, OPEN cannot become empty before $P_{s-\gamma}$ is discovered. For if it did there would be a last node $n' \in P_{s-\gamma}$ on OPEN (at least one node, the start node s, from $P_{s-\gamma}$ has surely entered OPEN) which is expanded and found to generate no new successors. This contradicts our assumption that n' lies on a solution path; every such node (except a goal node) has at least one successor also lying on a solution path. This argument is applicable to all best-first strategies.

THEOREM 1. *A^* is complete even on infinite graphs.*

Proof: Consider the set of nodes on the solution path $P_{s-\gamma}$; all these nodes are assigned finite f's and at all times at least one of them is on OPEN. If A^* does not return a solution, it must be that it does not terminate at all; it cannot exit with failure for the reason invoked in the last paragraph. Now, in any locally finite graph there is only a finite number of acyclic paths with bounded length; if A^* does not terminate, it must be chasing an infinite path. This, however, would be contrary to A^* node selection policy; due to the positive, δ-bound nature of the arc costs, every infinite path must have an unbounded cost and, since at all times at least one node on OPEN should be advertizing its bounded cost, A^* should stop its chase and select that node for expansion. ∎

The proof does not actually require that all arc costs be bounded away from zero; it is enough to demand that the cost of every infinite path is unbounded. This property will be assumed to hold henceforth. A similar condition if imposed on any best-first strategy would render it complete.

3.1.3 Admissibility—A Guarantee for an Optimal Solution

From the set splitting argument of Section 2.3.2, it should be clear that whenever $h(n)$ is an optimistic estimate of $h^*(n)$, A^* will return an optimal solution. This motivates naming this class of estimates: **Admissible heuristics.**

DEFINITION: *A heuristic function h is said to be* ***admissible*** *if*

$$h(n) \leqslant h^*(n) \qquad \forall n \tag{3.6}$$

In the following discussion we assume that Eq. (3.6) is satisfied. Our results will be cross-referenced to those found in Nilsson (1980).

LEMMA 1: (Nilsson, Result 2): *At any time before A^* terminates, there exists an OPEN node n' on $P^*_{s-\gamma}$ with $f(n') \leqslant C^*$.*

Proof: Consider any optimal path $P^*_{s-\gamma} \in \mathbf{P}^*_{s-\Gamma}$

$$P^*_{s-\gamma} = s, n_1, n_2, \ldots, n', \ldots, \gamma$$

and let n' be the shallowest OPEN node on $P^*_{s-\gamma}$ (there is at least one OPEN node on $P^*_{s-\gamma}$ because γ is not CLOSED until termination). Now, since all ancestors of n' are on CLOSED and since the path $s, n_1, n_2, \ldots, n'$ is optimal, it must be that the pointers assigned to n' are along $P^*_{s-n'}$, hence, $g(n') = g^*(n')$. Using now the admissibility of h, we obtain:

$$f(n') = g^*(n') + h(n') \leqslant g^*(n') + h^*(n') = f^*(n') \tag{3.7}$$

and since $n' \in P^*_{s-\gamma}$, we have (see Eq. (3.4)) $f^*(n') = C^*$ and

$$f(n') \leqslant C^* \tag{3.8}$$

■

LEMMA 2: (a Corollary of Lemma 1): *Let n' be the shallowest OPEN node on an optimal path $P^*_{s-n''}$ to any arbitrary node n'', not necessarily in Γ. Then*

$$g(n') = g^*(n') \tag{3.9}$$

stating that A^* has already found the optimal pointer-path to n' (i.e., n' is along $P^*_{s-n''}$) and that path will remain unaltered throughout the search.

Proof: It follows directly from the proof of Lemma 1, where the equality $g(n') = g^*(n')$ was established without using the fact that γ is a goal node. ■

THEOREM 2. (Nilsson, Result 4): *A^* is admissible.*

Proof: Suppose A^* terminates with a goal node $t \in \Gamma$ for which $f(t) = g(t) > C^*$. A^* inspects nodes for compliance with the termination criterion only after it selects them for expansion. Hence when t was chosen for expansion, it satisfied:

$$f(t) \leqslant f(n) \qquad \forall\, n \in OPEN$$

This means that, immediately prior to termination, all nodes on OPEN satisfied $f(n) > C^*$. This, however, contradicts Lemma 1 which guarantees the existence of at least one OPEN node (n') with $f(n') \leqslant C^*$. Therefore the terminating t must have $g(t) = C^*$, which means that A^* returns an optimal path. ■

Note that this proof is essentially identical to the subset-splitting argument used in Section 2.3.2 to show the admissibility of GBF^* with optimistic estimates. It should still hold, therefore, for any complete best-first strategy with optimistic estimates and order-preserving cost functions (Eq. 2.11).

3.1.4 Comparing the Pruning Power of Several Heuristics

The **power** of the heuristic estimate $h(\cdot)$ is measured by the amount of pruning induced by h and depends, of course, on the **accuracy** of this estimate. If $h(\cdot)$ estimates the completion cost precisely ($h = h^*$), then A^* will only explore nodes lying along optimal paths. On the other hand, if no heuristic at all is used ($h = 0$), a uniform-cost (or breadth-first) search ensues, which amounts to exhaustively expanding all nodes reachable from s by a path costing less than C^*. The more common cases lie somewhere between these two extremes.

In Section 1.1 we saw that a given problem may make use of different types of heuristics. In the 8-Puzzle, for example, we can use h_1, the number of misplaced tiles, or h_2, the sum of the Manhattan distances. In the Traveling Salesman problem (TSP) we had a choice between the minimum-spanning-tree (MST) approximation or the estimate obtained from solving the assignment problem. Clearly, the higher the h, the closer it is to h^* (as long as h remains admissible) and the more powerful it is expected to be. In the 8-Puzzle, for instance, h_2 is generally higher and never lower than h_1; hence it is expected to yield a more efficient search. No such preference can be proclaimed for the two TSP heuristics.

We now wish to cast these considerations into a formal setting.

DEFINITION: *A heuristic function h_2 is said to be* ***more informed than*** *h_1 if both are admissible and*

$$h_2(n) > h_1(n) \qquad \textit{for every nongoal node } n \tag{3.10}$$

Similarly, an A^* algorithm using h_2 will be said to be more informed than that using h_1.

Next we will show that using more-informed heuristics indeed implies more power, but first we need a couple of preliminary results.

THEOREM 3. (a necessary condition for node expansion, Nilsson, Result 5): *Any node expanded by A^* cannot have an f value exceeding C^*, that is,*

$$f(n) \leqslant C^* \qquad \textit{for all nodes expanded} \tag{3.11}$$

Proof: Assume the contrary, i.e., that n is selected for expansion from OPEN while $f(n) > C^*$. This would certainly contradict Lemma 1 which states that OPEN contains a node n' such that $f(n') \leqslant C^*$ and so n' should have been chosen for expansion instead of n. ■

THEOREM 4. (a sufficient condition for node expansion): *Every node on OPEN for which $f(n) < C^*$ will eventually be expanded by A^*.*

Proof: Suppose, at some stage, n is found on OPEN with $f(n) < C^*$. A^* terminates with $f(t) = C^*$ after selecting t for expansion from

OPEN. The fact that t, not n, was selected means either that n was expanded before t or that $f(n)$ has in the meantime been modified so as to exceed C^*. But f can only be modified downward, thus leaving the former alternative, the expansion of n as the only possibility. ■

Note that neither Theorem 3 nor 4 individually constitutes both a necessary and sufficient condition. This is so because the fate of OPEN nodes for which strict equality holds ($f(n) = C^*$) is not determined by either theorem, but remains to be decided by the tie-breaking rule employed in the particular implementation of A^*. We will see later that the set of nodes satisfying this equality is normally of relatively small size, and therefore we can dismiss the equality event from most calculations.

Theorem 3 identifies a set of nodes $\mathbf{S} = \{n : f(n) > C^*\}$ that are definitely destined to be *excluded* from expansion. This gives rise to a simple scheme of **cutting down on storage space** without compromising admissibility; any node known to belong to $\mathbf{S}$ could be safely removed from OPEN without affecting the performance of A^*. We do not know, of course, the exact value of C^*, but an upper bound on C^* would suffice, for if by some means (e.g., by a depth-first search) we find some path to a goal and it has a cost C', we know for sure that $C^* \leqslant C'$ and so every node satisfying $f(n) > C'$ definitely belongs to $\mathbf{S}$ and can be thrown away from storage. If, by chance, n will be regenerated later with a lower f, it will be treated as a new node that may or may not be expanded.

The expansion conditions in Theorems 3 and 4, although useful, do not provide adequate criteria for determining the efficiency of a given A^* algorithm. The reason is that they invoke two local parameters that may vary with the algorithm's state of development and are not, therefore, directly discernible from the external data. The first is the function $g(n)$ which is not truly a property of node n but depends on which path to n it is that A^* happens to have discovered at any given time. The second is the requirement (in Theorem 4) that n resides on OPEN before it is selected for expansion. Whether or not a given node ever enters OPEN depends not on n itself but on the behavior of A^* while exploring the paths leading to n. The next two results rectify these difficulties:

DEFINITION: *We say that a path P is* ***C-bounded*** *if every node n along this path satisfies $g_P(n) + h(n) \leqslant C$. Similarly, if a strict inequality holds for every n along P, we say that P is* ***strictly C-bounded****. When it becomes necessary to identify which heuristic was used in ascertaining the preceding inequality, we will use the notation C(h)-bounded.*

THEOREM 5. *A sufficient condition for A^* to expand a node n is that there exists some strictly C^*-bounded path P from s to n.*

Proof: Assume the contrary, that is, that there exists a strictly C^*-bounded path P from s to n and that at termination its final node n has

not yet been expanded. Let n' be the shallowest OPEN node on P (at termination). Since all ancestors of n' are on CLOSED, the g that A^* assigns to n' cannot be more costly than $g_P(n')$; hence,

$$f(n') = g(n') + h(n') \leqslant g_P(n') + h(n') < C^*$$

But if $f(n') < C^*$ at termination, then n' should have been chosen for expansion instead of the goal node which allegedly terminated the search with $f = C^*$. That contradicts our assumption that an OPEN node n' could exist on P and, hence, n must be expanded. ■

THEOREM 6. *A necessary condition for A^* to expand a node n is that there exists a C^*-bounded path from s to n.*

Proof: If A^* expands a node n, then n must be on OPEN, and simultaneously $f(n) \leqslant C^*$ (Theorem 4). Now consider the pointer path PP assigned to n at the time of its expansion. Each ancestor n' of n along PP is closed and, hence, was selected for expansion (perhaps more than once) some time in the past at which time (from Theorem 4) it satisfied $g'(n') + h(n') \leqslant C^*$. Its current $g(n')$ value along PP cannot be higher than its $g'(n')$ value at the time of expansion. Consequently every node n' along PP currently satisfies $f(n') \leqslant C^*$, meaning that PP itself is C^*-bounded. ■

Theorems 5 and 6 will play a major role in our analysis of A^*'s performance. For now we use them only to confirm our intuition regarding the desirability of making h as high as possible.

THEOREM 7. (Nilsson, Result 6): *If A_2^* is more informed than A_1^*, then A_2^* dominates A_1^*.*

Proof: Let A_2^* use heuristic h_2 and A_1^* use heuristic h_1 such that $h_2(n) > h_1(n)$ for every nongoal node n, and assume that A_2^* expands some node n. From Theorem 5 we know that there must exist a C^*-bounded path P from s to n if judged by h_2. Since $h_2(n') > h_1(n')$ for every node n' along P, P is *strictly* C^*-bounded when judged by h_1. Hence, using Theorem 5, A_1^* must also expand n. ■

The requirement of strict inequality between h_2 and h_1 remains a sore to aesthetics, because it is rarely satisfied, even in situations where h_2 is clearly superior to h_1. In the 8-Puzzle, for example, the number of misplaced tiles (h_1) *exactly* equals the sum of Manhattan distances (h_2) in many configurations, thus disqualifying h_2 as the "more-informed" heuristic of the two. This sore can be alleviated in various ways. First, if the two algorithms compared use the same tie-breaking rule and if the rule is purely structural (i.e., independent of the values of g and h), then the definition of more-informedness can be relaxed to also permit equalities between the corresponding heuristics. Second,

Theorem 12 demonstrates that under reasonable assumptions (satisfied in the 8-Puzzle) equalities between h_1 and h_2 only violate the dominance of A_2 on a small set of nodes for which $f_2(n) = C^*$. Third, in cases where it is required to find *all* the optimal paths, Theorem 6 constitutes both a necessary and sufficient condition and again the definition of more-informedness can permit equalities without affecting Theorem 7. Finally, in Chapter 6, we present a stochastic extension of the concept of more-informedness, to include cases in which one heuristic is known to be greater than another most of the time, but not always.

3.1.5 Monotone (Consistent) Heuristics

In the preceding section we implied that the effectiveness of A^* can be measured by how many nodes it manages to exclude from expansion. In actuality, however, a given node can be expanded many times, and so it is not the number of distinct nodes expanded but rather the number of *expansion operations* that should be our major concern. In this section we show that under certain conditions A^* never reopens nodes from CLOSED, and consequently the labor connected with reexpansion can be saved.

The condition that may enable us to forgo reopenings is a certain property of h that is satisfied by any h^* function, hence, the title **consistent**. In Section 3.1 we saw that every h^* function, aside from correctly estimating the optimal costs to Γ, also conforms to certain relations between $h^*(n)$ and h^* evaluated on n's descendants. In particular, Eq. 3.4 and Eq. 3.5 imply that the cheapest path constrained to pass through n cannot be less costly than the cheapest path available without this constraint, that is:

$$g^*(n) + h^*(n) \geqslant h^*(s) \qquad \forall(n) \tag{3.12}$$

or,

$$k(s,n) + h^*(n) \geqslant h^*(s) \tag{3.13}$$

This form of the **triangle inequality** for geodesics should hold for any pair of nodes not necessarily involving s. Thus, if n' is any descendant of n, we should have:

$$h^*(n) \leqslant k(n,n') + h^*(n') \qquad \forall(n,n') \tag{3.14}$$

which is automatically satisfied for nondescendant n' since $k(n,n') = \infty$.

It is now reasonable to expect that if the process of estimating $h(n)$ is consistent, it should inherit this geodesic property from $h^*(n)$ and satisfy:

$$h(n) \leqslant k(n,n') + h(n') \qquad \forall(n,n') \tag{3.15}$$

DEFINITION: *A heuristic function h(n) is said to be* **consistent** *if Eq. (3.15) is satisfied for all pairs of nodes n and n'.*

DEFINITION: *A heuristic function h(n) is said to be* **monotone** *if it satisfies:*

$$h(n) \leqslant c(n,n') + h(n') \qquad \forall n,n' \mid n' \in \mathrm{SCS}(n) \tag{3.16}$$

■

Monotonicity may seem, at first glance, to be less restrictive than consistency, because it only relates the heuristic of a node to the heuristics of its immediate successors. However, a simple proof (by induction on the depth-k descendants of n, exercise 3.2) should convince the reader that Eq. (3.16) implies Eq. (3.15). We, therefore, state the following theorem.

THEOREM 8. *Monotonicity and consistency are equivalent properties.* ■

It is also simple to relate consistency to admissibility.

THEOREM 9. *Every consistent heuristic is also admissible.*

Proof: We simply replace n' in Eq. (3.15) by any goal node $\gamma \in \Gamma$, obtaining

$$h(n) \leqslant k(n,\gamma) + h(\gamma) \qquad \forall n$$

Now, since $h(\gamma) = 0$ and $k(n,\gamma) = h^*(n)$ for some goal node $\gamma \in \Gamma^*$, we have the admissibility condition

$$h(n) \leqslant h^*(n)$$

confirmed. ■

We are now ready to demonstrate the advantages of using consistent or monotone heuristics.

THEOREM 10. (Nilsson, result 7) *An A^* algorithm guided by a monotone heuristic finds optimal paths to all expanded nodes, that is,*

$$g(n) = g^*(n) \qquad \forall n \in CLOSED \tag{3.17}$$

Proof: Assume that A^* selects for expansion a node for which $g(n) > g^*(n)$. Consider an optimal path $P^*_{s,n}$ from s to n; if n is the only OPEN node on $P^*_{s,n}$, then obviously all n's ancestors along $P^*_{s,n}$ have been expanded and we have $g(n) = g^*(n)$ (Lemma 2). The assumption $g(n) > g^*(n)$ implies that $P^*_{s,n}$ contains at least one additional OPEN node and, as usual, we let n' be the shallowest OPEN node on $P^*_{s,n}$. We shall now show that n', and not n, should be selected for expansion. Lemma 2 states that $g(n') = g^*(n')$ and therefore, using consistency, we have

$$f(n') = g^*(n') + h(n') \leqslant g^*(n') + k(n',n) + h(n)$$

The sum $g^*(n') + k(n', n)$ is equal to $g^*(n)$ because n' is an ancestor of n along $P^*_{s,n}$, and so

$$f(n') \leqslant g^*(n) + h(n)$$

We now see that the assumption $g(n) > g^*(n)$ implies

$$f(n') < f(n)$$

which should prohibit the choice of n over n'. Thus we must conclude $g(n) = g^*(n)$. ■

THEOREM 11. (Nilsson, result 8) *Monotonicity implies that the f values of the sequence of nodes expanded by A^* is non-decreasing.*

Proof: Let n_2 be expanded immediately after n_1. If n_2 resided on OPEN while n_1 was expanded, then $f(n_1) \leqslant f(n_2)$ follows immediately from the node selection rule. If it did not reside there, then n_2 must be a newcomer to OPEN, that is, a successor of n_1 for which we have $g(n_2) = g(n_1) + c(n_1, n_2)$ and for which monotonicity dictates:

$$f(n_2) = g(n_1) + c(n_1, n_2) + h(n_2) \geqslant g(n_1) + h(n_1) = f(n_1)$$

Thus the sequence of f values cannot decrease. ■

Monotonicity simplifies significantly the conditions for node expansion.

THEOREM 12. *If h is monotone, then the necessary condition for expanding node n is given by*

$$g^*(n) + h(n) \leqslant C^*$$

and the sufficient condition by the strict inequality

$$g^*(n) + h(n) < C^*$$

Proof: The necessary condition follows by combining Theorems 3 and 10. The sufficient condition is based on the nondecreasing nature of $g^* + h$ along an optimal path from s to any arbitrary node n. This is so because, if n' is the parent of n along any optimal path $P^*_{s,n}$, it must satisfy

$$g^*(n) = g^*(n') + c(n', n)$$

which, together with the monotonicity of h, yields

$$g^*(n) + h(n) = g^*(n') + c(n', n) + h(n) \geqslant g^*(n') + h(n')$$

This implies that if n satisfies $g^*(n) + h(n) < C^*$, all its ancestors along $P^*_{s,n}$ must also satisfy this inequality. Thus $P^*_{s,n}$ is strictly C^*-bounded and, according to Theorem 5, n will be expanded by A^*. ■

This result has an interesting implication on the significance of the tie-breaking rule. Suppose algorithms A_1^* and A_2^* employ two different tie-breaking rules and both are guided by the same monotone heuristic h. A_2^* may, in general, expand more nodes than A_1^*, but Theorem 12 demonstrates that the difference cannot exceed the number of nodes actually satisfying the equality $f(n) = C^*$.

In Chapter 4 we will argue that monotonicity is not an exceptional property but rather a common occurrence among admissible heuristics (e.g., all the heuristics discussed in the example of section 1.1 are monotone). Theorem 12 helps explain why the difference between the relation $f(n) \leqslant C^*$ and $f(n) < C^*$ is so often insignificant and why it is permissible, in certain analyses, to treat them both as equivalent. The equivalence would be further strengthened if $f(n)$ can be modeled as a continuous random variable, in which case the equality $f(n) = C^*$ is a rare event indeed.

An important corollary of Theorem 12 is that in guaranteeing the advantage of one monotone heuristic over another we no longer need to insist on strict inequality between the two, as in Eq. (3.10).

DEFINITION: *An algorithm A_2^* is said to* **largely dominate** *A_1^* if every node expanded by A_2^* is also expanded by A_1^* except, perhaps, some nodes for which $h_1(n) = h_2(n) = C^* - g^*(n)$.*

COROLLARY TO THEOREM 12. *If $h_2(n) \geqslant h_1(n)$ for all n and both are monotone, then A_2^* (using h_2) largely dominates A_1^* (using h_1).*

Proof: Let n be a node expanded by A_2^* and not by A_1^* then, from Theorem 12, $g^*(n) + h_2(n) \leqslant C^*$ and $g^*(n) + h_1(n) \geqslant C^*$. These, together with $h_2(n) \geqslant h_1(n)$, imply $h_1(n) = h_2(n) = C^* - g^*(n)$ ■

Note that in the absence of monotonicity, the advantage of A_1^* over A_2^* would be much less certain; every descendant of n that is reachable from n by a strictly $C^*(h_2)$-bounded path will also be expanded by A_2^* and possibly not by A_1^*. (See exercise 3.1.)

Consistency also plays an important role in establishing the **optimality** of A^* over other types of search algorithms that are provided with the same heuristic information. It is possible to show (Dechter and Pearl, 1983) that if h is consistent, then A^* **largely dominates every admissible algorithm** having access to that same h (see exercise 3.7b). On the other hand, if h is admissible but not consistent, then there are admissible algorithms that, using the same h, will grossly outperform A^* in some problem instances, regardless of what tie-breaking rule A^* invokes (see Section 3.4 and exercise 3.7a).

3.2 RELAXING THE OPTIMALITY REQUIREMENT

In the previous section, we have seen that A^* possesses some useful properties that make good use of the heuristics provided. In particular, if the available heuristic is admissible (i.e., an optimistic estimate of h^*), then A^* is guaranteed to find an optimal solution path if one exists.

Experience shows, however, that in many problems A^* spends a large amount of time discriminating among paths whose costs do not vary significantly from each other. In such cases, the admissibility property becomes a curse rather than a virtue. It forces A^* to spend a disproportionately long time in selecting the best among roughly equal candidates and prevents A^* from completing the search with a suboptimal but otherwise acceptable solution path. These observations cast doubt on the appropriateness of the small-is-quick paradigm in satisficing problems and raise the question as to whether a more direct estimate of the "promise" of a node would do better.

In this section we describe attempts to solve three basic questions, quoted here from *The Handbook of Artificial Intelligence* (Barr and Feigenbaum,1981, vol.1):

1. One may be more concerned with minimizing search effort than with minimizing solution cost. Is $f = g + h$ an appropriate evaluation function in this case?
2. Even if solution cost is important, the combinatorics of the problem may be such that an admissible A^* cannot run to termination. Can speed be gained at the cost of a bounded decrease in solution quality?
3. It may be hard to find a good heuristic function h that satisfies the admissibility condition; with a poor but admissible heuristic function, A^* deteriorates into blind search. How is the search affected by an inadmissible heuristic function?

3.2.1 Adjusting the Weights of g and h

To find alternatives to the rule $f = g + h$, it may be instructive to recap the way this rule originated to govern satisficing problems. We started with the small-is-quick paradigm, arguing that if we persistently pursue the shortest path, we will be safe from being trapped into searching along fruitless directions. We then saw that to identify unmistakenly the node belonging to the shortest path we need to add to $g(n)$ a second term, the remaining shortest distance $h^*(n)$. In the absence of an exact value for $h^*(n)$, we applied the face-value principle; we took its estimate $h(n)$ and added it to $g(n)$ as if it was $h^*(n)$ itself. The result of all this is an algorithm that goes about the optimization mission too seriously, contrary to our original intention of getting a quick, though perhaps suboptimal, solution.

One approach to rectify this predicament is to examine the effects of g and h separately. The effect of g is to add a breadth-first component to the search. Without h, A^* reduces to a purely breadth-first (or uniform cost) search, whereas without g, A^* ignores completely the distance already covered and bases its decisions purely on h, the estimate of the remaining proximity to the goal. The latter strategy looks ideal for cases where solutions are plentiful, equally desirable, and the heuristic function is trustworthy. However, if h is unreliable, it can lead to a disaster; h may assign misleadingly low values to hopelessly long blind alleys, and we would not have g around any more to remind us that we have been led astray.

To adjust the balance between these two tendencies, Pohl (1970) has suggested using a weighted evaluation function:

$$f_w(n) = (1-w)g(n) + w\,h(n) \tag{3.18}$$

Obviously, $w = 0, ½$, and 1 correspond to uniform-cost, A^* and BF^* (with $f = h$) strategies, respectively. By varying w continuously between 0 and 1, one can achieve any desired mixture between conservatism and radicalism, to match the degree of reliability one attributes to h.

If h is admissible, it is easy to show that f_w is admissible in the range $0 \leqslant w \leqslant ½$ but may lose its admissibility in the remaining interval $½ < w \leqslant 1$, depending on how far h is from h^*. Empirical results obtained for the 15-Puzzle (15 tiles in a 4×4 array) indicate (Pohl, 1970) that f_w achieves its best performance in the range $½ < w < 1$. More extensive simulations by Gaschnig (1979a) on the 8-Puzzle revealed that in most cases f_w achieves its optimal performance (minimum average number of nodes expanded) in the vicinity of $w = ½$ and increasing w beyond ½ may cause the expected number of nodes expanded to increase rather than decrease.

Pohl has provided some analytical explanation for the advantages of using g by analyzing a uniform tree with one goal node and assuming that the absolute error between h and h^* is bounded by a fixed integer e. If one now analyzes the worst-case performance of f_w, that is, assuming that a clever adversary distributes the error in such a way that the search exhibits its worst performance, one finds that the number of nodes expanded with $f = h$ is greater than that expanded using $f = g + h$. In Chapter 7, we present an average-case analysis of this phenomena and demonstrate that, in the model chosen, f_w indeed achieves its best performance at the admissibility limit $w = ½$, as was observed in Gaschnig's experiments.

These empirical and analytical findings strengthen the soundness of the small-is-quick paradigm. In the cases examined, the auxiliary optimization task represented by the $f = g + h$ rule seems to be producing its expected by-product: speed.

3.2.2 Two ε-Admissible Speedup Versions of A^*

The deterioration of the performance of $f_w = (1-w)g + wh$ for $w > ½$, coupled with the ever present risk of termination with a much costlier solution than is acceptable, prompted attempts to answer the second question: can speed be gained at the cost of a *bounded* decrease in solution quality? Two approaches will be described, **dynamic weighting** and $\mathbf{A}_\epsilon^*$. Both endow A^* with a stronger depth-first component while still guaranteeing that the solution found at termination will not be worse than the optimal one by a factor greater than $1 + \epsilon$, $(\epsilon \geqslant 0)$.

Dynamic Weighting (Pohl, 1973). Rather than keeping w constant throughout the search, it is natural to dynamically change w so as to weigh h less heavily as the search goes deeper. The approach devised by Pohl uses the evaluation function:

$$f(n) = g(n) + h(n) + \epsilon \left[1 - \frac{d(n)}{N} \right] h(n) \tag{3.19}$$

where $d(n)$ is the depth of node n and N is the (anticipated) depth of the desired goal node. At shallow levels of the search tree ($d << N$), h is given a supportive weight equal to $(1 + \epsilon)$, encouraging depth-first excursions. At deep levels, however, where termination is likely to occur, the search assumes an admissible, equal weight, character protected against premature termination.

It is not hard to see that if $h(n)$ is admissible, the algorithm is ϵ-admissible, that is, it finds a path with cost at most $(1 + \epsilon)C^*$. This follows by noting that before termination the shallowest OPEN node n' along any optimal solution path still has $g(n') = g^*(n')$ (see Lemma 2) and so,

$$\begin{aligned} f(n') &\leqslant g^*(n') + h^*(n') + \epsilon \left[1 - \frac{d(n')}{N} \right] h^*(n') \\ &\leqslant C^* + \epsilon h^*(n') \\ &\leqslant (1 + \epsilon)C^* \end{aligned}$$

Therefore, the algorithm cannot terminate with any goal node whose cost is greater than $(1 + \epsilon)C^*$.

A_ϵ^* – An Algorithm Using Search Effort Estimates†. A_ϵ^* uses two lists: OPEN and FOCAL. FOCAL is a sublist of OPEN containing only those nodes that

† © 1982 IEEE. Reprinted with permission, from *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 4, No. 4, pp. 392–399, July 1982.

do not deviate from the lowest f node by a factor greater than $1 + \epsilon$, that is,

$$FOCAL = \{n : f(n) \leqslant (1 + \epsilon) \min_{n' \in OPEN} f(n')\}.$$

The operation of A_ϵ^* is identical to that of A^* except that A_ϵ^* selects the node from FOCAL with the lowest h_f value, where $h_f(n)$ is a second heuristic function estimating the computational effort required to complete the search starting from n.

The rationale for this variation is simple. According to the estimates $f(\cdot)$ at hand, all nodes on FOCAL have roughly equal solution paths. Therefore, rather than spending time on deciding which among them is the best, it makes more sense to use the time to compute the remaining portion of the solution, and the node that promises to offer the easiest completion job is the one with the lowest h_f. Clearly, for $\epsilon = 0$, A_ϵ^* reduces to A^*.

The heuristic h used in *forming* FOCAL is of an entirely different nature than the FOCAL-heuristic h_f used for *selecting* nodes from within FOCAL. The former anticipates the reduction in *solution quality* due to the remaining part of the solution once it is found; the latter estimates the *computational effort* required to complete the search.

In certain cases, it makes sense to surmise that the search would be completed faster from nodes with lower h values and hence choose $h_f = h$. In other cases, additional information may be available regarding the structure of the search graph emanating from any given node (e.g., the height of the graph or its approximate branching degree) and could be incorporated into h_f.

THEOREM 13. *A_ϵ^* is ϵ-admissible, that is, it always finds a solution whose cost does not exceed the optimal cost by more than a factor $1 + \epsilon$.*

Proof: Adopting the following notation:

n_0 = a node in OPEN having the smallest f value
t = termination node (chosen for expansion and found to be a goal)
n_1 = shallowest OPEN node on an optimal solution path
$C(t) = f(t) = g(t)$ = cost of solution found

we have:

$f(n_1) \leqslant C^*$	*since h is admissible*
$f(n_0) \leqslant f(n_1)$	*because OPEN is f-ordered*
$f(t) \leqslant f(n_0)(1+\epsilon)$	*since t is chosen from FOCAL*

Therefore:

$$C(t) = f(t) \leqslant f(n_1)(1+\epsilon) \leqslant C^*(1+\epsilon) \qquad \blacksquare$$

We still have to prove that A_ϵ^* terminates, but the proof of this fact is identical to the proof that A^* terminates if all arc costs are positive and bounded away from zero.

Note that, unlike h, h_f is not required to be admissible, that is, it may overestimate the actual computational effort required without affecting the ϵ-admissibility of A_ϵ^*.

Comparing the Two Algorithms. Dynamic weighting is a simpler algorithm to implement; it maintains only one list and employs only one heuristic function. A_ϵ^*, on the other hand, offers several other advantages. Whereas dynamic weighting is limited to problems where the depth of the optimal solution (N), or at least a good upper bound to it, is known *a priori*, A_ϵ^* will maintain its ϵ-admissibility even when N is completely unknown. In addition, the complete separation of the two heuristics used by A_ϵ^* renders it a more flexible mechanism for incorporating sophisticated estimations of the computational cost required for completing the search. These may, in general, involve elaborate functions of the path from s to n, not always capturable by g and h. For example, if the Traveling Salesman problem is applied to a sparsely connected road map, the number of edges in the unexplored portion of the graph would usually constitute a more valid estimation of the remaining computational effort than the proportion of unexplored cities $1-\text{depth}(n)/N$, which guides the dynamic weighting algorithm.

Figure 3.1 represents a comparison between the performance of A_ϵ^* to that of dynamic weighting. Each coordinate represents the ratio of the number of nodes expanded by the corresponding algorithm to that expanded by A^* (with the same heuristic). The open circles represent Traveling Salesman problems generated by a random distribution of nine cities in the unit square. The solid circles represent "hard" problems where intercity distances were independently chosen from a uniform distribution over the interval (0.75, 1.25). The heuristic function in all cases was computed by

$$h = \max\left(\sum_i \min_j d_{ij}, \sum_j \min_i d_{ij}\right)$$

where d_{ij} stands for the distance between city i and city j, and where both the summation and the min operation range over the set of unvisited cities. The focal-heuristic employed by A_ϵ^* was simply the number of unvisited cities.

The chart indicates comparable performances for the two algorithms with a slight edge in favor of A_ϵ^*. Both algorithms exhibit their highest utility on "hard" problems with saving ranging between 60% and 90%.

3.2.3 R_δ^*—A Limited Risk Algorithm Using Information about the Uncertainty of h

Harris (1974) has convincingly argued that the condition of admissibility is too restrictive. Although it may be impractical to find a good estimate of h^* that never violates the admissibility conditions, "it is often easier to find a heuristic

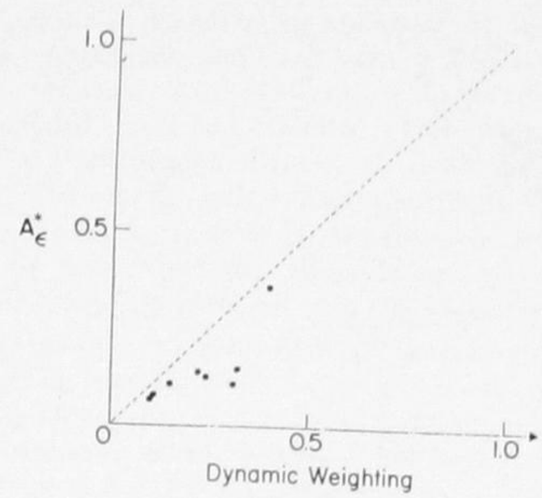

Figure 3.1
Comparison of the performances of A_ϵ^* and dynamic weighting.

that estimates the future cost well, but occasionally overestimates it." In these cases Harris shows that the extra cost of the resulting solution cannot exceed the amount by which h^* is overestimated, that is, if e bounds the overestimation, $h(n) - h^*(n) \leqslant e$, then A^* is e-admissible, $C(\mathcal{A}) - C^* \leqslant e$. This property enabled Harris to devise an algorithm called **band-width search** which was permitted to accept a wider variety of heuristic functions as long as they are known to be e-admissible.

We now wish to make the notion of e-admissibility even less restrictive. Harris' e-admissibility condition requires that a heuristic estimate be found that *never* overestimates h^* by more than e (to express this more precisely, $h(n) - h^*(n) \leqslant e$ should hold for all nodes found on OPEN at termination). This requirement, however, may rule out the use of very precise estimators that only *occasionally* overestimate h by more than any reasonable e.

Consider, for example, the case where all arc costs are known to be drawn independently from a common distribution function, uniform between 0 and 1. If N stands for the number of arcs between n and the goal (assuming a single solution path from each n on OPEN), then, for large N, it is known that $h^*(n)$ is most likely to fall in the neighborhood of $N/2$. However, the only *admissible* estimate that one can produce is $h = 0$, whereas the most *reasonable* heuristic, $h = N/2$, is not admissible and will not be permitted even by e-admissibility standards because it may, in the worst case, lead to an extra cost of $N/2$. Nevertheless, the likelihood of this event is extremely small and thus may be tolerated.

R_δ^* provides a generalized admissible search procedure for cases in which we wish to invoke likelihood considerations in the admissibility guarantee. We assume that in addition to estimating the value of h^*, we also have information regarding the uncertainty of the estimation process, and that this uncertainty

information is expressed in the form of a probability density function $\rho_{h^*}(x)$, which measures the relative likelihood that h^* will be found in the neighborhood of x. In cases where g^* is known (e.g., in tree searching where $g^* = g$), $\rho_{h^*}(x)$ also induces a density function on $f^* = g^* + h^*$:

$$\rho_{f^*}(y) = \rho_{h^*}(y - g^*) \tag{3.20}$$

Similarly, if $g(n) \neq g^*(n)$ and P_{s-n} is the best path found so far between s and some node n on OPEN, the cost of an optimal solution path constrained to continue P_{s-n} is given by the random variable:

$$f^+(n) = g(n) + h^*(n) \tag{3.21}$$

with density:

$$\rho_{f^+}(y) = \rho_{h^*}(y - g) \tag{3.22}$$

Note that OPEN always contains some node n for which $f^+(n) = f^*(n) = C^*$ (see Lemma 2).

Our task now is to devise a rule for scheduling nodes for expansion based on the information contained in the density function $\rho_{f^+}(y)$ attached to each node on OPEN. If these density functions were nonoverlapping (Figure 3.2(a)), we would simply select for expansion the node for which the density $\rho_{f^+}(y)$ signifies the lowest f^+ value, thus complying with the best-first principle. When overlaps exist, the decision is more involved, as can be seen in Figure 3.2(b). Which of the two nodes shown, n_1 or n_2, is the more promising (i.e., the "best") one? $f^+(n_1)$ has a lower mean, but n_2 advertises some possibility that its actual cost $f^+(n_2)$ is, in fact, appreciably lower than that of node n_1. An admissible algorithm, or even an ϵ-admissible one, would insist on expanding n_2, whereas intuition urges us to prefer n_1 because the density functions indicate that the event $f^+(n_1) > f^+(n_2)$ is rather unlikely.

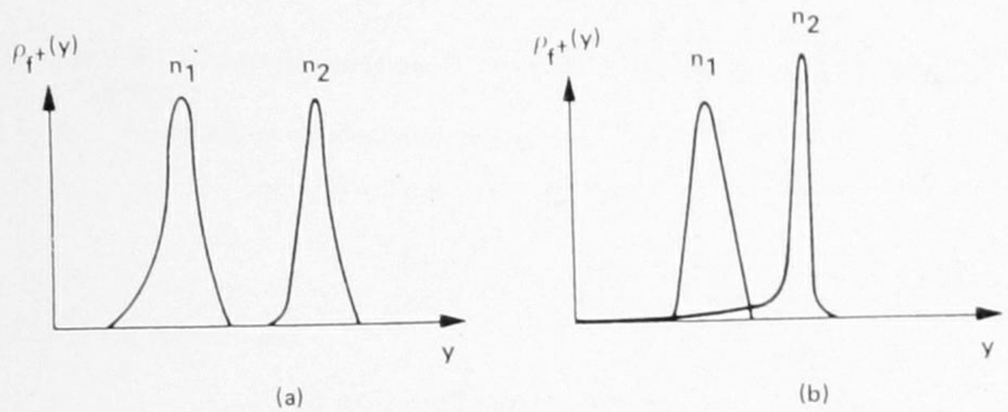

Figure 3.2
Probability density functions of $f^+ = g + h^*$ for two typical nodes in OPEN.

R_δ^* resolves this conflict by the following argument. Imagine that some algorithm has found a solution path with cost C, and that node n still resides on OPEN unexpanded. What is the basis of our *regrets* for not having explored n? We should have expanded n if it had exhibited a high potential for reducing the resultant cost C by a significant amount or, alternatively, if the **risk** of terminating the algorithm at cost C (without exploring the possibilities latent in n) had been above a given acceptable level. Otherwise we should be satisfied with the solution found. If our knowledge about the costs of solutions continuing P_{s-n} is represented by $\rho_{f^+}(y)$, the risk of missing a further cost reduction ought to be characterized by a risk-measure $R(C)$, associated with each node, which is a nondecreasing function of C, and can be computed from $\rho_{h^*}(y)$. Every node competing for expansion could then be characterized by a function $R(C)$ measuring the **risk associated with leaving that node unexplored**, and we can impose the requirement that the search will continue until the risk associated with every (unexplored) node on OPEN becomes lower than some acceptable level δ.

This is precisely what R_δ^* sets out to do, except that it pursues this objective in a best-first manner. It associates with every node on OPEN a **cost threshold** parameter $C_\delta(n)$, given by the solution to the equation:

$$R(C) = \delta \tag{3.23}$$

and chooses for expansion that node n from OPEN with the lowest $C_\delta(n)$. $C_\delta(n)$ represents the largest solution cost allowable before the risk associated with missing a potentially better solution emanating from n exceeds δ.

Figure 3.3 illustrates typical $R(C)$ curves for three nodes n_1, n_2, and n_3 and can serve to explain the rationale for expanding the node with the lowest

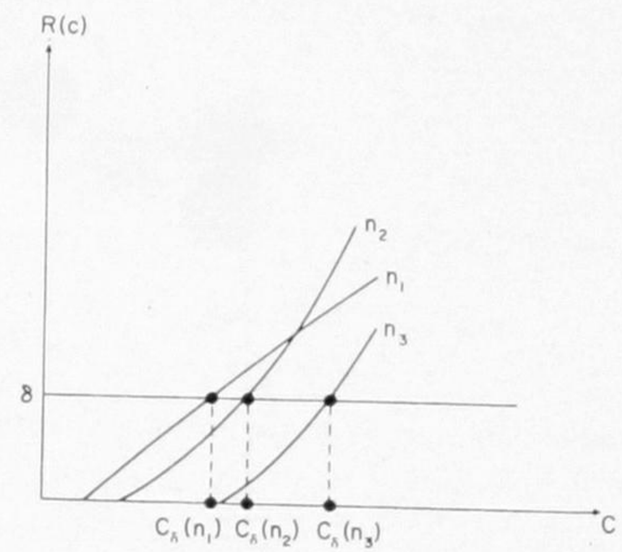

Figure 3.3
Typical $R(C)$ functions for three nodes in OPEN.

$C_\delta(n)$. If we choose n_1 for expansion and discover that it is a goal node (with $h(n_1) = 0$), then its cost $g(n_1)$ must be at most $C_\delta(n_1)$ (see Eq. 3.28). Hence, leaving n_2 and n_3 unexpanded is permissible because their associated risk measures $R_{n_2}[g(n_1)]$ and $R_{n_3}[g(n_1)]$ are both guaranteed to be below δ. This guarantee does not hold if we choose for expansion n_2 or n_3, which have higher cost thresholds than n_1 at the given level of δ. Note that because of possible crossings of the $R(C)$ curves, the order of node expansion should vary depending on the risk tolerance δ.

DEFINITION: *An algorithm is said to be* ***δ-risk-admissible*** *if it always terminates with a solution cost C such that $R(C) \leqslant \delta$ for every node left on OPEN.*

An alternative way of stating the δ-risk-admissible condition is that at termination, the cost of the solution found is not greater than $C_\delta(n)$ for each n on OPEN.

DEFINITION OF R_δ*: *R_δ* is an ordered search algorithm which is identical to A* except that it chooses for expansion that node from OPEN with the lowest cost-threshold $C_\delta(n)$.*

Note that with $\delta = 0$, R_δ* is identical to A* since it is guided by the solution to $R(C) = 0$ which is the lowest point on the tail of the density of f^+, that is, a lower bound to $g + h^*$. For $\delta > h^*$, R_δ* selects nodes by a criterion combining both the mean and the tail of the distribution.

Before demonstrating that R_δ* is indeed δ-risk-admissible, the reader may find it useful to examine some typical risk measures. Three common measures are defined as follows:

1. R_1—*The worst case risk:*

$$R_1(C) = \sup_{y:\rho_{f^+}(y)>0} (C - y) = C - g - h_a \tag{3.24}$$

where h_a is the lowest imaginable value of h^* (i.e., the lowest h^* with positive density)

2. R_2—*The probability of suboptimal termination:*

$$R_2(C) = P(C - f^+ > 0) = \int_{y=-\infty}^{C} \rho_{f^+}(y)\,dy \tag{3.25}$$

3. R_3—*The expected risk:*

$$R_3(C) = E[\max(C - f^+; 0)] = \int_{y=-\infty}^{C} (C - y)\rho_{f^+}(y)\,dy \tag{3.26}$$

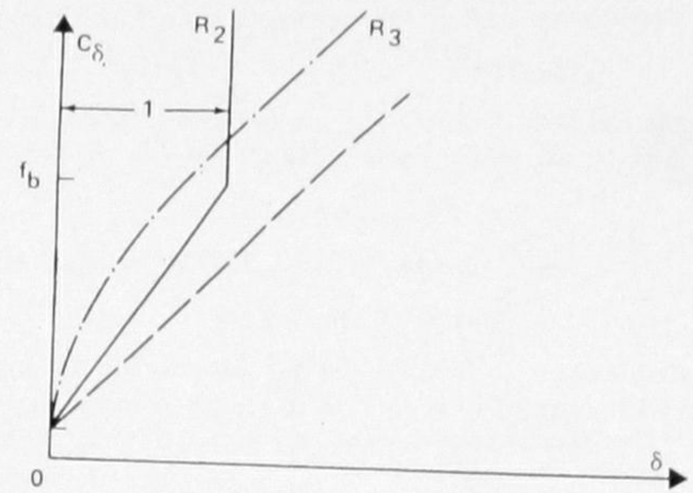

Figure 3.4
The horizontal axis represents the functions $R(C)$ for the three risk measures considered, when f^+ is uniformly distributed between f_a and f_b. Conversely, if the risk tolerance is fixed at δ, the vertical axis represents the cost threshold C_δ.

Figure 3.4 depicts the shape of these risk measures for the case where f^+ is uniformly distributed between f_a (an optimistic estimate) and f_b (a pessimistic estimate):

$$\rho_{f^+}(y) = \begin{cases} \dfrac{1}{f_b - f_a} & f_a \leqslant y \leqslant f_b \\ 0 & \text{elsewhere} \end{cases}$$

Symbolically, $R_3(C)$ is given by:

$$R_3(C) = \begin{cases} 0 & C \leqslant f_a \\ \dfrac{(C - f_a)^2}{2(f_b - f_a)} & f_a \leqslant C \leqslant f_b \\ C - \dfrac{(f_a + f_b)}{2} & f_b \leqslant C \end{cases}$$

The cost threshold C_δ associated with the expected risk measure R_3 is given by:

$$C_\delta = \begin{cases} f_a + \sqrt{2(f_b - f_a)} & 0 \leqslant \delta \leqslant \dfrac{f_b - f_a}{2} \\ \delta + \dfrac{f_b + f_a}{2} & \dfrac{f_b - f_a}{2} \leqslant \delta \end{cases} \tag{3.27}$$

which clearly demonstrates its role in rating nodes for expansion; if the risk-tolerance δ is higher than half the uncertainty range $f_b - f_a$, $R_\delta{}^*$ will rate nodes according to the mean values $(f_a + f_b)/2$. Under a lower tolerance for risk (i.e., $\delta < (f_b - f_a)/2$) $R_\delta{}^*$ will pay a greater attention to the optimistic estimate f_a until, when no risk at all can be tolerated ($\delta = 0$), the optimistic estimate f_a will become the sole criterion for node selection.

The exact shape of $\rho_{h^*}(n)$ can be computed only in those cases where the arc costs are generated by a well-defined probabilistic model and where the graph searched is of relatively simple structure. This should not prevent us, however, from computing good estimates for $C_\delta(n)$. Optimistic and pessimistic bounds to h^* are normally easy to compute for each node n, and moreover we often have a fairly good idea of where, between these bounds, h^* is most likely to concentrate. Using this information, the value of $C_\delta(n)$ can be estimated by making simplifying assumptions regarding the shape of $\rho_{h^*}(x)$, for example, uniform, triangular, or exponential. Since $R_\delta{}^*$'s performance is only guaranteed in probabilistic terms, these simplifications, being one additional level removed from exactitude, will have only second order effect on the performance guarantees.

We are now ready to prove some of the stated properties of $R_\delta{}^*$, assuming that $\rho_{h^*}(x)$ is known exactly.

THEOREM 14. *$R_\delta{}^*$ is δ-risk-admissible with respect to risk measures R_1, R_2, or R_3.*

Proof: Let $\delta > 0$ and assume $R_\delta{}^*$ terminates at node t with cost $g(t)$. Since $h(t) = h^*(t) = 0$, there remains no uncertainty regarding $f^*(t)$, and $C_\delta(t)$ is given by:

$$C_\delta(t) = \begin{cases} g(t) + \delta & \text{for } R_1 \text{ and } R_3 \\ g(t) & \text{for } R_2 \end{cases} \tag{3.28}$$

(The reader may wish to verify Eq. (3.28) by inspecting the behavior of the cost threshold functions in Figure 3.4 in the limit $f_b = f_a = g$.) Thus we conclude that for all three risk measures

$$g(t) \leqslant C_\delta(t).$$

At the same time the goal node t, having just been selected for expansion, must have the lowest $C_\delta(\cdot)$ on OPEN. Therefore, for each node n on OPEN we have:

$$C_\delta(n) \geqslant g(t)$$

which is the condition defining δ-risk-admissibility. ∎

It remains to show that $R_\delta{}^*$ terminates for all graphs with positive branch costs for which a solution path exists. This will be demonstrated at the end of the section.

Note that for the R_1 and R_3 measures, $R_\delta{}^*$ guarantees a termination risk that is even lower than δ. For these measures we, in fact, showed that:

$$g(t) \leqslant C_\delta(n) - \delta \qquad \forall\, n \text{ on OPEN} \tag{3.29}$$

or, that the cost found at termination would always be at least δ lower than the cost threshold of every node on OPEN. In particular, for R_1 we always have:

$$C_\delta(n) = \delta + g(n) + h_u(n)$$

where h_u is the lowest feasible value of h^*, and therefore:

$$g(t) \leqslant g(n) + h_u(n)$$

implying that the solution cost $g(t)$ is optimal or, equivalently, that $R_\delta{}^*$ is strictly admissible for all values of δ. This is to be expected since the node-rating criterion $C_\delta(n) = \delta + g + h_u$ is equivalent to that of A^* using the admissible heuristic $h = h_u$.

Note that the guarantees provided by $R_\delta{}^*$ are only as reliable as the density function describing the uncertainty of h^*. If this density function represents the user's subjective estimate of the magnitude of h^*, then $R_\delta{}^*$ only provides protection against the *subjectively* assessed risks associated with a premature termination. Alternatively, if $\rho_{h^*}(y)$ is obtained from a large statistical sample of problem instances, the risk protection of $R_\delta{}^*$ assumes statistical meaning with respect to the same problem ensemble.

The preceding discussion centered around the requirement of keeping the termination risk below a *fixed* tolerable level δ. For the risk measures, R_1 and R_3, it may be desirable to limit the termination risk by a fixed *percentage* of the solution cost. The results can easily be generalized to the case of proportional risk tolerance if instead of $C_\delta(n)$ we use the cost threshold $C'_\delta(n)$, where $C'_\delta(n)$ is the solution of

$$\frac{R(C')}{C'} = \delta \tag{3.30}$$

All the conditions necessary for proving Theorem 14 remain valid when $C'_\delta(n)$ is used as a node selection criterion and $0 \leqslant \delta \leqslant 1$.

3.2.4 $R_{\delta,\epsilon}^*$—A Speedup Version of $R_\delta{}^*$

The search of $R_\delta{}^*$, like that of A^*, is guided solely by the safeguards installed to keep the solution cost within an acceptable range. It uses no information regarding the computational effort required to reach a solution and therefore, like A^*, will suffer from the same deficiency that induced us to devise the $A_\epsilon{}^*$ algorithm: spending an unwarrantedly long time selecting the best among roughly equal solution paths.

To remedy this deficiency, the scheme employed by $A_\epsilon{}^*$ can be applied directly to $R_\delta{}^*$ at the price of risking only a slight deterioration in solution cost.

which clearly demonstrates its role in rating nodes for expansion; if the risk-tolerance δ is higher than half the uncertainty range $f_b - f_a$, $R_\delta{}^*$ will rate nodes according to the mean values $(f_a + f_b)/2$. Under a lower tolerance for risk (i.e., $\delta < (f_b - f_a)/2$) $R_\delta{}^*$ will pay a greater attention to the optimistic estimate f_a until, when no risk at all can be tolerated ($\delta = 0$), the optimistic estimate f_a will become the sole criterion for node selection.

The exact shape of $\rho_h{}^*(n)$ can be computed only in those cases where the arc costs are generated by a well-defined probabilistic model and where the graph searched is of relatively simple structure. This should not prevent us, however, from computing good estimates for $C_\delta(n)$. Optimistic and pessimistic bounds to h^* are normally easy to compute for each node n, and moreover we often have a fairly good idea of where, between these bounds, h^* is most likely to concentrate. Using this information, the value of $C_\delta(n)$ can be estimated by making simplifying assumptions regarding the shape of $\rho_{h^*}(x)$, for example, uniform, triangular, or exponential. Since $R_\delta{}^*$'s performance is only guaranteed in probabilistic terms, these simplifications, being one additional level removed from exactitude, will have only second order effect on the performance guarantees.

We are now ready to prove some of the stated properties of $R_\delta{}^*$, assuming that $\rho_h{}^*(x)$ is known exactly.

THEOREM 14. *$R_\delta{}^*$ is δ-risk-admissible with respect to risk measures R_1, R_2, or R_3.*

Proof: Let $\delta > 0$ and assume $R_\delta{}^*$ terminates at node t with cost $g(t)$. Since $h(t) = h^*(t) = 0$, there remains no uncertainty regarding $f^*(t)$, and $C_\delta(t)$ is given by:

$$C_\delta(t) = \begin{cases} g(t) + \delta & \text{for } R_1 \text{ and } R_3 \\ g(t) & \text{for } R_2 \end{cases} \tag{3.28}$$

(The reader may wish to verify Eq. (3.28) by inspecting the behavior of the cost threshold functions in Figure 3.4 in the limit $f_b = f_a = g$.) Thus we conclude that for all three risk measures

$$g(t) \leqslant C_\delta(t).$$

At the same time the goal node t, having just been selected for expansion, must have the lowest $C_\delta(\cdot)$ on OPEN. Therefore, for each node n on OPEN we have:

$$C_\delta(n) \geqslant g(t)$$

which is the condition defining δ-risk-admissibility. ∎

It remains to show that $R_\delta{}^*$ terminates for all graphs with positive branch costs for which a solution path exists. This will be demonstrated at the end of the section.

Note that for the R_1 and R_3 measures, $R_\delta{}^*$ guarantees a termination risk that is even lower than δ. For these measures we, in fact, showed that:

$$g(t) \leqslant C_\delta(n) - \delta \qquad \forall\, n \text{ on OPEN} \tag{3.29}$$

or, that the cost found at termination would always be at least δ lower than the cost threshold of every node on OPEN. In particular, for R_1 we always have:

$$C_\delta(n) = \delta + g(n) + h_a(n)$$

where h_a is the lowest feasible value of h^*, and therefore:

$$g(t) \leqslant g(n) + h_a(n)$$

implying that the solution cost $g(t)$ is optimal or, equivalently, that $R_\delta{}^*$ is strictly admissible for all values of δ. This is to be expected since the node-rating criterion $C_\delta(n) = \delta + g + h_a$ is equivalent to that of A^* using the admissible heuristic $h = h_a$.

Note that the guarantees provided by $R_\delta{}^*$ are only as reliable as the density function describing the uncertainty of h^*. If this density function represents the user's subjective estimate of the magnitude of h^*, then $R_\delta{}^*$ only provides protection against the *subjectively* assessed risks associated with a premature termination. Alternatively, if $\rho_{h^*}(y)$ is obtained from a large statistical sample of problem instances, the risk protection of $R_\delta{}^*$ assumes statistical meaning with respect to the same problem ensemble.

The preceding discussion centered around the requirement of keeping the termination risk below a *fixed* tolerable level δ. For the risk measures, R_1 and R_3, it may be desirable to limit the termination risk by a fixed *percentage* of the solution cost. The results can easily be generalized to the case of proportional risk tolerance if instead of $C_\delta(n)$ we use the cost threshold $C'_\delta(n)$, where $C'_\delta(n)$ is the solution of

$$\frac{R(C')}{C'} = \delta \tag{3.30}$$

All the conditions necessary for proving Theorem 14 remain valid when $C'_\delta(n)$ is used as a node selection criterion and $0 \leqslant \delta \leqslant 1$.

3.2.4 $R_{\delta,\epsilon}^*$—A Speedup Version of $R_\delta{}^*$

The search of $R_\delta{}^*$, like that of A^*, is guided solely by the safeguards installed to keep the solution cost within an acceptable range. It uses no information regarding the computational effort required to reach a solution and therefore, like A^*, will suffer from the same deficiency that induced us to devise the $A_\epsilon{}^*$ algorithm: spending an unwarrantedly long time selecting the best among roughly equal solution paths.

To remedy this deficiency, the scheme employed by $A_\epsilon{}^*$ can be applied directly to $R_\delta{}^*$ at the price of risking only a slight deterioration in solution cost.

Chapter 8

Strategies and Models for Game-Playing Programs

The skill of playing games such as chess, checkers, or GO has long been regarded as a distinctive mark of human intelligence. It is natural, therefore, that the general public continues to monitor the success of game-playing programs as a measure of progress in artificial intelligence. This fact has helped elevate the status of such programs from merely an expensive amusement to a level worthy of scientific pursuit. Game-playing research is today a thriving activity at many prestigious research institutes and universities.

Games, like the puzzles discussed in Chapter 1, offer a perfect laboratory for studying complex problem-solving methodologies. With a few parsimonious rules, one can create complex situations that require no less insight, creativity, and expertise than problems actually encountered in areas such as business, government, scientific, legal, and others. Moreover, unlike these applied areas, games offer an arena in which computerized decisions can be evaluated by absolute standards of performance and in which proven human experts are both available and willing to work toward the goal of seeing their expertise emulated by a machine. Last, but not least, games possess addictive entertaining qualities of a very general appeal. That helps maintain a steady influx of research talents into the field and renders games a convenient media for communicating powerful ideas about general methods of strategic planning.

We will not attempt to cover all the intricate subtleties that enter the technology of game-playing programs. Rather, our aim is to give a unified treatment of the various control strategies used in game searching and to quantify their performances using synthetic game models.

8.1 SOLVING AND EVALUATING GAMES

8.1.1 Game Trees and Game-Playing Strategies

Most games played by computer programs, including chess, checkers, and GO, are **two-player, perfect-information** games. There are two adversary players who alternate in making moves, each viewing the opponent's failure as his own success. At each turn, the rules of the game define both what moves are legal and what effect each possible move will have, leaving no room for chance. In contrast to card games in which the players' hands are hidden, or to the game of backgammon where the outcome of a die determines the available moves, each player has complete information about his opponent's position and about the choices available to him. The game begins from a specified initial state and ends in a position that, using a simple criterion, can be declared a win for one player and a loss for the other, or possibly as a draw.

A **game tree** is an explicit representation of all possible plays of the game. The root node is the initial position of the game, its successors are the positions that the first player can reach in one move, their successors are the positions resulting from the second player's replies, and so on. Terminal or leaf nodes are those representing WIN, LOSS, or DRAW. Each path from the root to a terminal node represents a different complete play of the game.

The correspondence between game trees and AND/OR trees is obvious. The moves available to one player from a given position can be represented by OR links, whereas the moves available to his opponent are AND links since a response must be contemplated to each one of them. Another way of obtaining this correspondence is to view each game position J as a problem statement: "find a winning strategy (for player 1) from J" or equivalently "show that player 1 can force a win from J." Clearly, if J admits the first player's moves, this problem is solved if a winning strategy can be found from *any one* of J's successors, hence the OR links. Similarly, if it is the opponent's turn to move from J, then J is solved if player 1 can force a win from *each and every one* of J's successors, hence the AND links. Thus, in games the process of problem reduction is completely dictated by the rules of the game; each legal move available to the opponent defines a subproblem or a subgoal and all these subproblems must be solved before the parent problem is declared solved.

For reasons that will become clear later, we call the first player MAX and his opponent MIN. Correspondingly we refer to game positions where it is MAX's or MIN's turn to move as MAX or MIN positions, respectively. The trees representing our games contain two types of nodes: MAX nodes (at even levels from the root) and MIN nodes (at odd levels from the root). Instead of using curved arcs, as in the AND/OR graphs of Chapter 2, we distinguish between MAX and MIN positions using a different node shape; the former is represented by squares and the latter by circles (see Figure 8.1). The leaf nodes in a game tree are labeled WIN, LOSS, or DRAW, depending on whether they represent a win, loss, or draw position from MAX's viewpoint.

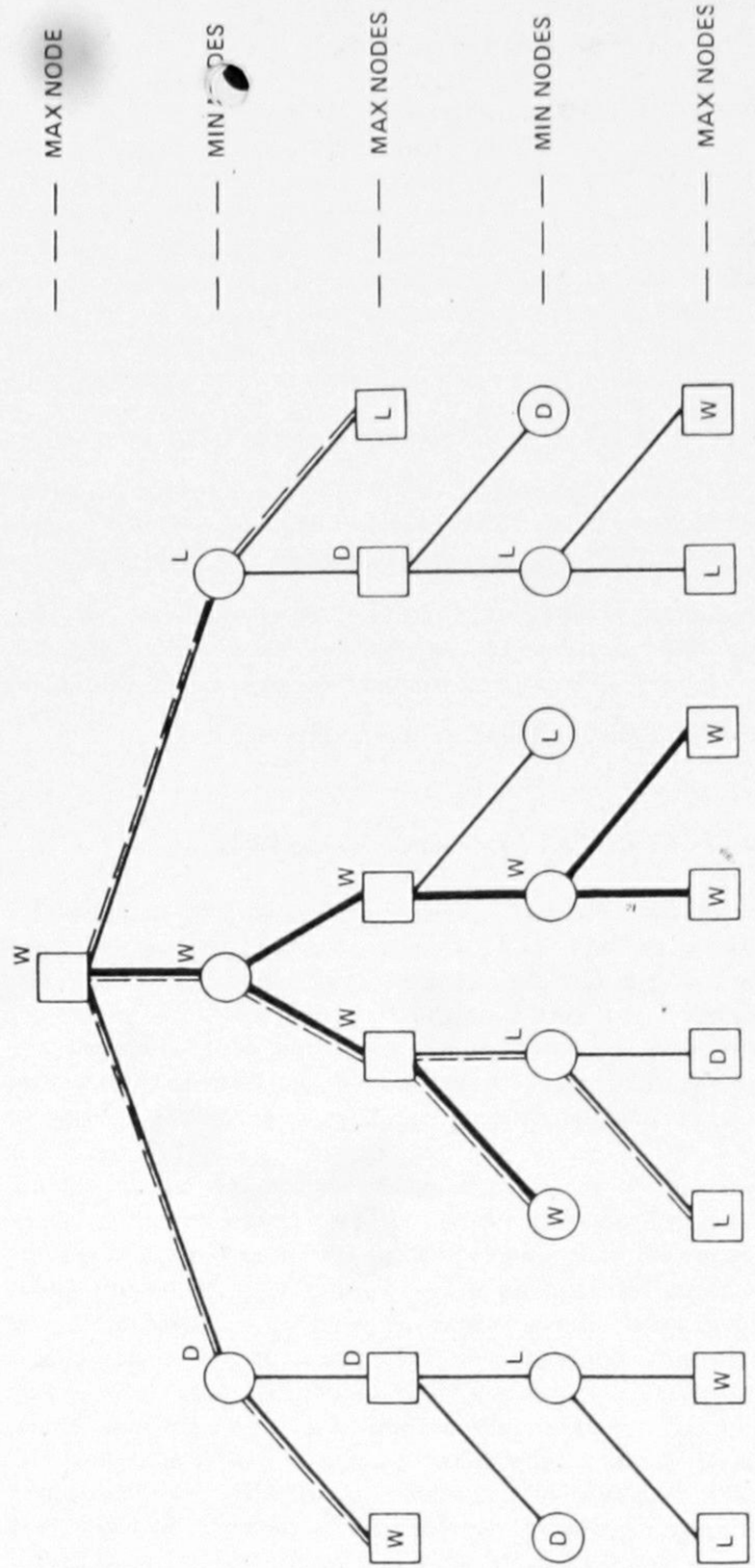

Figure 8.1
An evaluated WIN-LOSS-DRAW game tree showing a MAX strategy (boldface tree) and a MIN strategy (in broken lines).

Once the leaf nodes are assigned their WIN-LOSS-DRAW **status**, each node in the game tree can be labeled WIN, LOSS, or DRAW by a bottom-up process similar to the "solve" labeling procedure discussed in Chapter 1.

Status labeling procedure:

If J is a nonterminal MAX node, then:

$$\text{STATUS}(J) = \begin{cases} WIN & \textit{if any of J's successors is a WIN} \\ LOSS & \textit{if all J's successors are LOSS} \\ DRAW & \textit{if any of J's successors is a DRAW} \\ & \textit{and none is a WIN} \end{cases} \tag{8.1}$$

If J is a nonterminal MIN node, then:

$$\text{STATUS}(J) = \begin{cases} WIN & \textit{if all J's successors are WIN} \\ LOSS & \textit{if any of J's successors is a LOSS} \\ DRAW & \textit{if any of J's successors is a DRAW} \\ & \textit{and none is a LOSS} \end{cases} \tag{8.2}$$

The function STATUS(J) should be interpreted as the best terminal status MAX can achieve from position J if he plays optimally against a perfect opponent. Figure 8.1 depicts a simple game tree together with the status of all nodes. The status of the leaf nodes is assigned by the rules of the game, whereas those of nonterminal nodes are determined by the preceding labeling procedure.

Solving a game tree T means labeling the root node s as WIN, LOSS, or DRAW. Associated with each root label there is an optimal playing strategy which prescribes how that label can be guaranteed regardless of how MIN plays. A **strategy** for MAX is a subtree T^+ of T called a **solution tree** which is rooted at s and contains one successor of every nonterminal MAX node in T^+ and all successors of every nonterminal MIN node in T^+. A game-playing strategy T^- for MIN will contain, of course, the opposite types of nodes; one successor of every nonterminal MIN node and all successors of every nonterminal MAX node included in T^-. Of particular interest to us are **winning strategies**, strategies that guarantee a WIN for MAX regardless of how MIN plays. Clearly, a winning strategy for MAX is a solution tree T^+ whose terminal nodes are all WIN. Figure 8.1 shows a winning strategy for MAX (in heavy lines) and one nonwinning strategy for MIN (following broken lines).

Consider now an arbitrary pair of strategies; one for MAX, T^+, and one for MIN, T^-. It is easy to see that the two sets of terminal nodes associated with the two subtrees have exactly one leaf node in common. The intersection of the two strategies defines the unique play path that results if both players

adhere to their corresponding strategies and the one common leaf node is, in fact, the end position that results from this play.

Let (T^+, T^-) denote the leaf node common to strategies T^+ and T^-. Suppose MAX is forced to choose a strategy T^+ ahead of the play, to show it to the opponent, and then stick to it during the play. What T^+ would be his best choice? Being at such a disadvantage, MAX should reason as follows: If I choose T^+, then my opponent, knowing all my plans, would definitely respond so as to lead me toward the least favorable leaf in T^+ with label $\min_{T^-} \text{STATUS}(T^+, T^-)$. Now that I have the option of choosing T^+, I can guarantee myself $\max_{T^+} \min_{T^-} \text{STATUS}(T^+, T^-)$. On the other hand, suppose we reverse the roles and put MIN at the disadvantage of adhering to a predisclosed strategy T^-. By a similar argument MIN could guarantee that MAX would not achieve any status better than $\min_{T^-} \max_{T^+} \text{STATUS}(T^+, T^-)$.

An important consequence of our assumption of perfect information games is that these two guarantees are equal to each other and, moreover, they are given by the status of the root node as computed by Eqs. (8.1) and (8.2). Thus,

$$\text{STATUS}(s) = \max_{T^+} \min_{T^-} \text{STATUS}(T^+, T^-) \tag{8.3}$$

or

$$\text{STATUS}(s) = \min_{T^-} \max_{T^+} \text{STATUS}(T^+, T^-) \tag{8.4}$$

The validity of these two equalities can be easily proven using bottom-up induction on a general game tree (exercise 8.1). The interpretation of these alternate definitions, however, is rather profound; it implies that in perfect information games it does not matter if you choose a rigid plan ahead of time or make your decisions as the game goes along. Moreover, rather than conducting an optimization exercise over the enormous space of strategy pairs, one can find an optimal playing strategy using the status labeling procedure of Eqs. (8.1) and (8.2).

Although the significance of this result is mainly theoretical, we will find several occasions where it will be more convenient to view STATUS(s) as a product of optimization over strategies rather than the value returned by a labeling procedure. For example, such an occasion arises when we wish to answer the following question: Suppose someone claims that the root node of a certain game tree evaluates to a DRAW, what kind of information must he furnish to substantiate this claim? Had the claim been that s is a WIN, clearly we would only need to exhibit one winning strategy. Similarly, to defend the assertion "s is a LOSS," we need only demonstrate the existence of one winning strategy for MIN, that is, a MIN strategy with all LOSS leaf nodes. However, now that the claim is "s is a DRAW," would a single strategy suffice?

Eqs. (8.3) and (8.4) imply that *two* strategies are now needed. From Eq. (8.3) we see that as soon as we find a MAX strategy T^+ containing no LOSS leaves,

we immediately know that, no matter what MIN does, MAX can guarantee at least a DRAW. Moreover, as soon as we find a MIN strategy T^- with no WIN nodes, Eq. (8.4) implies that, no matter what MAX does, MIN can prevent him from obtaining a WIN. Thus, **two adversary strategies with compatible values** are both necessary and sufficient to verify that the game is a DRAW.

This result establishes an absolute limit on the number of terminal nodes that must be examined before a game tree can be solved. All the leaf nodes of two compatible strategies T^+ and T^- must be examined in case the game is a DRAW, whereas a single strategy is sufficient in case of a WIN or a LOSS. Equivalently, the task of solving a game can be viewed as the task of finding at most one pair of compatible strategies. We will show later that a pair of strategies is all that is required to certify the value of a more general game tree, even when the leaf nodes can take on more than three possible values.

8.1.2 Bounded Look-Ahead and the Use of Evaluation Functions

Size Considerations. The status labeling procedure described in the preceding section requires that a complete game tree, or at least a sizable portion of it, be generated. For most games, however, the tree of possibilities is far too large to be generated and evaluated backward from the terminal nodes in order to determine the optimal first move. A complete game tree for checkers, for instance, has been estimated as having about 10^{40} nonterminal nodes (Samuel, 1959). Generating the whole tree would require around 10^{21} centuries even if 3 billion nodes could be generated each second. The size of the game tree for chess is even more astronomical, estimated at about 10^{120} nodes or 10^{101} centuries at the same generation rate.

These numbers clearly demonstrate the absurdity of trying to generate the game tree exhaustively or even of hoping to find a winning strategy by luck. For, even if someone hands us a winning strategy for checkers, we would not know how to use it. A strategy should contain a prescribed response to each possible move of the opponent. Branching out once in every *two* levels of the game, the number of moves contained in a typical strategy is about the square root of the number of nodes in the game tree. That still leaves about 10^{20} moves for a single certified checkers strategy which poses an insurmountable requirement on storage space. Thus we must abandon all attempts to either find or retain a strategy that offers absolute end game guarantees.

Using Evaluation Functions. Having no practical way of evaluating the exact status of successor game positions, one may naturally resorts to heuristic approximations. Experience teaches us that certain features in a game position contribute to its strength, whereas others tend to weaken it. Thus we may be able to select a reasonably "good" move if we can list a sufficient number of features for each one of the resulting positions, aggregate their contributions into a single **evaluation function** e, and then choose the move leading toward the game position of highest merit.

Such a simplistic decision would obviously lead to poor play unless the evaluation function were extremely accurate, that is, if it correlated very closely with the actual status of the nodes evaluated. However, since e is based only on the static characteristics of game positions, it is very hard to achieve high levels of accuracy even using sophisticated evaluation functions.

Bounded Look-Ahead. An obvious generalization is not to use the board evaluation immediately, but use it after the play has extended through several rounds of move and counter move. In other words, we first explicate a reasonable portion of the game tree, assess the merit of the nodes on the search frontier using a **static evaluation function**, use these values to estimate the potential strength of ancestor nodes up to the immediate successors of the positions at hand, and finally choose the move that leads toward the successor with the highest potential strength. The most popular method of calculating the **backed-up values** of the ancestor nodes is based on the **face-value principle** discussed in Chapter 2, namely, the static evaluation of nodes on the search frontier are taken at their face value, as if these were true terminal payoffs given to MAX upon reaching any of these nodes.

The Minimax Rule. This assumption dictates that the backed-up values of ancestor nodes should be calculated using the following **minimax rule**:

1. The value $V(J)$ of node J on the search frontier is equal to its static evaluation function $e(J)$; otherwise
2. If J is a MAX node, $V(J)$ is equal to the maximum value of any of its successors.
3. If J is a MIN node, $V(J)$ is equal to the minimum value of any of its successors.

The minimax decision procedure is founded on two heuristics; the first is employed in the computation of $e(J)$, estimating the strength of a game position from its static characteristics. The second is used to compute the backed-up values while pretending that the frontier nodes were true terminal nodes and their evaluations true payoffs. Note that if the first heuristic is correct (i.e., $e(J) = \text{STATUS}(J)$), the second propagates the values exactly. If, however, the static evaluations are only rough estimates, the minimax propagation rule only offers a crude approximation to the method by which these estimates should be combined.

It is normally presumed that "the backed-up evaluations give a more accurate estimate of the true values of MAX's possible moves than would be obtained by applying the static evaluation function directly to those moves and not looking ahead to their consequences" (Barr and Feigenbaum, 1981). The underlying presumption is the notion that "the merit of a situation clarifies as it is pursued and that the look-ahead process can extend far enough that even rough board-evaluation formulas may be satisfactory" (Winston, 1977). The

appropriateness of these presumptions is critically analyzed in Chapter 10, where alternatives to minimax are also proposed.

8.1.3 MIN-MAX versus NEG-MAX Notations

So far the notation used in the preceding section evaluated games from one player's viewpoint. Both the WIN-LOSS status of the nodes considered as well as their assessed strength were viewed with respect to the objectives of the first player, MAX. That necessitated that we distinguish between MAX and MIN nodes, and that we define separate propagation rules for each node type.

An alternative approach, which leads to a more uniform notation called **NEG-MAX,** always evaluates a node from the viewpoint of the player whose turn it is to move at that node. A terminal node may assume a status of WIN, LOSS, or DRAW if the player whose turn it is to move wins, loses, or ties at this node, respectively. We call this mover-oriented status **MSTATUS**. The MSTATUS of a nonterminal node is dictated by those of its successors as follows:

$$\text{MSTATUS}(J) = \begin{cases} WIN & \textit{if any of } J\textit{'s successors has } \text{MSTATUS} = LOSS \\ LOSS & \textit{if all } J\textit{'s successors have } \text{MSTATUS} = WIN \\ DRAW & \textit{none of the above} \end{cases}$$

Eq. (8.5) combines Eqs. (8.1) and (8.2) into one rule, common to both MAX and MIN nodes. Moreover, comparing Eqs. (8.5) and (8.1), we see that the two rules become identical if we only complement the values of the successors' status. Equivalently, adapting the code

$$WIN = +1 \qquad DRAW = 0 \qquad LOSS = -1$$

MSTATUS(J) can be calculated by the formula

$$\text{MSTATUS}(J) = \max_{J'} \{ -\text{MSTATUS}(J') \mid J' \textit{ is a son of } J \} \tag{8.6}$$

hence the term NEG-MAX. The best move for either player is then to a node with the *minimum* MSTATUS value, because the minimum value to the player who moves from a successor of J means the highest value to the player moving from J.

The NEG-MAX notation also simplifies the description of the minimax rule. Instead of having a separate rule for MAX and MIN nodes, we now have a single rule for the NEG-MAX value, $MV(J)$ (connoting "mover-value") of J:

$$MV(J) = \begin{cases} Me(J) & \textit{if } J \textit{ has no successor} \\ \max_{J'} \{ -MV(J') \mid J' \textit{ is a son of } J \} \end{cases} \tag{8.7}$$

The NEG-MAX notation is useful for obtaining concise descriptions of game-searching algorithms and recursive performance formulas. The MIN-MAX notation, on the other hand, is more useful in explaining the rationale behind such algorithms and formulas. People find it easier to view the world from a fixed reference point rather than shifting back and forth between two adversaries. We will use the MIN-MAX notation in Chapter 8 and shift to NEG-MAX notation in Chapters 9 and 10.

8.2 BASIC GAME-SEARCHING STRATEGIES

Almost all game-playing programs use variants of the look-ahead minimax heuristic: the game tree is searched to some search frontier, a static evaluation function is computed to assess the strength values of nodes at that frontier, and minimax roll back is used to compute approximations of the strength values of shallower nodes. This procedure involves two major computational efforts: generating the partial game tree and evaluating the frontier positions. Highly sophisticated evaluation functions may permit us to get by with shallow search, whereas deep search may make up for deficiencies in the evaluation function.

Once we agree on the type of static evaluation function to be used, the search effort is directly proportional to the number of frontier nodes that are generated, examined, and evaluated. That number has become a standard measure of the complexity of game-searching procedures and this section describes several search strategies developed to reduce this number.

8.2.1 Exhaustive Minimaxing and the Potential for Pruning

The status of a node J and its minimax value $V(J)$ are both defined by a bottom-up induction which can be easily translated into a recursive search procedure. For example, we can write:

Algorithm: MINIMAX (Depth-First Version)

To determine the minimax value $V(J)$, do the following:

1. If J is terminal, return $V(J) = e(J)$; otherwise
2. Generate J's successors $J_1, J_2, \ldots, J_b$.
3. Evaluate $V(J_1), V(J_2), \ldots, V(J_b)$ from left to right.
4. If J is a MAX node, return $V(J) = \max[V(J_1), \ldots, V(J_b)]$.
5. If J is a MIN node, return $V(J) = \min[V(J_1), \ldots, V(J_b)]$.

This search procedure amounts to systematically traversing the search tree from left to right in a depth-first fashion. There is no need, of course, to gen-

erate all successors at once and keep them in storage until all are evaluated. Instead, the search can be executed in a backtracking style; successors are generated and evaluated one at a time and the value of the parent node updated each time a son is evaluated. This is accomplished in the following version of MINIMAX:

Algorithm: MINIMAX (Backtracking Version)

To determine the minimax value $V(J)$ of node J, do the following:

1. If J is terminal, return $V(J) = e(J)$. Otherwise;
2. For $k = 1, 2, \ldots, b$, do:
   a. Generate J_k, the kth successor of J.
   b. Evaluate $V(J_k)$.
   c. If $k = 1$, set $CV(J) \leftarrow V(J_1)$. Otherwise, for $k \geqslant 2$,
      set $CV(J) \leftarrow \max[CV(J), V(J_k)]$ if J is MAX or
      set $CV(J) \leftarrow \min[CV(J), V(J_k)]$ if J is MIN.
3. Return $V(J) = CV(J)$.

The variable $CV(J)$ represents the updated "current value" of node J. Step 2(b) is performed by calling MINIMAX recursively.

Note that in both versions of MINIMAX the evaluation of J is not completed until all its successors are evaluated. Consequently this procedure examines every node on the search frontier and evaluates every node in the search space.

To demonstrate that MINIMAX is wasteful and that exhaustive evaluation is not necessary, let us consider the case of bi-valued terminal nodes, that is $e(J) \in \{\text{WIN, LOSS}\}$. Calculating the minimax values of node J amounts to determining the WIN-LOSS status of J or to *solving* the game rooted at J. The following straightforward algorithm, which we call SOLVE, determines the status of J without evaluating every node in the tree.

Algorithm: SOLVE

To solve an arbitrary node J, do the following:

1. If J is a terminal node, return its WIN-LOSS status directly; otherwise
2. Start solving J's successors from left to right (calling SOLVE recursively).
3. If J is a MAX node, return a WIN as soon as one successor is found to be a WIN; return a LOSS if all successors of J are found to be a LOSS.
4. If J is a MIN node, return a LOSS as soon as one successor is found to be a LOSS; return a WIN if all successors of J are found to be a WIN.

Clearly SOLVE does not evaluate every node in the tree. It is a backtracking search that traverses the tree from left to right but skips all nodes that cannot provide useful information. For example, as soon as one successor of a MAX node is found to be a WIN, the father can be declared a WIN without

solving the rest of its children. The overall savings achieved by this method can be substantial and it depends, of course, on the order in which the successors are evaluated.

SOLVE is probably the simplest game-searching algorithm. It also lends itself readily to average case analysis (see Section 8.3) and can be shown to be optimal under rather general conditions (Tarsi, 1983). We therefore use SOLVE as a standard to gauge the performance of more intricate algorithms such as those that search game trees with multi-valued terminal nodes.

SOLVE draws its efficiency from two factors: the determination to examine only nodes that may influence the status of the top node being evaluated and our *a priori* knowledge that the terminal nodes are bi-valued. At first glance it may appear that the second is the dominant factor and that very little pruning can be achieved without knowing, for example, that the father node cannot obtain a status higher than WIN or lower than LOSS. It turns out, however, that although the bi-valued nature of the evaluation function is helpful, *it is not essential*. Comparable savings also can be achieved for multi-valued or continuous valued games, even when the range of values is not known in advance.

One way in which such savings can be realized is by the celebrated alpha-beta (α-β) pruning algorithm which is described in the next section.

8.2.2 The α-β Pruning Procedure: A Backtracking Strategy

Description of the α-β Procedure. The α-β pruning algorithm is the most commonly used procedure in game-playing applications, where it serves to speed up game searching without loss of information. The algorithm determines the minimax value of the root of a game tree by traversing the tree in a predetermined order, say from left to right, skipping all those nodes that no longer can influence the minimax value of the root.

The method is demonstrated in Figure 8.2, which shows a binary game tree of depth $d = 4$ with MAX and MIN nodes represented by squares and circles, respectively. The numbers inside the terminal squares represent evaluations of the game positions at the frontier of the search tree, whereas those at higher levels are the minimax values computed by the α-β procedure. The heavy branches represent the search tree actually generated by the α-β procedure as it traverses the game tree from left to right. Nodes not on that search tree are skipped ("or cut off") by α-β because they cannot provide useful information.

The rationale for node skipping can be explained by examining Figure 8.3(a), which depicts the traversal path reached by α-β while skipping the node labeled C of Figure 8.2. The purpose of exploring node B has been to find out if the value of A should be reduced below 10, which is the value established for A's leftmost son. However, the facts that one of B's sons has already attained the value 14 and that B is a MAX node imply that the value of B must be at least 14, regardless of any information that C may provide. Therefore, nothing can alter the fact that the value of A is exactly 10, and C can be cut off from the search.

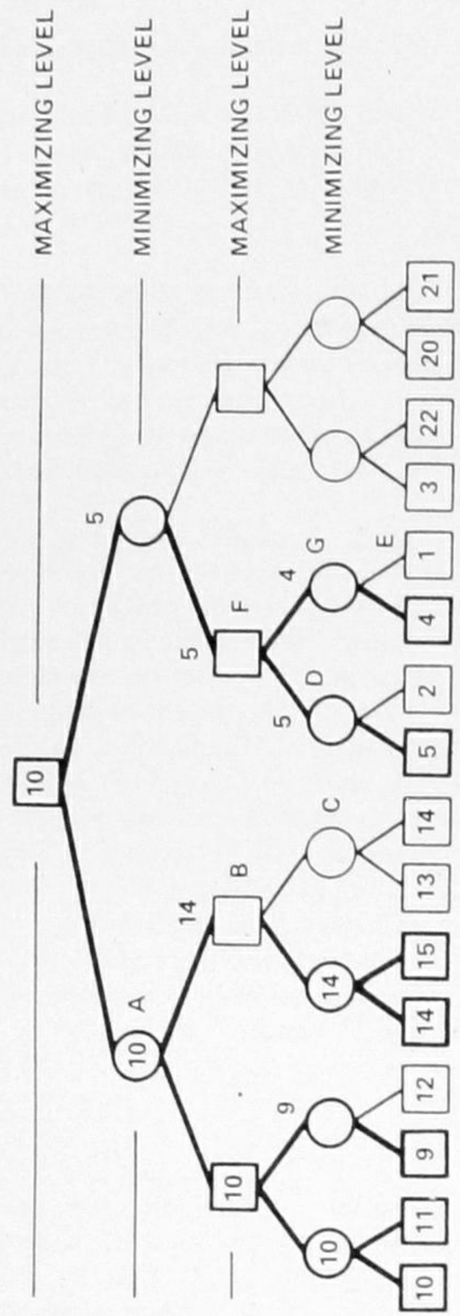

Figure 8.2
A game tree exhibiting alpha-beta (α-β) pruning.

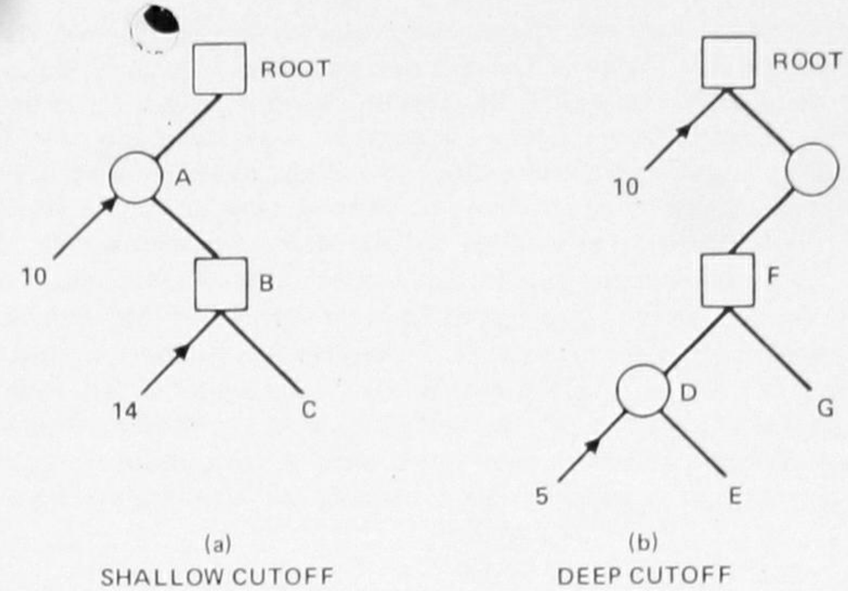

Figure 8.3
Shallow and deep cutoffs in alpha-beta search.

A similar argument can be used to justify cutting off the node labeled E in Figure 8.2. When this cutoff is decided, α-β has reached the path shown in Figure 8.3(b); the leftmost son of the root has delivered the value 10 and D is being explored to find out if it can increase the root value above 10. Clearly, as soon as one of D's sons reveals the value 5, D's value cannot exceed 5 and there is no way that D can cause the root value to rise above 10. Therefore E and D are discarded and the search moves to the next brother of D.

The example shown in Figure 8.3(a) demonstrates a **shallow cutoff** in the sense that the information responsible for cutting off node C came only from its siblings and uncles, but not from higher-level nodes. By contrast, the condition shown in Figure 8.3(b) demonstrates a **deep cutoff**; if it were not for the information received from its great-grandfather, the root node, E could not have been ignored.

In practice, these considerations translate into a very simple pruning scheme:

> Perform the backtracking version of minimax search with one exception; if in the course of updating the minimax value of a given node this value crosses a certain bound, then no further exploration is needed beneath that node; its current-value CV can be transmitted to its father as if all of its sons have been evaluated.

The cutoff bounds are adjusted dynamically and are transmitted top-down as follows:

1. **α-bound**—The cutoff bound for a MIN node J is a lower bound called α, equal to the **highest current value of all MAX ancestors of J**. The explora-

tion of J can be terminated as soon as its current value CV equals or falls below α.

2. **β-bound**—The cutoff bound for a MAX node J is an upper bound called β, equal to the **lowest current value of all MIN ancestors of J**. The exploration of J can be terminated as soon as its current value CV equals or rises above β.

In Figure 8.3(a), for example, the exploration of node B begins with a β-bound of 10 which is received from A. As soon as the value 14 is received from its left son, B's current value (14) exceeds its β-bound (10) and all further explorations beneath B can be terminated. B now notifies its parent A to proceed with the other sons. In our example, since A has no more sons to the right of B, the evaluation of A is completed and it transmits its current value, 10, to its father.

The exploration of node D (Figure 8.3(b)) begins with an α-bound of 10 which propagates down from the root. The value of D drops below 10 as soon as it receives the value 5 from its left son. This causes the exploration of D to terminate and the search continues at the right brother of D.

This last example also permits us to demonstrate how the bounds are **updated** so as to comply with their given definitions. Consider node F which begins its exploration with an α-bound of 10. The α-bound that F should deliver to its right son G must equal the highest value over all MAX's ancestors of G, namely, the maximum of 10 and the value delivered by D. Since D delivers the value 5, G receives an α-bound of 10. Had the minimax value D been 15, F would deliver down to G the α-bound of 15.

These pruning and bound-updating operations are summarized by the following recursive procedure which we call $V(J; \alpha, \beta)$. It receives two parameters, $\alpha < \beta$, and evaluates $V(J)$, the minimax value of node J, if the latter lies between α and β. Otherwise, it returns either α (if $V(J) \leqslant \alpha$) or β (if $V(J) \geqslant \beta$). Clearly, if J is a root of a game tree, its minimax value will be obtained by $V(J; -\infty, +\infty)$.

α-β Procedure: $V(J; \alpha, \beta)$

1. If J is terminal, return $V(J) = e(J)$. Otherwise, let $J_1, \ldots, J_k, \ldots, J_b$ be the successors of J, set $k \leftarrow 1$ and, if J is a MAX node, go to step 2; else go to step 2′.
2. Set $\alpha \leftarrow \max[\alpha, V(J_k; \alpha, \beta)]$.
3. If $\alpha \geqslant \beta$, return β; else continue.
4. If $k = b$, return α; else proceed to J_{k+1}, i.e., set $k \leftarrow k + 1$ and go to step 2.

2′. Set $\beta \leftarrow \min[\beta, V(J_k; \alpha, \beta)]$.
3′. If $\beta \leqslant \alpha$, return α; else continue.
4′. If $k = b$, return β; else proceed to J_{k+1}, i.e., set $k \leftarrow k + 1$ and go to step 2′.

A more concise definition of the α-β algorithm using NEG-MAX notation, and a proof that it returns the minimax value of the root node can be found in Knuth and Moore (1975).

Clearly, the efficiency of this search method depends on the order of the terminal values. For the values shown in Figure 8.2, only 7 terminal nodes are examined by a left-to-right search, whereas all 16 terminal nodes would be examined by a right-to-left search. In complex games, the difference between the best case and the worst case can be quite substantial, amounting to a factor of 2 in the depth of the look-ahead tree that a given computer system can afford to explore. This disparity warrants analysis of the *average* performance of α-β under the assumption that the terminal values are randomly ordered.

The analysis will be carried out in Chapter 9 and, strangely, it reveals that the pruning provided by α-β permits the search depth to extend roughly 33% deeper than that permitted by an exhaustive MINIMAX search, i.e., a third of the way between its worst case and best case values. However, before exposing this analysis, it is instructive to present α-β from a different perspective and to describe other algorithms that compete with α-β in game-searching applications.

Understanding α-β from an Influence Analysis Viewpoint. The pruning action of α-β draws its legitimacy from the fact that the pruned nodes cannot **influence** the root node, no matter what their value turns out to be. Let us examine now what information must be gathered by the search so that we can decide that a particular node no longer has any influence on the value of the root. That information will lead to a simple criterion for determining whether a given node is generated by α-β and will play an important role in the analysis of α-β's expected performance.

Assuming that at a certain stage of the search the traversal path P_{s-J} has reached the situation described in Figure 8.4, all the terminal nodes to the left of this path have been explored, the minimax values of all subtrees stemming to the left of P_{s-J} are known, a decision must now be made whether node J should be explored or pruned. The information gathered by the search is summarized by the parameters $\alpha(K)$, $\beta(L)$, $\alpha(M)$, $\beta(N)$, . . . where $\alpha(K)$ is the maximum of the values of the subtrees emanating from K to the left of P_{s-J}, $\beta(L)$ is the minimum of the values of the left subtrees emanating from L, and so on. For uniformity, we agree that, if a node Z on P_{s-J} has no sons to the left of P_{s-J}, its associated parameters will be $\alpha(Z) = -\infty$ or $\beta(Z) = +\infty$, depending on whether Z is MAX or MIN.

Our task now is to determine for what strings of parameters $\alpha(K)$, $\beta(L)$, ... we can declare J as having no influence on the root node value $V(s)$. We pretend that J is the last node to be evaluated before the root value is to be declared, and that $V(J)$ is a variable x, ranging from $-\infty$ to $+\infty$. The root value $V(s)$, if it is affected by $V(J)$, would correspondingly change its value along some curve $V(s) = F_J(x)$. If $F_J(x)$ is flat, J has no impact on s and can be pruned away; otherwise it must be explored lest $V(s)$ be determined erroneously. We call $F_J(x)$ the **influence function** of the root with respect to the node J.

The calculation of $F_J(x)$ is facilitated by the fact that, if J has any influence at all on s, that influence must be transmitted via the intervening nodes along

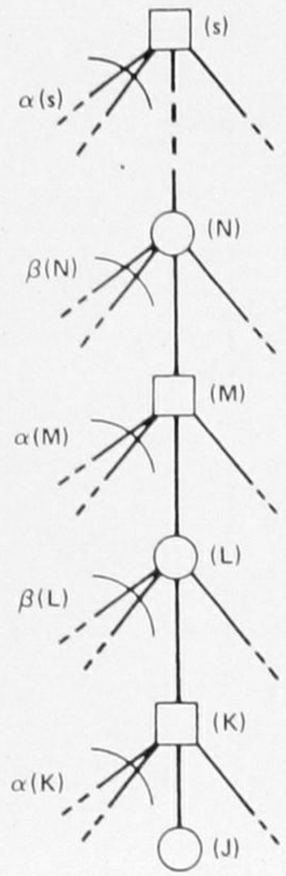

Figure 8.4
A traversal path in the midst of alpha-beta search.

P_{s-J}. In other words, in order for J to influence the root node, it must influence its father K, its grandfather L, and so on. Thus, instead of calculating $F_J(x)$ directly, we can compute it as a **cascade** of several influence functions in much the same way that we compute the output of a cascaded chain of electronic circuits, treating the output of one circuit as an input to its successor.

Let $f_{K,J}(x)$ characterize the direct influence of J on its father K, that is $V(K) = f_{K,J}[V(J)]$. If we know both $f_{K,J}(x)$ and $F_K(x)$, we can compute $F_J(x)$ using the composite function:

$$F_J(x) = F_K[f_{K,J}(x)] \tag{8.8}$$

Similarly, we may continue this process through all the intervening nodes between J and s, and obtain

$$F_J(x) = f_{s,}(\ldots(f_{L,K}(f_{K,J}(x)))) \tag{8.9}$$

Let us now examine the direct influence function $f_{K,J}(x)$ between a son J and its father K. This is an elementary min or max function which depends only on

the parameter α and β available to the father node. If the father K is a MAX node with parameter α, then

$$f_{K,J}(x) = f_\alpha^+(x) \triangleq \max(\alpha, x) \tag{8.10}$$

If the father K is a MIN node with parameter β, then

$$f_{K,J}(x) = f_\beta^-(x) \triangleq \min(\beta, x) \tag{8.11}$$

Thus, $F_J(x)$ can be written

$$F_J(x) = \ldots f_{\beta(N)}(f_{\alpha(M)}(f_{\beta(L)}(f_{\alpha(K)}(x)))) \tag{8.12}$$

The shape of the functions $f_\alpha^+(x)$, $f_\beta^-(x)$, and the pair cascade

$$L_{\alpha,\beta}(x) \triangleq f_\beta^-(f_\alpha^+(x)) = \begin{cases} x & \alpha \leqslant x \leqslant \beta \\ \alpha & x \leqslant \alpha \leqslant \beta \\ \beta & x \geqslant \alpha \geqslant \beta \\ \beta & \beta \leqslant \alpha \end{cases} \tag{8.13}$$

are shown in Figure 8.5. $L_{\alpha,\beta}(x)$ describes a linear 45° ramp truncated below α and above β. If $\beta \leqslant \alpha$, the ramp degenerates to a flat horizontal line at level β, a condition that we simply label FLAT. Note that the cascade $f_\beta^-(f_\alpha^+(x))$ is identical to $f_\alpha^+(f_\beta^-(x))$ if $\alpha \leqslant \beta$; the two only differ in the level of the horizontal line when $\alpha > \beta$. However, since for the purpose of influence analysis the level of a flat function is irrelevant, we may conclude that compositions of ramp functions are commutative:

$$f_\beta^-(f_\alpha^+(x)) = f_\alpha^+(f_\beta^-(x))$$

This allows us to group together all the f^+ functions in the expression for $F_J(x)$ in Eq. (8.12) and to write

$$F_J(x) = \ldots f_{\beta(N)}^-(f_{\beta(L)}^-(\ldots f_{\alpha(M)}^+(f_{\alpha(K)}^+(x))))$$

Note that the family of $f^+(\cdot)$ functions is **closed under composition**, that is, composing two functions $f_{\alpha_1}^+(x)$ and $f_{\alpha_2}^+(x)$ yields another member $f_{\alpha_3}^+(x)$ of the same family, where $\alpha_3 = \max(\alpha_1, \alpha_2)$. Similarly, the family of $f^-(\cdot)$ functions is also closed:

$$f_{\beta_1}^-(f_{\beta_2}^-(x)) = f_{\beta_3}^-(x) \qquad \beta_3 = \min(\beta_1, \beta_2)$$

That permits us to rewrite $F_J(x)$ as a single truncated ramp

$$F_J(x) = f_{B(J)}(f_{A(J)}^+(x)) = L_{A(J),B(J)}(x)$$

where

$$A(J) = \max[\alpha(K), \alpha(M), \ldots]$$

$$B(J) = \min[\beta(L), \beta(N), \ldots] \tag{8.14}$$

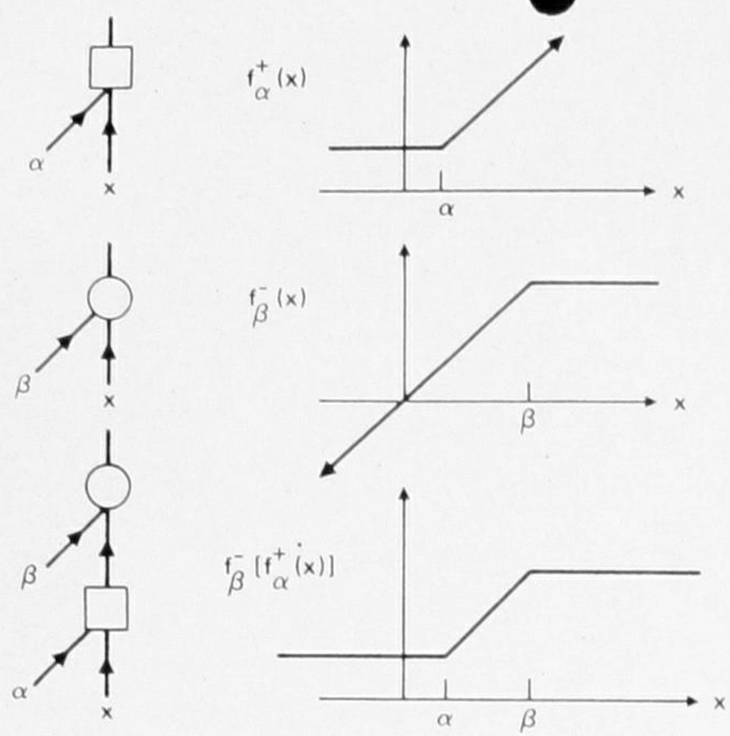

Figure 8.5
The influence functions $f_\alpha^+(x)$ and $f_\beta^-(x)$, and their composition $f_\beta^-(f_\alpha^+(x))$.

Thus the influence functions of all nodes are truncated 45° ramps each characterized by its own truncation points A and B. Obviously, $F_J(x)$ will be FLAT if $A(J) \geqslant B(J)$ in which case J can be pruned; otherwise J must be explored because its value may affect that of the root. This explains both the cutoff condition used by α-β and the process by which the bounds are updated; the α- and β-bounds that are passed down in the α-β search are exactly the parameters A and B above.

A Necessary and Sufficient Condition for Node Generation. The preceding discussion leads to a simple criterion for determining whether a given node is generated by the α-β algorithm:

For any node J, form the path leading from the root to J, and define the following quantities:

$A(J)$ = *the highest minimax value among all left-sons of MAX ancestors of J.*
$B(J)$ = *the lowest minimax value among all left-sons of MIN ancestors of J.*

J will be generated by α-β if and only if:

$$A(J) < B(J) \tag{8.15}$$

The reader can easily verify that in Figure 8.2 all nodes generated satisfy the preceding criterion, whereas all those satisfying $A(J) \geqslant B(J)$ can provide no information beyond that which has already been gathered by the search and will be cut off. For example, for the right-most leaf node we have:

$$A(J) = \max[\min(3, 22), 10] = 10$$

$$B(J) = \min\{20, \max[\min(5, 2), \min(4, 1)]\} = 2$$

and since $A(J) > B(J)$, it is not generated by the α-β search.

At first glance, it may seem strange that the flatness criteria $A(J) \geqslant B(J)$ coincides with the pruning criterion of α-β; after all, the former relies on the actual minimax values of the subtrees emanating from the ancestors of J, whereas α-β, exploiting its pruning power, does not in fact evaluate the minimax values of all these subtrees. In Figure 8.2, for example, the minimax value of node F is 2, but the current value with which α-β abandons F is as high as 5.

On deeper thought, however, it becomes clear that the two criteria should coincide. We can demonstrate this fact by reasoning inductively. Assume that J is the first node that satisfies the flatness condition $A(J) \geqslant B(J)$, then $V(J)$ cannot affect $V(s)$ and, moreover, it also *cannot affect the influence function of any node J' to the right of J.* Therefore, we can pretend that $V(J)$ is equal to the minimax value of J's left brother. That would make the minimax value of J's father identical to the current value assigned to the father by α-β while pruning J. Because this substitution leaves the influence functions of all future nodes unaltered, we conclude that the next node pruned by α-β must also satisfy the flatness condition. We can now repeat that same argument from the second FLAT node and show that the fictitious value that α-β assigns to this node's father also leaves all future nodes with the same influence function, and so on. A formal proof of the statement in Eq. (8.15) can be found in Chapter 9 (Appendix 9-B) as a special case of the more elaborate condition of SSS*.

The criterion $A(J) \geqslant B(J)$, first derived by Fuller, Gaschnig, and Gillogly (1973), has a definite analytical advantage over the equivalent pruning criterion expressed by the operational statement "the current value of J's father exceeds its β-bound." The latter depends on two local variables of the search algorithm whose values fluctuate widely during the course of the computation in an unpredictable manner. The former, on the other hand, explicitly expresses the cutoff condition in terms of the values of the terminal nodes, whose variability can more easily be assessed. For that reason, the criterion in Eq. (8.15) is useful in computing the average number of terminal nodes examined by α-β. One need only compute the probability $P[A(J) < B(J)]$ for every node J, then sum these probabilities over all terminal nodes. This analysis is carried out in Chapter 9.